Dataflow Pointcut for Integrity Concerns

Kazunori Kawauchi Hidehiko Masuhara
Graduate School of Arts and Sciences, University of Tokyo
Tokyo 153–8902 Japan
{kazu,masuhara}@graco.c.u-tokyo.ac.jp

Abstract

Some security concerns, such as secrecy and integrity, are sensitive to flow of information in a program execution. We proposed a new pointcut to aspect-oriented programming (AOP) languages in order to easily implement such security concerns as aspects. The pointcut identifies join points based on the origins of values, and can be used with the other kinds of pointcuts in existing AOP languages. This paper presents an example how the pointcut can be applied to an integrity concern in a web-application.

1 Introduction

Techniques to build secure software systems are crucial for rapid development of network-based software systems in open network environments. Many techniques have been proposed such as runtime inspection, program verification, and encryption. For example, a program verification technique can identify problems that potentially leak secret information to untrusted parties[9].

Application of those security techniques to a software system often becomes crosscutting concerns as security is usually related to several participants to the system, such as a resource to be protected, and trusted/untrusted third-party program and data. Aspect-oriented programming would be useful programming technique to modularize those security concerns. In fact, there have been several studies that apply AOP to modularize access control[3] and that propose an AOP language for supporting authentication[13]. However, as far as the authors' knowledge there have been few studies to support the flow of information in AOP languages although it is a crucial concept for ensuring secrecy and integrity of software systems[9].

Based on the above observation, we proposed a mechanism that supports the flow of information in an AOP language[8]. The mechanism offers a new pointcut primitive into AspectJ-like AOP languages in order to concisely determine the flow of information in aspect definitions. The mechanism is being implemented as an extension to AspectJ language. This paper presents, by taking an example security concern, how a security concern crosscuts web-applications and how our new pointcut primitive enables modularization of the concern. Detailed description of the new pointcut primitive and their implementation issues can be found in the other literature[8].

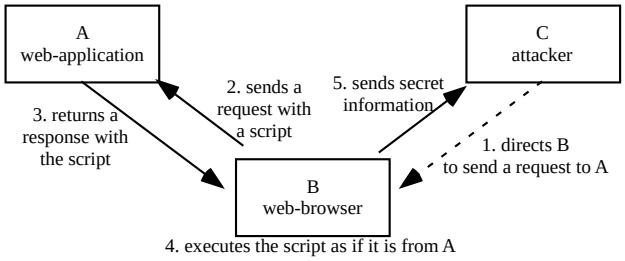

Figure 1: An Exploit of a Cross-site Scripting Problem

We propose a new kind of pointcut, called `dflow` pointcut, that identifies join points based on the origins of data. It is designed as an extension to AspectJ's pointcut language; `dflow` pointcuts can be used in conjunction with the other kinds of pointcuts in AspectJ. This makes it easy for the programmers to adopt `dflow` with minimal efforts.

The rest of the paper is organized as follows. Section 2 gives an example problem. Section 3 presents the design of the dataflow-based pointcut, and how it can solve the problem. Section 4 discusses related work. Section 5 summarizes the paper.

2 Example: Security Problem in Web-Applications

2.1 Cross-site Scripting Problem

Cross-site scripting is a security problem in web-applications. By exploiting a flaw of a web-application, an attacker can reveal secret information from the browser of the web-application's client[1]. The following scenario with three principals, namely (A) a web-application, (B) a web-browser of a client of A, and (C) an attacker, explains an attack (Fig. 1).

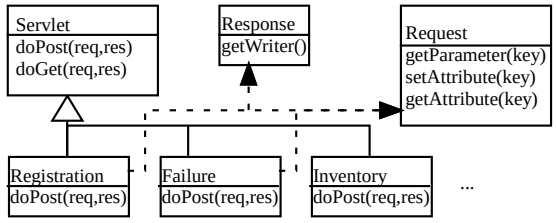

Figure 2: Structure of a Web Application

The problem could have been avoided if A did not return the malicious script as a part of the response to B. A solution to this problem on the A's side is not to generate pages by using a string that comes from any untrusted principal. A simple implementation is to remove special characters that constitute scripts from strings that come from untrusted principals, and to replace them with non-special (or quoted) characters. This solution is usually called *sanitizing.*

We here define a following sanitizing task for a web-application:

> Assume that the web-application is to generate a web page[1]. When a string that originates from anyone untrusted by the web-application appears in the generated page, it replaces special characters in the string with quoted ones.

2.2 How Sanitizing Crosscuts a Web Application

In the web-applications that dynamically generates many pages, the sanitizing can be a crosscutting concern because its implementation could involve with many parts of the program. Here, we assume a web-application on Java servlets framework, which defines a class for each kind of pages. Fig. 2 shows a class structure of the web-application. Each `Servlet` subclass represents a particular kind of pages. When a client browser sends a request for a URL, the framework automatically creates an instance of a respective `Servlet` subclass, and then invokes `doPost` (or `doGet`, etc.) with objects representing the request and a response. The method can read values to the input fields from the request object, and generates a new page by writing into a stream in the response object. Alternatively, a method can delegate another method to generate a page after performing some process.

Fig. 3 shows definitions of some of the `Servlet` subclasses. When `doPost` method of `Registration` class is invoked, it merely checks whether the requested ID is found in the database, and transfer itself to either `Inventory` page or `Failure` page. In `Failure` class, it displays a message of failure with the requested ID. It also places a link to the previous page by reading information in the attributes of the request.

The sanitizing task is to wrap the call to `getParameter` method in `Failure` class with a method that replaces special characters. Although the task needs to change only one place in the application, it crosscuts the application because the similar modifications need to be done in many sibling classes when those classes also use the results from `getParameter` for responding.

2.3 Usefulness of Dataflow

Since the sanitizing task crosscuts, it looks a good idea to implement it as aspects. However, it sometimes is not easy to do so with existing pointcut-and-advice mechanisms because they do not offer pointcuts to address dataflow, which is the primary factor in the sanitizing task.

The problem can be illustrated by examining the (incomplete) aspect definition in Fig. 4. The `respondClientString` pointcut is supposed to intercept any join point that prints an unauthorized string to a client. (By unauthorized, we mean that a string is created from one of client's input parameters.) With properly defined `respondClientString`, the task of sanitizing is merely to replace all special characters in the unauthorized string, and then continue the intercepted join point with the replaced string[2].

With existing kinds of pointcuts, it is not possible to write an appropriate `respondClientString` in a declarative manner.

- First, a straightforward definition is not appropriate. For example, the following pointcut will intercept strings that are to be printed as a part of a response to a client. However, it will intercept strings including 'authorized' ones:

```
pointcut respondClientString(String s) :
 call(* PrintWriter.print*(String))
 && args(s) && within(Servlet+);
```

- Second, even if one could write an appropriate pointcut definition, the definition is less declarative. For example, an appropriate `respondClientString` could be defined with auxiliary advice declarations that detect and trace unauthorized strings. However, those advice declarations are not declarative

[1] Another approach is to replace special characters when the web-application receives strings from browsers. It is not recommended because the replacement might affect the intermediate process unexpectedly. Also, it can not sanitize strings that come to the application via methods other than HTTP[2].

[2] In AspectJ, `proceed` is a special form to do so. When the formal parameters of the advice (i.e., `s`) is bound by `args` pointcut (e.g., `args(s)`), the arguments to the `proceed` (i.e., `quote(s)`) replace the original arguments in the continued execution.

```
class Registration extends Servlet {
  void doPost(Request req, Response res) {
    String id = req.getParameter("ID");       //read input field
    if (<id is found in the database>)
      <transfer to an Inventory page>
    else {
      req.setAttribute("PREV", req.getURL()); //store current URL as an
      <transfer to a Failure page>            //originating address of the
    }                                         //transferred page
  }
}

class Failure extends Servlet {
  void doPost(Request req, Response res) {
    PrintWriter out = res.getWriter();
    out.println("<HTML>...Login failed for: ");
    out.println(req.getParameter("ID"));      //read input field & print
    out.println("...<a href=");
    out.println(req.getAttribute("PREV"));    //back to the originating page
    out.println(">go back</a>...");           //by using the stored address
  }
}
```

Figure 3: Implementation of `Servlet` Subclasses
(Type names are abbreviated due to space restrictions.)

because they have to monitor every operations that involve with (possibly) unauthorized strings.

In order to define `respondClientString` in a declarative manner, a pointcut that can identify join points based on the origins, or *dataflow*, of values is useful. The next two reasons support this claim.

First, dataflow can use non-local information for determining the points of sanitizing. For example, the result of `getAttribute` in `Failure` class in Fig. 3 needs no sanitizing because if we trace the dataflow, it turns out to be originally obtained by `getURL` (i.e., not by `getParameter`) in `Registration` class.

Second, dataflow can capture derived values. For example, assume that a web-application takes a parameter string out from a client's request, and creates a new string by appending some prefix to the string, the new string will also be treated as well as the original parameter string.

3 Dataflow Pointcut

We propose a new kind of pointcut based on dataflow, namely `dflow`. This section first presents its syntax and example usage for the simplest case, and then explains additional constructs for more complicated cases.

3.1 Pointcut Definition

The following syntax defines a pointcut p:

$$\begin{aligned} p &::= \texttt{call}(s) \mid \texttt{args}(x,x,\ldots) \mid p\texttt{\&\&}p \mid p\texttt{||}p \\ &::= \texttt{dflow}[x,x](p) \mid \texttt{returns}(x) \end{aligned}$$

where s ranges over method signature patterns and x ranges over variables.

The second line defines new pointcuts. $\texttt{dflow}[x,x'](p)$ matches if there is a dataflow from x' to x. Variable x should be bound to a value in the current join point. (Therefore, `dflow` must be used in conjunction with some other pointcut, such as $\texttt{args}(x)$, that binds x to a value in the current join point.) Variable x' should be bound to a value in a past join point matching to p. (Therefore, p must have a sub-pointcut that binds a value to x'.) $\texttt{returns}(x)$ is similar to `args`, but binds a return value from the join point to variable x. This is intended to be used only in the body of `dflow`, as a return value is not yet available when a current join point is created.

By using `dflow`, the pointcut for the sanitizing task can be defined as follows:

```
pointcut respondClientString(String o) :
  call(* PrintWriter.print*(String))
  && args(o) && within(Servlet+)
  && dflow[o,i](
      call(String Request.getParameter(String))
      && returns(i));
```

```
aspect Sanitizing {
  pointcut respondClientString(String s) : ...; // incomplete

  Object around(String s) : respondClientString(s) {
    return proceed(quote(s));           //continue with sanitized s
  }

  String quote(String s) {
    return <replace all special characters in s>;
  }
}
```

Figure 4: (Incomplete) Aspect for the Sanitizing Task

The second line is not changed from the one in Section 2.3, which matches calls to print methods in `Servlet` subclasses, and binds the parameter string to variable `o`. The `dflow` pointcut restricts the join points to such ones that the parameter string originates from a return value of `getParameter` in a past join point.

3.2 Dataflow Relation

The condition how `dflow` pointcut identifies join points can be elaborated as follows: assume x is bound to a value in the current join point, `dflow[`x`,`x'`](`p`)` matches the join point if there exists a past join point that matches p, and the value of x originates from a value bound to x' in the past join point. By originating from a value, we mean the value is used for deriving an interested value. For example, when two strings are concatinated, the concatinated string originates from those two strings. We let the originating-from relation be transitive; e.g., the origins of the two strings are considered as the origins of the concatinated string as well.

The originating-from relation is defined as follows. Let v and w are two values. v originates from w when

- v and w are the identical value, or
- v is a result of a primitive computation using $u_1, \ldots, u_n$, and u_i originates from w for some i ($1 \leq i \leq n$).

The above definition is sufficient for languages only with primitive values. When a language also has compound values, such as arrays and objects, we need to extend the matching condition for `dflow`. Although it needs further study to give a feasible condition, we tentatively extended the definition in the following ways. `dflow[`x`,`x'`](`p`)` matches when the value of x or a value reachable from the value of x originates from the value of x' or a value reachable from the value of x'.

3.3 Excluding Condition

We also defined an extended syntax of `dflow` for excluding particular dataflows. We call the mechanism `bypassing`.

A motivation of `bypassing` can be explained in terms of the sanitizing task. Assume a `Servlet` subclass that manually quotes the client's inputs:

```
class ShippingConfirmation extends Servlet {
  void doPost(Request req, Response res) {
    PrintWriter out = res.getWriter();
    String address =
      quote(req.getParameter("ADDR"));
    out.print("...Please confirm address:");
    out.print(address);
    ...
  }
}
```

When an object of this class runs with `Sanitizing` aspect after filling its pointcut definition with the one in Section 3.1, the aspect intercepts method calls that print `address` in `ShippingConfirmation`. As `address` has already quoted string, it doubly applies `quote` to the quoted string.

The following extended `dflow` syntax excludes dataflows that go through certain join points:

$$p ::= \texttt{dflow[}x\texttt{,}x\texttt{](}p\texttt{)}\ \texttt{bypassing[}x\texttt{](}p\texttt{)}$$

Intuitively, a `bypassing` clause specifies join points that should not appear along with a dataflow. By using `bypassing`, the following pointcut avoids intercepting quoted strings:

```
pointcut respondClientString(String o) :
  call(* PrintWriter.print*(String))
  && args(o) && within(Servlet+)
  && dflow[o,i](
       call(String Request.getParameter(String))
       && returns(i))
     bypassing[q](call(String *.quote(String))
                  && returns(q));
```

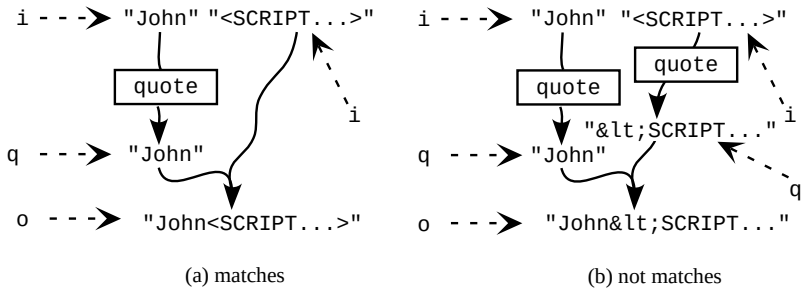

Figure 5: How dataflow is restricted in `dflow[o,i](...)  bypassing[q](...)`.

The `bypassing` clause requires that `o` (an argument to print method) to originate from `i` (a return value from `getParameter`) but not through `q` (a return value from `quote`) after `i`.

Precisely, `bypassing` requires existence of at least one dataflow that does not go through join points matching to the pointcut in the `bypassing` clause. Fig.5 illustrates computations that generate concatenated strings from two strings. Assume that the original strings at the top of the figure are the results of `getParameter` in the above example. Then the `dflow` pointcut with `bypassing` clause matches the computation (a) because there is a dataflow to the string at the bottom of the figure without going through `quote`. On the other hand, it does not match the computation (b) because all dataflows go through `quote`; i.e., there are no dataflows bypassing `quote`.

The semantics of `bypassing` clause can be defined by slightly extending the originating-from relation. Let v and w are two values. v originates from w bypassing x in p, when:

- there have been no such a join point that matches p and the value bound to x is identical to v, and
- either of the following conditions holds:
  - v and w are the identical value, or
  - v is a result of a primitive computation using $u_1, \ldots, u_n$, and u_i originates from w bypassing x in p for some i $(1 \leq i \leq n)$.

3.4 Explicit Dataflow Propagation

We provide an additional declaration form that specifies explicit propagation of dataflow through executions in external programs. This is useful in an open-ended environment, where a program runs with external code whose source programs are not available (e.g., a class library distributed in a binary format).

A declaration is written as a member of an aspect in the following form:

`declare propagate:` p `from` x`,`x`,... to` x`,`x`,...;`

where p and x range over pointcuts and variables. The form requests that, when a join point matching to p is executed, it will regard that the values of the `to`-variables originate from the values of the `from`-variables.

For example, assume that a program uses `update` method of `Cipher` class for encryption (or decryption), but the system has no access to the source code of the class. With the following declaration, the system will regard that the return value from `Cipher.update` originates from its argument. As a result, if a string matches `dflow` pointcut, the `Cipher` encrypted string of the string also matches to the `dflow` pointcut.

```
aspect PropagateOverEncryption {
  declare propagate:
    call(byte[] Cipher.update(byte[]))
    && args(in) && returns(out)
    from in to out;
}
```

The propagate declarations are designed to be reusable; i.e., once someone defined propagate declarations for a library, the users of the library merely need to import those declarations to track dataflow over the library.

The propagate declarations would be sufficient for the libraries that have only dataflows between input and output parameters. Coping with more complicated cases, such as the ones involving with structured data or the ones with conditional dataflows, is left for further study.

4 Related Work

There are systems that can examine dataflow in a program either in a static or dynamic manner (e.g., Confined Types[10] and taint-checks in Perl[14]). Those are useful for *checking* security enforcement. On the other hand, when a breach of the security enforcement is found by those systems, the programmer may have to fix many modules in the program without AOP support.

Information flow analyses (e.g., [11]) can detect a secret that can leak by indirect means, such as the conditional context and timing. For example, the following

code does not have direct dataflow from `b` to `x`, but information about `b` indirectly leaks in `x`:

```
if (b) { x = 0; } else { x = 1; }
```

As we have shortly discussed, our dataflow definition only deals with direct information flow. It does not regard a dataflow from `b` to `x`. Extending dataflow definition to include such indirect information flow, is left for future study.

Giving more expressiveness to pointcuts in AOP languages are studied in many ways. Some offer pointcuts that can examine calling context[5], execution history[12], and static structures of a program[4].

Demeter is an AOP system that can declaratively specify traversals over object graphs[6, 7]. It allows to examine relation between objects, but the relation is about a structure of data in a snapshot of an execution.

5 Conclusion

We presented `dflow` pointcut in aspect-oriented programming (AOP) languages and its application to sanitizing aspect for web-applications. Since the pointcut identifies join points based on the dataflow of values, it enables the programmers to write more robust pointcuts in aspects that are sensitive to information flow.

Although the pointcut primarily aims at security concerns, we believe that its applications are not limited to such. Our plan is to apply the pointcut to many programs, such as the other kinds of security concerns.

The design space of the `dflow` pointcut is large enough for further study. Especially, to find a right balance between the declarativeness of the pointcut and the runtime efficiency is crucially important. It will also be crucially important to give a formal framework of `dflow` pointcut so that we can reason about completeness of the semantics.

References

[1] CERT. Malicious HTML tags embedded in client web requests. Advisory Report CA-2000-02, CERT, Feb. 2000.

[2] CERT Coordination Center. Understanding malicious content mitigation for web developers. Tech tips, CERT, Feb. 2000.

[3] B. De Win, B. Vanhaute, and B. De Decker. Security through aspect-oriented programming. In B. De Decker, F. Piessens, J. Smits, and F. Van Herreweghen, editors, *Advances in Network and Distributed Systems Security*, volume 206 of *IFIP Conf. Proc.*, pages 125–138. Kluwer Academic Publishers, 2001.

[4] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD2003)*, pages 60–69. ACM Press, 2003.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[6] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Comm. ACM*, 44(10):39–41, Oct. 2001.

[7] K. J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[8] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In A. Ohori, editor, *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121, Beijing, China. (or Kanazawa, Japan or Nanjing, China.), Nov. 2003.

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

[10] J. Vitek and B. Bokowski. Confined types. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA99)*, pages 82–96. ACM Press, 1999.

[11] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[12] R. J. Walker and G. C. Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of the eighth international symposium on Foundations of software engineering for twenty-first century applications (FSE-8)*, volume 25(6) of *ACM SIGSOFT Software Engineering Notes*, pages 69–78, San Diego, California, USA, Nov. 2000.

[13] R. J. Walker and G. C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

[14] L. Wall and R. Schwartz. *Programming Perl*. O'Reilly and Associates, 1991.