dataflow pointcut in aspect-oriented programming

Hidehiko Masuhara
(University of Tokyo)

joint work with Kazunori Kawauchi
background: aspect-oriented programming (AOP)

• a paradigm for modularizing crosscutting concerns
e.g. security, optimizations, debugging

• **aspects**: modules of additional behavior / structure

• **pointcuts**: specifies “where”
motivation & approach

• existing AOP languages can not straightforwardly support some security concerns
  – key factor: flow of information

• our approach: extend AspectJ [Kiczales+2001] to directly support dataflow
  – by adding a new pointcut
a crosscutting concern: XSS prevention

cross-site scripting (XSS):
– an attack to steal secrets from a browser
– by exploiting a problem in a web app.

**copying request string into response**

prevention, a.k.a. **sanitizing**: quote such a string when used in a response
a crosscutting concern: XSS prevention wrt Servlet

sanitizing
replace out.print(s) with
out.print(quote(s)) when
– generating responses, and
– the arguments come from the browser’s request

• sanitizing is crosscutting: web apps. usually generate pages in many classes

prints to be replaced with
how AspectJ can implement sanitizing(1)

- a pointcut specifies “where to sanitize”
i.e., when PrintWriter’s print method is called from a subclass of Servlet
- one module for many classes

aspect Sanitizing {
  String around(String s) :
    call(* PrintWriter.print*(String))
    && within(Servlet+) && args(s){
      return proceed(quote(s));
    }
} ... }
how AspectJ can \_\_\_not\_\_\_ implement sanitizing(2)

• a modified aspect would
  – remember all strings got from a request
  – quote only remembered string at printing

```java
aspect Sanitizing {
  Set reqStrs = ...;
  after() returning (String s) : call(* Request.get(String)) {
    reqStrs.add(s);
  }
  String around(String s) :
    call(* PrintWriter.print*(String)) && within(Servlet+) && args(s) && if(reqStrs.contains(s)) {
      return proceed(quote(s));
    }
  }
}```
how AspectJ can implement sanitizing (2)

• a modified aspect would
  – remember all strings got from a request
  – quote only remembered string at printing

• **but not perfect!**
  
  can not detect
  
  “derived” strings;
  e.g., concatenated string

  ```java
  title = req.get("title");
  name = req.get("name");
  attn = title + " " + name;
  ...
  out.println(attn);
  ```

  not a string in request!
observations on sanitizing & AspectJ

- requires a pointcut to judge the conditions key: *flow of information*

whereas

- AspectJ pointcut can only reason about
  - instant properties (e.g., meth. name), and
  - flow of control (e.g., cflow)

replace out.print(s) with out.print(quote(s)) when
  - generating responses, and
  - the argument *comes from* the browser’s request
our proposal: a new dataflow-based pointcut \texttt{dflow}

issues:

- design
  - combination with the other pointcuts
  - clear semantics
  - exceptional cases for practical use
    - to exclude some dataflows
    - to handle “native” code
- efficient implementation
sanitizing with `dflow`

```
replace out.print(s) with out.print(quote(s)) when
  – generating responses, and
  – the argument comes from the browser’s request

aspect Sanitizing {
  String around (String s) :
    call(void print(String)) && args(s) &&
    dflow[ s, userinput ]
    ( call(String Request.get()) && returns(userinput) ){
      proceed(quote(s));
    } ... }
```

• declarative
• can be combined with the other pointcuts
how dflow works (1)

checks how values are generated wrt sub-pointcut

```java
@sweptSanitizing {
  String around (String s) {
    call(void print(String)) && args(s)
    && dflow[ s, userinput ]
    ( call(String Request.get())
    && returns(userinput) )
    proceed(quote(s));
  }
}
```
how \texttt{dflow} works (2)

it tracks back
\begin{itemize}
\item to "indirect" origins
\item across modules
\end{itemize}

```java
pect Sanitizing {
  String around (String s) :
    call(void print(String)) && args(s)
    && dflow[s, userinput]
    ( call(String Request.get())
      && returns(userinput) )
    proceed(quote(s));
  }
```
semantics of dflow

dflow[dst, src](p) matches jp if ∃jp’ in the past s.t.
  • jp’ matches p
  • p binds a val. in jp’ to src
  • dst originates from src

*jp (join point):
a point in execution
auxiliary constructs

- **bypassing** to exclude some dataflow
  e.g., avoid quoting strings twice

- **propagate** to manually connect dataflow through “native” code
implementations of dflow

• a proof-of-concept implementation
  – interpretive;
  based on Aspect SandBox interpreter
  [Wand+02,Masuhara+03]

• a practical implementation
  – under development
  – translator to plain AspectJ language
an interpreter-based implementation

• base implementation: AspectJ-like AOP
  – for each dynamic operation (e.g., method call) to be performed,
    • match each jp against pointcuts
    • run advice body when match
• extension for dflow
  – associate a tag to a value when the value originates from a dflow-specified value
how tags are manipulated

1. gives a tag to dflow (preprocess)
2. tags a value
3. propagates tags
4. checks tags
towards translation-based implementation

• basic strategy: 
  *insert code for tag operations*
  into target program
  – most operations can be done by AspectJ
  – finer-grained translation for propagation

• static analysis would eliminate operations for statically-known dataflow
discussion

• languages with information-flow analysis [JFlow, SLam, taint-Perl, etc.]
  – more than “dataflow” (cf. implicit flow)
  – not for modularization

• AOP w/ finer-grained jps [Walker+03]
  – could implement dataflow-tracking aspects
  – but less declarative,
    and less opportunity to optimize
conclusion

• **dflow**: dataflow-based pointcut
  – useful to declaratively modularize some security concerns
    • would be useful in more general situations
cf. cflow in AspectJ discriminates polymorphic code by calling context
  – smoothly integrated with existing AOP lang.
  – needs further semantics/implementation work for practical use
why not sanitize when it gets a request?

• because quoting depends on the destination at input web app.

on the destination at output web app.
can **dflow** track dataflow in the database?

- no,
- but it helps manual tracking by writing an aspect that
  - flags when it writes a record containing a request string
  - regards a flagged record as a request string
is it useful to concerns other than securities?

- cflow specifies execution context
- dflow specifies value context
if web app. concatenates request-string with valid HTML?

• need to modify the pointcut to capture such concat. operation instead of print operation

• only need to modify the pointcut