Capturing Returned Computable Values in AspectJ for Security Hardening

A. Boukhtouta  D. Alhadidi  M. Debbabi

Computer Security Laboratory (CSL)
Concordia Institute for Information Systems Engineering
Concordia University
Montreal, H3G 1M8, Quebec, Canada

Abstract

The main contribution of this paper is to present an extension to AspectJ compiler ajc-1.5.0 for security hardening. The extension consists of two pointcuts that can capture the returned computable values of methods in both the execution scope or following a method invocation. The returned values in programs are the results of operations done in the execution scope of methods. They are significant for the intra and inter procedural dataflow analysis where they represent the context transitivity between the caller and the called methods in a given program. Any misuse of them can allow the reflection of important data and the disclosure of secret information. Moreover, any alteration of these values can violate the integrity of programs and conduct to their misbehavior. In this paper, we detail the design and the implementation of these two pointcuts. Finally, we present a case study to demonstrate how the data integrity property can be satisfied in distributed systems using the newly implemented pointcuts.

Key words: Aspect-oriented Programming, Pointcut-Advice Model, AspectJ, Security Hardening, Data Integrity

1 Introduction

Application security hardening can be defined as any process, methodology, product, or combination thereof that is used to directly increase the security

1 This research is the result of a fruitful collaboration with the Department of National Defense, Bell Canada and the DND/NSERC Research Partnership Program.
of an existing software. Actually it is all about code modifications scattered across the whole software to fix security vulnerabilities. Accordingly, Aspect-oriented Programming (AOP) \[9\] appears to be a promising paradigm for security hardening. AOP allows implementing crosscutting concerns as separated modules to avoid tangled and scattered code. This facilitates software understanding, evolution and maintenance. AspectJ \[1\] is an aspect-oriented extension to the Java programming language. It has a beautiful technical design that is behind its success and its wide adoption by the AOP community. AspectJ comes with a new kind of modular unit, named aspect, which provides the developer with the ability to implement crosscutting concerns such as logging, synchronization, and persistence. The adopted model by AspectJ is the pointcut-advice model \[9\]. The fundamental concepts of this model are: join points, pointcuts, and advices. A join point is a fixed point in the execution of a program, e.g., a call of a constructor or a method, a reference to a field, etc. A pointcut is a concept that classifies join points in the same way a type classifies values. An advice is a code fragment executed when join points satisfying a particular pointcut are reached. This execution can be done before, after, or around a specific join point.

AspectJ appears to be a promising language for security hardening \[11\] because it has the ability to inject code in designated points that are located in the flow of a program. Accordingly, a security engineer can target susceptible points in programs without necessarily digging into the logic of them. This would contribute to the systemization of security hardening and alleviate the job of security developers in making changes to programs. In addition, AspectJ provides a great set of pointcuts that are both appropriate and relevant from a security hardening perspective. Nevertheless, it has not been engineered with security in mind and accordingly it lacks some important primitives from a security perspective. There are some contributions that suggest new pointcuts for security purposes such as the one of Masuhara and Kawauchi \[12\] and the one of Laverdière et al.\[11\]. In addition, Alhadidi et al. \[3\] have highlighted the need of security dedicated primitives that are important to express common security hardening practices. Among the needed primitives are those that capture computable values. This is exactly what we are addressing in this paper, i.e. extending AspectJ so as to capture the returned computable values in Java programs. The proposal is to integrate two new primitives in AspectJ, namely \textit{returns} and \textit{returned} pointcuts.

The output of a method call is known as the returned computable value. The returned values in programs are the results of operations done in the execution scope of a method. They are important since they reflect the computations done in methods for different purposes. They are also significant for the intra and inter procedural dataflow analysis \[12\] where they represent the context transitivity between the caller and the called method in a given program. Any misuse of them can allow the reflection of important data and the disclosure
of secret information. Moreover, any alteration of these values can violate the integrity of a program and conduct to its misbehavior. The need of capturing and monitoring the computed returned values during the execution of a program is essential to ensure its integrity, secrecy and privacy. This idea is the main motivation behind integrating \textit{returns} and \textit{returned} primitives in AspectJ. These pointcuts allow tracking the computable returned values of methods inside Java applications. Extending AspectJ with \textit{returns} and \textit{returned} pointcuts allows writing crosscutting concerns to enforce data integrity, secrecy and privacy, especially in distributed programs where the data transits through a network easily affected by a potential smart attacker.

The rest of the paper is organized as follows. In Section 2, an overview of the current literature is presented on the subjects that are related to the contribution of this paper: AOP and AspectJ compiler \textit{ajc}. The design and the implementation for the new pointcuts are detailed in Section 4. We illustrate in Section 5 a case study related to data integrity to demonstrate the importance of the new pointcuts. The paper discusses related work in Section 6. Concluding remarks as well as a discussion of future work are represented in Section 7.

2 Background

We present in the sequel an overview of the current literature on the subjects that are related to the contribution of this paper.

2.1 Aspect-oriented Programming

AOP depends on the principle of "Separation of Concerns", where issues that crosscut the application are addressed separately and encapsulated within aspects. There are many AOP languages that have been developed. We distinguish from them AspectJ [1] built on top of the Java programming language, AspectC [5] built on top of the C programming language, AspectC++ [15] built on top of the C++ programming language, AspectC# [10] built on top of the C# programming language, and the AOP version addressed for Smalltalk programming language [4]. AspectJ and AspectC++ are the most prominent AOP languages. The approach adopted by most of these languages, is called the pointcut-advice model. The fundamental concepts of this model are: join points, pointcuts, and advices.

A join point is a specific point in the execution of a program. A pointcut is a concept that classifies join points in the same way a type classifies values.
public aspect AuthorizationAspect
{
  pointcut access(): within(Account) && execution(* Withdraw*(..));

  before(): access()
  {
    System.out.println("Checking security for account access");
    boolean hasAccess = false;
    // hasAccess = CheckAccessRights();
    if (hasAccess)
    {
      // continue with withdrawal
    }
    else
    {
      System.out.println("Access not permitted");
      // throw (new SecurityException)
    }
  }
}

Fig. 1. Authorization Aspect Example

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>call(void Foo.m(int))</td>
<td>Picks out the join points where the method void Foo.m(int) is called.</td>
</tr>
<tr>
<td>execution( !public Foo.new(...) )</td>
<td>Picks out the join points at the execution of any non-public constructor of Foo.</td>
</tr>
<tr>
<td>set(!private <em>.Point.</em>)</td>
<td>Picks out the join points at which any non-private field of Point is assigned.</td>
</tr>
<tr>
<td>cflow(call (void Figure.move()))</td>
<td>Picks out the join points in the control flow of each call to void Figure.move().</td>
</tr>
<tr>
<td>args(*,int)</td>
<td>Picks out any join point where there are two arguments, the second of which is an integer.</td>
</tr>
<tr>
<td>within(com.bigboxco.*)</td>
<td>Picks out any join point where the associated code is defined in the package com.bigboxco.</td>
</tr>
</tbody>
</table>

Table 1
AspectJ Pointcut Examples

The pointcut expressions typically allow to pick out function calls, function executions, join points on the control flow ulterior to a given join point, etc. An advice is a code fragment executed when join points satisfying a particular pointcut are reached. This execution can be done before, after, or around a specific join point. At the heart of this model, is the concept of aspect, which embodies all these elements. Fig. 1 shows an authorization aspect written in
AspectJ where the pointcut access picks out any execution of any method whose name begins with Withdraw and it is within the Account class. A before advice is used to check access rights. The weaving is implemented statically in AspectJ by inserting the advice functionality in certain regions of a program that correspond to the join points matched by the advice pointcut.

AspectJ has a comprehensive and expressive pointcut specification language that allows specifying particular points in the control flow of the program where advices are to be applied. Some of them are presented in Table. 1. Finally, the aspect is composed and merged with the core functionality modules into one single program. This process of merging and composition is called weaving, and the tools that perform such process are called weavers.

### 3 High-level Description of AspectJ Compiler ajc

To understand the design and the implementation methodology of the proposed pointcuts, a high-level picture of AspectJ compiler ajc is presented in this section. It is composed, as appears in Fig. 2, of the following three primary modules:

* **Package org.aspectj.ajdt.core**: This is the compiler front-end that extends the eclipse Java compiler from org.eclipse.jdt.core. It compiles both AspectJ and pure Java source code into pure Java bytecode annotated.

---

**Fig. 2. Aspect.J Compiler Architecture**

- *Package org.aspectj.ajdt.core*: This is the compiler front-end that extends the eclipse Java compiler from org.eclipse.jdt.core. It compiles both AspectJ and pure Java source code into pure Java bytecode annotated.
with additional attributes representing any non-java forms such as advice and pointcut declarations.

- **Weaver**: This is the compiler back-end that provides the bytecode weaving functionality. It implements the transformations encoded in the attributes that represent any non-java forms as advice and pointcut declarations to produce woven class files.
- **Runtime**: These are the classes that are used by the generated code at runtime and must be redistributed with any system built using AspectJ.

To compile a pointcut, it should go through the phases of its life cycle which are:

- **Parsing**: It corresponds to build an abstract syntax tree where its nodes represent different declarations and designators provided in AspectJ syntax. Once nodes are set by the front-end AspectJ compiler, they can be formatted as patterns. A pattern, as appears in Fig. 3, is a syntactic representation that holds information about types, fields, methods, pointcuts, advices and aspects.

```
Pointcut Designator Node:
call(* Test.test(..))
```

<table>
<thead>
<tr>
<th>Result: KindedPointcut (Kind, Signature)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind: Method-call Joinpoint</td>
</tr>
<tr>
<td>Signature:</td>
</tr>
<tr>
<td>Member: Method</td>
</tr>
<tr>
<td>Modifiers: 0</td>
</tr>
<tr>
<td>Return Type: Any</td>
</tr>
<tr>
<td>Declaring Type: Test</td>
</tr>
<tr>
<td>Name: test</td>
</tr>
<tr>
<td>Parameters Type: ellipsis</td>
</tr>
<tr>
<td>Exceptions: Any</td>
</tr>
<tr>
<td>Annotations: Any</td>
</tr>
</tbody>
</table>

Fig. 3. Pointcut Parsing Example

- **Resolution**: It corresponds to the association of a parameter declared in an advice to an identifier enclosed by a pointcut.
- **Serialization**: The defined pointcuts are serialized into attributes in the class file.
• **Deserialization**: The weaver unpacks serialized attributes when a class file is loaded by reading input data stream and recording declared aspects. Once aspects are added, the crosscutting members are extracted as shadow mungers. A shadow munger is a representation of declared advices that enclose pointcut designators where each one has its own signature.

• **Concretizing**: The weaver concretizes a shadow munger by replacing its named pointcuts by their declarations. Declared pointcuts that are held in shadow mungers can be either named or anonymous. A named pointcut has a name and can be referred to by another pointcut. An anonymous pointcut has no name and cannot be referred to.

These phases are followed by weaving. The weaving concept is implemented on bytecode [2] rather than on the Java source code. The weaving process matches join points against defined shadow mungers. Every join point has a corresponding static shadow in the bytecode of the program. Shadows are either a single bytecode statement or a bounded region of bytecode. The matching is based on fuzzy boolean operations. A fuzzy boolean can take one of the following values: maybe, never, no, or yes. This kind of boolean is used since a matching decision cannot be taken directly in the case of some pointcuts, i.e., `args` and `cflow` pointcuts. Static or dynamic tests are needed in such cases. Finally, if a join point matches a shadow munger, advice code is inserted at these join points to modify the dynamic behavior of a program.

For illustration purposes, let us see how the weaving modifies the bytecode instructions for the function `f` in the example presented in Fig. 4. The aspect `A` has one before advice that prints the word `aspect` when its pointcut is matched. The pointcut is matched when the join point is either a method execution or a field set. In the method `f`, there are two join points that match the corresponding pointcut: the execution of `f` and the update of the field `a`. Hence, AspectJ weaver injects the advice at the beginning of `f` and before the bytecode instruction `putfield #2` which represents a field set join point. Accordingly, The instructions `invokestatic #26` and `invokevirtual #29` are injected at these places. The instruction `invokestatic #26` calls a static method named `aspectOf`, which is automatically generated by the front-end compiler once the aspect is compiled. It allows obtaining an instance of the aspect, which will serve as a receiver for the advice call. In the case of this example, the instance serves as a receiver for the instruction `invokevirtual #29` that corresponds to the advice method call.

In general, AspectJ has three types of pointcuts. Kinded pointcuts match directly a granular bytecode instruction or a set of bytecode instructions. For instance, a call pointcut matches an invoke bytecode instruction and an execution pointcut matches a bounded region of bytecode instructions in a method or a constructor execution. Scope matching pointcuts target a set of join points within a certain scope in the program. There are two kinds
Fig. 4. Weaving Process

of scopes: a static scope and a dynamic scope. A static scope is a syntactic location in a program such as a class or a package. The dynamic scope is a location in the control flow of a method call or a method execution. The aim of such pointcuts is to restrict join point location lookup inside a program. The pointcuts that belong to this class are within, withincode and cflow. Finally context matching pointcuts focus on providing contextual information such as object values during the runtime. These pointcuts are generally used in conjunction with kinded pointcuts. The pointcuts that belong to this class are: args, target and this.
4 New Pointcuts Design and Implementation

We have fully designed and implemented \textit{returns} and \textit{returned} pointcuts as extensions to \texttt{ajc-1.5.0} compiler of AspectJ. These pair of pointcuts can capture the returned computable values of methods. Such pointcuts can be appropriate primitives for integrating data concerns in Java code. The \textit{returned} pointcut is classified as a kinded pointcut whereas the \textit{returns} pointcut is classified as a context matching pointcut. In the sequel, we present a detailed description for the design and the implementation of these pointcuts.

4.1 Returned Pointcut

The \textit{returned} pointcut is a primitive that allows capturing the return bytecode instruction (\texttt{ireturn}, \texttt{areturn}, etc.) during the execution of methods. It has the following general form \texttt{returned(MethodPattern)} where \texttt{MethodPattern} represents a method or a set of methods. Given a method signature \texttt{ms} to a method \texttt{m} defined in a class \texttt{c} and a join point \texttt{jp}, which is in the scope of the method \texttt{m}. The join point \texttt{jp} matches the pointcut \texttt{returned(mp)} if and only if the method pattern \texttt{mp} matches the method signature \texttt{ms} and the join point \texttt{jp} corresponds to a return bytecode instruction.

The extension of the \texttt{ajc} compiler to support the \textit{returned} pointcut requires the following steps:

- Extend the pointcut parser to handle the new primitive \texttt{returned(mp)}.
- Create a new kind of join point for the return bytecode instructions.
- Implement a new join point shadow (RETURN).
- Extend the matching process to take into consideration the new join point for the new pointcut. The matching compares the method signature for the method that contains the created shadow against the method pattern enclosed by the \textit{returned} pointcut.

\begin{verbatim}
aspect example {
  before() : returned(* ..(..)) {
    System.out.println("matched");
  }
}
\end{verbatim}

Fig. 5. Returned Pointcut

For example in Fig. 5, we pick out all the return join points and print out a message before them using the \textit{returned} pointcut. The same \textit{returned} pointcut in Fig. 5 can be used with an after advice but in this case the weaving is done
differently. It corresponds to collect all possible return bytecode instructions inside all matched methods and insert a branch bytecode instruction, which points to a new block containing the advice instructions. Afterwards a new return instruction that corresponds to a new exit point of is added. Fig. 6 shows how the weaving is done in this case.

Fig. 6. After Advice with Returned pointcut

4.2 Returns Pointcut

The returns pointcut has brought forward by Masuhara et al. [12], where it is used as an entry to a dataflow pointcut to retrieve the returned value of a given method. It has the following general form `returns(Identifier)` where `Identifier` represents a variable name. Given a method `m` defined in a class `c`, an identifier `id`, and a join point `jp`. The join point `jp` matches the pointcut `returns(id)` if and only if the join point `jp` corresponds to a return or a call bytecode instruction and the type of the identifier `id` can be cast to the return type of the method `m`. If the join point `jp` corresponds to a call bytecode instruction, the method `m` is the method called inside this call bytecode instruction. On the other hand, if the join point `jp` corresponds to a return bytecode instruction, the method `m` is the method that contains this return bytecode instruction.

The implementation of the returns pointcut takes after the args pointcut implementation since this pointcut exposes a state. A question that can flash in potential readers’ mind is why we have not extended the functionality of the args pointcut to capture the returned value. A possible answer resides in its expressiveness since we are aiming to retrieve a returned value and we do not target retrieving a method argument or a field. The pointcut `returns` encloses an identifier that can be bound to a returned value, which represents
an advice local variable. As all implemented pointcuts, the \textit{returns} pointcut has a life cycle similar to the other known pointcuts as previously mentioned in Section 3. Since this pointcut is classified as a contextual matching pointcut, its implementation corresponds mainly to testing the context type. Hence, we add an additional static test, which checks the return type of methods. If the return type can match the type bound to the identifier enclosed by the \textit{returns} pointcut, a new temporal local variable is initialized and used to store the returned value. The \textit{returns} pointcut is designed in a way that can be used in conjunction with \textit{returned} and \textit{call} pointcuts. However, in the case of \textit{call} pointcuts, it should not be used in a before advice because a returned value cannot be retrieved before calling a method. In Fig. 7, the first advice allows matching the return bytecode instruction inside the method \texttt{test} and binding the returned value to the variable \texttt{i}. The second advice allows retrieving the returned value after calling the method \texttt{test} and binding it to the variable \texttt{j}.

![Fig. 7. Returns Pointcut](image)

AspectJ provides a special reference variable, \texttt{thisJoinPoint}, that contains reflective information about the current join point for the advice to use. This variable can only be used in the context of an advice. It includes a rich reflective class hierarchy of signatures, and can be used to access both static and dynamic information about join points. We have implemented a new method \texttt{getReturns} and added it to the available methods on the \texttt{thisJoinPoint} object to acquire the returned value. Fig. 8 shows how the returned values of all methods are collected in a list with the help of this method.

![Fig. 8. Reflection](image)
4.3 Comparison

In AspectJ, there are another ways to capture returned values of methods. First, an after returning advice allows capturing the returned value of a join point. For example, by declaring the advice \texttt{after returning(int s): call(int \ast \ast .foo(\ldots))}, we can capture the returned value after calling the method \texttt{foo} and store it in the variable \texttt{s}. Second, an around advice with \texttt{proceed} can also capture the returned value of a method. However, these ways have some shortcomings, which are detailed next.

- The after returning advice allows capturing a returned value of a method in an advice parameter. Advices cannot be named like pointcuts. Pointcuts are syntactic identities that provide flexibility since they can be named and called in different advices’ declarations. Accordingly, developers can reuse a declared pointcut in different places but this is cannot be done with a defined advice. To clarify this issue, let us consider the declarations in Fig. 9. It is clear from the example that the pointcut \texttt{intcalls} can be reused within different advices’ declarations. On the other hand, the \texttt{after returning} advice does not provide such a feature.

```
aspect example
{
  1: after returning(int s): call(int \ast \ast (\ldots))
  {\ldots}
  2: pointcut intcalls(int s): call(int \ast \ast (\ldots))\&\& returns(s)
     after(int ret): intcalls(ret)
  {\ldots}
}
```

Fig. 9. After Returning Advice versus Returns Pointcut

- Another way to capture the returned value of a method is by using an around advice with \texttt{proceed}. Around advice in AspectJ uses the special form \texttt{proceed} to continue with the normal flow of execution at the corresponding join point. This issue is clarified in the example presented in Fig. 10. Using the \texttt{returns} pointcut is more powerful than using around advice with \texttt{proceed} since the \texttt{proceed} form is very costly. The around advice with \texttt{proceed} calls a copy of the matched method. Such a behavior needs to weave a new method member into an aspect and calling it within an advice. On the other hand, the returned value is linked directly to an advice argument using the \texttt{returns} pointcut. This can be done easily by loading the returned value into the execution stack and storing it in the argument.
5 Data Integrity Concern

One of the properties that should be enforced in applications is data integrity, which is a basic issue in security hardening for distributed programs. Distributed programming techniques have emerged especially in web services and distributed computing. Data integrity refers to validity of data, which can be compromised through a malicious or an accidental altering. In this section, we present a data integrity service for distributed systems using the newly implemented pointcuts. These pointcuts $returned$ and $returns$ together with the $call$ pointcut compromise the main building block of the data integrity service.

In common Java distributed programming systems such as Remote Method Invocation (RMI) and Common Object Request Broker Architecture CORBA, a client calls functionalities implemented in a server. The server answers by sending returned values to the client. Providing the correct data to the client is important especially when it is related to e-business applications or critical computing operations. Hence, an improvement to distributed systems in terms of data integrity is needed. This can be done easily by the $returns$ and the $returned$ pointcuts. To clarify the idea, consider the model that appears in Fig. 11. In this model, we consider a standard server which implements a set of remote methods that can be invoked at any time by clients. Once a client invokes a method implemented in the server, the server runs the method. Before returning the value to the client, it creates a hash for it, which is recorded in a hash repository implemented in the data integrity service. The server then sends the value to the client. After receiving the value, the client creates a hash for the received value and sends the hash to the data integrity service, which acknowledges the hash. If the acknowledgement is positive, the client carries on the execution of its program. Otherwise it throws an exception. In order to understand the model, we present an RMI example that illustrates the data integrity model. The proposed example takes into consideration an interaction between a client and a server. However, it is possible to interpolate this example to a couple of clients by adding threads to handle concurrency issues. The example corresponds to an authentication between a client and a
The code that implements the data integrity service appears in Fig. 12. The aim behind the exposure of this example is to clarify the defined model. Some good programming practices have been ignored, e.g., considering access modifiers due to the large size of code.

Fig. 11. Data Integrity Service Model

The integrity service consists of three sub-components: IntegrityService, IntegrityServiceImpl and Repository. The interface IntegrityService specifies methods that can be invoked remotely. The class IntegrityServiceImpl is the implementation of the methods that are specified in the interface IntegrityService. The class Repository represents a recipient of hash values sent by the server and the client. It has two important methods Record and Compare that allow respectively setting the hash sent by the server and comparing the hash value sent by the client with the server hash value.

The code presented in Fig. 13 illustrates the test code in the server side, which implements the authentication service. The test code lies in three sub-components: Authentication, AuthenticationImpl and Record. The interface Authentication specifies a method that can be invoked remotely for authentication service. The class AuthenticationImpl is the implementation of the interface Authentication, where the authentication is done by accessing a database and checking whether the password is correct or not. The aspect Record captures the returned value before returning it to the caller, which is the client. The main part of the aspect Record is the set of actions taken in the advice body (lines 32 to 36). The returned value is bound to the advice local variable i, which is hashed. For sake of simplicity, we consider a simple hash method but in real world applications more sophisticated hash algorithms can be adopted. A repository is created and registered in the integrity service remotely (lines 33 to 35). The hash value is recorded in the repository through the integrity service by invoking remotely the Record method. Consequently, the corresponding aspect allows registering the hash of the value supposed to be returned to the caller in the integrity System. The integrity system has a
knowledge about what was sent by the server.

The code in the client simulates a login process done by a user. The client invokes remotely the method authenticate, which is implemented in the server side. An aspect Receive captures the returned value sent by the server. The code in Fig. 14 illustrates the authentication done in the client side. The code is composed of two sub-components: client and Receive. The class client simulates a login done by a user. The method login invokes the method authenticate that is implemented in the authentication server. The aspect Receive permits us to retrieve the returned value and binds it to the variable j. The hash value is computed using the same hash method implemented in the server. The client invokes the integrity service (line 15). By invoking the method Compare (line 16), the client checks whether the hash value computed in the server and stored in the integrity service is equal to the hash value computed in the client. If the checking is positive, the client carries on the execution of the program. Otherwise it throws an exception. As we can see, the checking is done in the client side. However, we can adopt an
interface Authentication {
  Boolean authenticate(String username, String passwd);
}

class AuthenticationImpl extends UnicastRemoteObject implements Authentication {
  Boolean authenticate(String user, String passwd)
  {
    try{
      String url = "jdbc:mysql://localhost:1114/Demo";
      Connection c = DriverManager.getConnection(url, "", "");
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select Password From Users where UserName=\""+user+\"\"");
      While(rs.next()){
        String pass = rs.getString("Password");
        if(pass.equals(passwd)){
          conn.close();
          return true;
        }
      }
      conn.close();
      return false;
    }
    catch (Exception e){
      e.printStackTrace();
    }
  }
}

aspect Record {
  before(Boolean i):
  returned(Boolean AuthenticationImpl.
              authenticate(String , String))
  & returns(i) {
    long k = Hash(i);
    Repository rep =new Repository("serverhost",
                                "clienthost");
    IntegrityService IS =(IntegrityService)Naming.
                          Lookup("IS");
    IS.setRepository(rep);
    IS.Record(k);
  }
}

Fig. 13. Serve-side Code

acknowledgment process between the integrity service and the client using a secure communication between the two entities. An additional implementation can be done in the client and the data integrity service.

In the defined model, two important assumptions have been taken into consideration: the existence of a potential strong attacker and the reliability of the hash algorithms used to hash exchanged data. Regarding the first assumption, the implementation of the proposed model makes the task of such attacker dif-


Fig. 14. Client-side Code

ficult since this attacker does not have a knowledge about the hidden running data integrity service. The second assumption is important since the unreliability of hash algorithms can mislead the data integrity service. However, the occurrence of such event tends to have a null probability because reliability is considered a security property in hash algorithms.

6 Related Work

Most of the contributions that explore the usability of AOP for injecting security code into applications are presented as case studies that show the relevance of AOP languages for application security. We present in the following an overview on these contributions.

By means of an example of access control, De Win et al. [17] have investigated how well AOP can deal with the separation of security concerns from an application. In order to construct a more generic solution, they have suggested to abstract relevant pointcuts out of the aspect implementation. Besides, the feasibility to build a security aspect framework has been discussed. Viega et al. [16] have developed a general-purpose aspect-oriented extension to the C programming language that can be used to separate security policies from the code. The authors have pointed out that building in security into an application is preferable to the "penetrate-and-patch" approach where problems are addressed in an ad-hoc manner when flaws emerge. The aspect language specifies structured transformations that are inserted in the code at well-defined points. Points of interest for such insertion could be calls to functions, function definitions, and pieces of functions.
A case study has been described [14] by Ramachandran and his colleagues to incorporate multilevel security system (MLS) using aspects. A MLS has two goals: First, it is intended to prevent unauthorized personnel from accessing information at a classification they are not authorized to. Second, it is intended to prevent personnel from changing the classification of information they do have access to. AspectJ is used to intercept Java library calls in order to enforce MLS policy. The authors have described how AspectJ can actually go further than conventional object-oriented approach to achieve stronger enforcement of MLS. Huang et al. have introduced an implementation of a reusable and generic aspect library [7]. This aspect library is based on AspectJ and common Java security packages. It contains the following typical categories of security aspects: encryption, decryption, authentication, authorization, and security audit.

Masuhara and Kawauchi [12] have defined a dataflow pointcut for security purposes. The pointcut identifies join points based on the origins of values. Cross-site scripting (XSS) problem in web-applications is an example presented by them to clarify the need for such a pointcut. Local variables set and get pointcuts [3] have been introduced to protect the privacy and integrity of sensitive data. They allow to track the values of local variables inside a method. Many security hardening practices require the injection of code around a set of join points or possible execution paths. Examples of such cases would be the injection of security library initialization/deinitialization, privilege change, and logging. M-A. Laverdière et al. [11] have proposed new pointcuts that enable the identification of a common join point to a set of other join points where code can be injected efficiently once for all of them.

Kaul et al. [8] have introduced a distributed storage management system. They have used AspectJ to implement crosscutting concerns such as as persistence control, database connection pooling, authorization, exception handling and logging in distributed storage systems. The contribution have investigated adding crosscutting modules for metadata management for large scale distributed data storage. The authors have proved the usefulness of using AspectJ in distributed systems since it diminishes the amount of tangled and scattered code. Although the authors have defined a distributed storage management system, they have assumed that AspectJ provides enough constructs to achieve concerns related to security issue. Yonezawa et al. [6] have proposed a finer grained join point model, and an experimental AOP language named PitJ. In their alternative join point model, a join point represents a point-in-time, or an instant when an action in execution begins or ends. For example, a call join point represents a point-in-time at the beginning of invocation. The point-in-time join point model additionally defines reception join point as a point-in-time at returning from a method. Such join points give developers the ability to target different points in the program without using AspectJ-like advice declarations, i.e., before, after and around. This is an interesting
approach since it allows reusing pointcut declarations. However, it would be more interesting to fit the proposed join point model to AspectJ, which represents the most popular language used by the AOP community. Nishizawa et al. [13] have presented DJcutter, an extension to AspectJ, where they have implemented a remote pointcut [13]. The remote pointcut is a construct that identifies join points in the execution of a program running in remote hosts. The contribution mainly has focused on highlighting the usefulness of the remote pointcut in terms of performance and has not elaborated on security issues in distributed systems.

7 Conclusion

This paper have presented the design and the implementation of two pointcuts returned and returns as an extension to AspectJ compiler ajc. The returned pointcut identifies the return bytecode instruction located in the execution scope of a given method. The returns pointcut binds the returned value to an identifier defined in an advice. We have illustrated the usefulness of these pointcuts through a case study that addresses a data integrity service that checks whether the data exchanged in a distributed system is not corrupted. The service extends the existing RMI calls. However, it can be used with other Java systems such as CORBA. As a future work, we are intending to propose, design and implement more security hardening constructs as extensions to AspectJ and other AOP code production compilers.

References


