SECURE REFACTORING
Improving the Security Level of Existing Code

Katsuhisa Maruyama
Department of Computer Science, Ritsumeikan University
1-1-1 Noji-higashi Shiga 525-8577, Japan

Keywords: Refactoring, software restructuring, software security, program analysis, software design.

Abstract: Software security is ever-increasingly becoming a serious issue; nevertheless, a large number of software programs are still defenseless against malicious attacks. This paper proposes a new class of refactoring, which is called secure refactoring. This refactoring is not intended to improve the maintainability of existing code. Instead, it helps programmers to increase the protection level of sensitive information stored in the code without changing its observable behavior. In this paper, four secure refactorings of Java source code and their respective mechanics based on static analysis are presented. All transformations of the proposed refactorings can be designed to be automated on our refactoring browser which supports the application of traditional refactorings.

1 INTRODUCTION

Software security is ever-increasingly becoming a serious issue since software comes to play an essential role in the real world and our lives much depend on software (Devanbu and Stubblebine, 2000; Viega and McGraw, 2001). A lot of code is running on a computer that can access open communications networks like the Internet, and it is exchanging data (or objects) or providing application services over the networks. Additionally, various kinds of mobile code snippets (e.g., a Java applet or JavaScript) are being downloaded over the networks, and they are running on personal computers that exist all over. In this context, most software programs are no longer isolated from malicious adversaries. In fact, software programs including security vulnerabilities have been repeatedly incurring the leaks of confidential information or the discontinuation of e-commerce services. Nevertheless, there are an enormous number of vulnerable software programs with unnecessary privileges, known flaws, or weak access control settings on resources. They have been designed and implemented without regard to how an attacker breaks themselves (Viega and McGraw, 2001; Howard and LeBlanc, 2002; Hoglund and McGraw, 2004; McGraw, 2006).

To reduce the risk posed by attacks which try to exploit security vulnerabilities, insecure software programs including vulnerable code should be eliminated as much as possible. For this, two approaches can be considered in general. One is that secure software not including vulnerable code will be created from scratch. No one might be able to create insecure software programs if formal methods that support the whole of software development are reasonably available. However, it is hard to in advance design and implement secure software, and then it takes long time to replace all existing insecure programs with newly created ones. The other approach is that existing insecure software will be changed into secure one by partially modifying its design or code. This might not be able to perfectly remove vulnerabilities from existing software. Instead, it can decrease the likelihood of threats (e.g., information disclosure or tampering with data) as early as possible. This work employs the latter approach since the author emphasizes more feasible and practical solution to a large number of vulnerable software programs existing in the real world.

From the point of view of security improvement by changing the design of existing code, it is worth both discussing the impact of refactoring on security and exploring refactorings which make software more

Maruyama K. (2007).
SECURE REFACTORING - Improving the Security Level of Existing Code.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 222-229
DOI: 10.5220/0001339102220229

secure. Refactoring is the process of altering the internal structure of existing code without changing its external (observable) behavior (Opdyke, 1992; Fowler, 1999; Mens and Tourwé, 2004). If refactoring attains design changes that lead to security improvement, programmers (developers or maintainers) can obtain a systematic way to remove several vulnerabilities from an existing software program and to provide secure one with the same behavior as before.

This paper proposes a new class of security-concern refactoring, which is specially called *secure refactoring*. Its goal is to improve the security level of existing code irrespective of its maintainability. It detects security flaws (inadvertent code flaws (Landwehr et al., 1994)) existing in software programs and removes vulnerable code or fragile design that could be accidentally or intentionally exploited to damage assets. This process is performed by repeatedly applying a series of small refactoring transformations each of which preserves the behavior of existing code. In this paper, four secure refactorings of Java source code and their mechanics are presented. The proposed refactorings might be able to all increase the protection level (confidentiality and integrity) of sensitive information. The main contribution of this paper is to collect several transformations of refactorings which attain security improvement and which will be codified into automated tools.

The remainder of the paper is organized as follows. Section 2 describes the general concept of software security and security concerns in traditional refactorings. Section 3 defines secure refactoring and presents four transformations of the refactoring. Section 4 describes conventional studies related to secure programs. Finally, Section 5 concludes with a brief summary and future work.

2 SECURITY CONCERNS

Software security has been a hot topic for some time, and thus various kinds of studies about it have been carried out (McGraw, 2006). However, few studies have tried to discuss the impact of modification (or evolution) of software on its security quality. This section first describes security aspects the paper deals with and Java's security mechanism. Then it presents several traditional refactorings which affect the security level of existing software.

2.1 Software Security

According to the book (Gollmann, 2006), security is about the protection of assets and protection measures are distinguished between prevention, detection, and reaction. This paper focuses attention 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, the security level of software is measured based on the potential existence of vulnerable code which might violate confidentiality or integrity of sensitive data. Software including vulnerable code is called insecure one. Code fragments with serious security vulnerabilities provide more opportunities for success of unauthorized access to sensitive data. Conversely, it is more difficult for unauthorized users to break code with less vulnerability even if vulnerable fragments slightly remain in the code. If a code change removes vulnerable code fragments from a software program and never inserts new vulnerable ones into the program, such change makes the program more secure, that is, the security level of the software is improved (increased).

Every refactoring that will be proposed in this paper deals with source code written in Java programming language. Java's security model revolves around the idea of a sandbox, which is responsible for protecting a number of resources (e.g., a file and memory) according to a user-defined security policy (Oaks, 2001). This mechanism is less vulnerable to external attacks if executing code does not contain security vulnerabilities. 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 obtain 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 constructs such as fields, methods, classes, and interfaces (Gosling et al., 1996; Oaks, 2001). 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. Moreover, Java provides the keyword `final` so that any program

code cannot accidentally or intentionally change the properties of a construct. A final variable can be set only once. A final field can be also set only once by a constructor of the class defining it. A final method can be neither overridden nor hidden. No subclass can be derived from a final class.

2.2 Security Concerns in Refactorings

Refactoring is useful for improving the quality characteristics of existing software. According to the definition widely used in the software engineering community, its main goal is to improve the maintainability of software. Unfortunately, the traditional definition is not so much concerned with security characteristics of software (Smith and Thober, 2006). Thus, programmers can find few refactorings that make existing code more secure without changing its observable behavior.

It is worth investigating the effects on the security level of software programs, resulting from the application of traditional refactorings. Some refactorings in Fowler's catalog (Fowler, 1999) can increase the security level of existing code. For example, the Encapsulate Field refactoring converts the accessibility setting of a field from public into private, and adds accessors (setting and getting methods) for the field. If a field is declared public, any client (attacker in most cases) easily obtains and modifies the value of the field. This refactoring makes it harder for any client to access the value of the encapsulated field; therefore the security level of the refactored code will be increased. If such field is set at creation time and will be never altered, the Remove Setting Method refactoring can be further applied, which removes a setting method for the field and declares the field final. A private final field not having its setting method prevents a client from changing the value of the field.

The Encapsulate Collection refactoring replaces a setting method for a collection (e.g., an array, list, or set) with adding/removing methods for the collection, and rewrites a getting method of the collection so that it returns either its unmodifiable view or its copy. The code after this transformation does not allow any client to freely change the contents of the encapsulated collection. In addition, the Encapsulate Classes with Factory refactoring in Kerievsky's catalog (Kerievsky, 2004) reduces the possibility of malicious access from any client by hiding classes which do not need to be publicly visible. No attacker can access sensitive data stored in the encapsulated classes since any attacker has no way to know a name of such class and a reference to its instance.

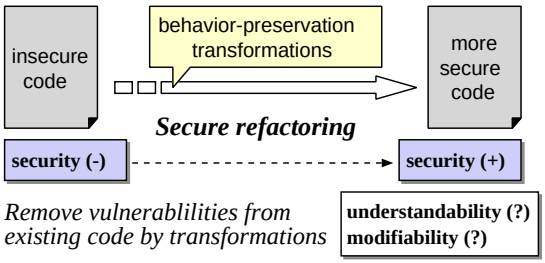

Figure 1: Overview of secure refactoring.

3 SECURE REFACTORING

Secure refactoring is a collection of safe transformations each of which certainly increases the security level of the refactored code. All the safe transformations ensure no change of observable behavior of the refactored code. This means that the resulting set of output values for the same set of legal (not malicious) input values is the same before and after refactorings. Figure 1 shows an overview of this refactoring. For the definition of traditional refactoring, secure refactoring is defined as follows:

> A change made to the internal structure of software to make it more secure without changing its observable behavior.

Both the secure and traditional refactorings have the same property of preserving the external behavior in a series of their transformations. A significant difference between them is that the secure refactoring is intended for security improvement of existing code while the traditional refactoring emphasizes the maintainability of the code. Of course, understandable or modifiable code is also useful for security improvement since security vulnerabilities of such code will be easily detected and fixed. However, many programmers with knowledge of software security often encounter situations in which they should quickly eliminate security bugs and flaws from executing code without regard to its maintainability. The secure refactoring is designed to be used in this case. By employing this refactoring on software development or maintenance, the programmers will be able to easily obtain software that is more secure than before without tedious checks and bug injection.

This section will explain four secure refactorings and their mechanics in detail. Each refactoring has a name, motivation, mechanics, and examples. The motivation describes why the refactoring should be applied and the result of its application. The mechanics show a situation in which the refactoring should be applied (called *security smell*), and they present concise procedures of how to carry it out. The examples

```
// before refactoring (original)
public class Member {
  private String name;
  public Member() { }
  public void setName(String n) {
    name = n;
  }
  public String getName() {
    return name;
  }
}
```

```
// after refactoring (more secure)
public class Member {
  private final String name;
  public Member(String n) {
    name = n;
  }
  public String getName() {
    return name;
  }
}
```

Figure 2: Introduce Immutable Field.

show code snippets before and after the application of the refactoring.

3.1 Introduce Immutable Field

One of the important tenets of software design is information hiding or encapsulation (Parnas, 1972). This is useful for building secure software since it hides inside its sensitive data and minimizes accessibility of the data. In other words, secure code should make a field private and provide accessors with the lowest possible access level consistent with the code that has to access the field (Bloch, 2001).

This concept is satisfied by using the Encapsulate Field refactoring that was already proposed. It prevents other objects from directly accessing the encapsulated field. However, its value might be still modified through its public (or non-private) getting method although its setting method would be deleted (or strongly restricted). To protect any field from malicious modification, it should become immutable unless there is a reasonable reason that other objects must modify it. In summary, a field must be declared final if its value is set only once. This improvement is named an Introduce Immutable Field refactoring.

Figure 2 shows two code snippets in this refactoring. The underlines attached to the code indicate differences between the two versions. As compared with the original code, in the refactored code the keyword `final` was added to the field `name` and the value of `name` is always set in the constructor of the class `Member` that defines `name`. The setting method for `name` appearing in the original code was deleted.

The mechanics are shown as follows:

Security smell: In a target class C, there is a mutable field f the value of which is set only once.

1. If f is declared public, use Encapsulate Field (206)[1].
2. Find all calls to a setting method of f.
3. Check to see if the calls can be both moved all together to the succedent position of a constructor call creating an instance of C that defines f and executed immediately after the constructor call.

   $\Rightarrow$ *If any call cannot be coupled with its corresponding constructor call, cancel this refactoring.*
4. For all constructors of C, add a parameter that can pass on the value of f, by using Add Parameter (275). Create an assignment for putting the received value into f in each of the constructors.
5. Remove the setting method of f and all the calls to the method.
6. Add the keyword `final` to f in its declaration. If all fields within C can be immutable, it should be considered that the class becomes immutable.

3.2 Replace Reference with Copy

The value of a final field can be never changed once it is set. However, data stored in this field can be changed if its declarative type is both reference (not primitive) and mutable. Consider a code snippet shown on the top side of Figure 3. The value of the field `name` cannot be basically changed since it is declared final and has no setting method of `name`. However, any class makes it a success to modify the value of the field `lastname` if the class can access an instance of the class `Member`. Consequently, internal data stored in `name` can be altered by exploiting the following simple code:

```
member.getName().setLastname("X");
```

where `member` is an instance of `Member`.

A major flaw is that `Member` carelessly reveals a reference holding an instance of the class `Name`. An attacker can obtain this reference by calling the method `getName()`, and then change internal data by exploiting the obtained reference. To fix this security flaw, the code of `Member` should not return the reference to an original instance that contains sensitive data. Instead, it should provide an instance that contains (partial) copies of the sensitive data. This improvement is named a Replace Reference With Copy refactoring, which is one of variations of Fowler's Encapsulate Collection (208).

[1]A parenthetic number indicates the page number of Fowler's catalog (Fowler, 1999).

```
// before refactoring (original)
public class Member {
  private final Name name;
  public Member(String fn, String ln) {
    name = new Name(fn, ln);
  }
  public Name getName() {
    return name;
  }
}

public class Name {
  private String firstname;
  private String lastname;
  public Name(String fn, String ln) {
    firstname = fn;
    lastname = ln;
  }
  public void setLastname(String ln) {
    lastname = ln;
  }
}
```

```
// after refactoring (more secure)
public class Member {
  private final Name name;
  public Member(String fn, String ln) {
    name = new Name(fn, ln);
  }
  public Name getName() {
    return new Name(
           name.firstname, name.lastname);
  }
  void setLastname(String ln) {
    name.setLastname(ln);
  }
}
```

Figure 3: Replace Reference With Copy.

A code snippet after the application of this refactoring is shown in the bottom side of Figure 3, in which the code of `Name` is the same as before. In the refactored code, any attacker cannot modify the original value of `lastname` without tampering the code of `Name`. Thus, the original value never damages although its copy can be modified.

The mechanics are shown as follows:

Security smell: In a target class *C*, there is a public final field f that holds either a reference to an instance *i* storing sensitive data or a public getting method m_f that returns the reference to *i*.

1. If f is declared public, use Encapsulate Field (206). The getting method newly created is also called m_f.
2. Find all code fragments using the reference obtained through m_f, and check to see if each of the fragments alters the value of *i*.

   ⇒ *If any code fragment alters the value, consider moving the fragment to either C or a class existing in the same package as C by using* Extract Method (110) *and* Move Method (142).

```
// before refactoring (original)
public class Member {
  public String getInfo(Password pw) {
    if (pw.check()) {
      return "Confidential info";
    } else {
      return "No info";
    }
  }
}

public class Password {
  private final String passwd = "SECRET";
  private final boolean result;
  public Password(String pw) {
    result = passwd.equals(pw);
  }
  public boolean check() {
    return result;
  }
}
```

```
// after refactoring (more secure)
public final class Password {
  private final String passwd = "SECRET";
  private final boolean result;
  public Password(String pw) {
    result = passwd.equals(pw);
  }
  public final boolean check() {
    return result;
  }
}
```

Figure 4: Prohibit Overriding.

3. Check to see if all the altering fragments are enclosed in the package containing *C*.

   ⇒ *If some of the fragments remain outside the package, cancel this refactoring.*
4. Modify m_f so that it returns a copy of *i*.
5. Add a new method with the default accessibility setting into *C*, which is used to alter the value of *i*.

Guidelines similar to this refactoring were proposed (Wheeler, 1999; Bloch, 2001), which suggest that secure code should never return a mutable instance to potentially malicious code. However, these guidelines do not present concrete mechanics.

3.3 Prohibit Overriding

Polymorphism in object-oriented programming is a powerful mechanism in which an instance can send a message without knowing a class of the receiving instance. Therefore, it makes code more flexible to future extension. However, the excessive use of polymorphic calls with inheritance and method overriding is dangerous since such calls allow an attacker to have a chance to execute malicious code.

Consider a code snippet shown in the top side of Figure 4. This well-mannered code fragment using the two classes `Member` and `Password` creates an instance of `Password` by giving a letter string indicat-

```
public class Fake extends Password {
  public Fake(String pw) {
    super(pw);
  }
  public boolean check() {
    return true;
  }
}
```

Figure 5: Malicious class using overriding.

ing a password, and then calls the method `getInfo()` with the created instance. The code fragment can obtain sensitive information only if the given letter string is equal to a right password[2]. At a glance, this password authentication seems to be secure. However, a clever adversary can easily see confidential information without knowing the right password by using both the malicious class shown in Figure 5 and the following code:

```
Password fake = new Fake("X");
System.out.println(member.getInfo(fake));
```

where `member` is an instance of `Member`. The class `Fake` is derived from `Password` and its method `check()` always returns true. The instance `member` might be obtained from a class which is deservedly contained in a framework containing `Member`.

To avoid this attack, secure code should not allow an arbitrary (possibly malicious) class (`Fake` in this case) to redefine a method existing in the code. Simply stated, the keyword `final` will be attached to the method which has to block its overriding, as shown in the bottom side of Figure 4. This improvement is named a Prohibit Overriding refactoring.

The mechanics are shown as follows:

Security smell: In a target class C, there is a call to a method m of a class C_i for an instance which is given by external code, and any class can be derived from C_i.

1. Find all subclasses of C_i and check to see if respective methods of the subclasses override m.

   $\Rightarrow$ *If overriding methods exist, consider applying the Replace Inheritance with Delegation (352) to replace them with delegation methods. If some of them remain, cancel this refactoring.*

2. Add the keyword `final` to m in its declaration. If there is no subclass of C_i, make C_i final.

[2]Actually, a right password and sensitive information must be stored in protected files instead of being hard-coded.

3.4 Clear Sensitive Value Explicitly

According to Java's garbage collection mechanism, it is inadvisable for a running program to preserve sensitive information in its memory area if it will be never used in the future (SunMicrosystems, 2000). To protect such information from a stealer, it should be explicitly cleared at the earliest possible time.

A Clear Sensitive Value Explicitly refactoring finds statements which finally use (refer) the value of a variable storing sensitive data, and identifies locations where the needless value should be deleted. This process is performed by analyzing data dependencies (Ferrante et al., 1987) for a target code. A final-use statement corresponds to a final-use node shown as follows:

> For a variable v in the target code, s_v denotes a statement either writing or reading the value of v, and b_v denotes a conditional statement (e.g., `if` or `while`) a branch of which includes s_v (b_v dangles s_v). A node n_v corresponds to s_v or b_v on the control flow graph (CFG) of the code, and $P(n_v)$ is a reachable path from a node which is one of nodes executed immediately after n_v. If there is at least one $P(n_v)$ which does not include any nodes which correspond to statements writing or reading the value of v, n_v is defined as a **final-use node** for v.

Figure 6 depicts several patterns of final-use nodes (gray ones) for a variable v of interest. The nodes labeled with def and use mean statements writing and reading the value of v, respectively. New statements clearing the value of v should be inserted immediately after all final-use nodes for v. In Figure 6, the triangles indicate the respective insertion points.

Figure 7 shows code snippets in this refactoring. In this case, the statement "`score = -1`" was inserted (corresponding to Figure 6(d)). An attacker cannot inspect the value of the variable `score` after executing the inserted statement.

The mechanics are shown as follows:

Security smell: In a target class C, there is a method m which keeps the sensitive value of a local variable (a parameter) v until the execution of m is terminated.

1. Check to see if the value of v is stored in an immutable instance.

   $\Rightarrow$ *If an immutable instance is used, replace it with the use of a mutable instance.*

2. Find final-use statements for v.

3. Insert new statements that update the values of v with dummy ones.

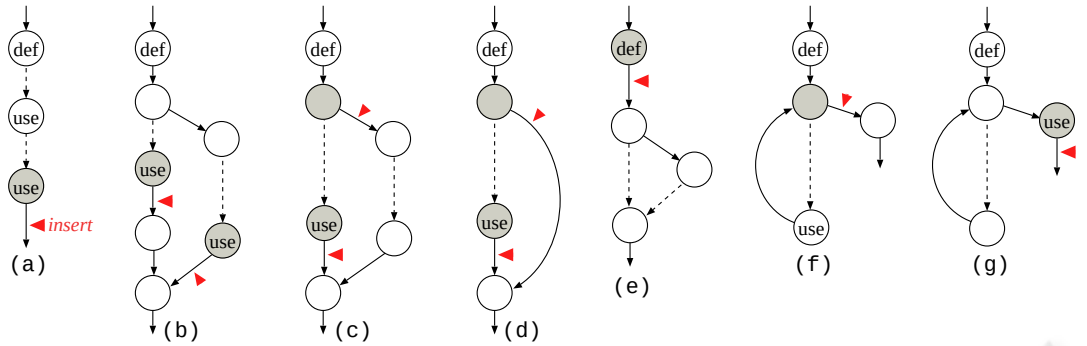

Figure 6: Final-use nodes (gray) and locations (triangles) where new statements should be inserted.

```
// before refactoring (original)
public class Member {
  public void printMessage(int score) {
    if (score < 60) {
      print(score);
      printMessage();
    }
  }
}
```

```
// after refactoring (more secure)
public class Member {
  public void printMessage(int score) {
    if (score < 60) {
      print(score);
      score = -1;
      printMessage();
    } else {
      score = -1;
    }
  }
}
```

Figure 7: Clear Sensitive Value Explicitly.

Here supplemental remarks for the step 1 will be mentioned. For example, if confidential information such as a password is stored in an instance of the immutable class `String` (more precisely, `java.lang.String`), `String` must be converted into an array of `char` since the value stored in a `String`-typed variable cannot be explicitly deleted. In this case, several utility methods of the `java.util.Arrays` class can be used for comparing the contents of two arrays and deleting these contents.

Moreover, we note that this refactoring might be adversely affected by compiler optimization since several compilers will remove the inserted statements.

4 RELATED WORK

There are two main approaches that are available for increasing the security level of existing software. One approach is to use a checker which automatically (or semi-automatically) examines source (or object) code of a target software program before it runs. For example, Jslint (Viega et al., 2000) is capable of detecting security vulnerabilities by using 12 rules for writing source code. However, it does not provide a way of how to remove the detected vulnerabilities. For most programmers, some secure programming guidelines (McGraw and Felten, 1998; Wheeler, 1999; Viega and Messier, 2003; Whittaker and Thompson, 2001; Graff and van Wyk, 2003) are well-known rather than code checking. These guidelines are secure and insecure code patterns (or idioms), which are not formally created but pragmatically collected in realistic programming. If vulnerable patterns appear in software, the programmers should remove them according to the guidelines by hand. In other words, there is still no systematic mechanism and automated tool for supporting secure programming. The proposed secure refactoring has the potential to formalize secure programming since it provides mechanics (explicit conditions and procedures) of detection and/or correction which are all expected to be implemented in an automated tool. Moreover, it seems to be a particular kind of secure programming which preserves the behavior of existing code.

The other approach does not directly increase the security level of existing software. Instead, it provides a secure environment in which untrusted software programs safely run. The Java sandbox (Oaks, 2001) is one of the famous mechanisms that implement such environment. This approach can reduce the burdens imposed upon programmers since it needs no troublesome or error-prone modification of existing software. The weak point of the approach is that it requires the installation of an execution environment and such environment is too general or typical to capture various kinds of vulnerabilities or sophisticated attacks. Moreover, it must pay the overhead cost for dynamically gathering and analyzing much information about an executing program.

Unfortunately, there are few studies about refactoring concerning security. One of the studies pro-

vides a method that refactors an existing program into distinct modules with high and low security (Smith and Thober, 2006). The proposed refactoring identifies code which depends on high security inputs by using program slicing based on information flow of C programs, and extracts a new module including the code. Although the detailed mechanics and target programming languages are different between this work and mine, these aims are almost the same.

5 CONCLUSION

Although many programmers want to easily remove vulnerabilities from existing software programs, there are still few systematic mechanisms which are intended to be codified into automated tools. This paper has proposed secure refactoring which improves the security level of existing code without changing its observable behavior. Two main issues will be tackled in future work.

- For the proposed secure refactorings to be practically applied to realistic software development, tool support is considered crucial. Moreover, a running implementation is essential for demonstrating the feasibility of automation of the secure refactorings. The author is currently working on integration of the proposed refactorings into our developed tool, Jrbx (Java refactoring browser with XML) (Maruyama and Yamamoto, 2005)[3]. Moreover, the implementation of the refactorings might require elaboration of the presented mechanics.
- It is not guaranteed that each transformation of the proposed secure refactorings neither changes the behavior of an existing code nor inserts new vulnerabilities into the code. In addition to a theoretical proof, the author will make a large number of experiments with various programs to check the ability of the refactoring transformations.

REFERENCES

Bloch, J. (2001). *Effective Java: Programming Language Guide*. Addison-Wesley.

Devanbu, P. T. and Stubblebine, S. (2000). Software engineering for security: A roadmap. In *ICSE '00: The Future of Softw. Eng.*, pages 227–239.

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

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gollmann, D. (2006). *Computer Security, 2nd ed.* John Wiley & Sons.

Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. Addison-Wesley.

Graff, M. G. and van Wyk, K. R. (2003). *Secure Coding: Principles and Practices*. O'Reilly & Associates Inc.

Hoglund, G. and McGraw, G. (2004). *Exploiting Software: How to Break Code*. Addison-Wesley.

Howard, M. and LeBlanc, D. (2002). *Writing Secure Code, Second Edition*. Microsoft Press.

Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley.

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

Maruyama, K. and Yamamoto, S. (2005). Design and implementation of an extensible and modifiable refactoring tool. In *Proc. IWPC'05*, pages 195–204.

McGraw, G. (2006). *Software Security: Building Security in*. Addison-Wesley.

McGraw, G. and Felten, E. (1998). Twelve rules for developing more secure java code. *Javaworld*. http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Trans. Sofw. Eng.*, 30(2):126–139.

Oaks, S. (2001). *Java Security, 2nd ed.* Addison-Wesley.

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

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058.

Smith, S. F. and Thober, M. (2006). Refactoring programs to secure information flows. In *Proc. PLAS'06*, pages 75–84.

SunMicrosystems (2000). Security code guidelines. http://java.sun.com/security/seccodeguide.html.

Viega, J. and McGraw, G. (2001). *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley.

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

Viega, J. and Messier, M. (2003). *Secure Programming Cookbook for C and C++*. O'Reilly & Associates Inc.

Wheeler, D. A. (1999). Secure Programming for Linux and Unix HOWTO. http://www.dwheeler.com/secure-programs/.

Whittaker, J. A. and Thompson, H. H. (2001). *How to Break Software Security*. Addison Wesley.

[3]Jrbx can be downloaded from http://www.jtool.org/.