Toward secure refactoring of object-oriented programs

Marwan Mahmoud
Department of Computer and Information Technology, Faculty of Applied Studies, King Abdulziz University, Jeddah, Saudi Arabia
E-mail: mmamahmoud@kau.edu.sa

Abstract. Security is a significant issue in software systems. The initially secure design of a software system could be at risk of a refactoring transformation. Consequently, via a refactoring transformation, the software could be considered vulnerable, and the refactoring process can undergo security-aware refactoring. This work deals with three security approaches to assist programmers in refactoring their software systems while simultaneously sustaining or possibly improving the security of their software system. In addition, this work deals with security-aware refactoring at the design stage rather than refactoring that requires a full source code. The main aim of this work is to provide programmers with knowledge about secure refactoring processes to improve the long-term maintainability of the software system in a secure way while sustaining the external behavior of the system.

1. Introduction
Software program refactoring is a process of improving the internal structure of the software system while sustaining the observable behavior [1]. Refactoring can be implemented via program modification, which describes a way of refactoring in arranged steps.

Software system security is a process to make software programs secure in a way that protects the confidential data to be accessed by unauthorized users [2]. The main critical issue in software program security is not allowing confidential data to be accessible by unauthorized activities or users.

The use of distributed systems is increasing globally. Many internet users have access to software systems remotely. In this case, the software systems must be free of security flaws. A security flaw can be made by programmers while editing a software code and can result in problems that may increase the ability of the attacker to access private data in the program. Most insecure software systems are a result of a lack of required security knowledge for the programmers who can occasionally make mistakes even if they are experts at security. Existing studies focus on security issues at the design or implementation level. However, this work focuses on unexpected security problems arising when refactoring the software program.

This work focuses on four approaches of a secure refactoring software system. Section 2 describes the security issues in refactoring and states rules for secure refactoring using the access control security approach [3]. Section 3 points to refactoring software programs in low and high-security modules using the information flow security approach [4]. Section 4 works on the design and implementation of a secure refactoring tool by considering specific security issues [5]. Section 5 in this work studies secure refactoring at the design stage of the software system using the access control security approach [6]. Section 6 discusses secure refactoring in either the implementation or design stages. Section 7 concludes this work.
2. Identifying refactoring security issues

Secure refactoring is similar to traditional refactoring, but secure refactoring warns programmers of the security issues [1][7]. Therefore, the importance of refactoring is sustaining external behavior. The tool developed by Maruyama and Tokoda [3] focuses on some security rules in programming and how programmers can avoid security flaws in their programs.

One of the most critical factors in security is preventing unauthorized users from accessing the program’s data. More chances to access data means that software programs are an easy target for attackers. Easily accessible codes increase the possibility of unauthorized users to access essential data. The security level of the program is measured by controlling the accessibility of the external codes. As a result, the program code could become vulnerable when refactoring allows changes in the access control of the program’s data.

This study focuses on critical security issues in traditional refactoring [1]. For example, Move Method refactoring changes private fields to non-private in modified code. The security level of the software program decreases in this case as well as in the refactoring of Push Down Method, Move Field, Extract Class, Pull Up Method, and Pull Up Field.

A security-aware refactoring method is a result of formulating a process that determines the effect of existing refactoring on the source code. This approach focuses on the ability to access program code that contains confidential data by providing an example in Java programming language and proposing a technique to detect the security issues.

Programming languages have security elements built in. Java has four levels to access construct (interfaces, methods, classes, and fields) [8][9]. When a construct is declared as protected, the construct can be accessed from the class that defines the construct, from the classes within the package of the defining class, and from subclasses of the class that define the construct. If no access level is assigned to the construct, then construct level access becomes default. In this case, a construct can be accessed from the class that defines the construct in addition to any class within the package.

A private level construct is accessible only from inside the class that defines the construct. On the other hand, a public construct is accessible from any class.

```java
// before refactoring (original)
class Store {    
    private int income;    
    private int outcome;    
    private int profit;    
    private void adjust() {        
        income = 0;        
        outcome = 0;        
        profit = 0;        
        return outcome();    
    }    
    void adjust() {        
        if (profit > 0)        
            profit = 0;        
    }    
    void record() {        
        microProfit(income, outcome);    
    }    
}    
class Shop extends Store {    
    }    

// after refactoring (modified)
class Shop {    
    private int income;    
    private int outcome;    
    private int profit;    
    private void adjust() {        
        income = 0;        
        outcome = 0;        
        profit = 0;        
        return outcome();    
    }    
    void adjust() {        
        if (profit > 0)        
            profit = 0;        
    }    
    void record() {        
        microProfit(income, outcome);    
    }    
}    
class Shop extends Store {    
    void adjust() {        
        if (profit > 0)        
            profit = 0;        
    }    
}
```

**Figure 1.** Example of Push Down Method refactoring [3].

Figure 1 shows an example of Push Down Method refactoring. In this case, the programmer should be aware of changing the access level of any field. In Figure 1, the `adjust()` method moved from a class.
store by refactoring to subclass shop. This refactoring affects the private field profit changing it to a protected field.

A secure refactoring tool in Java called Jstart supports the Push Down Method and Push Up Method refactoring as a plug-in on Eclipse. This tool depends on an existing tool called Jxplatform [10], which converts the source code into an XML document and generates program dependence and control flow graphs. Figure 2 shows the architecture of the Jstart tool. A code transformer checks each refactoring precondition to rewrite the XML document contents converted from the original source code. The approach then recovers modified code from the rewritten XML document. A security evaluator creates an access level graph for target methods before and after refactoring. When access levels of any fields are lowered, the security evaluator creates a warning message regarding the field’s security level. The warning message contains source code before and after refactoring in addition to the security message. After the message appears, the programmers can decide whether to cancel or accept the refactoring.

Figure 2. The Architecture of The Proposed Jstart Tool [3].

The Jstart tool has limitations. First, although information flows contain possible security risks, this approach does not consider the implicit flow of information because of the difficulty of the implementation. Second, the approach isolates dependences by using analysis that depends on classes and ignoring aliases of named instances. Third, Jstart creates an access level graph that considers data flow in local variables; as a result, not all field values are distinguished at different points.

3. Secure program refactoring

The tool developed by Smith and Thober [4] assumes input and output data in the source code labeled as either low-security or high-security input or output. The refactoring operation in this approach contains an “identifying sensitive data in the code” task and an “applying refactoring to create a new component that contains all high-security data” task.

An example using C and C++ in Figure 3 can explain the process of this approach. Figure 3 shows the login code that accepts a username and password as keyboard input and then authenticates the user by validating the information on the file system. The username and password stored in the file system are considered high-level security data.
3.1. Identifying sensitive data in the code

There are many ways to use slice criteria, which can divide such code into different parts. However, this approach works on a high-security input. As a result, this approach uses forward slicing and starts from a high-security input. Figure 4 shows all affected code by high-security input.

The high slice is a part of the code that contains high-level security data. The slicing criterion is a part of the code that related to `getPasswd()` method. In this case, the high-security slice has been identified.

```
1: int main() {
2:     char uname[15];
3:     char passwd[15];
4:     char* syspasswd;
5:     // ...
6:     ....
7: ....
8:     ....
9:     if (strcmp(syspasswd, passed)) {
10:        printf("access granted");
11:    } else {
12:        printf("access denied");
13:     exit(0);
14:    }
15:    // Execution proceeds */
16: }
17: 
18: char* getPasswd(char* uname, char* file) {
19:    /* Returns the password for `uname` in `file` */
20: }
```

---

Figure 3. Login code [4].

---

```
1: int main() {
2:     char uname[15];
3:     char passwd[15];
4:     char* syspasswd;
5:     ....
6:     scanf("%s",uname);
7:     scanf("%s",passed);
8:     syspasswd = getPasswd(uname,"/etc/passwd");
9:     if (strcmp(syspasswd, passed)) {
10:        printf("access granted");
11:    } else {
12:        printf("access denied");
13:     exit(0);
14:    }
15:    // Execution proceeds */
16: }
17: 
18: char* getPasswd(char* uname, char* file) {
19:    /* Returns the password for `uname` in `file` */
20: }
```

---

Figure 4. forward slicing method to identify high slice [4].

---

Figure 5. Refactored code [4].
3.2. Applying refactoring
The next step is refactoring the high slice into a new component. The high component contains a combination of high-security codes. A low component is a part of the code that contains a combination of low-security codes. Sometimes, a high component must contain codes not presented in the high slice—for example, variable declaration.

Figure 5 shows the refactored code where the login class is created, and the `getPasswd` method moved to a private section of the new class due to the high security contents. Method authenticate becomes accessible as a public method in the new class, and the variable `syspasswd` is deleted because it is useless after refactoring. Figure 5 shows high-security operations in low components having been changed to call public methods only in the high components as in line 4 and line 8 of low component code.

3.3. Approach limitation
This approach works perfectly if the program can be easily divided into two parts. However, most programs use low and high components in addition to libraries that are shared. As a result, this approach needs more techniques to deal with complex programs.

4. Designing a secure refactoring tool
The tool developed by Widhiartha and Maruyama [5] designed a secure refactoring that addresses some security issues in refactoring. This tool is used as a plug-in on Eclipse and uses two existing plug-ins for analyzing the source code: Abstract Syntax Tree (AST) and Java Development Tool (JDT). The two most important steps of this approach are designing a secure refactoring process and designing a secure refactoring tool.

4.1. Designing a secure refactoring process
The process of refactoring starts when the programmer selects part of the code to be refactored; the programmer can then select a secure refactoring method option. In this case, the tool creates a dialog in a new window for the programmer, which contains the source code before and after refactoring. The programmer can determine if he/she wants to continue with the refactoring process.

4.2. Designing the secure refactoring tool
This approach implements three security issues of refactoring [11] into a Secure Refactoring Tool to increase source code security level.

4.2.1. Changing to secure class. The first secure refactoring concern is changing the `java.util.Random` class to the secure class `java.security.SecureRandom`, in particular, when the class is used to generate a key in cryptography because random numbers are predictable and can assist the attacker in accessing confidential data. Class `java.security.SecureRandom` uses a technique to generate a strong pseudo-random number. The Java programmer then only needs to substitute `java.util.Random` class with `java.security.SecureRandom` as shown in Figure 6.

Figure 6. The method of secure random refactoring [5].

Figure 7. The method of transient refactoring [5].
4.2.2. Transient refactoring. Java serializes objects for transmission or storage. Serializing objects means converting the object into byte streams to be transmitted as needed to network programming or stored in a file. In this case, private data is vulnerable [12]. To avoid this security issue, Java has a variable declaration called Transient that can avoid storing the value in a file. The operation is shown in Figure 7.

4.2.3. Changing from string to character array. String variables in Java are undeletable from memory. In this case, programmers should not use passwords or any sensitive data as a string because it leads to vulnerability [13]. To avoid this security issue, this approach uses a password string as a character array and then adds fake values to the arrays after use as shown in Figure 8. However, this operation has some restrictions because not all functions that work on strings can be adopted to work on arrays.

![Figure 8](image_url) Figure 8. The method of changing string to array refactoring [5].

4.3. Approach limitation
This approach works on the syntax level, but it needs to consider the source code semantic levels as well.

5. Refactoring effects on the security critical design
This approach was developed by Alshammari et al. [6] and differs from the previous approaches because it works in the design phase of the development. The critical point of this approach is using metrics to assess the effect of traditional refactoring rules on the security of software program design by measuring the metrics before and after refactoring.

5.1. Identifying security refactoring issues in program design
This approach distinguishes the effect of traditional refactoring rules on the security level of programs’ design into non-classified and classified features as shown in Table 1; it then studies their effects on the accessibility of private data.

5.2. Assessment of security refactoring issues
This approach studies some refactoring rules that affect the security level (positively or negatively) of the software program design. Table 1 shows the expected impact of the refactoring on the security of the design. This approach applied security metrics on each refactoring rule, and it presents the overall impact on the security of the design.

Figure 9 shows the bank system hierarchy in the design stage. After applying Extract Super Class refactoring method to “CustomerAccount” class in the original design, the hierarchy is refactored to be as shown in Figure 10. The approach in this example applies one of the security metrics, which counts the number of security-critical super-classes to the number of the security-critical classes of the system in the design stage. In this example, the security metric of the original design that contains four critical classes and two critical super-classes is equal to 0.5. The security metric of the refactored design in Figure 10 shows that there are four critical classes, and there is one critical super-class. As a result, the security metric value is decreased to 0.25, and applying Extract Super Class refactoring method increases the security level of the software system in the design stage.
5.3. Approach result
This approach shows that refactoring can increase the overall security level of a software program design as it includes 20 refactoring rules that increase the security level of the software program design. On the other hand, 12 refactoring rules that decrease the security level and four refactoring rules that have no effect on the security of the software program design are included.
Table 1. Assessment of security refactoring issues on the software program design.

| Security Refactoring | Identifier Effect | Overall Security Impact |
|----------------------|-------------------|-------------------------|
| Encapsulate Classified Field | Changes the access modifier of classified public fields to private. | D |
| Encapsulate Non-Classified Field | Changes the access modifier of non-classified public fields to private. | N |
| Inline Classified Field | Contains two classified fields or more into one classified field if they are always navigated | D |
| Inline Non-Classified Field | Contains two non-classified fields or more into one classified field if they are always navigated | I |
| Extract Classified Field | Creates a new classified field from an existing classified field if no information can be used separately | I |
| Extract Non-Classified Field | Creates a new non-classified field from an existing classified or non-classified field if no information can be used separately | D |
| Pull Up Non-Classified Field | Moves subclasses have the same non-classified field to the upper level | I |
| Pull Up Non-Classified Field | If a subclass in the lower level has a non-classified field then this rule moves this field to that superclass. | D |
| Push Down Non-Classified Field | If a classified field is used by only some subclasses then this rule moves this field to those subclasses. | D |
| Move Classified Field | Moves a classified field to a new class. | D |
| Move Non-Classified Field | Moves a non-classified field to a new class. | D |
| Hide Classified Method | Makes a classified public method private if not used by another class. | D |
| Hide Non-Classified Method | Makes a non-classified public method private if not used by another class. | N |
| Inline Classified Method | Combines two classified methods or more into one classified method if they are always used together. | D |
| Inline Non-Classified Method | Combines two non-classified methods or more into one classified method if they are always used together. | I |
| Extract Classified Method | Creates a new classified method from an existing classified method if its information can be used separately. | I |
| Extract Non-Classified Method | Creates a new non-classified method from an existing classified or non-classified method if its information can be used separately. | D |
| Finder Classified Method | Declares a classified method as "final" to prevent it from being extended. | D |
| Finder Non-Classified Method | Declares a non-classified method as "final" to prevent it from being extended. | N |
| Pull Up Classified Method | If two subclasses have the same classified method then this rule moves this method to their superclass. | I |
| Pull Up Non-Classified Method | If two subclasses have the same non-classified method then this rule moves this method to their superclass. | D |
| Push Down Classified Method | If a subclass in the lower level has a classified method then this rule moves this method to those subclasses. | D |
| Push Down Non-Classified Method | If a subclass in the lower level has a non-classified method then this rule moves this method to those subclasses. | D |
| Move Classified Method | Moves a classified field to a new class. | D |
| Move Non-Classified Method | Moves a non-classified field to a new class. | D |
| Inline Classifier Class | Combines two critical classes or more into one critical class if they are always used together. | D |
| Inline Non-Critical Class | Combines two non-critical classes or more into one critical class if they are always used together. | I |
| Extract Critical Class | Creates a new critical class from an existing critical one. | I |
| Extract Compound Part Critical Class | Creates a new compound part critical class from an existing critical class. | D |
| Extract Non-Critical Class | Creates a non-critical class from an existing critical class. | D |
| Finder Critical Class | Declares a critical class as "final" to prevent it from being extended. | D |
| Finder Non-Critical Class | Declares a non-critical class as "final" to prevent it from being extended. | N |
| Extract Critical Method | If two critical subclasses have similar critical features, this rule creates a critical method. | D |
| Extract Non-Critical Method | If two critical subclasses have similar non-critical features, this rule creates a non- critical method. | D |
| Extract Critical Subclass | If two critical subclasses have similar critical features, this rule creates a critical subclass. | D |
| Extract Non-Critical Subclass | If two critical subclasses have similar non-critical features, this rule creates a non-critical subclass. | D |

D = Decrease security metrics  P = Increase security metrics  N = No impact on the security metrics

6. Discussion

This work focuses on three security approaches in refactoring: access control approach, information flow approach, and specific security issues approach. The access control approach is applied in both the implementation and design stages. The other approaches are applied in the implementation stage only.

The access control security approach in the implementation stage is applied as a tool that compares the source code before and after refactoring. It then warns the programmer if the control access level of any construct is lower during the refactoring. This security-aware refactoring tool is helpful to consider some security concerns during refactoring, but this tool is limited to six refactoring methods. In addition, this tool does not suggest an alternative use of refactoring that helps the developer to apply the refactoring without any security concerns.

The information flow security approach pointed out in this work is a refactoring tool that checks the information flow in the source code, and distinguishes high-security data from low-security data, and presents it as a new method using the Move Method refactoring to make the source code more secure. This tool is limited due to the process of applying specific refactoring methods to a specific kind of source code. If the source code is complex and very difficult to be divided into two security levels, then this tool becomes useless.

The specific security issues approach presented in this work as a refactoring tool include refactoring the source code to recover three particular security issues. The results suggest the refactored source code to the developer. This tool is very limited to three specific security issues of the source code.
Access control security in the design stage stated here as a security-aware refactoring shows the refactoring effects on the security of software program. The outcome can warn the programmer which refactoring method leads to lower or higher access control of the software program construct as well as which refactoring method has no impact on the security of the software program. This approach is beneficial to the programmer to consider security awareness during some refactoring methods, and it is more helpful than the tools applied in the implementation stage. However, it is limited to the design stage.

This work focuses on four different proposed approaches of secure refactoring in different stages, and it is important that programs should not depend on a specific approach for secure refactoring. They should consider secure refactoring in different stages. Also, there is a need for a comprehensive secure refactoring approach that considers different programming stages as a guideline for the programs of object-oriented programs.

7. Conclusion

Even though refactoring improves the long-term maintainability of the software system, some refactoring decreases the security level of the software program. This work focuses on four approaches that can be implemented to address security level decreasing caused by refactoring. Three approaches are presented here as supportive tools to the programmer and allow the programmer to make the final decision about the refactoring process considering the information provided. Two approaches give an example of secure refactoring on Java programming language while one approach gives an example in C and C++ programming language. This work covers some limitations of each security-aware refactoring approach. One approach in this work focuses on the refactoring security issues at the software system design stage. This work shows that a comprehensive approach for secure refactoring that covers multiple programming stages is needed.

References

[1] Martin Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed. Addison-Wesley Professional, 2018.
[2] M. Bishop, Computer Security: Art and Science, 1st ed. Addison-Wesley Professional, 2002.
[3] K. Maruyama and K. Tokoda, “Security-Aware Refactoring Alerting its Impact on Code Vulnerabilities,” in 2008 15th Asia-Pacific Software Engineering Conf., 2008, pp. 445–452.
[4] S. F. Smith and M. Thober, “Refactoring Programs to Secure Information Flows,” in Proc. of the 2006 Workshop on Programming Languages and Analysis for Security, 2006, pp. 75–84.
[5] P. Widhiartha and K. Maruyama, “Design and implementation of a secure refactoring tool,” in 2nd Annual Int.l Conf. Information and Communication Technology Seminar, 2006.
[6] B. Alshammari, C. Fidge, and D. Corney, “Assessing the Impact of Refactoring on Security-Critical Object-Oriented Designs,” in 2010 Asia Pacific Software Engineering Conf., 2010, pp. 186–195.
[7] W. Opdyke and R. Johnson, “Refactoring Object-Oriented Frameworks,” 1992.
[8] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, The Java Language Specification, 7th ed. Addison-Wesley Professional, 2013.
[9] S. Oaks, Java Security, 2nd ed. O’Reilly Media, 2001.
[10] K. Maruyama and S. Yamamoto, “Java-XML Tools Project.” [Online]. Available: http://www.jtool.org/. [Accessed: 19-Dec-2020].
[11] P. Widhiartha and K. Maruyama, “Security Concern Refactoring,” in ICTS 2005, 2005.
[12] “JDKTM 1.1.3 Documentation,” Java Development Kit. [Online]. Available: https://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/. [Accessed: 19-Dec-2020].
[13] B. Rubin, “Java security, Part 1: Crypto basics,” IBM Developer, 2002. [Online]. Available: https://www.ibm.com/developerworks/java/tutorials/j-sec1/j-sec1.html. [Accessed: 19-Dec-2020].