Traceability for the Maintenance of Secure Software

Yijun Yu [1], Jan Jürjens [1], John Mylopoulos [2]
[1] *Department of Computing, The Open University*
[2] *Department of Computer Science, University of Toronto*
{y.yu,j.jurjens} @open.ac.uk, jm@cs.toronto.edu

Abstract

Traceability links among different software engineering artifacts make explicit how a software system was implemented to accommodate its requirements. For secure and dependable software system development, one must ensure the linked entities are truly traceable to each other and the links are updated to reflect true traceability among changed entities. However, traditional traceability relationships link recovery techniques are not accurate enough. To address this problem, we propose a traceability technique based on refactoring, which is then continuously integrated with other software maintenance activities. Applying our traceability technique to the proven SSL protocol design, we found a significant vulnerability bug in its open-source implementation. The results also demonstrate the level of accuracy and change resilience of our technique that enable reuse of the traceability-related analysis on different implementations.

Keywords *traceability, refactoring, maintenance, security*

1 Introduction

Requirements traceability is defined as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" [10]. Existing traceability approaches aim to recover traceability links that connect elements of certain software engineering artifacts in requirements, design and implementation [1, 8, 4, 13]. In general, none of them can recover accurate requirements traceability links that preserve the semantics of traced elements.

On the other hand, high assurance is required in secure and dependable software systems development. A single inaccurate requirement traceability link assumed by developers may already be exploited by malicious attackers. To assure high trustworthiness, software using such mechanisms must be analyzed thoroughly. In [15], we proposed an approach for establishing that the design of crypto-based software based on the security extension to UML (UMLsec[14]) satisfies relevant security requirements using automated theorem provers. In [16, 17], we showed how one can link a Java-based implementation of a crypto-protocol to its representation in UMLsec using assertions. In such experience, it is however non-trivial to insert the right assertions at the right place in the program. As the implementation or the used libraries evolve, the instrumentation may no longer guarantee correct traceability links. Moreover, it is unclear whether and how such assertions can be reliably transferred to a different implementation of the protocol.

This work was motivated by the need to accurately trace the design to the implementation of a crypto-based software. By accurate traceability, we mean that the implementation is verified to satisfy the specification in the design, without introducing any false relations between them. We propose an approach to maintain accurate traceability through refactoring. We have developed a change resilience refactoring language and tools in order to maintain accurate traceability in a process continuously integrated with other software maintenance activities. Through accurate and change-resilient traceability, the analysis of implementation errors of a design model can be reused to analyse a different implementation of the same design.

To demonstrate the effectiveness of our proposal, we show how to apply refactoring-based traceability to cryptographic protocol implementations. Our method was applied to JESSIE and JSSE, open-source implementations of the Java Secure Sockets Extension in order to establish accurate traceability. For different versions of the implementation as well as different implementations of the same protocol design, we demonstrate that our refactoring tool enables reuse of the test cases for vulnerability analysis and aspects for security hardening.

The next section presents the properties of traceability required by secure software maintenance, followed by an explanation of our refactoring-based traceability approach. Section 3 illustrates the rationale and implementation of the tool support by a running example. Section 4 explains the approach presented at the hand of an application to the Internet security protocol, SSL.

2 Traceability for maintenance

To be useful for maintenance of dependable software systems, traceability needs to be as *accurate* as possible. When a model element is changed, false negative traceability may lead to neglects of some updates; whereas false positive traceability may lead to unnecessary updates. Using search-based traceability techniques [1, 4], precision (i.e., whether the keywords match with the selected document) and recall (i.e., whether all matching documents are selected) metrics determine the accuracy of recovered traceability links. When precision and recall are below 100%, as are in usual cases, false traceability is inevitable.

To support software evolution, accurate traceability also needs to be as *resilient to change* as possible, that is, traceability links remain true even when the models change. With change resilient traceability, one does not need to update a link as long as it can still accurately relate the changed artifacts. Otherwise, traceability links have to be rediscovered whenever changes happen.

Taking advantages of accuracy and change resilience, the traceability can be applied to secure and dependenable software maintenance, in many useful ways: (1) By checking whether all elements in the design are traced faithfully into the implementation, one can tell whether an implementation has correctly carried out a given design. This is a direct application of traceability. (2) While accurate traceability to the same design has been established for two implementations, through transitive relations, one can trace between elements in these two implementations. Such indirect traceability, whenever possible, can help to derived parts of the implementation from the other. Consider existing test cases used to validate correctness of one implementation to the design. If such test cases do not exist for another implementation of the same design, one can construct them based on the ones exist in this implementation. (3) The traceability process can then be continuously integrated with other interactive and automated maintenance activities. Whenever changes to the models are committed to a shared repository as a result of other maintenance activities, necessary traceability steps will be triggered in order to maintain the benefit of (1) and (2). The change resilience property can help reduce the effort in such maintenance due to the fact that the traceability may not always need to be updated.

Software refactoring [9, 21] changes the internal structure of an implementation without changing its external behavior. Thus in this work, we propose to use refactoring steps to obtain and maintain accurate traceability. We first show it possible to obtain accurate traceability among design and implementation elements using refactoring steps; then we illustrate how to improve change-resilience of refactoring steps using declarative refactoring scripts that can be translated into context-sensitive ones. Then we use derived traceability and continuous integration to explain benefits of refactoring-based traceability support.

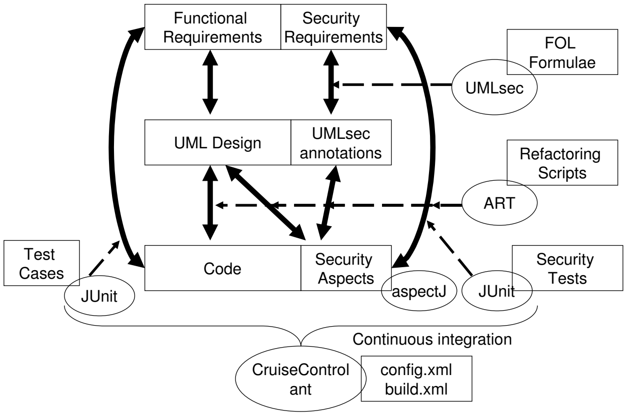


Figure 1. Maintain Traceability

Figure 1 shows the big picture of how our tools are integrated to support traceability for maintenance of secure software.

2.1 Refactoring for accuracy

Using refactoring, we can create accurate traceability between design and implementations. In a round-trip, one can (1) convert the identifiers/methods to names at the design level, and (2) convert the names on the design level to identifier/method names in the implementation. Through a sequence of refactoring steps, every occurrence of a selection of program elements can be transformed into their counterpart design elements.

Since refactoring maintains external behavior unchanged, one can transform a program entity into another without worrying about losing accuracy in behavior. Applying refactoring for a number of steps, the resulting program produces the same results. Therefore, the traceability transitively from the original program to the resulting program remains accurate. The resulting program is typically more abstract than the original because refactoring steps are used to improve understandability of the program.

To make sure that the resulting program maps to the design element accurately, additional program understanding tasks may need to be performed by the analyst. In our case study of the security protocol implementation, for example, a variable named after the design element "R_C" should be a random seed, which can only be verified by finding out that it was assigned by the returned value of a random number generation function in the library. Once the relation between the refactored program element and the design element is confirmed by the analyst, the original program entity also accurately traces to this design element, no matter whether it was originally named "Random" or it was originally a sequence of statements.

2.2 Refactoring for change resilience

Having accurate traceability links between design symbols and program entities established, one would maintain them even when some design symbols or program entities change. Due to the fact that refactoring steps can be applied with a certain precondition, that is, they are applicable only when the program meet a certain pattern in a certain context. We need to specify the minimal requirement of the application context of refactoring steps such that they are still applicable even if the original program changes. It is also preferable to have traceability links automatically maintained.

Modern refactoring-supported IDE's, e.g., Eclipse, supports automatically record and replay applied refactoring steps. When the source code is not changed, in other words, these recorded steps can be replayed if the code is exactly the same. However, when code changes slightly, they often fail to replay. For example, an "Extract Method" refactoring can substitute a selection of statements into a method. If there is a statement inserted before the selection, then the refactoring will not be replayed.

In order to allow refactoring on changed code, we designed a declarative refactoring specification language. Combining with the changed code, the specification script pinpoints the exact context for the refactoring steps. In addition, the declaration can characterise an applicable context using fewer parameters by virtue of regular expressions. These parameters and expressions were initially generated from the refactoring steps recorded in the IDE using a transformation utility. By changing all spaces into all possible separators in the selection criteria, for example, a pretty formatting or obfuscation of the program will not block the application of the "Extract Method" refactoring. The change resilience can be further improved by adapting the regular expressions in the specification, for example, by ignoring any renaming to local variables. Existing clone detection algorithms could also be applied to allow for slightly modified code to match.

2.3 Derived traceability

When there are two implementations of the same design, accurate traceability may enable reusing the analysis results of one implementation for the analysis of the other. Artifacts including refactoring scripts, test cases, aspects, can improve the understanding of one implementation, they can also be useful in analysing the other implementation if the traceability links between design and both implementations are bijective. Take a test case for example, if all its program elements P can be traced to design elements D, and all these relevant design elements can be traced to those in another implementation Q, then it is possible to reuse the test case by substituting elements in P with the counterparts in Q. Suppose that a program element $p \in P$ is traced to a design element $d \in D$ through a sequence of refactoring steps R_1, and a program element $q \in Q$ is traced to the same design element d through another sequence of refactoring steps R_2. The substitution of p to q can be achieved by first applying the refactoring R_1 to the test case, then apply the refactoring R_2 inversely.

Most refactoring steps are invertible as they are equivalent transformation of programs. For example, "Rename variable" from A to B is the invert step to "Rename variable" from B to A. For more complex refactoring steps (e.g., "Extract Method"), the invert step is a different kind of refactoring (e.g., "Inlining Method"). Thus it is possible to allow traceability links to be composed as the *derived traceability*.

2.4 Continuous integration

Continuous integration[1] has been adopted by our process where the regression test subprocess is augmented with the regressive refactoring: whenever code or model changed in the repository – e.g., a developer committed a set of changes – the continuous integration script will check out the change set into a sandbox to conduct various automated build and tests. Adding our refactoring scripts to the continuous integration script enable the regression security engineering. The error report subprocess is also augmented with an explanation of the counter example of potential attack traces and the mismatch between the UMLsec model and the implementation code.

3 Tool support

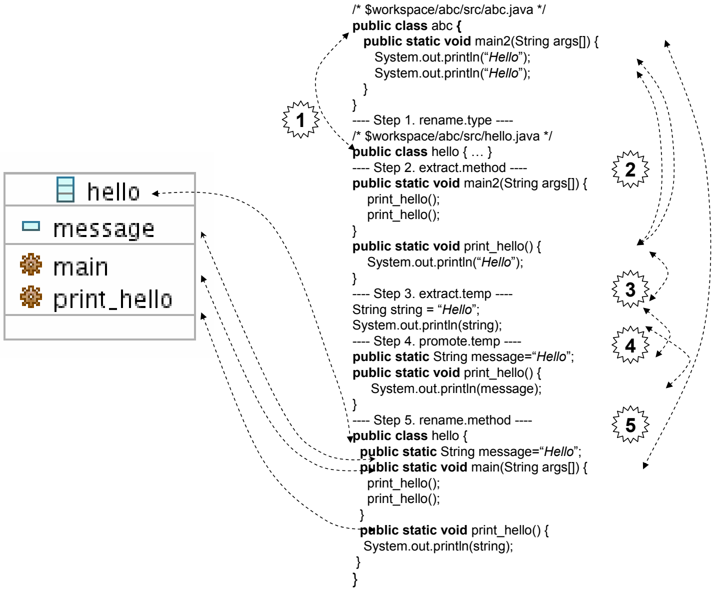
In this section, we explain the tools we implemented support the traceability for our case study. These tools are built on top of the Eclipse IDE, the CruiseControl continuous integration tool and our UMLsec tools.

A running example To illustrate, Fig. 2 shows a series of refactoring steps applied to a small "Hello World" program. The example is explained in the context of Eclipse IDE, where a number of refactoring steps are supported in the tool.

Assume that the source file abc.java is initially located at a folder src in the project abc. The refactoring steps are applied as follows. Step 1: Class abc is renamed to

---

[1] See M. Fowler and M. Foemmel. Continuous integration. http://www.martinfowler.com/articles/ continuousintegration.html



```
/* $workspace/abc/src/abc.java */
public class abc {
  public static void main2(String args[]) {
    System.out.println("Hello");
    System.out.println("Hello");
  }
}
---- Step 1. rename.type ----
/* $workspace/abc/src/hello.java */
public class hello { … }
---- Step 2. extract.method ----
public static void main2(String args[]) {
   print_hello();
   print_hello();
}
public static void print_hello() {
   System.out.println("Hello");
}
---- Step 3. extract.temp ----
String string = "Hello";
System.out.println(string);
---- Step 4. promote.temp ----
public static String message="Hello";
public static void print_hello() {
    System.out.println(message);
}
---- Step 5. rename.method ----
public class hello {
  public static String message="Hello";
  public static void main(String args[]) {
   print_hello();
   print_hello();
  }
  public static void print_hello() {
  System.out.println(string);
 }
}
```

Figure 2. Refactoring for traceability

hello and abc.java is renamed to hello.java accordingly. This refactoring step is called *rename.type*. Step 2: The first statement System.out.println is extracted into the body of a new method print_hello(). All instances of the selected statement are substituted at once, resulting in a 2-to-1 mapping. This step is called *extract.method*. Step 3: The expression ”Hello” is explicitly assigned to a new local (temporary) variable string. This step is called *extract.temp*. Step 4: The temp variable string is promoted into a field named as message. Finally, Step 5: The method main2 is renamed to a new method name main. This last step is called *rename.method*.

After these steps, the refactored program is traced to the UML class model intended by the designer, as shown to the left of Fig. 2. The traceability between the design elements (class names, method names and field names) and identifier/method names are established. Moreover, each refactoring step is a program transformation that preserves the behavior before the step. Note that the refactoring-based traceability is not one-to-one mapping between the source and the target. In other words, a single refactoring step can update multiple references in the design/program. As each refactoring step is well-known, one can understand the traceability between the original design and the final implementation.

Refactoring support in Eclipse The general refactoring engine in Eclipse is provided by a set of plugins called the refactoring Language Toolkit (LTK)[2], which allows one (1) to perform refactoring steps, (2) to save the history of refactoring steps into an XML-based script, and (3) to apply a refactoring script automatically. The plugins are applicable to any programming or specification language. The Java Development Tool (JDT), for example, instantiates LTK with a number of Java-specific refactoring steps. Fig. 3 shows a snippet from the refactoring script, which briefly specifies the *rename.type* and *extract.method* steps used at the first two steps.

```
<?xml version="1.0"?>
<session version="1.0">
 <refactoring comment="..."
  id="org.eclipse.jdt.ui.rename.type"
  description="Rename type 'abc'"
  project="abc" input="/src&lt;abc.java[abc"
   name="hello" ... />
 <refactoring comment="..."
  description="Extract method 'print_hello'"
  id="org.eclipse.jdt.ui.extract.method"
  project="abc" input="/src&lt;{hello.java"
  name="print_hello" selection="64 28" ... />
 ...
</session>
```

Figure 3. Eclipse refactoring script, Cf. Fig. 2

Every refactoring is recorded as an XML element refactoring, whose attributes specify the step. Every step has an identifier attribute ID, indicating the type of the step. For example, here org.eclipse. jdt.ui.rename.type is a name internally used by JDT to identify the *rename.type* refactoring. For readability, in the remainder of the paper we omit the common prefix and simply call it rename.type. The target of a refactoring step for rename.type is a new class name, whereas the target for extract.method is a new method name. They are completely specified by the name attribute. The source of a refactoring step is suggested by attributes including project, input and optionally selection. The values of these attributes typically indicate the context of a step. The project attribute specifies the subject project of the refactoring step; inherited from LTK, the input attribute is a composite of the source folder, the source package, and the source class name which are separated by delimiters "<", and "["; the selection attribute, when used, specifies the exact offset and length of the string selected for the refactoring.

In our example the extract.method refactoring is only applicable if the selection of a substring of 28 characters starting from the offset 64 in hello.java matches the statement to extract, character by character including the white spaces. Given such strict specifications of refactoring contexts in Eclipse, one can see that existing refactoring scripts are inadequate if source code has been modified by evolution or by previously applied refactoring steps, or source code from different library implementation is used. For example, it is required to modify the offset/length value if an extract.temp step has been applied earlier.

---

[2] http://www.eclipse.org/articles/article.php?file= Article-Unleashing-the-Power-of-Refactoring

```
@{org.eclipse.jdt.ui.rename.type,
  project="abc",  source="src",  package="",
  class="abc",   name="hello"
}
@{ org.eclipse.jdt.ui.extract.method,
  project="abc",  source="src",  package="",
  class="hello",  method="main",
  toclass="hello",  name="print_hello",
  regexp="S.*("Hello");",  count="1"
}
```

Figure 4. Our spec. for refactoring (cf. Fig. 3)

Refactoring scripting In the subsection, we present our new refactoring engine that overcomes the limitation of the native Eclipse JDT refactoring steps. It makes the refactoring steps reusable for maintaining design traceability in different legacy code.

One would reuse the traceability information discovered when linking the implementation to the UML model for example if one wants to apply the refactoring steps defined for one implementation (e.g., JESSIE 1.0.1) to a different version of that implementation (e.g., JESSIE 1.0), or to a different library (e.g., JSSE). To this end, we create a refactoring plugin that can apply parameterized refactoring steps[3]. Our refactoring tool is implemented on top of LTK refactoring plugins, which supports languages beyond Java. In order to keep the changes to the existing refactoring engine limited, we invoke the context-specific refactoring steps in JDT by instantiating a scripting template with the parameters derived from our specifications.

In [19], Krueger classified software reusability as five connected facets: abstraction, classification, selection, specialization and integration. Our traceability refactoring engine supports this view.

Our declarative specification language *abstracts* away context-sensitivity of existing refactoring steps and can describe any refactoring step supported by LTK. Corresponding to Fig. 3, Fig. 4 lists two refactoring steps in our specification language.

In the record of our refactoring specification, most fields have evident meaning and usage as they correspond to the attributes in the Eclipse refactoring scripts. We introduce new fields to compute the context of the source element, such as source, package. Optionally, the field condition indicate a *selection* to be refactored by a generic condition (e.g., a regular expression). Such selections increase the chance of reusability for context-sensitive refactoring steps when changes happen to the code. We can actually derive the condition from the concrete context. For example, by replacing white spaces with arbitrary number of white spaces. In this way, even if a programmer or a code formatter inserted indentation spaces, the selection can still be matched. Also we introduce the count field to selectively refactor some instances of matching selection rather than the first matched one. The selection parameter gets *specialized* from the other parameters by parsing the selection source using existing API in the IDE.

As refactoring consists of a sequence of steps, we *classify* existing refactoring steps by context-sensitivity and discuss its impact on exchangeability and invertibility. Context-free steps are more reusable whereas context-sensitive ones require care. For round-trip traceability, all refactoring steps need to be invertible. Since refactoring steps are behavior semantics-preserving, inverting them is generally feasible. Context-free steps are already invertible without consulting code (e.g., by inverting the source/target of the *rename.type* step). For context-sensitive steps, we made them invertible with the aid of code and the editing command stack because the information in the refactoring specification alone is not sufficient.

In Table 1, we list some JDT refactoring steps that have been parameterized in our refactoring tool. We also show which JDT steps are considered change resistant and a brief description on how such limitations are resolved.

Our tool delegates the domain-specific (here Java) refactoring *integration* tasks to LTK. During the integration, we support programmers to preview the effects of refactoring if they choose to, and to avoid manually constructing the specification from the saved refactoring history in Eclipse. The implementation of our refactoring plugin adds two command buttons to the Eclipse GUI, one of them performs all refactoring steps automatically, while the other brings up a dialogue for each step to preview the effects of refactoring. This allows us to verify if there are any potential maintenance problems arising from the step. For example, when renaming a variable to R_C, we can see a warning message from the Eclipse IDE that by programming convention, it is not recommended to let the name a variable start with capital letters. Such a renaming does not affect programmers because they can always edit the original source code. We also implemented a headless tool to invoke the functionality of the automated button as a RCP command. The arguments of the command provides the name of refactoring specification file. In this way, our automated refactoring step can be integrated continuously with other processes.

Another utility program we implemented is a transformation that converts an XML-based refactoring script from Eclipse IDE into our own specification language. The translation is done automatically by converting the XML attributes of the <refactoring> tag into fields of one record in our specification language. Special parsing to XML attributes such as "input" is performed as well, to encode the context to ease reuse. Whitespaces in "selection" attribute are globally substituted with an equivalent regular expres-

---

[3] These automated refactoring tools (ART), including their source code and examples in the paper, can be fetched from the project subversion repository http://computing-research.open.ac.uk/repos/art (username: guest, password: checkout).

Table 1. Some refactoring steps parameterized by our refactoring tool

ID	change resilient?	context	source selection	specified in Eclipse	our specification
org.eclipse.jdt.ui.rename.project	no	workspace	project	project	project
org.eclipse.jdt.ui.rename.folder	no	project	folder	folder	folder
org.eclipse.jdt.ui.rename.package	no	folder	package	package	package
org.eclipse.jdt.ui.rename.type	no	package	class	class	class
org.eclipse.jdt.ui.rename.method	no	class	method	method	method
org.eclipse.jdt.ui.move.method	no	class	method	method	method
org.eclipse.jdt.ui.extract.method	yes	class	statements	(offset, len)	(regexp [, count])
org.eclipse.jdt.ui.rename.temp	yes	method	variable	(offset, len)	(regexp [, count])
org.eclipse.jdt.ui.extract.temp	yes	method	expression	(offset, len)	(regexp [, count])
org.eclipse.jdt.ui.promote.temp	yes	method	expression	(offset, len)	(regexp [, count])

sions. After translation, one can further simplify the regular expressions to enhance change-resilience.

Continuous integration We extend the CruiseControl system by adding tasks to the ANT build and test scripts. A daemon process on the build/test machine periodically monitors whether there is any change to the repository. Whenever changed artifacts (including the code, the model, the test cases, the refactoring scripts and the security aspects and assurance test cases) are committed, the event triggered a run of the extended ANT build.xml script.

```
<project name="jessie" default="test" basedir="jessie">
    <target name="build" depends="refactoring"/>
    <target name="test" depends="build"/>
    // the following tasks are augmented
    <target name="umlsec"/>
    <target name="refactoring"/>
    <target name="saspect" depends="test"/>
</project>
```

The dependencies between the targets of the build.xml are straightforward. Before one can build the new system, the modified code must be refactored such that the changes committed by the programmers are synchronised with the model. The UMLsec security check for model vulnerability is done after the system is built and refactoring is done. Finally, security assessment are performed to validate the security requirements and security hardening aspects are performed to enforce vulnerability checks.

4 Example Application: SSL

We will explain the approach presented in this paper at the hand of an application to the open source implementation of the Internet security protocol SSL. SSL is the de facto standard for securing http connections, which however has been the source of several significant security vulnerabilities in the past and is therefore an interesting target. In this paper, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (cf. Fig. 5). We have used automated theorem provers to verify the UMLsec model of the SSL protocol against the relevant security requirements such as secrecy and authenticity using our tools [16].

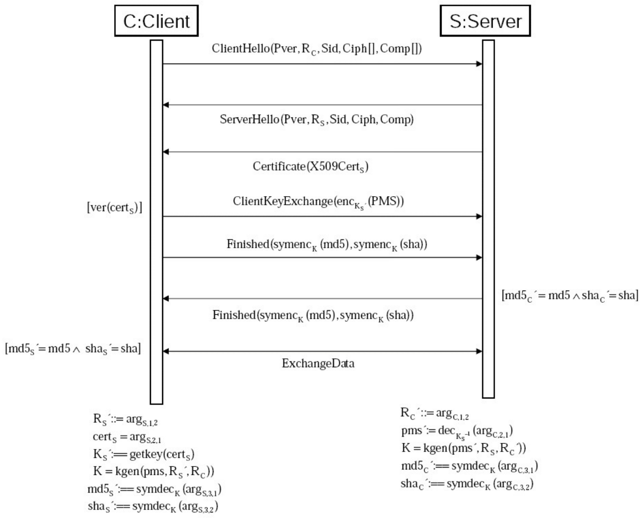


Figure 5. The SSL handshake protocol

The JESSIE Project The whole JESSIE project currently consists of about 5 MB of code, but the part directly relevant to SSL consists of less than 700 KB in about 70 classes. The implementation of the SSL protocol in JESSIE is only briefly documented by the comments in the program. Many important design elements in UMLsec (cf. Fig. 5) are missing in the program document.

Trace design to implementation After the security analysis of JESSIE version 1.0.1, we have identified 19 distinct symbols used in design models for cryptographic handshake protocols [18]. Table 2 presents 9 instances of such mapping. The first column shows the names of symbols as used in the cryptographic protocol model. The second column shows the names of corresponding methods in the JESSIE library. The third column shows the identifiers that are the target names of the refactoring steps. The types of the refactoring step is shown in the last column.

For each message in the sequence of an execution of the SSL handshake protocol (see Fig. 5), we perform a series of refactoring steps to establish traceability in the JESSIE 1.0.1 implementation. Table 3 lists the first four messages steps S1 to S4 in the message sequence protocol. One can also see that in general there does not need to be a one to one correspondence between the design and the code. For all the 19 symbols, 7 messages and 3 checks in Fig. 5, in total we have defined 27 refactoring steps in the specification to maintain the traceability between the protocol design and the JESSIE 1.0.1 code. The third column shows count of changed segments by the refactoring steps. Using `diff`, each block of changes, even when it contains multiple lines, is counted as one. When the number is larger than the number of steps, changes have happened to more than one places on average. The last column shows performance, i.e., how much time in seconds it took to perform the refactoring steps using our tools. Given the significant pay-off provided by the fact that the behavior of the code is preserved during the complex refactoring steps, such kind of performance figures do not impose a bottleneck within the overall process. On the contrary, much more time is spent on the security analysis and the manual creation of the refactoring steps, which will be paid back by reusing the scripts on different implementations.

The cryptographic protocol analysis requires for example that all the messages related to the cryptographic check Veri(X509Cert_s) (see Fig. 5) be intercepted and logged. The difficulty with applying AOP for such an instrumentation is that the joinpoint for such a check Veri(X509Cert_s) does not exist in SSLSocket.java as a method. Instead, the check is implemented as a group of statements of the whole doClientHandshake() method (lines 1518 through 1557 in the JESSIE library version 1.0.1). Therefore, an "extract.method" refactoring is necessary. Similarly, the cheVal joinpoint as a group of statements in SSLSocket.java (lines 1571–1604) needs to be refactored as a method.

Table 2. Mapping messages to methods

Symbols	Program methods	Identifiers	Refactoring op.
1. C	clientHello	C	rename.type
2. S	serverHello	S	rename.type
3. P_{ver}	session.protocol version	P_ver	extract.temp
4. R_C R_S	clientRandom serverRandom	R_C R_S	rename.temp rename.temp
5. S_{id}	sessionId sessionId	S_id S_id	rename.field rename.temp
6. Ciph[ ]	session.enabledSuites	Ciph	extract.temp
7. Comp[ ]	comp	Comp	extract.temp
8. Veri	Lines 1518–1557	Veri	extract.method
9. D_{nb} D_{na}	getNotBefore() getNotAfter()	D_nb D_na	rename.method rename.method

Table 3. Refactor the protocol (cf. Fig. 5)

Messages in sequence	op.	diff	Time (sec)
S1: $C \rightarrow S$: ($P_{\text{ver}}, R_C, S_{\text{id}}$, Ciph[ ], Comp[ ])	7	31	13.891
S2. $S \rightarrow C$: ($P_{\text{ver}}, R_S, S_{\text{id}}$, Ciph[ ], Comp[ ])	5	20	9.437
S3. $S \rightarrow C$: Certificate[X509Cert$_s$]	2	2	1.474
S4. C : Veri(X509Cert$_s$)	2	2	3.854
...	...	...	...
Total of 7 messages and 3 checks	27	86	40.303

Table 4. Test cases exposing vulnerability

Message	Example Test Case
S1	`Case1: ClientHello(TLSv1, clientRandom1, [B@b012a558, enabledSuites1, zlib)` `Case2-4: ClientHello(TLSv1, clientRandom2, [B@b01b0558, enabledSuites2, zlib)`

S4	`Case1,2: cheVal((107,2,2),(108,3,2))==True`
	`Case3:   cheVal((107,2,1),(107,3,1))!=False`
	`Case4:   cheVal((107,2,3),(107,3,1))!=False`

Vulnerability analysis and hardening Using a number of test cases, in the JESSIE 1.0.1 implementation, we found a significant security vulnerability as Veri(X509Cert_s) is not always invoked when the certificate message is received, which is a required and essential security check according to the protocol specification. It is needed because otherwise a man-in-the-middle attacker could insert a forged certificate containing his own public key into the communication and thereby decrypt the session key that is encrypted using that key, and thus eavesdrop on the encrypted communication in that session without being noticed by the communication partners. Additional checks can be inserted into the protocol to harden its security. For example, using an aspect to crosscut every joinpoint of the program where a certificate is received, we found nothing is called by the program to check the issuing date. Therefore we find it is necessary to instrument the program with the functionality to check validity of the certificate against its date range issued by OpenSSL.

Table 4 highlights the vulnerability by showing the execution log of 4 different test cases. If the certificate was checked, in Case 3 and 4, the cheVal should report false in a correct implementation. However, we found they reported true instead.

The vulnerability we found from the refactored program do exist in the original program, however, it was hidden from the joinpoint model of our security aspect before refactoring. Therefore, fixing such vulnerability by weaving an aspect on the refactored code can also, in fact, improve the implementation of the original program. After renaming checkValidity to cheVal, the aspect in Fig. 6 inserts an additional check on the validity of certificate date (cheVal). Also, the refactored Veri is called right *after* a certificate is obtained through the pointcut expression certificate(). Without these refactoring steps, this aspect cannot be weaved with the original program. This aspect is derived from the protocol design model introduced earlier assumes the existence of a method for Veri. This method is created from the given implementation using an *extract.method* refactoring for the doClientHandshake method to extract 58 lines of code into a new public method Veri in the SSLSocket class. The extracted Veri method is then called in the advice to reimplement the already existing check. In addition to this check, we then first introduced an additional cheVal method into the SSLSocket class and then moved it into the aspect module using the *Move Method* refactoring step. After these refactoring opertaions, the date validity check is performed before the existing certificate check.

```
public aspect CryptoProtocolSecurity {
 pointcut certificate():
      call(* Certificate.Certificate(..));
 Object around(): certificate() {
  X509Certificate[] X509Cert_s = (X509Certificate[])
      proceed();
  SSLSocket s = (SSLSocket) thisJoinPoint.getThis();
  for (int m=0; m<pCs.length;m++) {
   assert cheVal(pCs[m].D_nb(), pCs[m].D_na()):
      "+++ The date is invalid +++";
  }
  s.Veri(X509Cert_s);
  return X509Cert_s;
 }
}
```

Figure 6. A security aspect, cf.Table 4

From this example, one can see that refactoring serves two purposes. First, it reveals the control flow for instrumenting the program as a joinpoint. Second, it makes it possible to modularize the check into a security hardening aspect for reuse.

Reuse derived traceability Having studied one implementation of the cryptographic protocol in JESSIE 1.0.1, we aim at reusing our vulnerability analysis in the reference implementation of the same protocol in JESSIE 1.0.0, as well as in JSSE, a library in the standard JDK since version 1.4. The source code of JSSE library (after 1.6) can be checked out from the OpenJDK repository[4].

To perform the model-based security analysis as explained above on a different version of JESSIE or a different library (JSSE), one only needs to modify the specifications of the refactoring steps that provide the traceability of the model to the implementation level, without making any other adjustments to our refactoring engine and the analysis code, such as test cases and aspects.

Table 5. Reused refactorings, cf. Table 3

Messages	JESSIE 1.0.1		JESSIE 1.0.0		JSSE 1.6	
	op.	diff	op.	diff	op.	diff
S1	7	31	7	33	5	23
S2	5	20	5	21	4	16
S3	2	2	2	2	2	2
S4	2	2	2	2	2	2
...	...	...	...	...	...	...
Total	27	86	27	89	21	68

We have shown in Table 3 how many refactoring steps were applied to JESSIE 1.0.1 (released on October 12, 2005 according to its CVS repository) according to a maintainable refactoring specification. Table 5 shows how many steps in Table 3 can be reused on JESSIE 1.0.0 (released on June 9, 2004 according to its CVS repository), and JSSE 1.6 (released on May 8, 2007).

Inside the org.metastatic.jessie.provider package in JESSIE the 1.0.1 version has got 24 code block differences compare to that of 1.0.1 version. Due to these changes, the selection-sensitive steps in the refactoring history script saved from Eclipse cannot be applied to JESSIE 1.0.0. After converting the script into our specification language, all of them become reusable in our enhanced refactoring engine (cf. the column JESSIE 1.0.0). The only necessary change made to our original refactoring specification for JESSIE 1.0.1 was a global substitution of the project attribute for all steps from jessie-1.0.1 to jessie-1.0.0. Table 5 compares the number of diff blocks for themselves before and after refactoring. The numbers is differed slightly because of the evolution changes to the variable Ciph.

On the contrary, even after we performed a global substitution of the project name, for the JSSE 1.6 case, we found that most of the steps cannot be applied as is. The doHandshake protocol is mainly implemented in the class SSLSocket of the JESSIE 1.0.1 library, whereas in the JSSE library implementation in the OpenJDK 1.6 (hereafter called JSSE 1.6), the protocol is mainly implemented in the class sun.security.ssl.HandshakeMessage. Nevertheless, the naming of the symbols can be traced to the implementation.

Table 6 lists some mapping from the symbols in Table 2 to their naming in the JSSE library. The difference to the earlier table for the JESSIE project is mainly in the second column, that is, the source of the refactoring steps given in the third column.

To reuse the existing refactoring steps, we have to instantiate their specifications with different parameters for

---

[4] https://openjdk.dev.java.net/svn/openjdk/jdk/trunk/j2se/src/share/classes/sun/security/ssl

Table 6. Traceability in JSSE, cf. Table 2

Symbols	JSSE 1.6	op.
1. C	HandshakeMessage.ClientHello	rename.type
2. S	HandshakeMessage.ServerHello	rename.type
3. P_{ver}	protocolVersion	extract.temp
4. R_C	clnt_random	rename.temp
R_S	svr_random	rename.temp
5. S_{id}	sessionId	rename.temp
6. Ciph[ ]	cipherSuites	extract.temp
7. Comp[ ]	compression_methods	extract.temp
8. Veri	CertificateVerify.verify()	rename.method
9. $D_{\text{notBefore}}$	cert.getNotBefore()	rename.method
D_{notAfter}	cert.getNotAfter()	rename.method

its source (i.e., project, folder, package, class) and its context (i.e., regexp, count). In some cases even the type of refactoring step needs to be changed. For example, Veri(X509Cert$_s$) is refactored by the *extract.method* step in JESSIE (Table 2, and by the *rename.method* step in JSSE (Table 6). Such changes do not influence the target name attribute for the steps because they are derived from the same protocol design.

As part of the library release, two model-based unit tests for the message sequences in JESSIE 1.0.1 were provided: testclient.java and testserver.java. After refactoring, we were able to reuse them for the two other implementation libraries as well.

Moreover, the model-based security aspect we implemented for JESSIE 1.0.1 can also be reused without change. When weaving in the security aspect, we could determine that it did not further harden the security for JSSEbeyond the existing implementation since the security check implemented in the aspect is already correctly enforced in JSSE. This is confirmed by the logs of the two test cases that were reused. These test cases also helped us to verify that the messages are sent and received in a way consistent with the message sequence chart (Fig. 5), on both sides of client and server, regardless of the implementation library.

5 Related work

Traceability and model synchronization Software maintenance makes use of related models at different stages of development. Example models are goal trees for requirements, UML diagrams for design and source code for implementation. When some model elements change, it is necessary to *synchronize* the change on related elements in order to maintain all models consistent [12]. Existing traceability approaches aim to recover traceability links that connect elements of certain software engineering artifacts in requirements, design and implementation [1, 8, 4, 13]. Search-based techniques recover traceability links between documents and code with a precision below 100% [1, 13]; a probability-model based approaches relies on a softgoal-interdependency graph to recover traceability links between functional and non-functional requirements [4]; a scenario-driven approach generates traceability links from observations of system executions [8]. Other work on requirements tracing includes [23]. In general, none of them can recover accurate requirements traceability links. Though efficient techniques have been proposed to account for incremental update of traceability links recovered from search-based approaches, these incrementally maintained traceability links are still as inaccurate [13]. Graph transformation-based techniques [12] may accurately trace structural semantics, yet another mechanism is required to trace behavioral semantics.

Reverse engineering Existing reverse engineering frameworks were proposed to improve accuracy of traceability for reference architecture [22] and for known design patterns [2]. In our previous work [25], refactoring were proposed to enable accurate abstraction of behavioral implementations such that they can be compared to the goal-oriented requirements. In this work, refactoring is not only used for comparing the source and target, but also for transforming the source into the target.

Refactoring scripts Dig et al [5] first studied the evolution of component API that can be replayed as refactoring steps. They argued that the refactoring of library components may indeed change the behavior of the overall system especially when the client of the components are not refactored accordingly. For example, a function 'foo' may be renamed to 'bar' in the library, yet the call site of the function may still try to invoke 'foo', only to find broken contracts. Therefore, it is useful to keep track of (or detect in Dig's case) the refactoring steps as a script such that they can be replayed at the client side. Our tool supports tracking refactoring steps by translating the refactoring steps recorded by the IDE into change resilient refactoring specifications. Comparing with [5]'s work, our use of refactoring is not for replaying the changes, rather for maintaining the traceability between design elements and implementation regardless of changes. Though the RefactorCrawler tool [5] cannot be used directly, we can make use of the refactoring preview dialog code in the MolhadoRef tool [6].

Refactoring for aspects In [11, 20], specialized refactoring actions are defined mainly for aspect-orientation. In this work, we expand the scope to any general-purpose refactoring steps supported by existing tools. We have exploited the opportunity to perform aspect-oriented instrumentation in order to harden the security that require general-purposed refactoring actions. In [3], Binkley et al proposed a number of aspect-aware refactoring transformations to convert object-oriented programs into aspect-oriented ones. If the design element is implemented by crosscutting code, then Binkley et al's technique may be applied to our work to maintain the traceability between such elements. Since refactoring alone does not change the behavior of the system, aspects derived from such refactoring transformations must not change the behavior. Consequently, they cannot improve the security of existing implementation. In our work, we employ AOP to instrument the code with additional functionality to enforce security hardening. Therefore our aspect is introduced by a different purpose.

Tracing and validating aspects In [7], Antonio Castaldo D'Ursi et al discussed the difficulty of static analysis where multiple aspects that potentially interfere with each other through intertype declarations. Such problems are well known in the AOP community as *aspect interference* problems. Our case study only introduced one security hardening aspect for vulnerability check, which certainly did not expose interference. Yet it is possible in general, if the software systems have used aspects, or more than one aspects were refactored from the legacy system (using e.g., Binkley's [3] methodology). In a separate work [24], we proposed a goal-based testing framework to trace and validate aspects according to their goal-oriented requirements. In the security and mission-critical domain, such test-based validation alone may not be adequate. It is thus an open research issue to investigate how aspect interference can be prevented. Our tracing framework presented here helps simplify the task by relating the scope of aspects to the associated requirement/design elements.

6 Conclusions

We showed that refactoring can be used to support the maintenance of accurate traceability links. In order to maintain such traceability resilient to changing design and implementation, we enhanced the Eclipse refactoring engine in an automated refactoring tool support. The traceability refactoring process, together with our UMLsec analysis tools, are integrated with other maintenance activities continuously. Supported by the derived traceability in test cases and aspects, a traceability-related vulnerability found in one implementation can be effectively verified in another. The proposed approach was applied to three implementations of SSL protocol (i.e., JESSIE1.0.1, 1.0.0 and JSSE1.6) and actually detected a security vulnerability in JESSIE1.0.1, which was further confirmed in JESSIE1.0.0, and rejected in JSSE1.6.

Acknowledgements: This work is partly funded by the Royal Society through an international joint project with TU Munich. The authors would like to thank H. Lin and C. Li for discussions on the Jessie project. Our ART tool uses the LTK refactoring dialog implemented in by Dig et al in [6].

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. de Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[2] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *TSE*, 31(2):137–149, 2005.

[3] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Trans. Software Eng.*, 32(9):698–717, 2006.

[4] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *ICSE'05*, pages 362–371. ACM, 2005.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[6] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE*, pages 427–436, 2007.

[7] A. C. D'Ursi, L. Cavallaro, and M. Monga. On bytecode slicing and aspectJ interferences. In *FOAL '07*, pages 35–43, New York, NY, USA, 2007. ACM.

[8] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Trans. on Software Engineering*, 9(2), 2003.

[9] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[10] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *RE*, pages 94–101. IEEE, 1994.

[11] J. Hannemann. *Role-Based Refactoring of Crosscutting Concerns*. PhD thesis, Univ. Brit. Col., 2005.

[12] I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *ICSM '04*, pages 252–261, 2004.

[13] H. Jiang, T. N. Nguyen, and I. Chen. Incremental latent semantic indexing for effective, automatic traceability link evolution management. In *ICSE'08*, 2008.

[14] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[15] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE*, pages 322–331. IEEE/ACM, 2005.

[16] J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *ASE*, pages 167–176. IEEE/ACM, 2006.

[17] J. Jürjens and Y. Yu. Tools for model-based security engineering: Models vs. code. In *ASE*. IEEE/ACM, 2007.

[18] D. Kirscheneder. Method for comparison of Java implementations and UML models. Technical report, TU Munich, 2006.

[19] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[20] R. Laddad. *Aspect Oriented Refactoring*. Addison-Wesley Professional, 2006.

[21] T. Mens and T. Tourwe. A survey of software refactoring. *TSE*, 30(2):126–139, 2004.

[22] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *TSE*, 27(4):364–380, 2001.

[23] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127, 2004.

[24] Y. Yu, N. Niu, B. Gonzlez-Baixauli, W. Candillon, J. Mylopoulos, S. Easterbrook, J. C. S. do Prado Leite, and G. Vanwormhoudt. Tracing and validating goal aspects. In *RE'07*, pages 53–56. IEEE, 2007.

[25] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse engineering goal models from legacy code. In *RE'05*, pages 363–372, 2005.