Reverse Engineering Environment for Teaching Secure Coding in Java

Young Lee[1], Jeong Yang[2]

[1]Dept. of Electrical Engineering and Computer Science
Texas A&M University-Kingsville
Kingsville, TX 78414, U.S.A
young.lee@tamuk.edu

[2]Dept. of Computing and Cyber Security
Texas A&M University-San Antonio
San Antonio, TX 78224 U.S.A.
jeong.yang@tamusa.edu

Abstract

Few toolsets for program analysis and Java learning system provide an integrated console, debugger, and reverse engineered visualizer. We present an interactive debugging environment for Java which helps students to understand the secure coding by detecting and visualizing the data flow anomaly. Previous research shows that the earlier students learn secure coding concepts, even at the same time as they first learn to write code, the better they will continue using secure coding practices. This paper proposes web-based Java programming environment for teaching secure coding practices which provides the essential and fundamental skills in secure coding. Also, this tool helps students to understand the data anomaly and security leak with detecting vulnerabilities in given code.

1. Introduction

Visualizing the interactions among objects in object-oriented software is difficult in the presence of inheritance, polymorphism, and dynamic binding. This visualization includes an object's state information in a collection of variables, and object's behaviors implemented by methods that use those variables. [2]
This project focuses on a teaching environment for teaching secure coding practices to the students using the proposed reverse-engineering tool. Our tool supports Java source code development along with synchronized static and dynamic visualizations, interactive debugging in a web-based programming environment. It aims to help students better understand dynamic control flow and data flow of Java programs by detecting the code of the security leak. This paper presents an initial evaluation of this tool to investigate its effectiveness and user satisfaction through quantitative and qualitative experiments.
Data flow analysis can identify data flow anomalies in the sequence of actions performed upon a program's data elements [1]. Tools based on data flow analysis of object-oriented software must analyze the object's behavior as well as object's status. This research aims to develop an approach for performing dynamic data flow analysis for object-oriented programs synced with source code, class diagram, object diagram, and sequence diagram.
We present an interactive debugging environment for Java which helps students to understand the secure coding by detecting and visualizing the data flow anomaly.

This paper makes several contributions about the reverse engineering tools for secure coding.
These contributions are:
1. Proposed reverse engineered tool presents a dynamic and static visualization for UML diagrams and data flow.
2. The paper presents a synced approach for dynamic visualizations (data flow, sequence diagram, object diagram) and static visualizations (class diagram, source code) to understand how the Object-Oriented design affect the behavior of objects at run-time.
3. A case study for detect data anomalies is presented. This case study allows to students to understand what causes the data anomaly or security leak.

2. Reverse Engineering Environment

The proposed tool visualizes the integrated structure and behavior of Java program in a platform-independent web-based environment. It provides synchronized static and dynamic visualization of Java programs such as class diagram, object diagram, sequence diagram, and data flow diagram. An overview of the system (see figure 1) and applied design principles are presented in [3, 4].
Through those synchronized diagrams, students get a better perspective of the structure of the Java code, the behavior, and interaction of the objects, and data flow along with control flow.

3. Case Study: Data Anomaly Detection

Overridden methods and polymorphism can cause a problem called the yo-yo effect with a simple inheritance hierarchy that is only three-level classes deep [2]. With an overridden method 'bounced' up and down among levels of an inheritance hierarchy, overridden methods and polymorphism can result in a data flow anomaly. The reverse-engineered diagrams such as class diagram, object diagram, sequence diagram, and data flow diagram can efficiently visualize the data flow along with control flow for these types of overridden methods in the presence of dynamic binding and polymorphism. Therefore, it is anticipated that the diagram helps programmers perform with less difficulty and required effort in tracing the sequence of calls and detecting data flow anomalies.
The arrows on figure 2 (a) show the overriding: h() and i() methods of class B override the methods in class A. The table shows the state variable definitions and uses for some of the methods in the hierarchy. Suppose that A is an actual type of an object o and call d(), which calls g(), which calls h(), which calls i(), which finally calls j(). In this case, the variables u and w are first defined in h(), then used in i() and j(). But, when class B is an actual type of object o and a call d() is made. This time B's version of h() and i() are called, u and w are not given values, and thus the call to j() of class A can result in a data flow anomaly. Figure 2 (b) is a yo-yo graph of this situation and illustrates the actual sequence of calls in case of actual type A, B, and C. In figure 3 (b) shows that the variable u and w are declared and used without the defining the values.

Figure 4 shows the user interface of the proposed reverse engineered tool. This tool visualizes each step of program execution on the class diagram, object diagram, sequence diagram, and data flow diagram. Students can step forward and step backward to see how each line of code affects the control flow (method call in sequence diagram), object status (object diagram), where the methods and variables are defined (class diagram), and define-use path (data flow diagram).

4. Conclusion

This project focused on a teaching environment for teaching secure coding practices to the students using the proposed reverse-engineering tool. The case study showed that the tool visualized the data anomalies of the Java source code with synchronized static and dynamic visualizations, and data flow diagrams. In conclusion, the proposed reverse engineered tool helps students better understand dynamic control flow and data flow of Java programs by detecting and visualizing the code of the ta anomalies and security leak.

References

[1] Scott F. Smith and Mark Thober, "Refactoring Programs to Secure Information Flows," ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (2006)
[2] Offutt, J., Alexander, R., Wu, Y., Xiao, Q., Hutchinson, C., "A fault model for subtype inheritance and polymorphism," In: Proceedings of the 12th International Symposium on Software Reliability Engineering. pp. 84-93, (2001)
[3] Jeong Yang, Young Lee, David Hicks, and Kai Chang, "Enhancing Object-Oriented Programming Education using Static and Dynamic Visualization," IEEE Frontiers in Education 2015: Launching a New Vision in Education Engineering, pp. 806-810, (2015)
[4] Jeong Yang, Young Lee, and David Hicks, "Synchronized Static and Dynamic Visualization in a Web-Based Programming Environment," IEEE International Conference on Program Comprehension (ICPC), May 16-17, (2016)

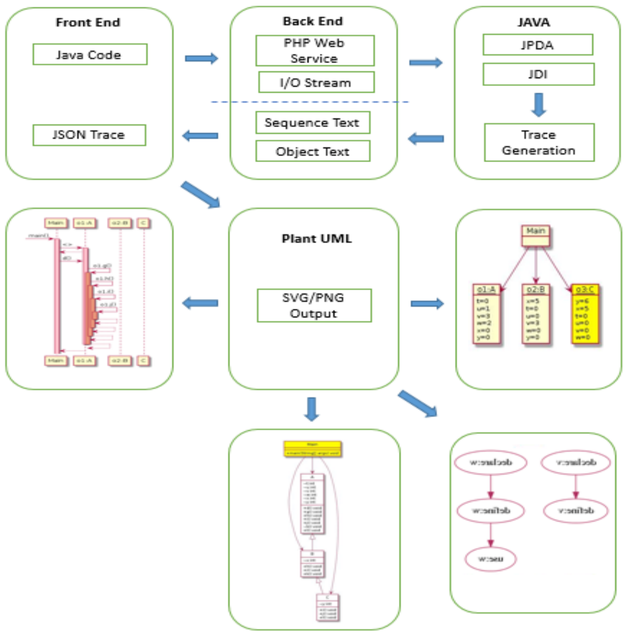
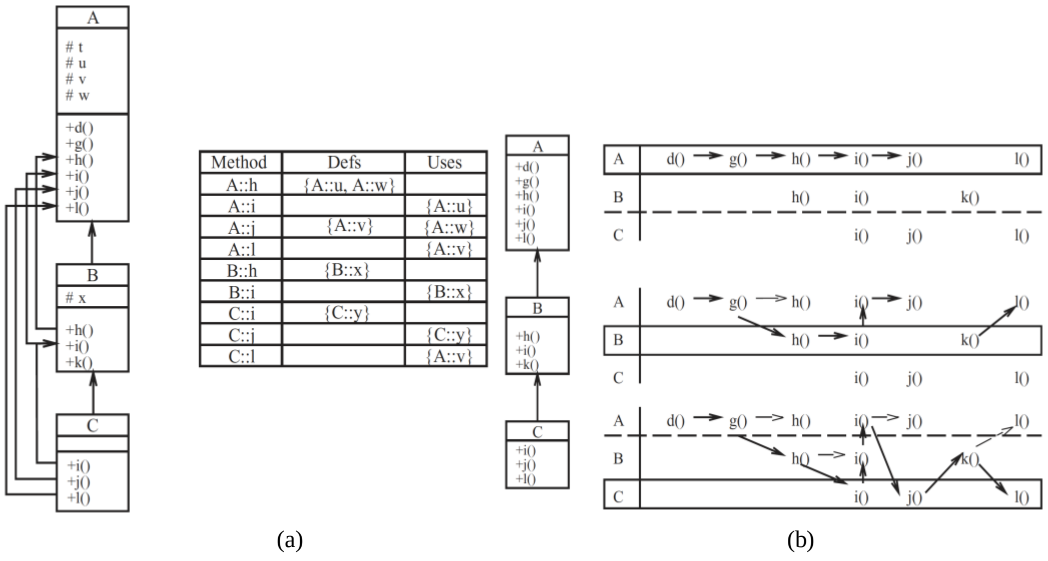
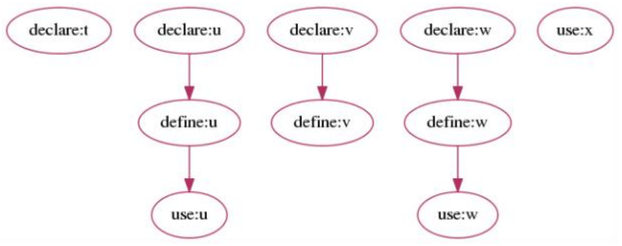


Fig. 1 Architecture Overview

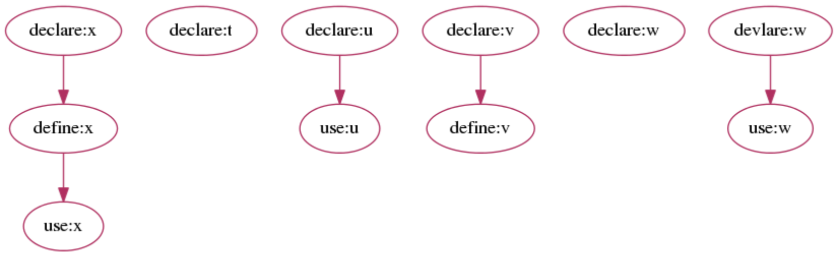


Method	Defs	Uses
A::h	{A::u, A::w}	
A::i		{A::u}
A::j	{A::v}	{A::w}
A::l		{A::v}
B::h	{B::x}	
B::i		{B::x}
C::i	{C::y}	
C::j		{C::y}
C::l		{A::v}

Fig. 2 Data flow anomalies with polymorphism. Source from Offutt, 2011

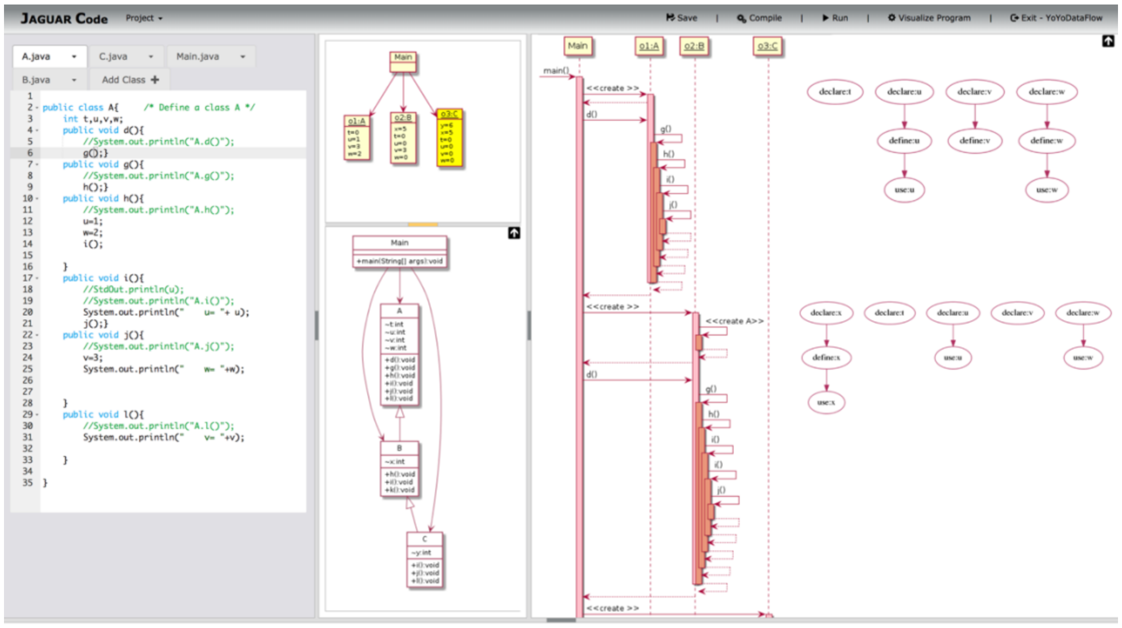


(A) Actual type of Class A



(B) Actual type of Class B

Fig. 3 Data Flow Diagram (Define-Use Path)



```
public class A{     /* Define a class A */
    int t,u,v,w;
    public void d(){
        //System.out.println("A.d()");
        g();}
    public void g(){
        //System.out.println("A.g()");
        h();}
    public void h(){
        //System.out.println("A.h()");
        u=1;
        w=2;
        i();

    }
    public void i(){
        //StdOut.println(u);
        //System.out.println("A.i()");
        System.out.println("    u= "+ u);
        j();}
    public void j(){
        //System.out.println("A.j()");
        v=3;
        System.out.println("    w= "+w);


    }
    public void l(){
        //System.out.println("A.l()");
        System.out.println("    v= "+v);

    }

}
```

Fig. 4 User Interface of Reverse Engineered Tool