Automated checker for detecting method-hiding in Java programs

M. Z. I. Nazir, M. Alqaradaghi, and T. Kozsik

Abstract—**Method overriding is a valuable mechanism that happens when an instance method is defined in a subclass and has the same signature and return type as an instance method in the superclass. However, in Java, if those methods are static, then instead method hiding happens, which is a programming weakness and may produce unexpected results. Static analysis is an approach in software testing that examines code to identify various programming weaknesses throughout the software development process without running it.**

This paper addresses the detection of method-hiding problem in Java programs. We implemented a new automated checker under the SpotBugs static analysis tool that can detect the mentioned problem. According to our results, the checker precisely detected the related issues in both custom test cases and real-world programs.

Index Terms—**Java, method-hiding, precise automated checker, static analysis, SpotBugs tool**

I. INTRODUCTION

FREQUENT cyberattacks on IT infrastructures drive cybersecurity research [1]. It is crucial to keep software free of weaknesses. Method overriding (also called late binding, run-time polymorphism, and dynamic polymorphism) happens when an instance method is defined in a subclass and has the same signature (method's name, parameters' numbers, and parameters' types) and return type as an instance method in one of the superclasses. In this case, the method in the subclass will override the one in the superclass. This programming mechanism is valuable. It enables a class to derive from a superclass that exhibits similar behavior and subsequently customize and/or extend the behavior as required [2, 3]. While the compiler in Java does not require the *@Override* annotation[1] to be present for the overridden method. Doing so is advised for the following reasons:

1) The compiler will produce an error if the method is not present in one of the superclasses, informing the programmer that this is not actually overriding and that he must fix it.

2) If the overridden method is static, the compiler will generate another type of error [2] which will instruct about the necessity of removing that annotation because it is not possible to override a static method. Omitting the *@Override* annotation in the latter case will make the compiler ignore this issue, leading to the problem of method hiding.

More specifically, method hiding happens when a subclass defines a static method with the same signature and the return type as a static method in the superclass. The method in the superclass, in this case, hides the one in the subclass [2]. Overriding and hiding methods have distinct differences in determining which method is called from a specific location. In the case of overriding, the method called is determined during runtime based on the specific instance of the object being used. On the other hand, hiding determines the method called during compile time by considering the specific qualified name or method invocation expression used at the call [3].

Method hiding is neither considered an error nor a compilation failure. However, according to the SEI CERT Oracle Coding Standard for Java, method hiding should be avoided because it often leads to unexpected results, especially when programmers mistakenly expect method overriding. This has been clarified under Rule 06. Methods (MET) MET07-J [4]. Moreover, according to the same web page, no free automated static analysis tool can detect this issue in Java code. Static analysis approaches save time, effort, and money by identifying software flaws and security vulnerabilities early in the software development process [5, 6]. These techniques are capable of identifying a wide variety of security flaws and vulnerabilities, from simple programming errors to more complex concerns like access control difficulties [5].

The motivations of this paper are:

- According to the TIOBE index [7], Java is still one of the most widely used programming languages despite some decline in popularity.
- Java is used to create many long-lasting programs that we use on a daily basis. It is crucial to keep these applications up to date and fix any flaws.
- Static analysis techniques are useful for finding code flaws and security issues.

The contributions of the paper are:

- Design and implement an automated checker named *FindHidingMethod* under the SpotBugs static analysis tool (SB) [8], which raises an issue when finding method hiding bugs in Java programs.
- Assess our approach and report the assessment results using *recall*, *false alarm rate*, and *precision* metrics.

M. Z. I. Nazir was with the Department of Programming Languages and Compilers, ELTE, Eötvös Loránd University, Budapest, Hungary. He is now with the Technical University Munich (e-mail: bsvncs@inf.elte.hu).

M. Alqaradaghi is with the Department of Programming Languages and Compilers, ELTE, Eötvös Loránd University, Budapest, Hungary and Northern Technical University, Kirkuk, Iraq (e-mail: alqaradaghi.midya@inf.elte.hu).

T. Kozsik is with the Department of Programming Languages and Compilers, ELTE, Eötvös Loránd University, Budapest, Hungary
(e-mail: kto@inf.elte.hu).

[1] *@Override* annotation instructs the compiler that you intend to override a method in the superclass.

[2] error: static methods cannot be annotated with the *@Override*.

DOI: 10.36244/ICJ.2024.2.3

B. Problem Statement

The issue of defining a static method in a subclass that has the same signature as a static method in the superclass is known as method hiding. Here, the superclass method is hidden by the subclass method. It is important to avoid method hiding because it can lead to confusion and unexpected behavior, especially when programmers mistakenly expect method overriding. Listing 1 presents simple Java code for method overriding versus method hiding.

LISTING 1
METHOD OVERRIDING VERSUS METHOD HIDING

```
1 class SuperClass {
2       public static void methodHiding() {
3               System.out.println("methodHiding (SuperClass)");
4       }
5       public void methodOverriding() {
6               System.out.println("methodOverriding (SuperClass)");
7       }
8 }
9 class HidingVsOverriding extends SuperClass {
10      public static void methodHiding() {
11              System.out.println("Method Hiding (SubClass)");
12      }
13      public void methodOverriding() {
14              System.out.println("Method Overriding (SubClass)");
15      }
16 }
17 public class MainClass {
18      public static void main(String[] args) {
19              SuperClass bs3 = new HidingVsOverriding();
20              bs3.methodOverriding();
21              bs3.methodHiding();
22      }
23 }
```

```
Results:
Method Overriding (SubClass)
Method Hiding (SuperClass)
```

As we can see here, method hiding may create confusion. It may become a source of a programming error when a static method is called using an instance of the subclass (an object) because, in this case, an inexperienced or incautious programmer may expect dynamic binding of the call to a method implementation defined in the dynamic type of the object (the subclass). However, even though some of the IDEs today provide hints that an instance should not call static methods and attributes, rather it should be called by the class (because static methods and attributes belong to the class and not to an instance), we have to assume that some people have nothing more than their compiler and a simple text editor, which will not catch such issues.

II. RELATED WORKS

No automated static analysis tool available for free can find methods hiding problems within Java programs [4]. However, some static analysis tools target various issues regarding the method overriding mechanism, which is very similar to the method hiding's problem. In this section, we present them.

PMD source code analyzer [9] targets the problem of useless overriding methods [10]. The related checker only raises an issue when an overridden method does not do any more than the method it overrides, marking it as useless. SB targets various issues related to method overriding in Java [11]. We will go through them in detail. The first one is when there is a call for an overridable method that performs a security check. This is considered an issue because the overridden method may compromise it and omit the checks. We have implemented this checker, in our previous work [12].

Due to the similarity of the upcoming SB's rules, we will explain them by grouping them into two groups. The first is related explicitly to how *equals*, *compareTo*, and toString methods are overridden. SB raises an issue when **a)** the *hashCode* method is not being overridden by the class overriding the *equals* method. This is an issue because, according to the contract of those two methods, equal objects should have equal hashcodes (i.e., calling the *hashCode* method on each of the two objects must produce the same integer result). **b)** a class defines a covariant version of the *equals* method. **c)** a class defines a covariant version of *compareTo* method. The last two rules state that the parameters of *equals* and *compareTo* methods must have type *java.lang.Object* otherwise, it is considered a bug **d)** a class defines a *toString* method that is not actually the one in the *java.lang.Object* class. The latter is probably what the programmer intended. The second rules group targets different overriding related issues. Those rules will raise an issue **e)** when a method overrides a method included in an Adapter class that implements a listener defined in the *java.awt.event* or *javax.swing.event* package. SB considers this an issue because this method will not be called when an event occurs. **f)** when an overriding method changes the superclass contract related to the Liskov Substitution Principle defined in a superclass. This is an issue since a subclass instance can be cast and handled as an instance of the superclass. **g)** when a class overrides an *equals* method in a superclass, and both methods use the *instanceof* operator to decide whether two objects are equal. This is problematic since it is important to ensure those two equal methods are symmetrical, i.e., a.equals(b) == b.equals(a). If B is a subtype of A, then there is a good chance that this method's equivalence connection is not symmetric. A's equals method verifies that the argument is an instance of A, and B's equals method verifies that the argument is an instance of B.

SonarQube static analysis tool (Sonar) [13] targets some of the previously mentioned issues, as shown in Table 1. However, it also targets other method overriding related issues of Java, listing them as rules [14]. We present the most important ones. An issue will be raised when these rules are violated. **a)** while not mandatory, using the *@Override* annotation on compliant methods improves readability by making it explicit that methods are overridden. According to this rule, *@Override* should be used to override and implement methods; **b)** In JUnit testing, to make sure that the test cases are set up and cleaned up consistently, the overriding implementations of *setUp* and *tearDown* methods should call the parent implementations explicitly because those two methods provide some shared logic that is called before all test cases. This logic may change over the lifetime of your codebase; **c)** a record class has an array field and is not overriding *equals, hashcode*, or *toString* methods.

This is an issue because array fields are compared by their reference, and overriding *equals* is highly appreciated to achieve the deep equality check. The same strategy applies to *hashCode* and *toString* methods; **d)** although overriding the *clone* method without implementing the *Cloneable* interface can be helpful if a programmer wants to control how subclasses clone themselves, it's probably a mistake. So, this rule suggests that classes that override clone should implement *Cloneable* and call the *super.clone* method. **e)** a class implementing the interface *Cloneable* but does not override the *clone* method is considered an issue because *Cloneable* is a marker interface that defines the contract of the *clone* method, which is to create a consistent copy of the instance. Since the compiler cannot enforce the definitions of marker interfaces (because they have no own API), when a class implements Cloneable but does not override the clone method, it likely violates the contract for Cloneable. Finally, **f)** the *Object.finalize* method should not be overridden. Relying on overriding it to release resources or update the program's state is highly discouraged because there is no guarantee that this method will be called as soon as the last references to the object are removed, which may lead to many issues. Table 1 presents the summary of the previously explained related works.

TABLE I
RELATED WORKS SUMMARY

The issue/rule	PMD	SB	Sonar	Ours
Useless overriding methods	✓		✓	
The methods that perform security checks must be declared private or final		✓		
hashCode method is not being overridden by the class that is overriding the *equals* method		✓	✓	
A class defines a covariant version of the *compareTo* method		✓		
A class defines a covariant version of the *equals* method		✓	✓	
A class defines a *toString* method that is not actually the one in the *java.lang.Object* class		✓		
A class overrides a method implemented in the superclass Adapter wrongly		✓		
Do not use the *instanceof* operator to decide whether two objects are equal		✓	✓	
Method overrides should not change contracts		✓	✓	
@Override annotation should be used for overriding and implementing methods			✓	
Junit test cases should call super metthods			✓	
equals, *hashcode*, and *toString* methods should be overridden in records containing array field			✓	
Classes that override *clone* should implement *Cloneable* and call the *super.clone* method			✓	
"Cloneables" should implement clone			✓	
The *Object.finalize* method should not be overridden			✓	
Never declare a class method that hides a method declared in a superclass or super interface				✓

Method hiding is considered as a weakness [4]. Moreover, many state-of-the-art static analysis tools focus on various issues regarding method overriding, which is the basis of method hiding. Still, none of these tools focus on method hiding. Our work sheds light on this problem and implements a checker that raises an issue when finding one.

III. RESEARCH METHODOLOGY

This section thoroughly describes our checker design process and the steps involved in developing the custom test cases.

A. Checker design

Listing 2 presents the pseudocode of the implementation of our checker named *FindHidingMethod*, using SB version 4.7.3. SB is written mainly in Java, so we implemented our checker using Java. The process starts by visiting each class in the program, then getting a list of all its superclasses, i.e., the parent class, grandparent class, etc., until reaching the last superclass, which is always the Object class. For each of the visited classes, our checker will check each of the methods and raise an issue when it finds a hidden subclass. More specifically, an issue will be raised when there is a subclass-superclass pair that includes methods with the same name, both of which are static, non-private, and not main (because it is an odd case of a static method that may exist in a superclass-subclass pair). However, the checker also considers the possibility of the method being a constructor and some other odd cases where it will be excluded (not reporting as an issue).

The checker has been developed successfully and has passed our team's internal review and the SB tool's public reviews. For further information about the implementation coding, please refer to the public review of our checker implementation on the official website of the SB tool [15].

LISTING 2
THE PSEUDOCODE OF OUR CHECKER

```
1 procedure FindHidingMethod( aClass ) is
2  foreach method in declared methods of class aClass loop
3   if method is static and non-private and not SpecialCase then
4    foreach superClass in superclasses of aClass loop
5     foreach superMethod in declared methods of class superClass lo
6      if signature(method) = signature(superMethod) then
7       report
8      end if
9     end loop
10   end loop
11  end if
12 end loop
13 end

14 function SpecialCase( method ) is
15  return   method is "non-private void main ( String[] )"
16       or method is "non-private void main( )"
17       or method is a constructor
18       or method is static_initializer_block
19       or method is a generated method
20 end
```

Complexity Analysis: SB static analysis tool, which we built our checker under, inspects the Java byte code for different programming vulnerabilities and weaknesses. Since the bytecode of the subclass does not contain the bytecode of any methods of the superclass (i.e., inherited methods), we had to analyze all the superclasses. The latter leads to O(n) complexity (where n is the number of superclasses), then goes over each of the methods, leading to a second loop with O(m) complexity (where m is the number of methods in each class). Combined together, it gives O(n*m), i.e., a quadratic time complexity. However, we do not generate the bytecode for each method every time we check for the vulnerability of that method; rather, we call a built-in method of the SB tool's environment, called *visitClassContext* [15], only once for the class. It scans all the methods and information related to the class. This balances out the double loop, producing an efficient checker (i.e., with linear complexity).

An Exception to Rule 06. Methods MET07-J: According to the SEI CERT Oracle Coding Standard for Java web page, which includes a description of the targeted issue of this paper, there is a case that should not be considered a violation of this rule, i.e., it should not be counted as a method hiding issue. This exception only applies when an API's hidden methods are called; in this scenario, all calls to hidden methods make use of qualified names or method invocation expressions that clearly indicate which particular method is being called [3]. The previously mentioned exception case has not been considered through our checker's implementation for two reasons:

1) When there is a method hiding in a program, it is considered unsafe, regardless of whether the related static method is being called.
2) The produced Java byte codes for calling a static method using a class name and calling it using a class instance are the same. Since SB uses bytecode to inspect the flaws in programs, any checker built under SB is not able to differentiate these two as well. More specifically, when using an instance variable for calling a static method, the JVM smartly fixes it when producing the bytecode. It makes it seem like the calling was happening on the class name. See Listing 3, where both invocation types (from lines 10 and 15) produce the bytecode **invokestatic.**

Therefore, whenever there are two identical static methods in any superclass-subclass pair, our checker will raise an issue and report the second method as a bug, no matter if any of the methods are being called or not (or whether they are called on a fully qualified name (class name) or an instance). This, however, can be considered a limitation of our implementation, and it may be addressed by implementing a checker under a static analysis tool that inspects the program source code instead of the bytecode (for example, under PMD Source Code Analyzer).

Note: Checkers implemented under SB may interact with the SB framework and be able to use its properties using either the *OpcodeStackDetector* abstract class or the *Detector* Interface. However, the previously explained scenario also made us decide to choose the *Detector* interface, which is a lower level since using the *OpcodeStackDetector* will only add extra complexity to our program.

B. Custom test cases' design

To assess our generated checker and attain thorough coverage of the problem being studied. The following points are covered by the test cases that our team has built:

1) Flawed and unflawed: To assess our checker performance in both true positive (TP) and false positive (FP) aspects [3], we designed non-compliant (NC), flawed test cases, and compliant (C), unflawed test cases.
2) Unambiguous: the test cases are written in a clear and concise way, leaving no room for misinterpretation.
3) Validated expectations: every test case has a predetermined expected result.
4) Test objective: every test case has a distinct goal that identifies the precise component of the problem being investigated.
5) Independence: We created separate test cases to isolate and identify problems more effectively.

As a result, we produced 11 C and 9 NC test cases. Next, we go into detail on the design of these test cases.

NC test cases: These are the flawed constructs. These test scenarios are considered insecure and involve real issues, so our checker should report them. We could cover every scenario in which methods in Java applications might be hidden by creating nine NC test cases.

C test cases: These are the unflawed constructs, i.e., include safe coding scenarios to test if our checker successfully ignores them. To cover all possible scenarios, we have designed eleven test cases.

You can check the test cases from the public review of our checker implementation on the official website of the SB tool [15]. The word *good* is used in the file name and/or method name of the C test cases, while the word *bad* is used in the NC test cases.

[3] TP is the number of flawed constructs that are detected correctly by our checker, while FP is the number of unflawed constructs that are mistakenly reported by the checker.

LISTING 3
THE PRODUCED BYTECODE OF CALLING A STATIC METHOD USING A CLASS INSTANCE AND A CLASS-QUALIFIED NAME

```
1 Invocation.java
2
3 class Super {
4           public static void staticMethod() {
5           }
6 }
7 public class Invocation {
8           public void invocationOnInstance() {
9                       Super sup = new Super();
10                      sup.staticMethod();
11          }
12
13          public void invocationOnClass() {
14                      Super sup = new Super();
15                      Super.staticMethod();
16          }
17 }

Compiled from "Invocation.java"

 public class Invocation {
 public Invocation();
   Code:
     0: aload_0
     1: invokespecial #1        // Method java/lang/Object."<init>":()V
     4: return

 public void invocationOnInstance();
   Code:
     0: new           #7        // class Super
     3: dup
     4: invokespecial #9        // Method Super."<init>":()V
     7: astore_1
     8: aload_1
     9: pop
    10: invokestatic  #10       // Method Super.staticMethod:()V
    13: return

 public void invocationOnClass();
   Code:
     0: new           #7        // class Super
     3: dup
     4: invokespecial #9        // Method Super."<init>":()V
     7: astore_1
     8: invokestatic  #10       // Method Super.staticMethod:()V
    11: return
}
```

IV. RESULTS AND DISCUSSION

This section presents our checker's analysis results of both bug types. The first type is intentional bugs; the custom test cases that have been explained in the previous section, while the second are real-world bugs.

A. Analyzing the custom test cases

For evaluation purposes, our team designed custom test cases. They have been explained in detail in Section 3.2. We used three metrics to represent our checker's performance in identifying methods hiding issues in Java programs. Formulas 1 through 3 present these metrics respectively. To calculate the metrics values, we first ran our checker on the test cases and then computed the number of TP and FP.

$$Recall = TP/NC \quad (1)$$

$$False\ alarm\ rate = FP/C \quad (2)$$

$$Precision = TP/(TP + FP) \quad (3)$$

Higher recall and precision values, while a lower value of false alarm rate indicates better performance. All values fall in the interval [0, 1].

Table 2 presents the results of running our checker on the custom test cases. It achieved optimal performance results.

TABLE II
ASSESSMENT RESULTS OF ANALYZING CUSTOM TEST CASES

Metrics	Values
NC test cases	9
C test cases	11
TP	9
FP	0
Recall	1.00
False alarm rate	0.00
Precision	1.00

B. Analyzing real-world software

We analyzed seven different pieces of software to determine how well our checker performed in identifying the target bug of this paper in real-world software. The software are: SB itself, maven-javadoc-plugin [17], mybatis-3 [18], spark [19], cayenne [20], Apache Hadoop [21], and Apache Dubbo [22] (In Table 3, they have been renamed to P1 through P7, respectively). Hence, we could only use the precision metric here because it is not straightforward to calculate the number of NC and C constructs when it comes to real-world software.

The analyzed software has been chosen arbitrarily from the GitHub web page. Out of 30 projects, the presented ones include method-hiding weaknesses. This indicates the popularity of this issue; it appears in 23% of the arbitrarily chosen projects. Table 3 presents the results, which revealed that our checker gave the highest possible precision for the analyzed programs. The numbers of TP and FP have been decided by manually reviewing the output report of the checker. You can check the second author's GitHub repository to find the reports of running the checker on the presented software [23].

TABLE III
ASSESSMENT RESULTS OF ANALYZING THE REAL-WORLD SOFTWARE

METRICS	P1	P2	P3	P4	P5	P6	P7
TP	4	2	6	38	2	35	6
FP	0	0	0	0	0	0	0
PRECISION	1.0	1.0	1.0	1.0	1.0	1.0	1.0

V. CONCLUSION

With the help of the SpotBugs static analysis tool, we have created and implemented a new checker called "FindHidingMethod" that can identify the issue of method hiding in Java programs. Our approach has been evaluated, and the results revealed that it was very precise when detecting related issues in the analyzed test cases and real-world programs.

REFERENCES

[1] M. Ufuk., Review of some recent European cybersecurity research and Innovation Projects, Infocommunications Journal, Vol. XIV, pp. 70–78, 2022, DOI: 10.36244/ICJ.2022.4.10.

[2] Java Documentation, Overriding and hiding methods, accessed on December 2023, https://docs.oracle.com/javase/tutorial/java/IandI/override.html

[3] Java Language Specification, Inheritance, Overriding, and Hiding, accessed on December 2023, https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.8

[4] SEI CERT Oracle Coding Standard for Java, MET07-J. Never declare a class method that hides a method declared in a superclass or super interface, accessed on December 2023, https://wiki.sei.cmu.edu/confluence/display/java/MET07-J.+Never+declare+a+class+method+that+hides+a+method+declared+in+a+superclass+or+superinterface

[5] B. Chess and J. West, Secure Programming with Static Analysis, Addison-Wesley, USA, 2007.

[6] M. Alqaradaghi, G. Morse and T. K., Detecting security vulnerabilities with static analysis – a case study, Pollack Periodica, Vol. 17, pp. 1–7, 2021, DOI: 10.1556/606.2021.00454.

[7] TIOBE Index for October, accessed on October 2023, https://www.tiobe.com/tiobe-index

[8] SpotBugs, Find Bugs in Java Programs, accessed on December 2023, https://spotbugs.github.io/

[9] PMD Source Code Analyzer., accessed on December 2023, https://pmd.github.io/

[10] Useless Overriding Method, accessed on December 2023, https://pmd.github.io/pmd/pmd_rules_java.html

[11] SpotBugs Bug Description, accessed on July 2024, https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html

[12] M. Alqaradagh, M.Z.I. Nazir, and T. Kozsik, Design and Implement an Accurate Automated Static Analysis Checker to Detect Insecure Use of SecurityManager. Computers, 12(12), p. 247., Computers 2023, 12(12), 247; DOI: 10.3390/computers12120247

[13] SonarQube Static Code Analysis Tool., accessed on December 2023, https://www.sonarsource.com/products/sonarqube/

[14] Sonar rules, Java static analysis, accessed on January 2024, https://rules.sonarsource.com/java/

[15] M.Z.I. Nazir, Public Review of MET07-J Checker, accessed on December 2023, https://github.com/spotbugs/spotbugs/pull/2467

[16] visitClassContext method, accessed on July 2024, https://github.com/spotbugs/spotbugs/blob/cc2bad5559f662dd997059606abc9d7e659f2a45/spotbugs/src/main/java/edu/umd/cs/findbugs/Detector.java#L36

[17] maven-javadoc-plugin, Apache Maven Javadoc Plugin., accessed on December 2023, https://github.com/apache/maven-javadoc-plugin

[18] mybatis-3, MyBatis SQL mapper framework for Java, accessed on January 2024, https://github.com/mybatis/mybatis-3

[19] Apache Spark – A unified analytics engine for large-scale data processing, accessed on December 2023, https://github.com/apache/spark

[20] cayenne, Mirror of Apache Cayenne, accessed on January 2024, https://github.com/apache/cayenne

[21] Apache Hadoop, accessed on January 2024, https://github.com/apache/hadoop

[22] Apache Dubbo, The Java implementation of Apache Dubbo. An RPC and microservice framework, accessed on January 2024, https://github.com/apache/dubbo

[23] M. Alqaradaghi, Running FindHidingMethod checker on real-world software, accessed on January 2024, https://github.com/Midya-ELTE/Running-FindHidingMethod-checker-on-real-world-software/tree/main

M. Z. I. Nazir was born in Lahore, Pakistan, in 2001. He received the B.Sc. degree in informatics from ELTE, Eötvös Loránd University, Budapest, Hungary, in 2023. Currently, he is studying M.Sc. informatics at Technical University Munich. He worked previously at different companies in Hungary. On microservices at Ericsson, on networks at Thermofisher, and on file transfer at CERN.

M. Alqaradaghi was born in Baghdad, Iraq. She received a B.Sc. degree in Information Technology from the Middle Technical University, Baghdad, Iraq, in 2006 and an M.Sc. degree in Computer Science from Sam Higginbottom University, India, in 2015. She is now a Ph.D. candidate at the Department of Programming Languages and Compilers, ELTE, Eötvös Loránd University, Budapest, Hungary. Her research focuses on finding security vulnerabilities in Java code using static analysis tools. From 2015 to 2019, she worked as a Teaching Assistant at the Northern Technical University, Kirkuk, Iraq. Since 2019, she has been working as an instructor in the programming languages lab in the Department of Programming Languages and Compilers, ELTE.

T. Kozsik is currently an Associate Professor with the Department of Programming Languages and Compilers, Eötvös Loránd University (ELTE). His research interests include formal verification, programming paradigms (i.e., functional programming, concurrent programming, and quantum computing), static analysis, refactoring, and domain-specific programming languages.