2008 15th Asia-Pacific Software Engineering Conference

Security-Aware Refactoring Alerting its Impact on Code Vulnerabilities

Katsuhisa Maruyama
Department of Computer Science
Ritsumeikan University, Kusatsu, Japan
maru@cs.ritsumei.ac.jp

Kensuke Tokoda
Graduate School of Science and Engineering
Ritsumeikan University, Kusatsu, Japan
toko@fse.cs.ritsumei.ac.jp

Abstract

Security is still a serious issue for many software systems. Even if software has the correct security features in its initial implementation, recurring modifications (e.g., refactoring) could deteriorate such features. We found several refactoring transformations which might make existing software vulnerable, and organized them as security-aware refactoring. This refactoring presents information useful for programmers to determine if they could accept or should cancel it, based on a criterion assessing the changes of accessibility of data stored in the target program. To demonstrate the feasibility of the proposed refactoring, we have developed a prototype of an automated refactoring tool detecting possible code vulnerabilities regarding the accessibility criterion. The new refactoring provides programmers with an environment in which they safely improve the maintainability of existing software without missing the intrusion of unexpected security vulnerabilities.

1. Introduction

Software provides an increasing number of e-commerce services in the real world. At the same time, software systems are no longer monolithic creations [6]. A service is comprised of software components that collaborate on distributed platforms such as the web. Various kinds of mobile code snippets (e.g., JavaScript) are being downloaded over the networks and they are running on personal computers. Additionally, in the construction of web services which accept the execution of external code, public APIs should assume that several connections from clients are untrusted; otherwise the discontinuation of the services could incur.

To prevent attacks in this situation, programs must not contain security flaws. A security flaw is a mistake made while a programmer (a developer or a maintainer) writes or modifies a software program, and could become a vulnerability. A vulnerability is a problem that can be exploited by an attacker [13] or a weakness that makes it possible for a threat to occur [4]. Unfortunately, there are an enormous number of vulnerable software programs with unnecessary privileges or weak access control settings on assets. In particular, a recent Java platform supports several script languages including JavaScript. In such a platform, Java and JavaScript programs can invoke themselves and each other. If a Java program which is a framework (or a skeleton) importing downloaded JavaScript code contains awkward code with improper accessibility settings (e.g., public mode) of confidential data (e.g., HTTP cookies or passwords), malicious JavaScript code can steal them easily.

This risk results from the fact that not all programmers have knowledge sufficient to create a secure system. Although some are experts on security, they are not perfect and sometimes make mistakes. We emphasize the need for security-concern support for implementation (construction) of software. Even if software has the correct security features in its requirements specification, the actual program could contain serious security vulnerabilities that could pose a potential security risk. Software consisting of programs with vulnerabilities cannot be considered secure.

From the point of view of software security in the implementation phase, it is worth discussing the impact of program modifications (source code changes) on security characteristics of software. To our knowledge, almost all existing studies concern themselves with how a programmer creates a secure program from scratch or fixes a security flaw. For example, several guidelines for secure programming [4, 12, 17, 18, 28] and catalogs of implementation-level attack patterns [13, 29] have been documented. On the other hand, little attention has been given to accidental security vulnerabilities resulting from recurring program modifications in a maintenance task. Of course, the existing security guidelines and catalogs are useful for modifying programs. However, these are insufficient for retaining security characteristics of existing software or preventing the intrusion of security vulnerabilities in its modifications. We believe that many programmers require automated support specific to various kinds of program modifications and a systematic way to eliminate a possible risk on vulnerabil-

1530-1362/08 $25.00 © 2008 IEEE
DOI 10.1109/APSEC.2008.57

ity intrusion although they should be responsible for carefully applying refactorings.

This paper proposes *security-aware refactoring* which provides information so that programmers can easily know the impact on its application of the security vulnerabilities of the modified code. Refactoring is the process of improving the maintainability of software without changing its external behavior [8, 22]. The reason why we address the problem on security vulnerabilities in refactoring is that refactoring is a particular kind of program modification with sophisticated mechanics. The mechanics are concise, step-by-step description of how to carry out a refactoring.

In the paper, the impact is measured based on a criterion assessing the changes of accessibility of data stored in the target program. The criterion results from our preliminary investigation which shows that several traditional refactorings could deteriorate the security characteristics of existing programs. The security-aware refactoring helps the programmers to obviate the occurrence of inadvertent code flaws [15] in its behavior-preserving transformations. Moreover, the paper shows a prototype of an automated refactoring tool implementing the proposed refactoring, which is built as an Eclipse plug-in. The tool provides programmers with an environment in which they safely improve the maintainability of existing software without missing the intrusion of unexpected flaws that could be accidentally or intentionally exploited to damage assets.

The remainder of the paper is organized as follows. Section 2 defines the security-aware refactoring and describes security concerns in traditional refactorings. Section 3 proposes a criterion assessing security characteristics with respect to the access level of data and explains a mechanism detecting the impact of code changes in refactoring on security vulnerabilities. Section 4 shows the implementation of a security-aware refactoring tool. Section 5 describes existing studies related to software security in the implementation phase of software development. Finally, Section 6 concludes with a brief summary and future work.

2. Security-Aware Refactoring

The process of the security-aware refactoring which programmers do is the same as that of the traditional refactoring [8, 22], but the traditional refactoring never provides security warnings in its process. We newly define security-aware refactoring as follows:

> A change made to the internal structure of software to make it easier to understand and cheaper to modify without both changing its observable behavior and missing unexpected changes of its security level.

Figure 1 shows the concept of the security-aware refactoring. Every transformation of this kind of refactoring

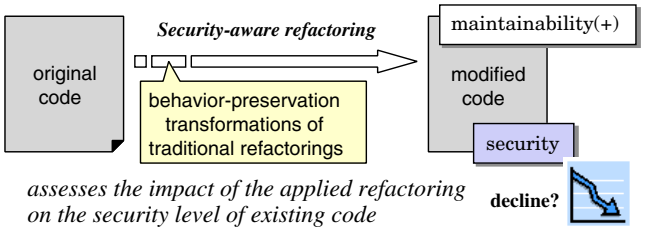

Figure 1. Security-aware refactoring.

ensures no change of observable behavior of the modified code. It also provides programmers with information on how the security level of the modified code is decreased as a result of its application. Accordingly, they obtain a chance to cancel or undo undesirable modification of the applied refactoring if they cannot accept its impact on the security characteristics.

Here, we should reveal more about security in this paper. According to Gollmann's book [10], security is about the protection of assets and protection measures are distinguished between prevention, detection, and reaction. This paper focuses on the prevention, and assumes that secure code neither discloses sensitive data (information) to unauthorized users (i.e., confidentiality) nor allows them to change the data (i.e., integrity). Other aspects (e.g., availability) are considered out of scope although they are also related to software security.

In this paper, a vulnerability is considered a possible problem that might violate the confidentiality or the integrity of particular data. Note that this vulnerability could sometimes remain in programs if a programmer recognizes its existence. This is because our goal is not to absolutely detect rejectable vulnerabilities but to advise programmers to check the change of the ease of access to the data. For this reason, the security level of software is relatively determined based on the potential existence of internal code which allows external code to access the data. The more generous the access to data is, the more opportunities for success of unauthorized accesses are. An attacker easily steals or tampers with the data by exploiting code with weak access control. Conversely, it is more difficult for an unauthorized user to access the data if its access is severely limited. If a refactoring (code change) relaxes the access to data in a program, the security level of the modified program is decreased. This refactoring could make the program more vulnerable.

To verify that several refactorings could actually deteriorate the security characteristics of existing software in their applications, we made a preliminary investigation with traditional refactorings presented by Fowler's catalog [8]. In the catalog, we found risky refactorings with respect to security characteristics. The Move Method refactoring makes a private field non-private if the moved method will have to

use it within the modified code. It gives an attacker a chance to access the field since its access level will be weaker than before. The Move Field, Pull Up Field, Pull Up Method, Push Down Method, and Extract Class refactorings have similar effects on the accessibility of the moved element (field or method) or elements related to it.

3. Vulnerability Detection Mechanism

To implement the security-aware refactoring, it is needed to formulate a mechanism which assesses the impact of each existing refactoring with respect to software security. However, the security aspects are intricately intertwined with each other, and it is hard to complete a deep investigation into various kinds of impact. Therefore, we focus on the accessibility to program elements which can affect the confidentiality and the integrity of data existing in the program (e.g., the value of a field). This seems to be worthwhile as the first attempt although our focus covers a small part of security characteristics.

In the paper, we present a simple criterion which assesses the security impact on the accessibility. This section explains the criterion in detail, and proposes a mechanism detecting the intrusion of code vulnerabilities.

3.1. Java Security

The proposed refactoring treats source code written in Java programming language, so we briefly describe here Java's security model which revolves around the idea of a sandbox. The Java sandbox is responsible for protecting a number of resources (e.g., a file and memory) according to a user-defined security policy [20]. This mechanism is less vulnerable to external attacks if executing code does not contain security flaws. However, awkward code including improper accessibility settings or needlessly flexible structure might violate the protection given by the sandbox with no check. Therefore, it is important to write or retain secure code in addition to the use of the secure platform.

As a security component built in programming language itself, Java provides four possible access levels for constructors such as fields, methods, classes, and interfaces [11, 20]. If a construct is declared `private`, it is accessible from only inside the class defining the construct. If no access modifier is attached to a construct, it is deemed to have a default (package) level. In this access level, the construct is accessible from any class in the same package as the class defining the construct. If a construct is declared `protected`, it is accessible from the class defining the construct, classes within the same package as the defining class, or subclasses of the defining class. A `public` construct can be accessed from any class. For top-level (non-nested) classes, only public and default can be specified.

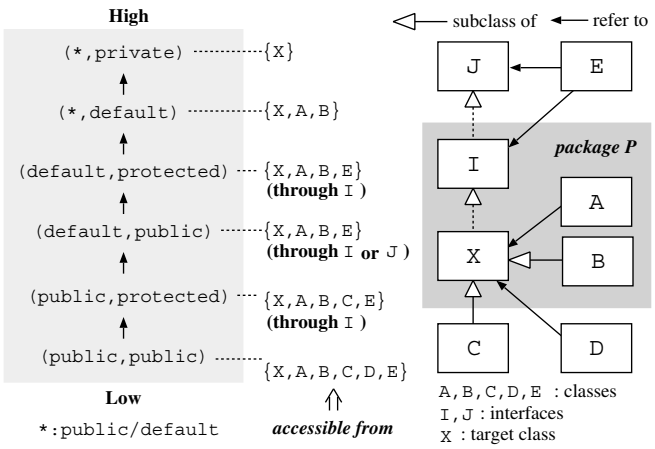

Figure 2. Access level criterion.

3.2. Access Level

In general, (sensitive) data are stored in variables (fields, local variables, or formal parameters of methods) or are observed as arguments required by method calls (actual parameters or return values of methods). These entities except for a field have local scope and cannot be accessed from a program element existing outside its local-scope. A program element can access the value of a particular field based on its access modifier. It can also read or modified the value of a field through a method accessing it.

Accordingly, we assume that the confidentiality and integrity of sensitive data within the code is measured by means of the access levels of a field storing the data and its access methods. In this paper, confidentiality means that the value of a field can be read directly or through a method reading it (e.g., a getting method). On the other hand, integrity means that the value of a field can be written directly or through a method writing it (e.g., a setting method). Our mechanism does not distinguish confidentiality and integrity although these are distinct security aspects. In the mechanism, a field preserving sensitive data is defined in a top-level class. Interfaces are not treated since their members are implicitly declared public. Nested classes are not also treated since secure code does not need to use them[1].

Figure 2 depicts a criterion that measures the impact of the possible access levels for fields and their access methods in the code. It is represented by a lattice of pairs of access modifiers. In the pair `(c,m)`, `c` denotes the access modifier of a class (`X` in Figure 2) containing a member (a field or a method) of interest, and `m` denotes the access modifier of the member. The asterisk (`*`) denotes a wild-card that can be replaced with either public or default. The order of the pairs was basically determined based on the range of accessible

[1] A nested class should be translated into a class accessible to any class in the package defining the nested class [28].

program elements. For example, a target member whose access modifier is private in the class X (see the top pair in Figure 2) can be accessed from only program elements in X. On the other hand, a target member whose access modifier is public in the public class X (see the bottom pair in Figure 2) can be accessed from all the classes X, A, B, C, D, and E. Class E can possibly access X through interface I or J. If an applied refactoring makes sensitive data more accessible, that is, the accessibility is changed from a higher pair to a lower one, the security level of the modified code will be decreased since the refactoring makes the data easier to read or write. Conversely, if the accessibility is changed from a lower pair to a higher one, the security level of the modified code will be increased.

The effect of the application of an explanatory refactoring is explained here. Figure 3 shows code snippets before and after the application of the Push Down Method. This refactoring moved a method `adjust()` from a class `Store` to its subclass `Shop`. As a result, a private field `profit` in the class `Store` became protected since the moved method `adjust()` will use it within the modified code. In this case, the access levels of `profit` is decreased as follows:

`(default,private)` → `(default,protected)`.

The value of `profit` will be observed or altered by using a malicious subclass of `Shop` if such subclass can be embedded into the target program as a component.

3.3. Access Level Graphs

To truly protect (sensitive) data, we should consider the accessibility of a variable which stores such data and all variables which might store some sort of information on the data (e.g., a copy of the data) [23]. In Figure 3, programmers should be aware of decreasing of the access level of the field `income`, as well as the field `profit` which is directly rewritten. They would often miss this decreasing since the access modifier of `income` was not explicitly modified. An attacker could figure out the value of `income` by knowing the value of `profit` (and the value of `outgo`). This is a comparatively trivial but possible example of the disclosure of sensitive data.

We assume here that an attacker has knowledge of source code of the target program which may contain sensitive data. Based on this assumption, the detection mechanism introduces the concept of information flow [4, 5, 23]. The goal of information flow control is to enforce noninterference [9], which intuitively means that confidential data may not interfere with observable data. For example, the modified code shown in Figure 3 includes possible information flows `income` → `profit` and `outgo` → `profit` through the parameters `i` and `o` of the method `calcProfit()`. The mechanism would extract explicit information flows from data dependencies of program dependence graphs

```
// before refactoring (original)
class Store {
  private int income;
  private int outgo;
  private int profit;
  Store(int i, int o) {
    income = i;
    outgo = o;
  }
  int getOutgo() {
    return outgo;
  }
  void adjust() {
    if (profit < 0)
      profit = 0;
  }
  private void calcProfit(int i, int o) {
    profit = i - o;
  }
  void record() {
    calcProfit(income, outgo);
  }
}
class Shop extends Store {
}
```

```
// after refactoring (modified)
class Store {
  private int income;
  private int outgo;
  protected int profit;
  Store(int i, int o) {
    income = i;
    outgo = o;
  }
  int getOutgo() {
    return outgo;
  }
  private void calcProfit(int i, int o) {
    profit = i - o;
  }
  void record() {
    calcProfit(income, outgo);
  }
}
class Shop extends Store {
  void adjust() {
    if (profit < 0)
      profit = 0;
  }
}
```

Figure 3. Push Down Method **refactoring.**

(PDGs) [7], and manages the propagation of access levels by using an access level graph (ALG) proposed in the paper. An ALG consists of a set of nodes corresponding to variables (to be precise, variable declarations without the duplication of the same variable) in a program and directed edges representing flow relations between two nodes. A variable is either a field, parameter, or return value. A field holds its explicit access level. The access levels of a parameter or a return value is the same as the access level of the method involving it since the value can be obtained or observed by using method invocation or overriding. Each node inherits the access level of its corresponding variable.

Figure 4 shows an ALG of the original code shown in the top of Figure 3. The gray rectangle denotes the region of each method. The label of each node indicates the name of a variable (e.g., `profit`) or the return value (e.g., `$getOutgo`) of a method. In the mechanism, flow relations

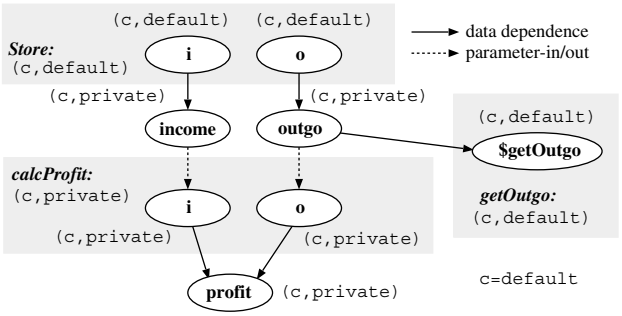

Figure 4. Access level graph (ALG).

are represented by data dependence edges or parameter-in/out edges which exist in a program. These are derived from assignments or method invocations.

Assignment

The value stored in a variable in the right-hand side is permitted to flow into a variable in the left-hand side. The mechanism assumes that an attacker has a chance to derive all the right-hand values from the left-hand value based on information flow models for assignments [4] [2].

See the assignment statement "`profit = i - o`" of the method `calcProfit()` in the top code of Figure 3. For this assignment, two data dependencies (explicit flows of information) `i` → `profit` and `o` → `profit` are extracted, and then included in the ALG, shown in Figure 4.

Method invocation

A callsite passes zero or more arguments to the called method, and receives zero or one return value. An ALG employs parameter-in/out edges of a system dependence graph (SDG) [14, 27], which represent data flows (parameter passing) between formal and actual parameters.

See the method invocation "`calcProfit(income, outgo)`" of the method `record()` in the top code of Figure 3. For the parameter `i` in this invocation, the three data dependencies `income` → `income_out`, `income_out` → `i_in`, and `i_in` → `i` lead to a parameter-in edge `income` → `i` in the ALG shown in Figure 4. A parameter-in edge `outgo` → `o` is similarly included in the ALG.

3.4. Access Level Transition

To explain a transition of access levels of data, we consider here a simple ordering relation between two access levels: `public` (low) and `private` (high). An attacker can obtain the value with the low access level while he/she should never obtain the value with the high access level.

[2] In most cases, a left-hand value results from the calculation of multiple right-hand values. Thus, an attacker does not always obtain the exact value of each of the right-hand variables although he/she knows the value of the left-hand variable.

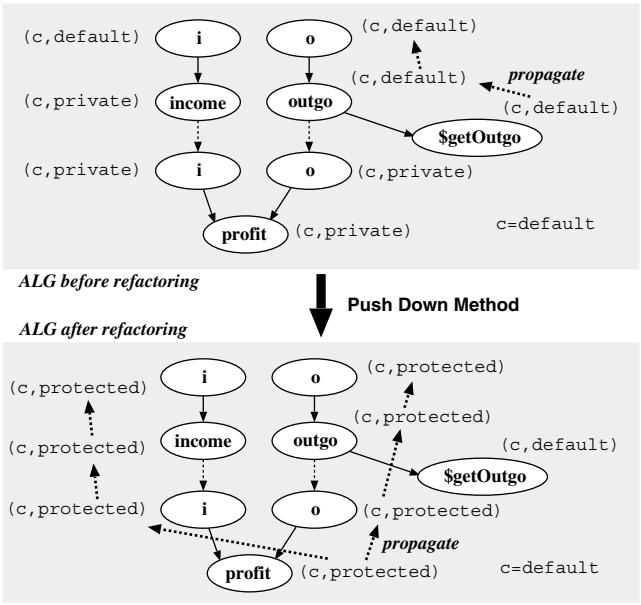

Figure 5. ALGs with access level transitions.

Unfortunately, the value with the high access level is not always protected in this situation. For example, consider that there is a data dependence from a variable x with the high access level to a variables y with the low access level, due to an assignment statement (e.g., $y = x$). In this case, he/she can (partially) guess the value of x although x is strongly protected. This is because the value of x reaches y and y is easily observed.

To detect a vulnerability posing this risk, we define a backward transition of edges in an ALG. For an edge $x \rightarrow y$, the backward transition indicates that the access level of y potentially destroys that of x. In general, the lowest access level is backward propagated through edges in an ALG. In other words, the access level of a node n ($\in N$) is hauled down by the lowest one of nodes which n reaches. The potential access level of n is summarized as follows:

$$PAL(n) = lowest(\{\ m \in N \mid n \rightarrow^* m\ \})$$

where N is a set of all nodes in the ALG and $\rightarrow^*$ indicates a reflexive transitive closure of an ALG edge. The function $lowest(S)$ returns the lowest access level for every node in a node set S. The access levels are compared based on the criteria shown in Figure 2.

Figure 5 depicts two ALGs whose nodes have potential access levels for the original and modified code shown in Figure 3. In the top ALG, the access level `(default,default)` of the return value `$getOutgo` was propagated to the nodes with the field `outgo` and the parameter `o` of the constructor `Store()`. Similarly, in the bottom ALG, the access level `(default, protected)` of the field `profit` was propagate to all nodes except for `$getOutgo`.

3.5. Vulnerability Detection using ALGs

The detection mechanism examines the change of the access levels of each field by using two ALGs generated from source code before and after an applied refactoring. A refactoring tool in general captures the code changes resulting from the applications of their provided refactorings. Therefore, it easily selects corresponding nodes for the same program element among the two ALGs. For example, in Figure 3, the tool grasps the fact that the applied refactoring moved the method `adjust()` from the class `Store` to its subclass `Shop` and changed the access modifier of the field `profit`. In other words, it knows that respective nodes `profit` of the two ALGs are equivalent, and informs that the nodes except for the node `profit` and the edges are all preserved in the two ALGs.

The mechanism detects the decreasing of the access levels of fields in a modified program as follows:

1. It receives pairs of corresponding nodes which the refactoring tool provides.
2. For each pair (n_b, n_a) for a field v, it repeats:
   (a) It compares the potential access levels $PAL(n_b)$ and $PAL(n_a)$.
   (b) If $PAL(n_a)$ is lower than $PAL(n_b)$, it creates a security warning denoting that the applied refactoring makes data of v more accessible from an attacker than before.

In Figure 5, the decline of the access levels of all the three fields, `profit`, `income`, and `outgo`, are detected.

```
profit: (c,private) → (c,protected),
income: (c,private) → (c,protected),
outgo: (c,default) → (c,protected),
```

where `c` is equal to `default`. Almost all programmers are aware of the decline with respect to `profit` since its access modifier was explicitly rewritten in the application of the Push Down Method refactoring. However, they tend to miss the decline with respect to `income` or `outgo`. The detection mechanism can bring this possible risk to their attentions.

4. Java Security-Aware Refactoring Tool

For the proposed security-aware refactoring to be practically applied to realistic software development, tool support is considered crucial. Moreover, a running implementation is essential for demonstrating its feasibility. For this, we have developed Jsart (Java security-aware refactoring tool) which supports two refactorings (Push Up Method and Push Down Method).

Figure 6 shows an overall architecture of the security-aware refactoring tool. The tool is built as an Eclipse plug-in and additionally utilizes Jxplatform [1]. Jxplatform is

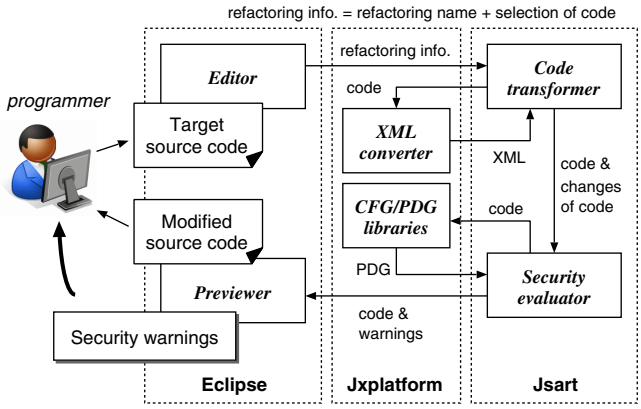

Figure 6. Architecture of the tool.

a tool platform which provides APIs for converting source code into an XML document (in XSDML format [16]) and retrieving an XSDML document corresponding to source code of interest. It contains CFG/PDG libraries which generate a control flow graph [3] and a program dependence graph [7] for each method from an XSDML document.

In Figure 6, each arrow denotes flow of data elements (source code of a program, an XML document, a refactoring name, the selection of code, and changes of code). A code transformer controls several modules to check preconditions for each of the refactorings, to rewrite the contents of an XSDML document converted from source code, and to recover the modified code from the rewritten XSDML document. A security evaluator constructs ALGs for respective methods by using the CFG/PDG libraries and the whole ALG by combining them. The evaluator compares the access levels of corresponding nodes in ALGs which generated from two version of source code before and after an applied refactoring, as mentioned in Section 3.5. If the access level of a particular field will be lower than before the applied refactoring, the evaluator creates a message of security warnings with respect to the field. A previewer displays the original and modified source code, and informs programmers on the resulting security warnings.

Figure 7 depicts a snapshot of a preview dialog after a programmer applied Push Down Method refactoring to the method `adjust()` in the original code shown in Figure 3. The original code (left) and its modified code (right) are displayed on the top of the dialog. The moved method is highlighted in `Store` in the original code and its subclass `Shop` in the modified code. The messages for security warnings are displayed on the bottom of the dialog. In this example, the dialog informs the decreasing of the access levels of `income`, `outgo`, and `profit`. Each of the messages is tree-structured. By unfolding a message tree, he/she checks the details of the decreasing of access levels.

After reviewing the provided information on security warnings, a programmer can determine if he/she accepts or

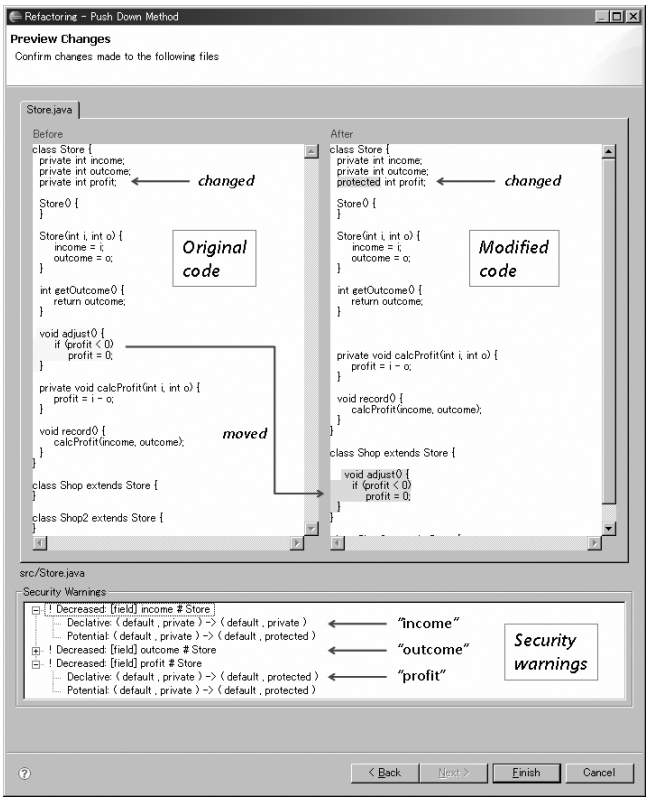

Figure 7. Screenshot of a preview dialog.

cancels the applied refactoring. He/she pushes the "Cancel" button to cancel the refactoring or the "Back" button to re-input different options. If all the warnings could be accepted, he/she would push the "Ok" button. In this case, the code on the actual Eclipse editor will be replaced with the modified code.

5. Related Work

There are several approaches available for assessing the security characteristics of software. A popular approach is based on coding checks for software programs. Moreover, semi-structured documents help programmers to check their programs by hand. This section describes these approaches.

5.1. Coding Checker

McGraw and Felten proposed 12 rules for writing security-critical Java code [17]. Jslint [26] is an automated tool that can statically analyzing a program and detect its security vulnerabilities by using these 12 rules. One of the rules states that secure code should limit access to classes, methods, and variables. For example, a public method will be vulnerable if it is called by only the class defining it. This rule might be related to the criterion proposed in Section 3.2.

Of course, this tool checks whether the modified code after the application of a refactoring contains such security vulnerabilities. Moreover, we could make a difference between the results of two checks for code before and after the application of a refactoring. This difference might indicate the impact of refactoring on the accessibility of data. However, Jslint does not keep track of code modifications whereas Jsart always grasps them in refactorings. Therefore, programmers using Jslint must identify the modified program elements by themselves. Moreover, Jslint detects security vulnerabilities which do not satisfy the predefined security policy but has no concern with the change of the security levels of two versions of a program. For example, consider that a programmer makes a private field protected in the Push Down Method refactoring. Jslint gives no warning if such field is accessed from a subclass of the class defining the field since this change is essential. A warning about the change of accessibility is significant for the programmer to construct a secure program. Jsart always produces such a warning.

5.2. Secure Programming Guidelines

A set of design and implementation guidelines which emphasize the programmer's viewpoint for writing secure programs is in general called secure programming. For example, Bishop presents 18 implementation rules that programmers should note to eliminate common security programming problems [4]. Moreover, several security patterns [24] and attack patterns [13, 29] can be considered well-documented secure programming guidelines. These guidelines have been pragmatically collected based on experiences of actual software development. Each of them shows explanatory security flaws or vulnerabilities, and it exploits that might be caused by the vulnerabilities. It also presents mitigation that can prevent or limit the exploits. Unfortunately, there is still no systematic mechanism for measuring security characteristics based on secure programming guidelines except for Jslint.

6. Conclusion

Although refactorings improve the maintainability of an existing program, some of them deteriorate its security characteristics. This paper proposed security-aware refactoring which gives information on the decreasing of the security level of the modified program. We have developed a tool detecting the decreasing of access levels of fields in two kinds of security-aware refactorings. The new refactorings provide programmers with an environment in which they safely improve the maintainability of existing software without missing the intrusion of unexpected vulnerabilities.

The current version of Jsart has several limitations. First, it treats no implicit flow of information due to its trou-

blesome implementation although such flow poses possible risks. Secondly, it extracts data dependencies by using class-based analysis and leaves an alias of instances. Thus, it neither distinguishes multiple instances generated from the same class nor identifies different names referring to the same instance. Third, it constructs an ALG based on flow-insensitive analysis except data-flows through local variables. Thus, the values of a field (or a parameter) at different execution points are not distinguished.

The followings are issues to be tackled in future work.

- Only two security-aware refactorings for assessing its impact on the access levels of data are currently implemented. We found other risky refactorings with respect to a polymorphism mechanism (with inheritance and method overriding). For example, the Extract Subclass, Extract Superclass, Form Template Method, and Replace Conditional with Polymorphism refactorings allow an attacker to have a chance to execute code of a hostile subclass. New criteria such as subclassing will be introduced for demonstrating the effectiveness of the tool.
- The above limitations of Jsart would give programmers excessive security warnings. For example, we know that Jsart cannot detect the exact impact on security vulnerabilities of a program if it contains dead-code, object references using aliases, or polymorphic calls. To make the warnings more exact, control-flow analysis for exceptions [25] and/or flow-sensitive alias analysis [21, 30] should introduced. The concept of conditional probability or entropy-based analysis [4] might mature the propagation of access levels of data in ALGs.
- Our mechanism assumes that Java's accessibility settings enforce confidentiality and integrity of data. However, this assumption seems to be awkward. We are planning to apply the mechanism to security-typed languages such as Jflow [19] and its successor Jif [2].

References

[1] Java-XML Tools Project. http://www.jtool.org/.

[2] Jif: Java information flow. http://www.cs.cornell.edu/jif/.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[5] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[6] P. T. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *ICSE '00: Proc. The Future of Software Engineering*, pages 227–239, 2000.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. Symposium Security and Privacy*, pages 11–20, April 1982.

[10] D. Gollmann. *Computer Security, 2nd ed.* John Wiley & Sons, 2006.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[12] M. G. Graff and K. R. van Wyk. *Secure Coding: Principles and Practices*. O'Reilly, 2003.

[13] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.

[14] S. Horwitz, T. Ball, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[15] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws, with examples. *ACM Computing Surveys*, 26(3):211–254, 1994.

[16] K. Maruyama and S. Yamamoto. A case tool platform using an XML representation of Java source code. In *Proc. SCAM'04*, pages 158–167, 2004.

[17] G. McGraw and E. Felten. Twelve rules for developing more secure Java code, 1998. http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html.

[18] S. Microsystems. Security code guidelines, 2000. http://java.sun.com/security/seccodeguide.html.

[19] A. C. Myers. Jflow: Practical mostly-static information flow. In *Proc. POPL'99*, pages 228–241, January 1999.

[20] S. Oaks. *Java Security, 2nd ed.* Addison-Wesley, 2001.

[21] F. Ohata and K. Inoue. JAAT: Java alias analysis tool for program maintenance activities. In *Proc. ISORC 2006*, pages 232–244, 2006.

[22] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign, 1992.

[23] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[24] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security And Systems Engineering*. John Wiley & Sons, 2006.

[25] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Proc. ICSM'98*, pages 358–367, 1998.

[26] J. Viega, G. McGraw, T. Mutdosch, and E. W. Felten. Statically scanning Java code: Finding security vulnerabilities. *IEEE Software*, 17(5):68–74, 2000.

[27] N. Walkinshaw, M. Roper, and M. Wood. The Java system dependence graph. In *Proc. SCAM'03*, pages 55–64, 2003.

[28] D. A. Wheeler. Secure programming for linux and unix howto. http://www.dwheeler.com/secure-programs/.

[29] J. A. Whittaker and H. H. Thompson. *How to Break Software Security*. Addison Wesley, 2001.

[30] J. Woo, J. Woo, I. Attali, D. Caromel, J.-L. Gaudiot, and A. L. Wendelborn. Alias analysis for Java with reference-set representation. In *Proc. ICPADS*, pages 459–466, 2001.