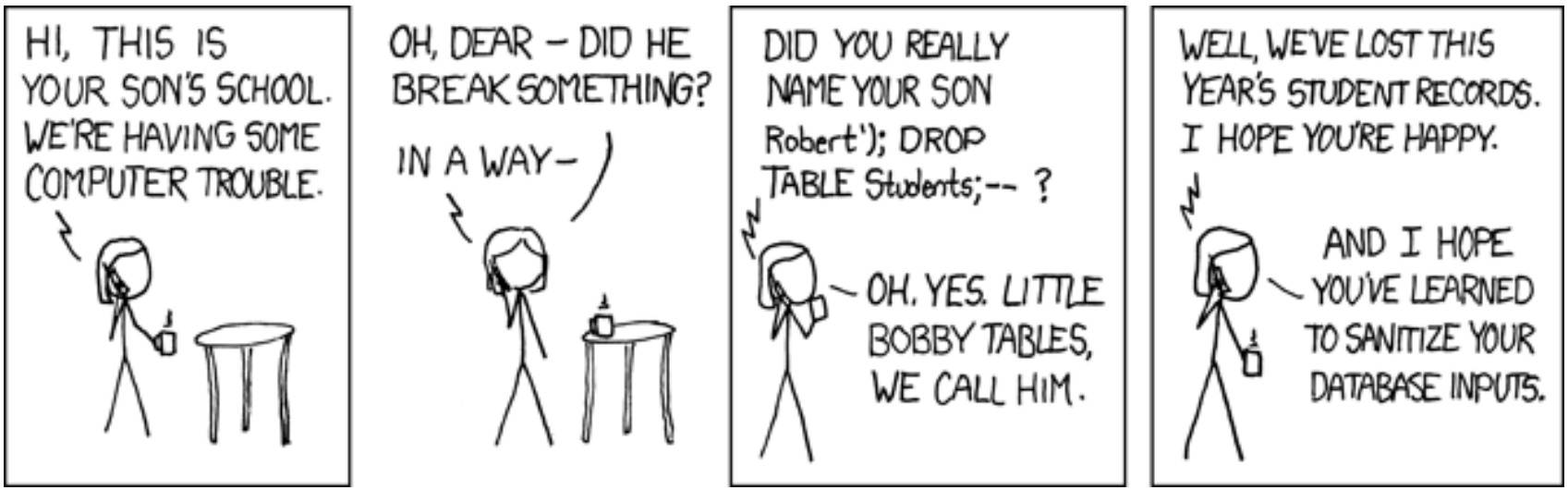
Type-based Prevention of Code Injection Attacks

Robert Grabowski

joint work with Martin Hofmann (LMU München)
and Keqin Li (SAP Research Sophia-Antipolis)

Oberseminar, TCS
December 3, 2010

Problem: code injection




```
executeSQL("INSERT INTO Students VALUES ('"+name+"');");

executeSQL("INSERT INTO Students VALUES ('Robert'); DROP TABLE Students; --');");
```

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed

very common vulnerability, comes in many forms:

- SQL injection

```
"INSERT INTO t VALUES ('" + name + "');"
```

- Cross-site scripting (XSS)

```
"<script> alert('" + msg + "'); </script>"
```

- include injection

```
<?php include($filename); ?>
```

- ...

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed

very common vulnerability, comes in many forms:

- SQL injection

```
"INSERT INTO t VALUES (''); DROP TABLE t; -- ');"
```

- Cross-site scripting (XSS)

```
"<script> alert('" + msg + "'); </script>"
```

- include injection

```
<?php include($filename); ?>
```

- ...

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed

very common vulnerability, comes in many forms:

- SQL injection

```
"INSERT INTO t VALUES (''); DROP TABLE t; -- ');"
```

- Cross-site scripting (XSS)

```
"<script> alert(''); form.submit('http://...'); </script>"
```

- include injection

```
<?php include($filename); ?>
```

- ...

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed

very common vulnerability, comes in many forms:

- SQL injection

```
"INSERT INTO t VALUES (''); DROP TABLE t; -- ');"
```

- Cross-site scripting (XSS)

```
"<script> alert(''); form.submit('http://...'); </script>"
```

- include injection

```
<?php include("http://evil.com/malicious.php"); ?>
```

- ...

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed

very common vulnerability, comes in many forms:

- SQL injection

```
"INSERT INTO t VALUES (''); DROP TABLE t; -- ');"
```

- Cross-site scripting (XSS)

```
"<script> alert(''); form.submit('http://...'); </script>"
```

- include injection

```
<?php include("http://evil.com/malicious.php"); ?>
```

- ...

Countermeasures

runtime protection

- PHP, Perl: "tainted" tag on untrusted strings
- JavaScript: browser refuses to execute cross-domain code

→ very interpreter-dependent

static program analysis

- dataflow analysis: calculate precise string values
- information flow analysis: determine origin of strings

→ may be hard to understand, too restrictive

most common in practice: programming guideline

- easy to understand "best practice" for safe programming

Countermeasures

runtime protection

- PHP, Perl: “tainted” tag on untrusted strings
- JavaScript: browser refuses to execute cross-domain code

→ very interpreter-dependent

static program analysis

- dataflow analysis: calculate precise string values
- information flow analysis: determine origin of strings

→ may be hard to understand, too restrictive

most common in practice: programming guideline

- easy to understand “best practice” for safe programming

Countermeasure: Best Practices

example: generate HTML page with data from GET request:

```
String name = httpRequest.getParameter("name");
output("<script> alert('" + name + "'); </script>");
```

`name` may close ' ' quotes of `alert`, followed by malicious code

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or "best practice" for the programmer:

  always use escape function on untrusted strings that are embedded in the output

Countermeasure: Best Practices

example: generate HTML page with data from GET request:

```
String name = httpRequest.getParameter("name");
output("<script> alert('" +escapeToJs(name)+ "'); </script>");
```

`name` may close ' ' quotes of `alert`, followed by malicious code

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or "best practice" for the programmer:

  always use escape function on untrusted strings that are embedded in the output

Countermeasure: Best Practices

example: generate HTML page with data from GET request:

```
String name = httpRequest.getParameter("name");
output("<script> alert('" +escapeToJs(name)+ "'); </script>");
```

`name` may close ' ' quotes of `alert`, followed by malicious code

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or "best practice" for the programmer:

  always use escape function on untrusted strings that are embedded in the output

obvious problem: programmer is responsible for following guideline

Goal: type-based enforcement of guidelines

We want to provide a type system that ensures that a Java programmer follows a given programming guideline .

Goal: type-based enforcement of guidelines

We want to provide a type system that ensures that a Java programmer follows a given programming guideline .

- types: familiar to programmers; extension of class types

Goal: type-based enforcement of guidelines

We want to provide a type system that ensures that a Java programmer follows a given programming guideline .

- types: familiar to programmers; extension of class types
- working tool: analysis of real Java code

Goal: type-based enforcement of guidelines

We want to provide a type system that ensures that a Java programmer follows a given programming guideline.

- types: familiar to programmers; extension of class types
- working tool: analysis of real Java code
- not a goal:
  evaluate guideline with respect to security properties

Overview

1. Guideline formalization

2. String analysis
   - Core language: FJEU with strings
   - Type and effect system
   - Inference and demo

3. Improving precision of the analysis

Concrete application scenario: Java servelet

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);

  // case 2: JavaScript embedding
  if (showAlert) {
    output("<script>");
    output("  alert('" + escapeToJs(input) + "');");
    output("</script>");
  }
}
```

ensure use of escape methods, depending on embedding position

Concrete application scenario: Java servelet

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);

  // case 2: JavaScript embedding
  if (showAlert) {
    output("<script>");
    output("  alert('" + escapeToJs(input) + "');");
    output("</script>");
  }
}
```

ensure use of escape methods, depending on embedding position

- consider only immutable `String` objects
- classify strings according to origin and contents

Concrete application scenario: Java servelet

Input: untrusted

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);

  // case 2: JavaScript embedding
  if (showAlert) {
    output("<script>");
    output("  alert('" + escapeToJs(input) + "');");
    output("</script>");
  }
}
```

ensure use of escape methods, depending on embedding position

- consider only immutable `String` objects
- classify strings according to origin and contents

Concrete application scenario: Java servelet

Input: untrusted

C1: sanitized for HTML

Literal: trusted

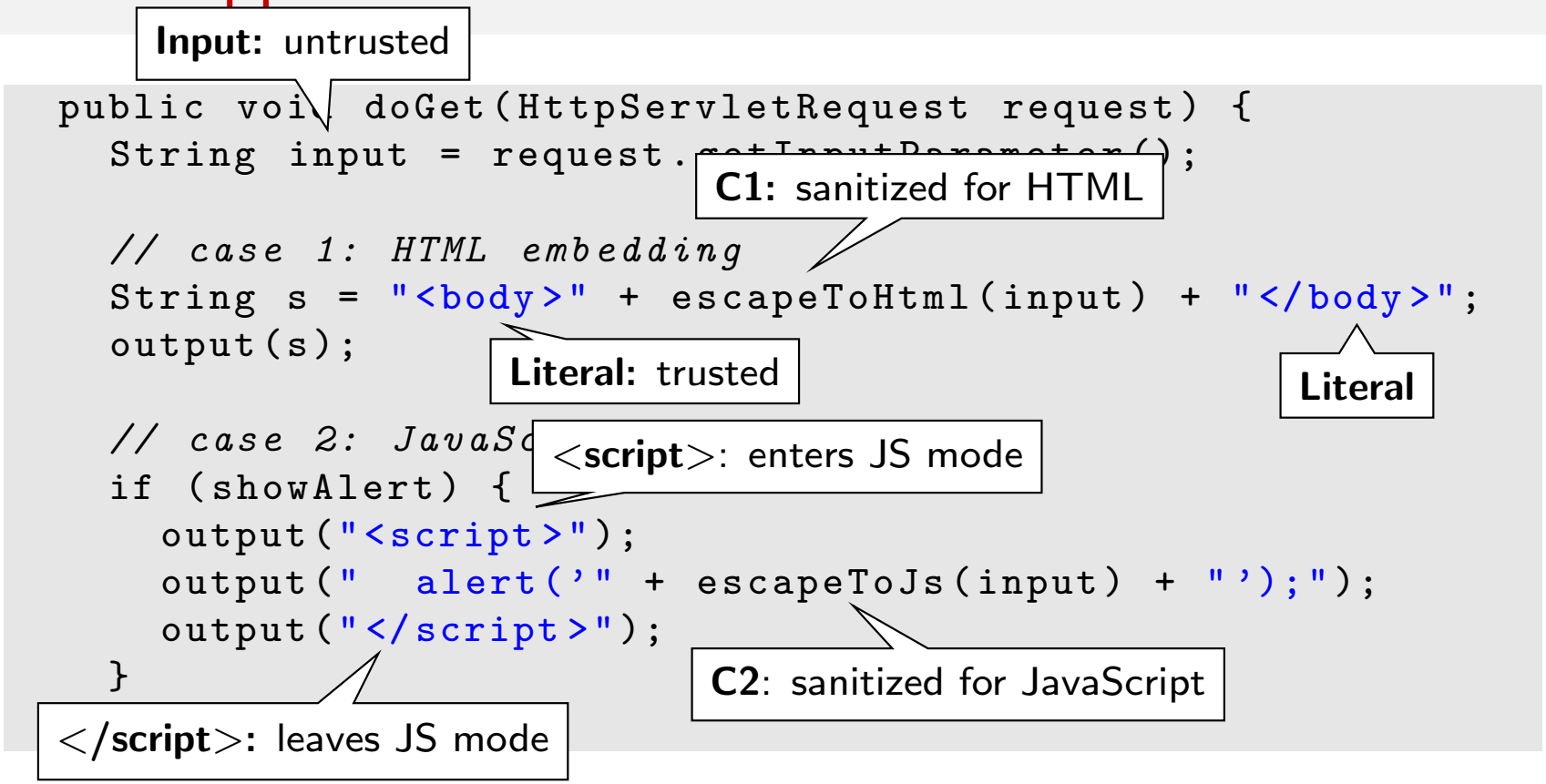
Literal

```
public void doGet(HttpServletRequest request) {
  String input = request.getInputParameter();

  // case 1: HTML embedding
  String s = "<body>" + escapeToHtml(input) + "</body>";
  output(s);

  // case 2: JavaScript embedding
  if (showAlert) {
    output("<script>");
    output("  alert('" + escapeToJs(input) + "');");
    output("</script>");
  }
}
```

ensure use of escape methods, depending on embedding position

- consider only immutable `String` objects
- classify strings according to origin and contents

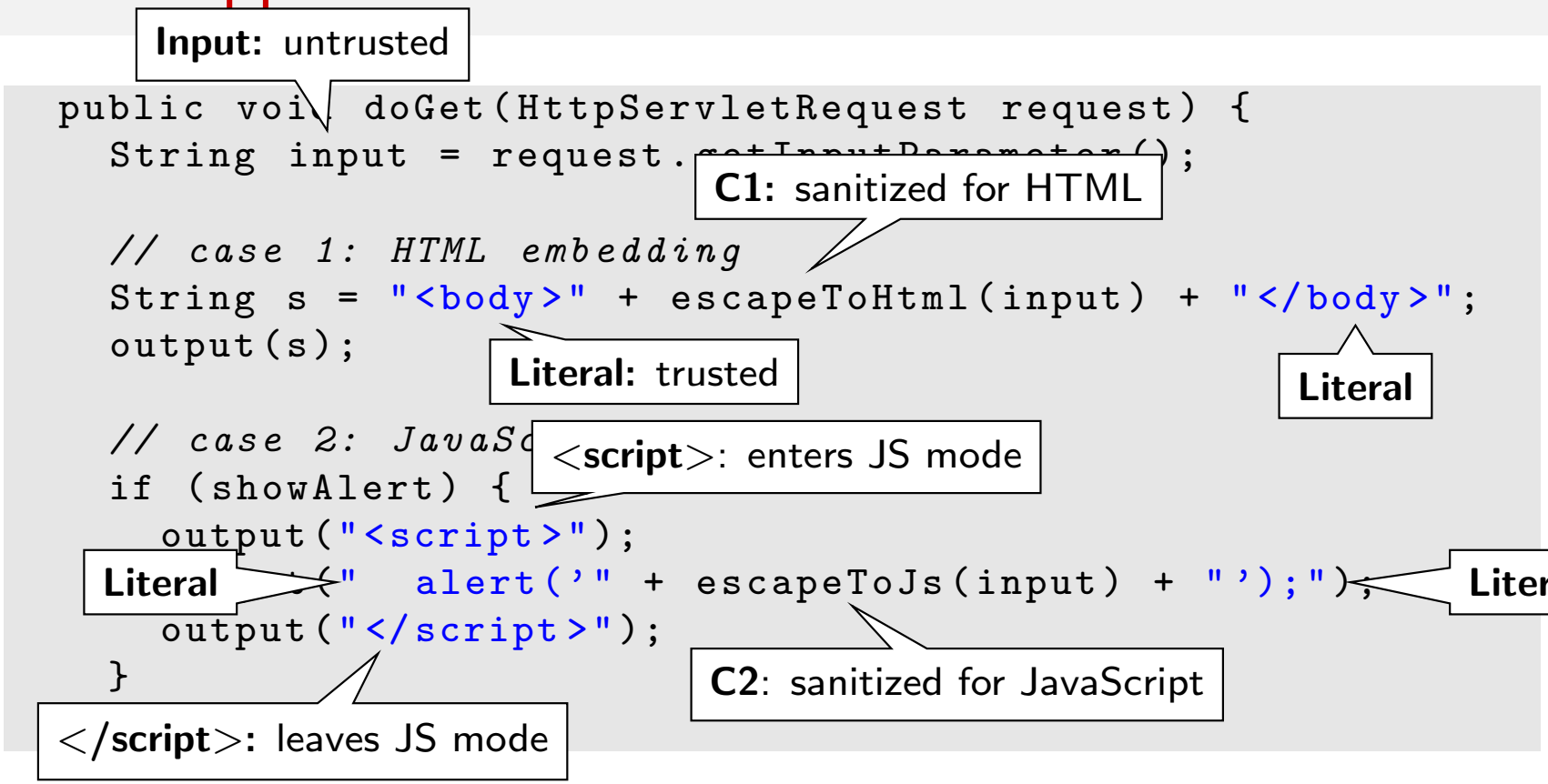
Concrete application scenario: Java servelet



ensure use of escape methods, depending on embedding position

- consider only immutable `String` objects
- classify strings according to origin and contents

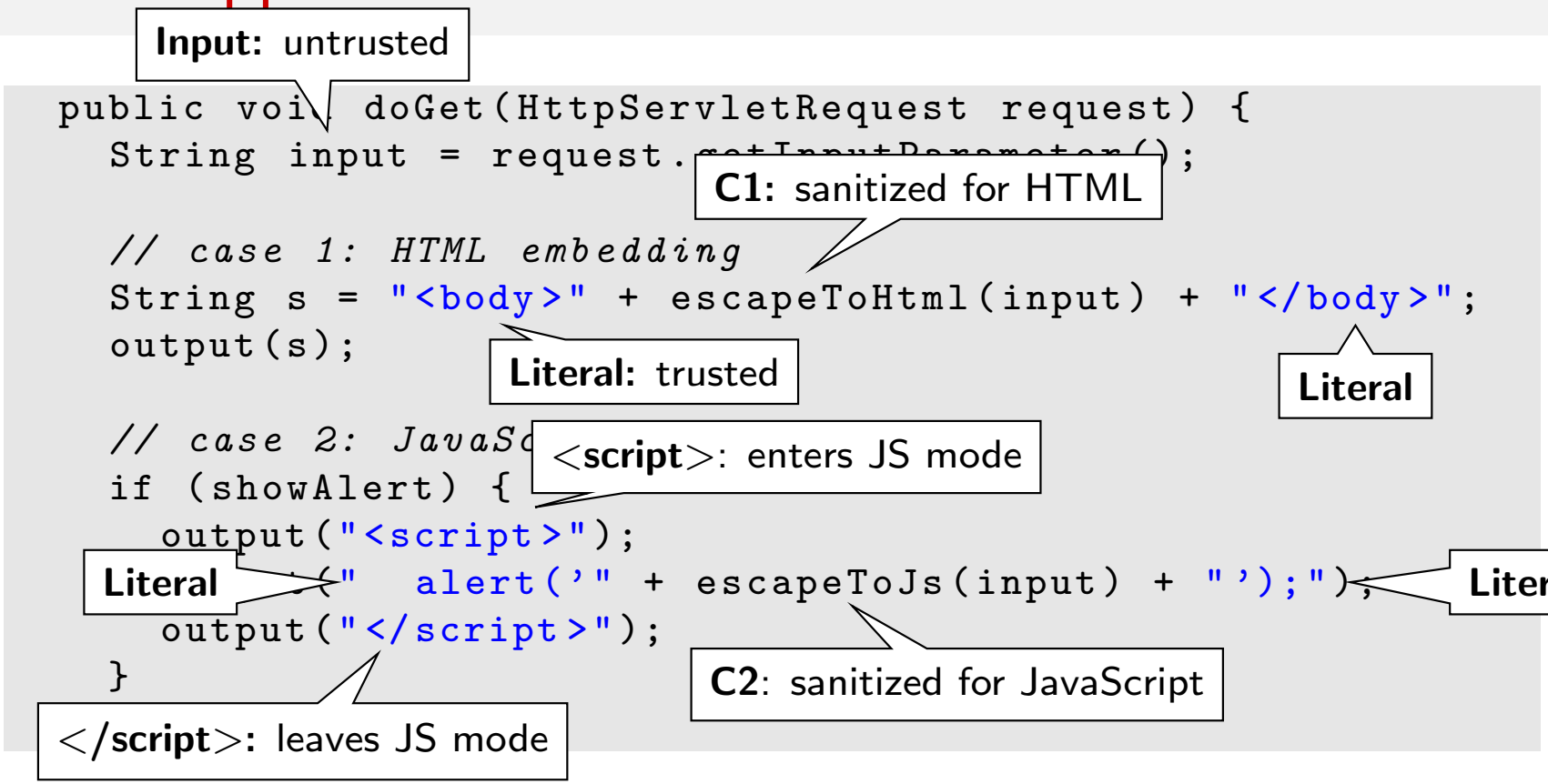
Concrete application scenario: Java servelet



ensure use of escape methods, depending on embedding position

- consider only immutable `String` objects
- classify strings according to origin and contents

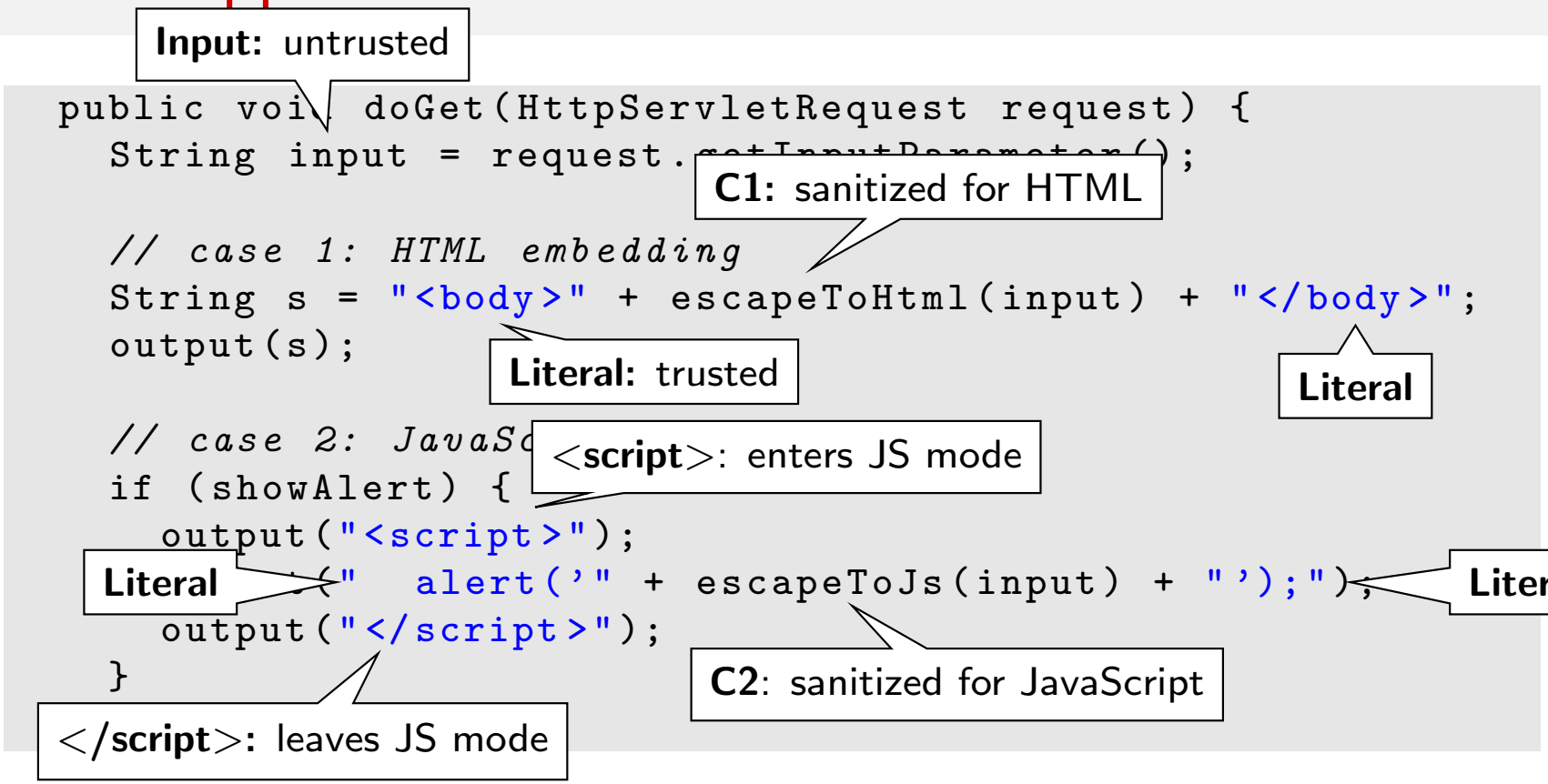
Concrete application scenario: Java servelet



ensure use of escape methods, depending on embedding position

- consider only immutable `String` objects
- classify strings according to origin and contents
- calls to `output` induce abstract output sequence

Concrete application scenario: Java servelet

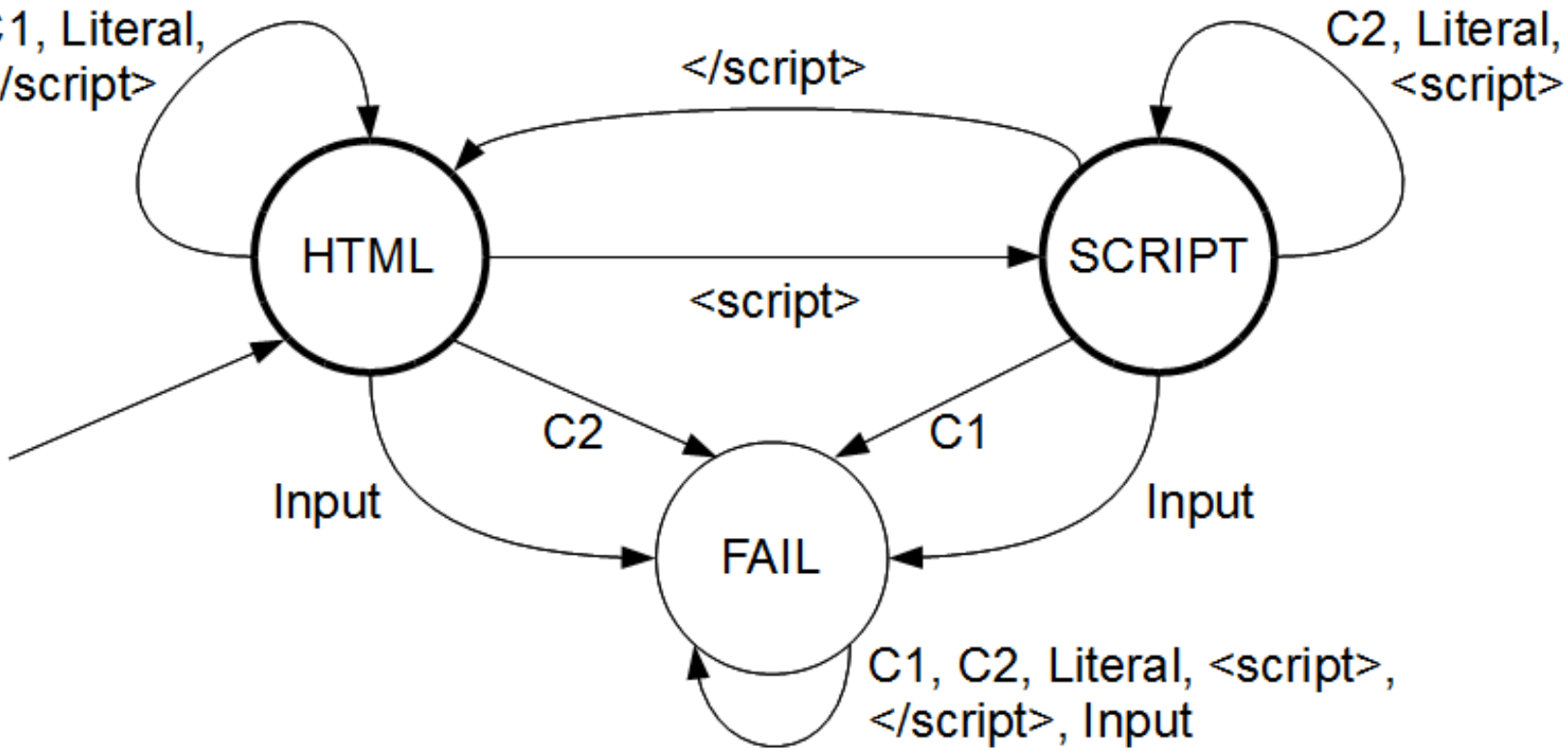


ensure use of escape methods, depending on embedding position

- consider only immutable `String` objects
- classify strings according to origin and contents
- calls to `output` induce abstract output sequence
- define set of allowed output sequences as regular language

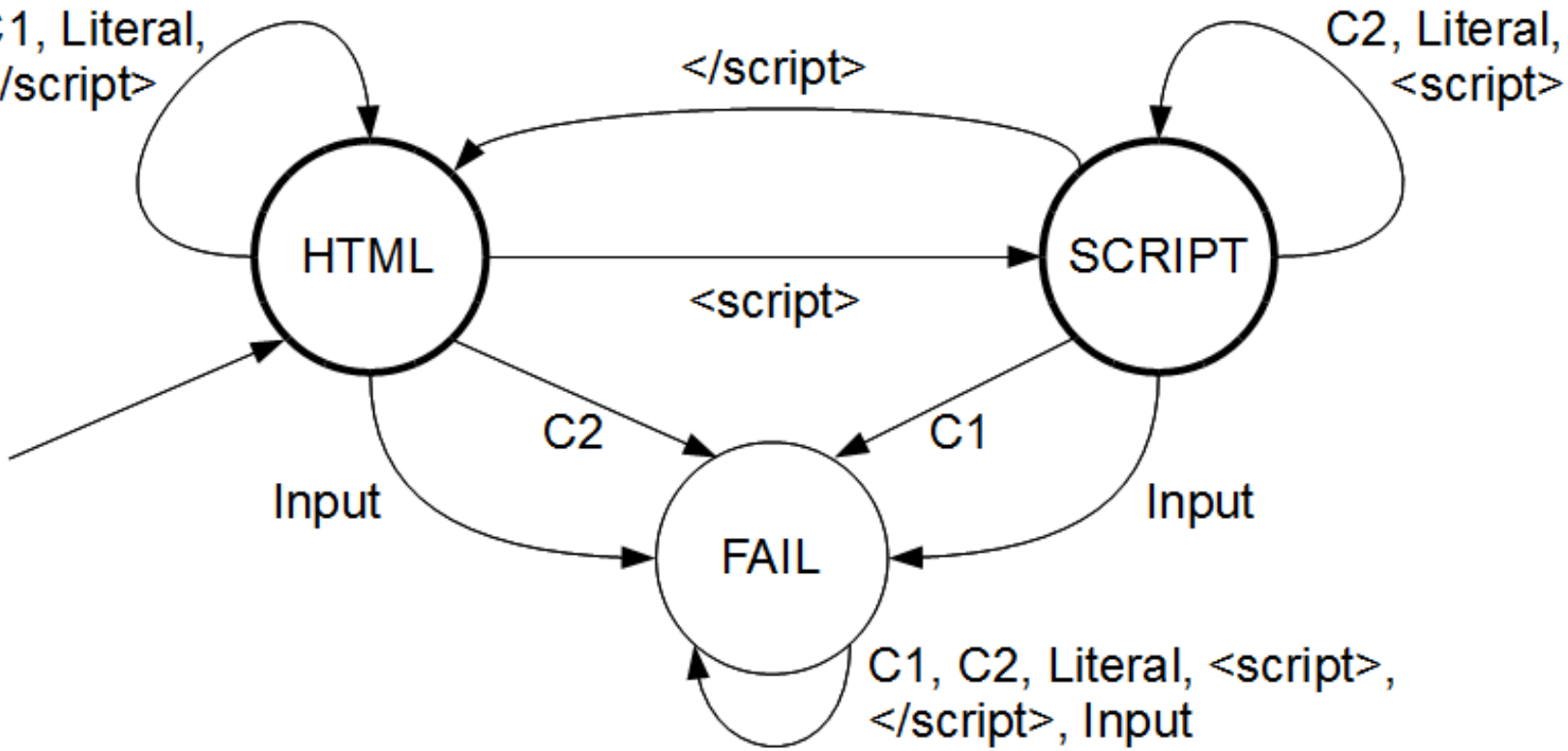
Guideline expressed as finite state machine

automaton $G = (Q, q_0, \delta, F)$ with alphabet $\Sigma =$ classifications



Guideline expressed as finite state machine

automaton $G = (Q, q_0, \delta, F)$ with alphabet $\Sigma =$ classifications



- accepted words = allowed program output
- we derive syntactic monoid: compact representation of words

Equivalence classes

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	**SCRIPT**	**FAIL**
[Literal] = [ϵ]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

Equivalence classes

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	**SCRIPT**	**FAIL**
[Literal] = [ϵ]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

- $[w]$ is allowed if $\delta(q_0, w) \in F$:

  [ϵ],[C1],[<script>],[</script>],[C1<script>]

Syntactic monoid

Monoid M:

- elements: equivalence classes $[w]$
- multiplication: concatenation

$$[w_1][w_2] = [w_1 w_2]$$

- neutral element: $[\mathit{Literal}] = [\epsilon]$

The type system is defined with respect to M.

String analysis for FJEUS

- FJEUS:
  Featherweight Java with Extended Updates and Strings

$$
\begin{array}{rcl}
e & ::= & x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ x_1 = x_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
& & \texttt{null} \mid \texttt{new}\ C \mid x.f \mid x_1.f := x_2 \mid x.m(\overline{x}) \mid \\
& & \text{"}str\text{"} \mid x_1 + x_2 \mid \texttt{output}(x)
\end{array}
$$

String analysis for FJEUS

- FJEUS:
  Featherweight Java with Extended Updates and Strings

$$
\begin{array}{rcl}
e & ::= & x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ x_1 = x_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
& & \texttt{null} \mid \texttt{new}\ C \mid x.f \mid x_1.f := x_2 \mid x.m(\overline{x}) \mid \\
& & \text{"}str\text{"} \mid x_1 + x_2 \mid \texttt{output}(x)
\end{array}
$$

- operational semantics with output trace / effect

$$(s, h) \vdash e \Downarrow v, h' \ \& \ t$$

- values v: object location (or `null`)
- traces t: sequence of string objects

Instrumented string semantics

Normal objects

- contain fields, methods

String objects

- contain immutable string value,
  are tagged with some class $[w]$ from monoid
- new strings are tagged in the way the should, e.g.
  - "$\langle$script$\rangle$" literal evaluates to $[\mathit{Script}]$-tagged string
  - getInputParameter() returns $[\mathit{Input}]$-tagged string
- concatenation of string objects $x_1 + x_2$:
  - if x_1 is tagged with $[w_1]$ and x_2 is tagged with $[w_2]$, then $x_1 + x_2$ is a new string tagged with $[w_1 w_2]$
- the built-in expression $\texttt{output}(x)$ has the output effect x

Type and effect system

- FJEU type system extended with string types:

$$\begin{array}{rrcl} \text{string type} & & & \\ \text{refinements:} & U, V & \subseteq & M \\ \text{types:} & \tau & ::= & C \mid \texttt{String}_U \end{array}$$

- class table specifies field and method types, inheritance (implicit in the following)

- typing judgement:

$$\Gamma \vdash e : \tau \,\&\, U$$

Interpretation of typing judgement

If

$$(s, h) \vdash e \Downarrow v, h' \;\&\; t$$

and

$$\Gamma \vdash e : \texttt{String}_U \;\&\; V$$

...and a couple of other premises...

then

- v is string object tagged with element from U, and
- concatenated trace t is string tagged with element from V

Some typing rules

Literal typing:

$$\frac{}{\Gamma \vdash \text{"}abc\text{"} : \texttt{String}_{\{[\epsilon]\}} \,\&\, \{[\epsilon]\}} \qquad \frac{}{\Gamma \vdash \text{"<script>"} : \texttt{String}_{\{[Script]\}} \,\&\, \{[\epsilon]\}}$$

Some typing rules

Literal typing:

$$\frac{}{\Gamma \vdash \text{"}abc\text{"} : \texttt{String}_{\{[\epsilon]\}} \,\&\, \{[\epsilon]\}} \qquad \frac{}{\Gamma \vdash \text{"<script>"} : \texttt{String}_{\{[Script]\}} \,\&\, \{[\epsilon]\}}$$

Subtyping:

$$\frac{\Gamma \vdash e : \texttt{String}_{U} \,\&\, V \qquad U \subseteq U' \qquad V \subseteq V'}{\Gamma \vdash e : \texttt{String}_{U'} \,\&\, V'}$$

Some typing rules

Literal typing:

$$\frac{}{\Gamma \vdash \text{"}abc\text{"} : \texttt{String}_{\{[\epsilon]\}} \,\&\, \{[\epsilon]\}} \qquad \frac{}{\Gamma \vdash \text{"<script>"} : \texttt{String}_{\{[Script]\}} \,\&\, \{[\epsilon]\}}$$

Subtyping:

$$\frac{\Gamma \vdash e : \texttt{String}_U \,\&\, V \qquad U \subseteq U' \qquad V \subseteq V'}{\Gamma \vdash e : \texttt{String}_{U'} \,\&\, V'}$$

Concatenation rule follows semantics:

$$\frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \texttt{String}_{UU'} \,\&\, \{[\epsilon]\}}$$

Some typing rules

Literal typing:

$$\frac{}{\Gamma \vdash \text{"}abc\text{"} : \texttt{String}_{\{[\epsilon]\}} \,\&\, \{[\epsilon]\}} \qquad \frac{}{\Gamma \vdash \text{"<script>"} : \texttt{String}_{\{[Script]\}} \,\&\, \{[\epsilon]\}}$$

Subtyping:

$$\frac{\Gamma \vdash e : \texttt{String}_U \,\&\, V \qquad U \subseteq U' \qquad V \subseteq V'}{\Gamma \vdash e : \texttt{String}_{U'} \,\&\, V'}$$

Concatenation rule follows semantics:

$$\frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \texttt{String}_{UU'} \,\&\, \{[\epsilon]\}}$$

$$\boxed{UU' = \{[ww'] \mid [w] \in U \wedge [w'] \in U'\}}$$

Some typing rules

Literal typing:

$$\frac{}{\Gamma \vdash \text{"}abc\text{"} : \texttt{String}_{\{[\epsilon]\}} \,\&\, \{[\epsilon]\}} \qquad \frac{}{\Gamma \vdash \text{"<script>"} : \texttt{String}_{\{[Script]\}} \,\&\, \{[\epsilon]\}}$$

Subtyping:

$$\frac{\Gamma \vdash e : \texttt{String}_U \,\&\, V \qquad U \subseteq U' \qquad V \subseteq V'}{\Gamma \vdash e : \texttt{String}_{U'} \,\&\, V'}$$

Concatenation rule follows semantics:

$$\frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \texttt{String}_{UU'} \,\&\, \{[\epsilon]\}}$$

Output effect:

$$\frac{\Gamma(x) = \texttt{String}_U}{\Gamma \vdash \texttt{output}(x) : \mathit{Void} \,\&\, U}$$

Some typing rules

Literal typing:

$$\frac{}{\Gamma \vdash \text{"}abc\text{"} : \texttt{String}_{\{[\epsilon]\}} \,\&\, \{[\epsilon]\}} \qquad \frac{}{\Gamma \vdash \text{"<script>"} : \texttt{String}_{\{[Script]\}} \,\&\, \{[\epsilon]\}}$$

Subtyping:

$$\frac{\Gamma \vdash e : \texttt{String}_U \,\&\, V \qquad U \subseteq U' \qquad V \subseteq V'}{\Gamma \vdash e : \texttt{String}_{U'} \,\&\, V'}$$

Concatenation rule follows semantics:

$$\frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \texttt{String}_{UU'} \,\&\, \{[\epsilon]\}}$$

Output effect:

$$\frac{\Gamma(x) = \texttt{String}_U}{\Gamma \vdash \texttt{output}(x) : \mathit{Void} \,\&\, U}$$

Effect concatenation:

$$\frac{\Gamma \vdash e_1 : \tau \,\&\, V \qquad \Gamma, x : \tau \vdash e_2 : \tau' \,\&\, V'}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau' \,\&\, VV'}$$

External functions

External functions are assigned a trusted signature:

$$\mathit{getInputParameter}() : \mathtt{String}_{\{[\mathit{Input}]\}} \;\&\; \{[\epsilon]\}$$

$$\mathit{escapeToHtml}(x : \mathtt{String}_{\{[\mathit{Input}]\}}) : \mathtt{String}_{\{[C1]\}} \;\&\; \{[\epsilon]\}$$

Example: rejected program

$$
\begin{aligned}
&\mathit{main}\ () : \mathit{Void}\ \&\ \{[\epsilon], [C1], [\mathit{Script}], [/\mathit{Script}], [C1\mathit{Script}]\} = \\
&\quad \texttt{let}\ u = \mathit{getInputParameter}()\ \texttt{in} \\
&\quad \texttt{let}\ e = \mathit{escapeToHtml}(u)\ \texttt{in} \\
&\quad \texttt{let}\ \_ = \texttt{output}(\text{"<script>"})\ \texttt{in}\ \texttt{output}(e)
\end{aligned}
$$

Example: rejected program

$$main\ () : Void\ \&\ \{[\epsilon], [C1], [Script], [/Script], [C1Script]\} =$$

$$\texttt{let}\ u = getInputParameter() \ \texttt{in}$$

effect $= \{[\epsilon]\}$

$\Gamma(u) = \texttt{String}_{\{[Input]\}}$

$$\texttt{let}\ e = escapeToHtml(u) \ \texttt{in}$$

$$\texttt{let}\ \_ = \texttt{output}(\text{"<script>"}) \ \texttt{in}\ \texttt{output}(e)$$

Example: rejected program

$$\mathit{main}\ () : \mathit{Void}\ \&\ \{[\epsilon], [C1], [\mathit{Script}], [/\mathit{Script}], [C1\mathit{Script}]\} =$$

$$\texttt{let}\ u = \mathit{getInputParameter}()\ \texttt{in}$$

effect $= \{[\epsilon]\}$

$$\Gamma(u) = \texttt{String}_{\{[\mathit{Input}]\}}$$

$$\texttt{let}\ e = \mathit{escapeToHtml}(u)\ \texttt{in}$$

effect $= \{[\epsilon]\}$

$$\Gamma(e) = \texttt{String}_{\{[C1]\}}$$

$$\texttt{let}\ \_ = \texttt{output}(\text{"<script>"})\ \texttt{in}\ \texttt{output}(e)$$

Example: rejected program

$$main\ () : Void\ \&\ \{[\epsilon], [C1], [Script], [/Script], [C1Script]\} =$$

$$\texttt{let}\ u = getInputParameter()\ \texttt{in}$$

effect $= \{[\epsilon]\}$

$\Gamma(u) = \texttt{String}_{\{[Input]\}}$

$$\texttt{let}\ e = escapeToHtml(u)\ \texttt{in}$$

effect $= \{[\epsilon]\}$

$\Gamma(e) = \texttt{String}_{\{[C1]\}}$

$$\texttt{let}\ \_ = \texttt{output}(\text{"<script>"})\ \texttt{in}\ \texttt{output}(e)$$

effect $= \{[Script]\}$

effect $= \{[C1]\}$

Example: rejected program

$$main\ () : Void\ \&\ \{[\epsilon], [C1], [Script], [/Script], [C1Script]\} =$$

$$\texttt{let}\ u = getInputParameter()\ \texttt{in}$$

effect $= \{[\epsilon]\}$

$\Gamma(u) = \texttt{String}_{\{[Input]\}}$

$$\texttt{let}\ e = escapeToHtml(u)\ \texttt{in}$$

effect $= \{[\epsilon]\}$

$\Gamma(e) = \texttt{String}_{\{[C1]\}}$

$$\texttt{let}\ \_ = \texttt{output}(\text{"<script>"})\ \texttt{in}\ \texttt{output}(e)$$

effect $= \{[Script]\}$

effect $= \{[C1]\}$

overall effect: $[\epsilon][\epsilon][Script][C1] = [Script][C1] = [Input]$

$\longrightarrow$ not a subeffect of declared effect

String type inference

FJEUS formalizes Java:

- class types specified in program
- here: inference of string refinements

intra-procedural:

- simply calculate types and effects "from left to right"

inter-procedural: infer method signatures

- for each formal string argument create type variable $U \subseteq M$
- typing rules generate set constraints
- solved externally
- easy, small base set (monoid)

Implementation

based on fjavac compiler [Tse & Zdancewic '05]

- Java compiler implemented in Ocaml
- contains normal class type checker

programmer may specify refined string types and method effects

- syntax: standard Java annotations
- use "?" annotation for types and effects that shall be inferred
- no annotation for "any string" or "no output effect"; compatibility with Java runtime library

inference: set constraints as Datalog rules

- solved with Succinct Solver [Nielson, Seidl, Nielson '03]
- solver output checked again

Demo

Improving precision of the analysis

building on earlier work [Beringer, G., Hofmann '10]

1. context-sensitive method analysis
2. class types refined with regions

Context-sensitive method analysis

```
   String appendLn(String s) { return s + "\n"; }

16 String x = appendLn("Your name is:");
17 String y = appendLn(sanitizedInput);
```

$$appendLn : \texttt{String}_{\{[\epsilon],[C1]\}} \rightarrow \texttt{String}_{\{[\epsilon],[C1]\}}$$

Context-sensitive method analysis

```
    String appendLn(String s) { return s + "\n"; }

16  String x = appendLn("Your name is:");
17  String y = appendLn(sanitizedInput);
```

- use different method types for different calls of the method

call site	*type*
16	$\mathit{appendLn} : \texttt{String}_{[\epsilon]} \to \texttt{String}_{[\epsilon]}$
17	$\mathit{appendLn} : \texttt{String}_{[C1]} \to \texttt{String}_{[C1]}$

Context-sensitive method analysis

```
15   String doSomething(String sanitizedInput) {
16     String x = appendLn("Your name is:");
17     String y = appendLn(sanitizedInput);
18   }

54   doSomething(escapeToHtml(input));

80   doSomething(escapeToJs(input));
```

- use different method types for different calls of the method

call site stack	*type*
16 :: 54	$appendLn : \texttt{String}_{[\epsilon]} \rightarrow \texttt{String}_{[\epsilon]}$
17 :: 54	$appendLn : \texttt{String}_{[C1]} \rightarrow \texttt{String}_{[C1]}$

Context-sensitive method analysis

```
15   String doSomething(String sanitizedInput) {
16     String x = appendLn("Your name is:");
17     String y = appendLn(sanitizedInput);
18   }

54   doSomething(escapeToHtml(input));

80   doSomething(escapeToJs(input));
```

- use different method types for different calls of the method

call site stack	*type*
16 :: 54	$\mathit{appendLn} : \texttt{String}_{[\epsilon]} \to \texttt{String}_{[\epsilon]}$
17 :: 54	$\mathit{appendLn} : \texttt{String}_{[C1]} \to \texttt{String}_{[C1]}$
16 :: 80	$\mathit{appendLn} : \texttt{String}_{[\epsilon]} \to \texttt{String}_{[\epsilon]}$
17 :: 80	$\mathit{appendLn} : \texttt{String}_{[C2]} \to \texttt{String}_{[C2]}$

Context-sensitive method analysis

```
15   String doSomething(String sanitizedInput) {
16     String x = appendLn("Your name is:");
17     String y = appendLn(sanitizedInput);
18   }

54   doSomething(escapeToHtml(input));

80   doSomething(escapeToJs(input));
```

- distinguish method types by call context from finite set Cxt
- method call rule relies on context switch function

$$\phi : Cxt \times Cls \times Mtd \times PP \rightarrow Cxt$$

  to select the method type for the invoked method
- inference generates new method type for unseen contexts

Region types

refine class types with sets of regions:

$$\Gamma \vdash e : C_R$$

regions further classify objects:

- represent disjoint sets of possible concrete memory locations
- e is a location that is in one of the regions in R
- two locations typed with disjoint sets do not alias

Region types

refine class types with sets of regions:

$$\Gamma \vdash e : C_R$$

regions further classify objects:

- represent disjoint sets of possible concrete memory locations
- e is a location that is in one of the regions in R
- two locations typed with disjoint sets do not alias

increases typing precision:

$$\mathit{fty}(\texttt{List}, r, \texttt{elem}) = \texttt{String}_{\{[\mathit{Input}]\}} \qquad \mathit{fty}(\texttt{List}, s, \texttt{elem}) = \texttt{String}_{\{[C1]\}}$$
$$\mathit{fty}(\texttt{List}, r, \texttt{next}) = \texttt{List}_{\{r\}} \qquad \mathit{fty}(\texttt{List}, s, \texttt{next}) = \texttt{List}_{\{s\}}$$

Region type system

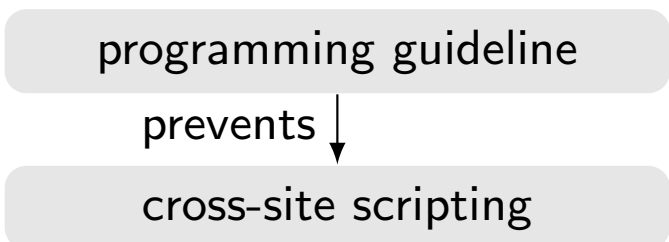
- type system parametrized with abstraction principles from pointer analysis
  - choice of regions for new objects
  - contexts
- inference: follows existing analysis [Whaley & Lam '04]
  - generate constraints for points-to relations as Datalog rules
  - use external, highly optimized solver
  - interprete resulting relations as types and verify them with type system
- for more information: see LPAR paper

Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

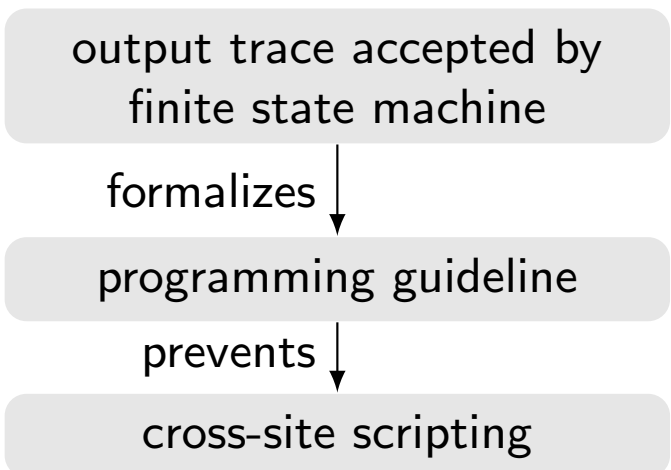
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks



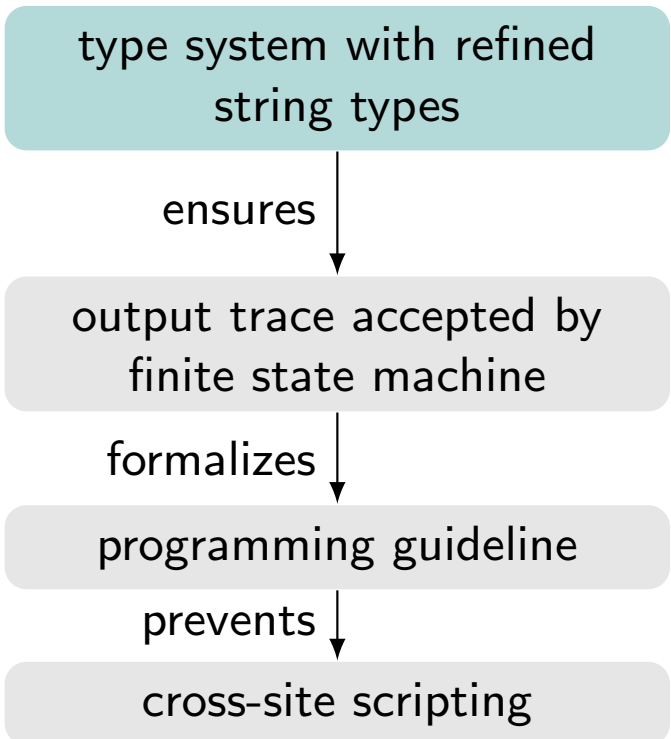
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks



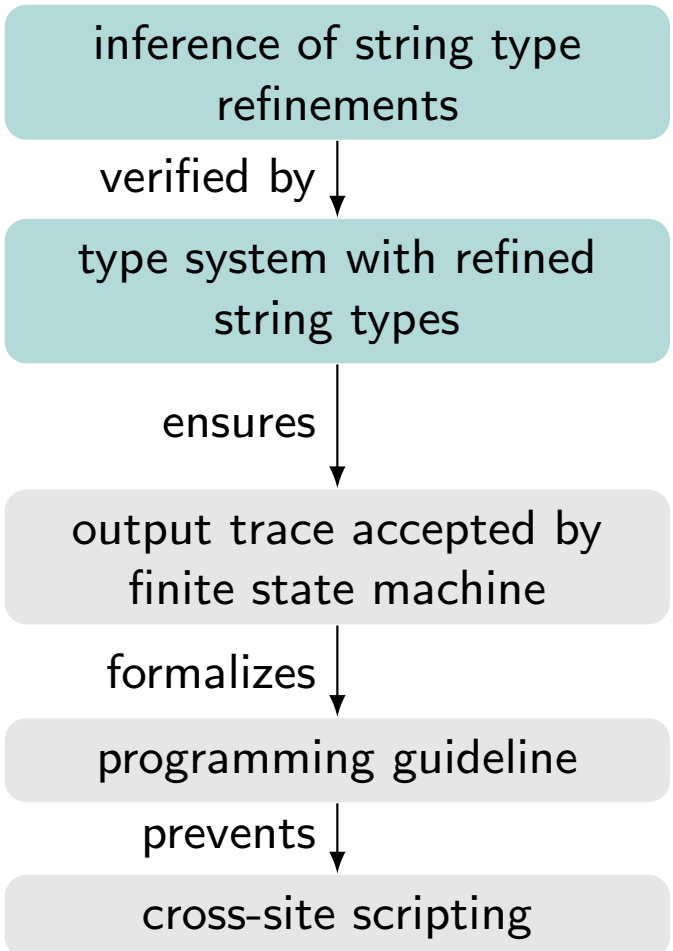
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks



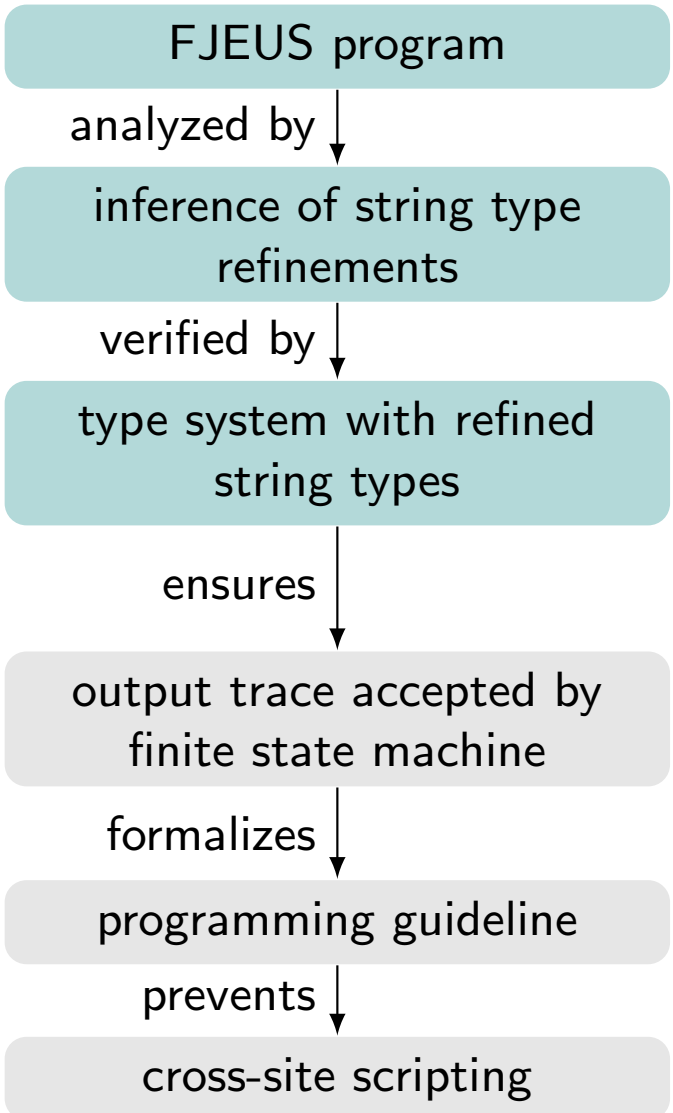
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

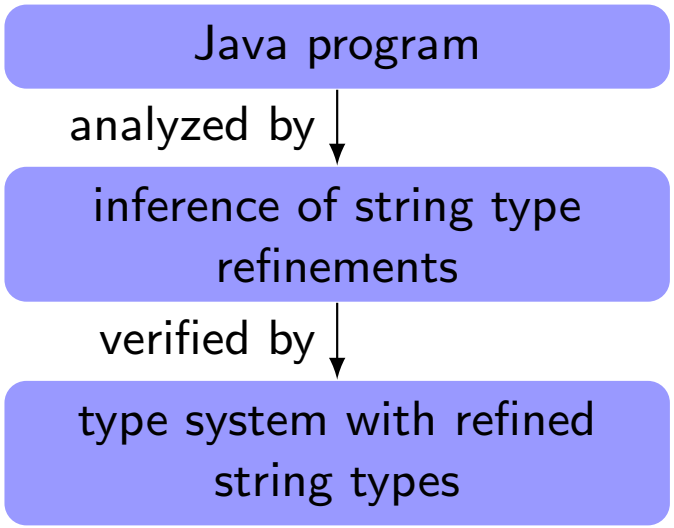


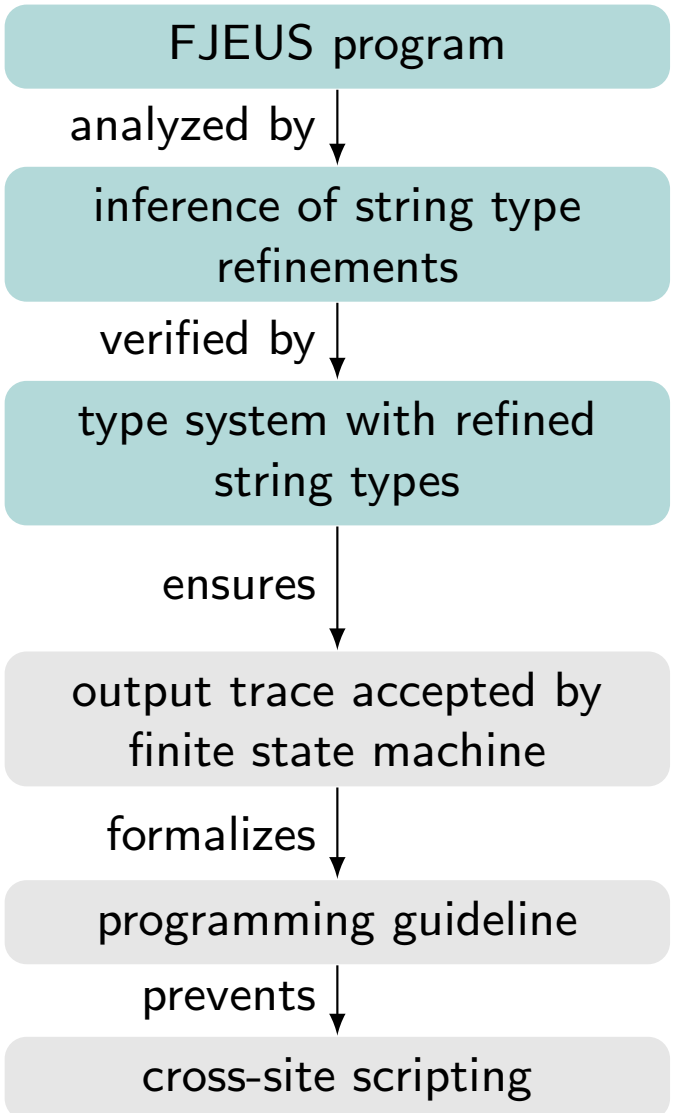
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks



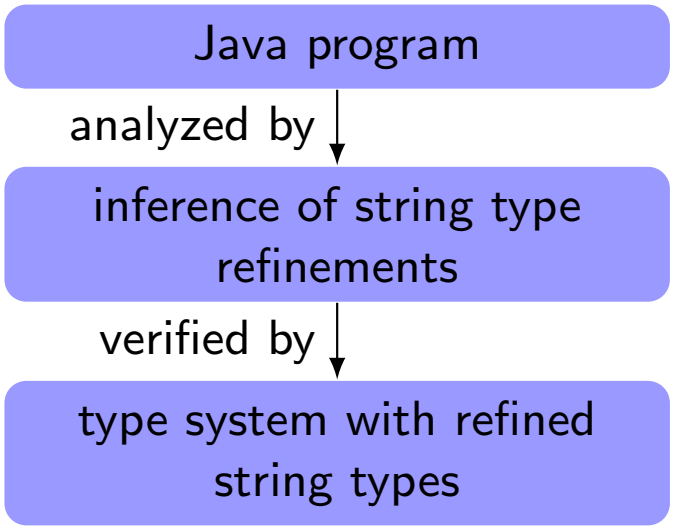
Summary

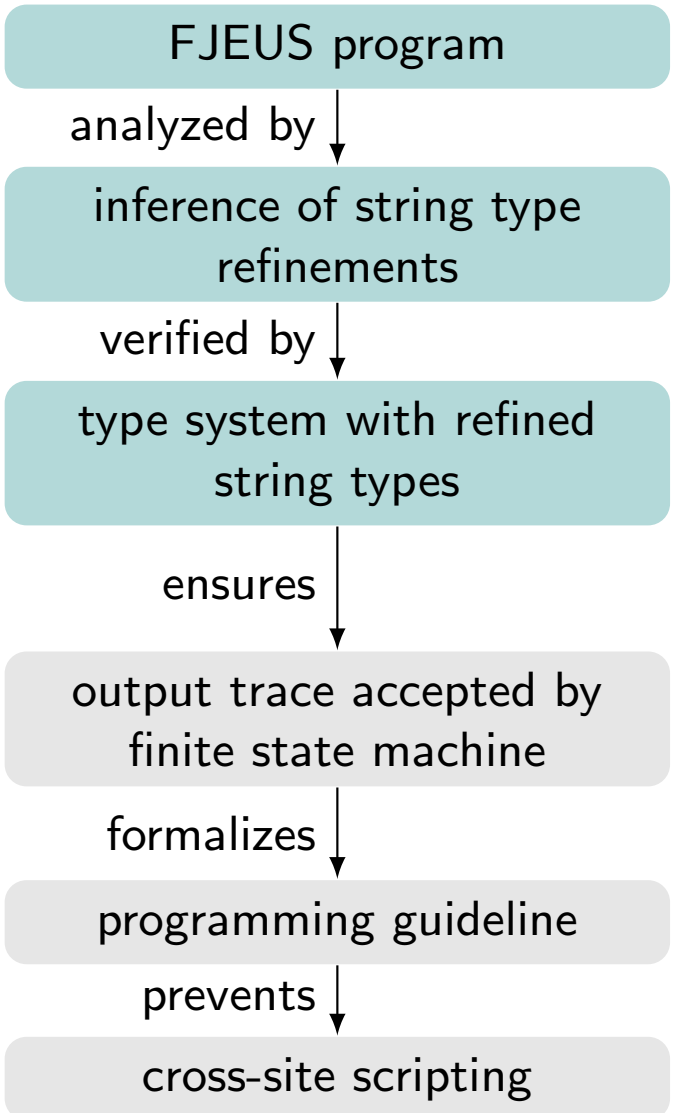




type-based enforcement of string handling guidelines to prevent XSS attacks

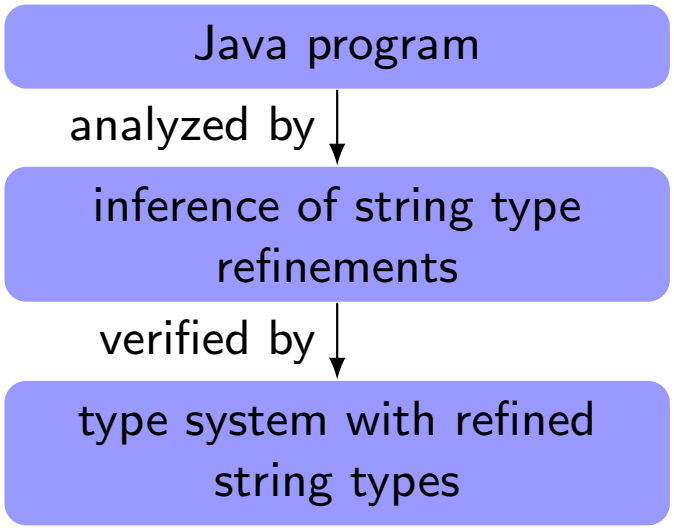
Summary and Future Work

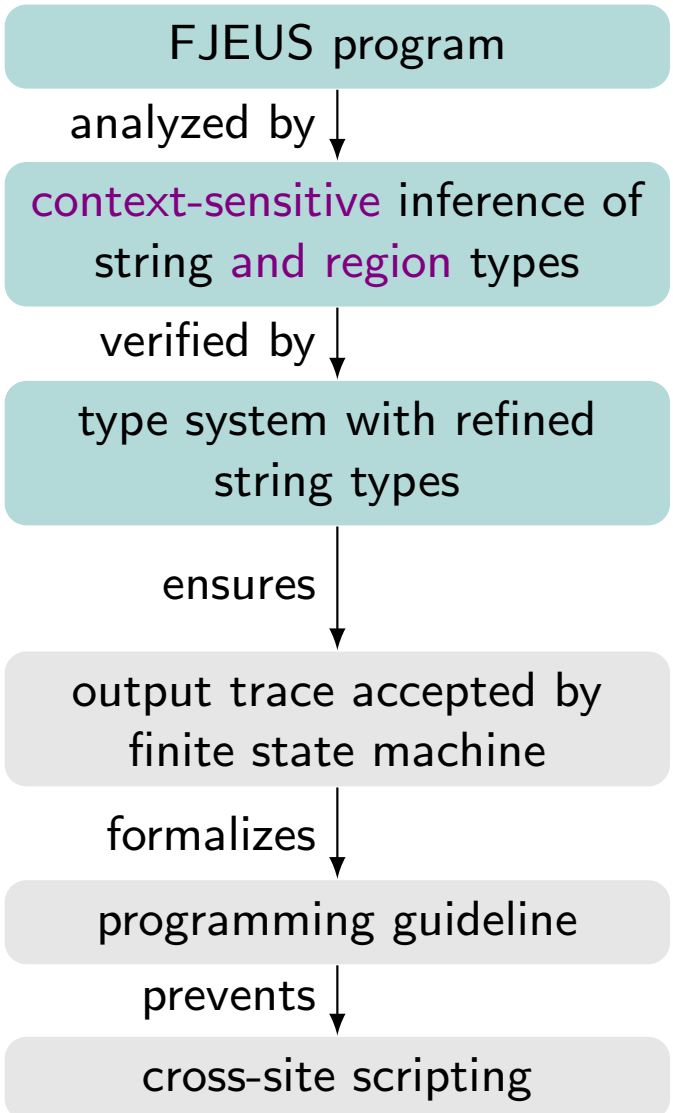




type-based enforcement of string handling guidelines to prevent XSS attacks

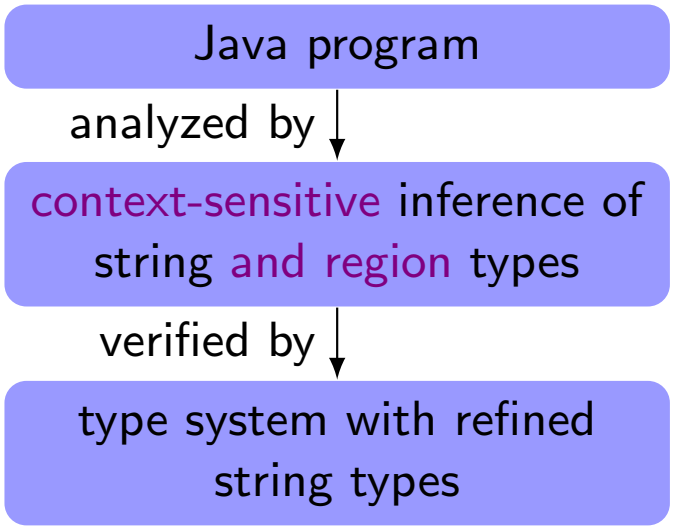
Summary and Future Work

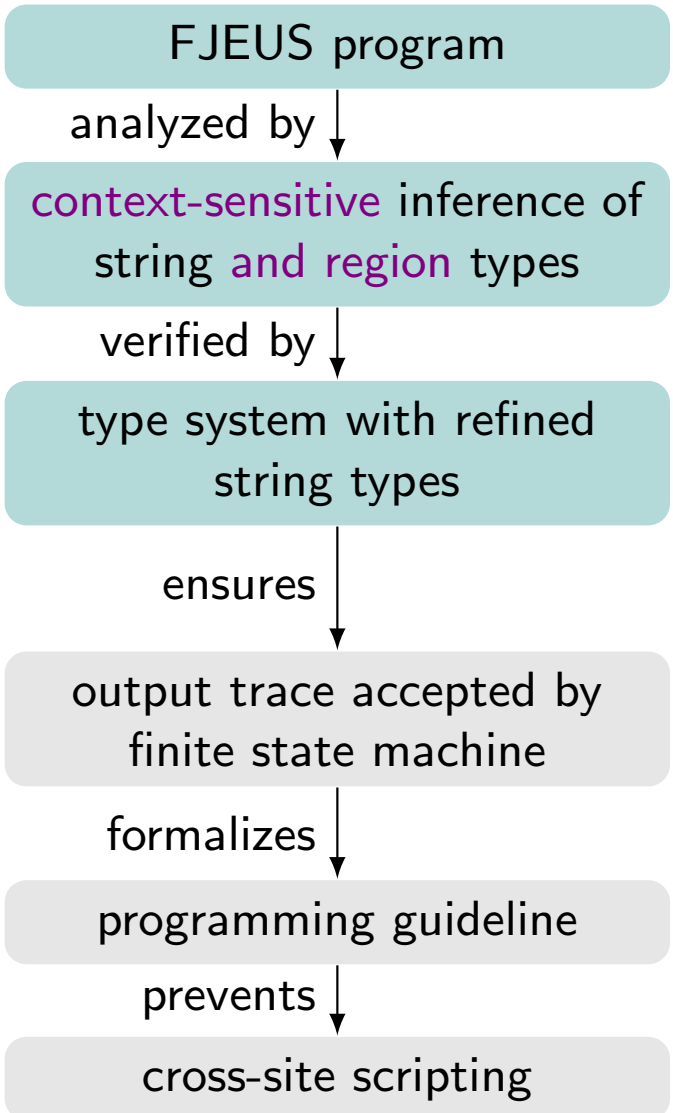




type-based enforcement of string handling guidelines to prevent XSS attacks

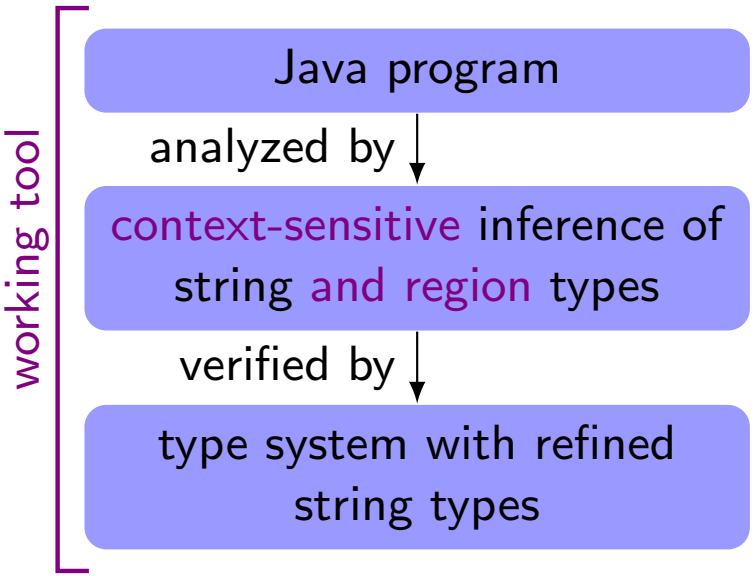
Summary and Future Work

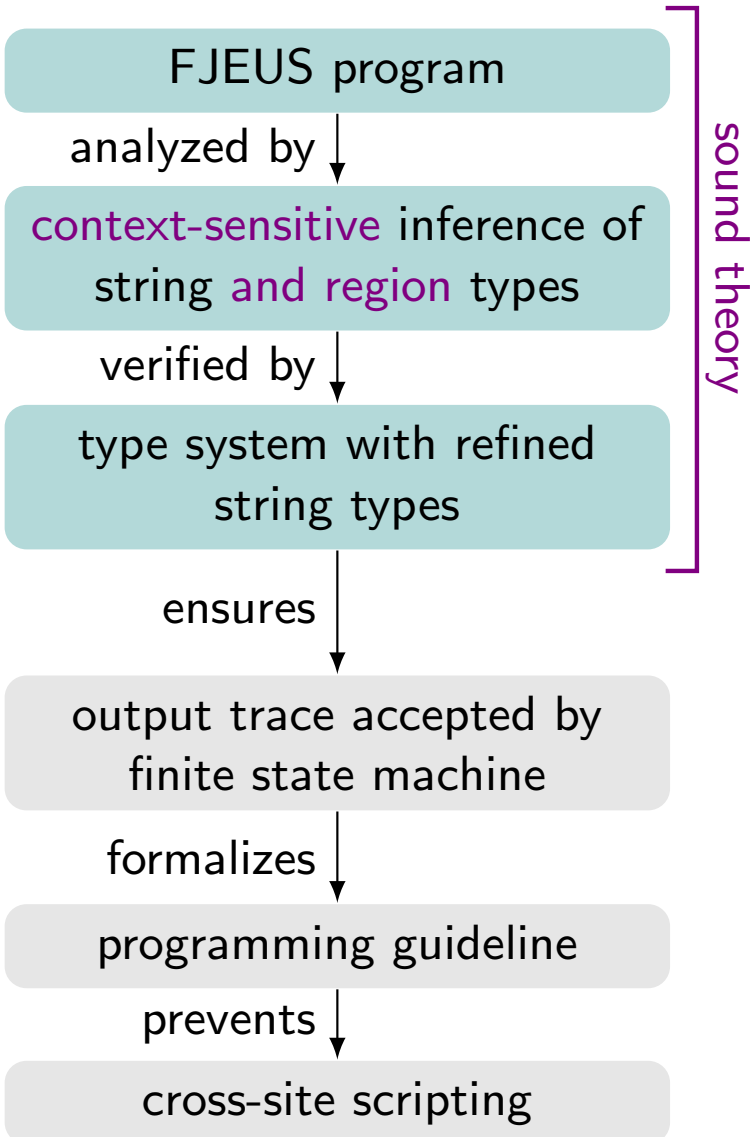




type-based enforcement of string handling guidelines to prevent XSS attacks

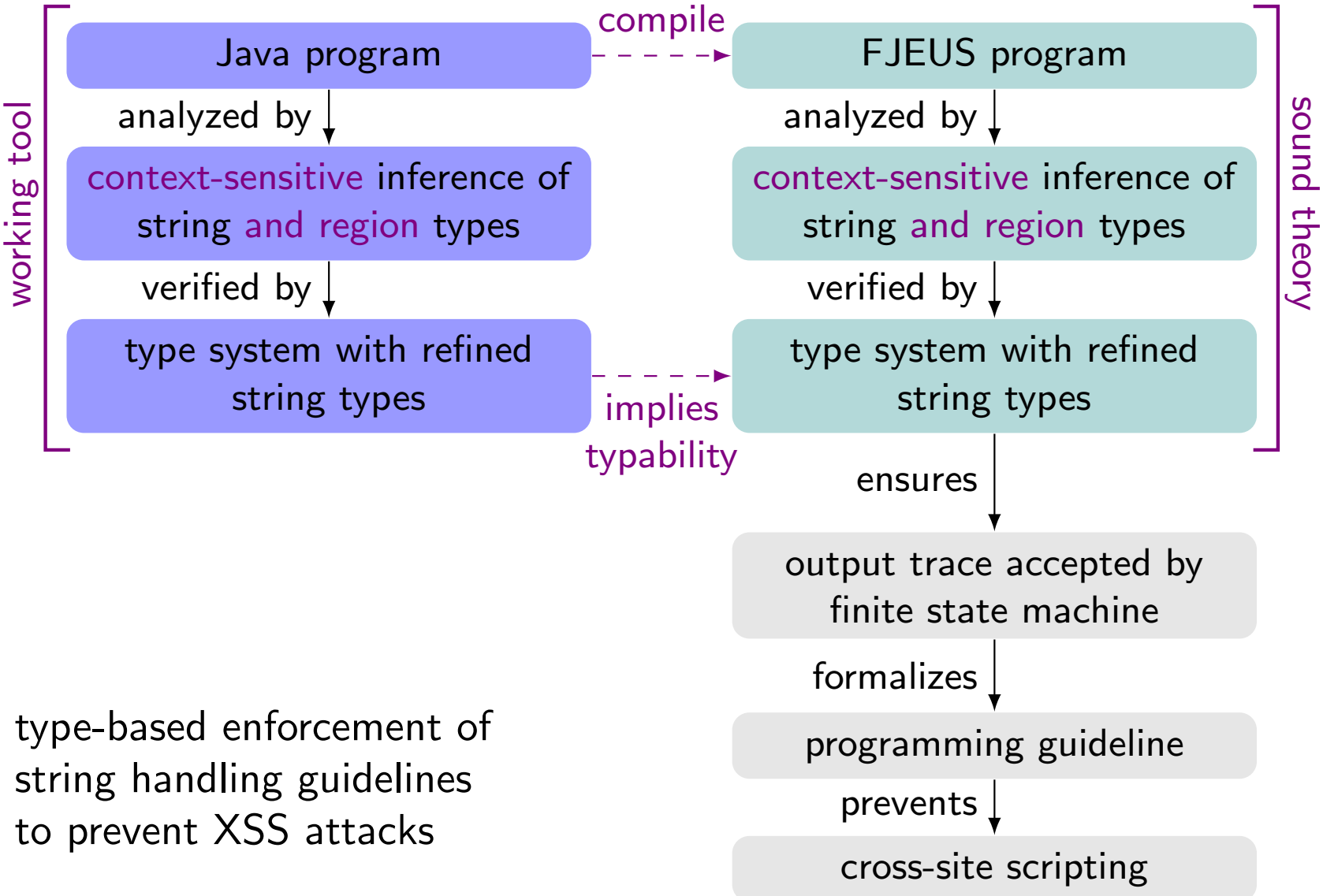
Summary and Future Work





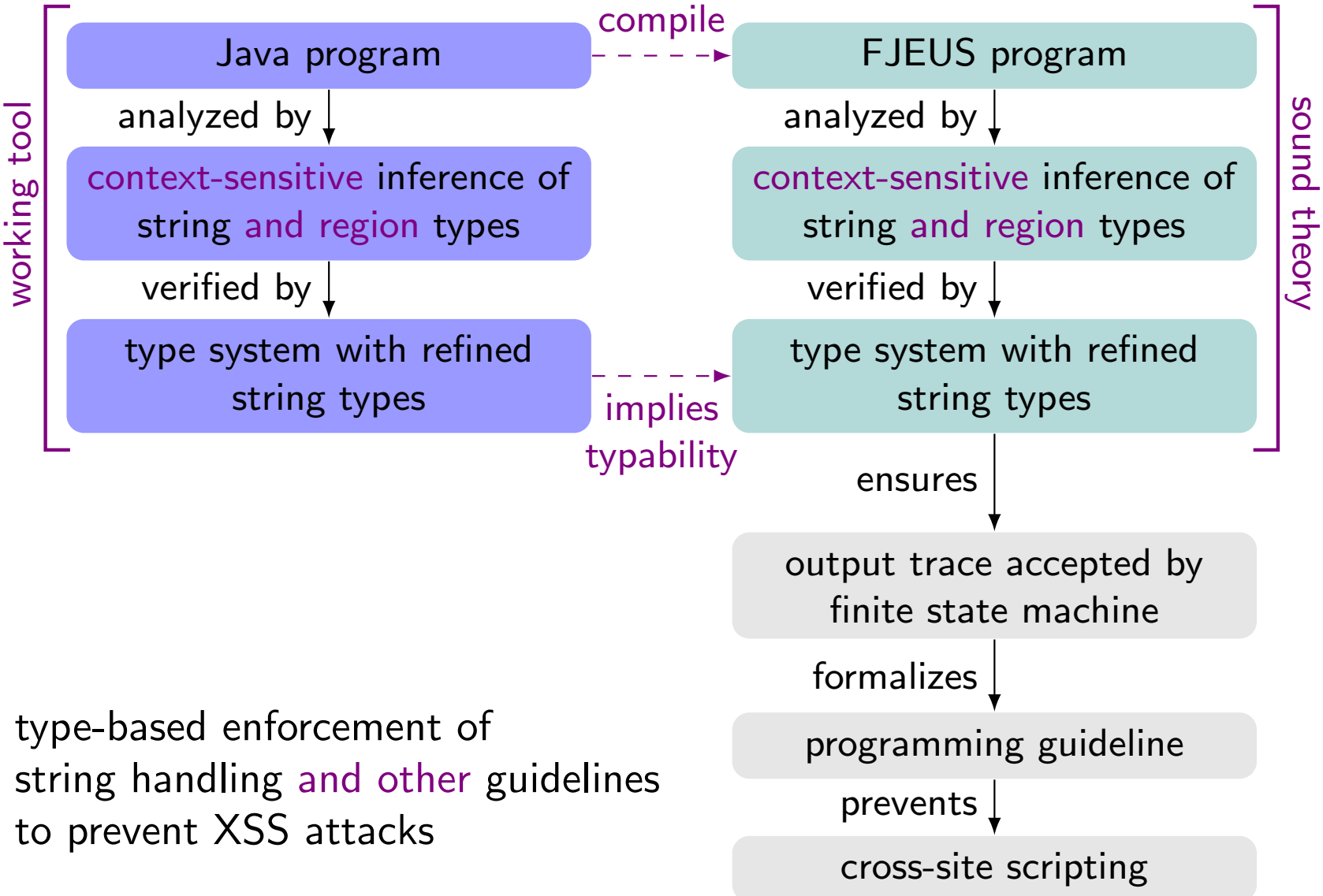
type-based enforcement of
string handling guidelines
to prevent XSS attacks

Summary and Future Work



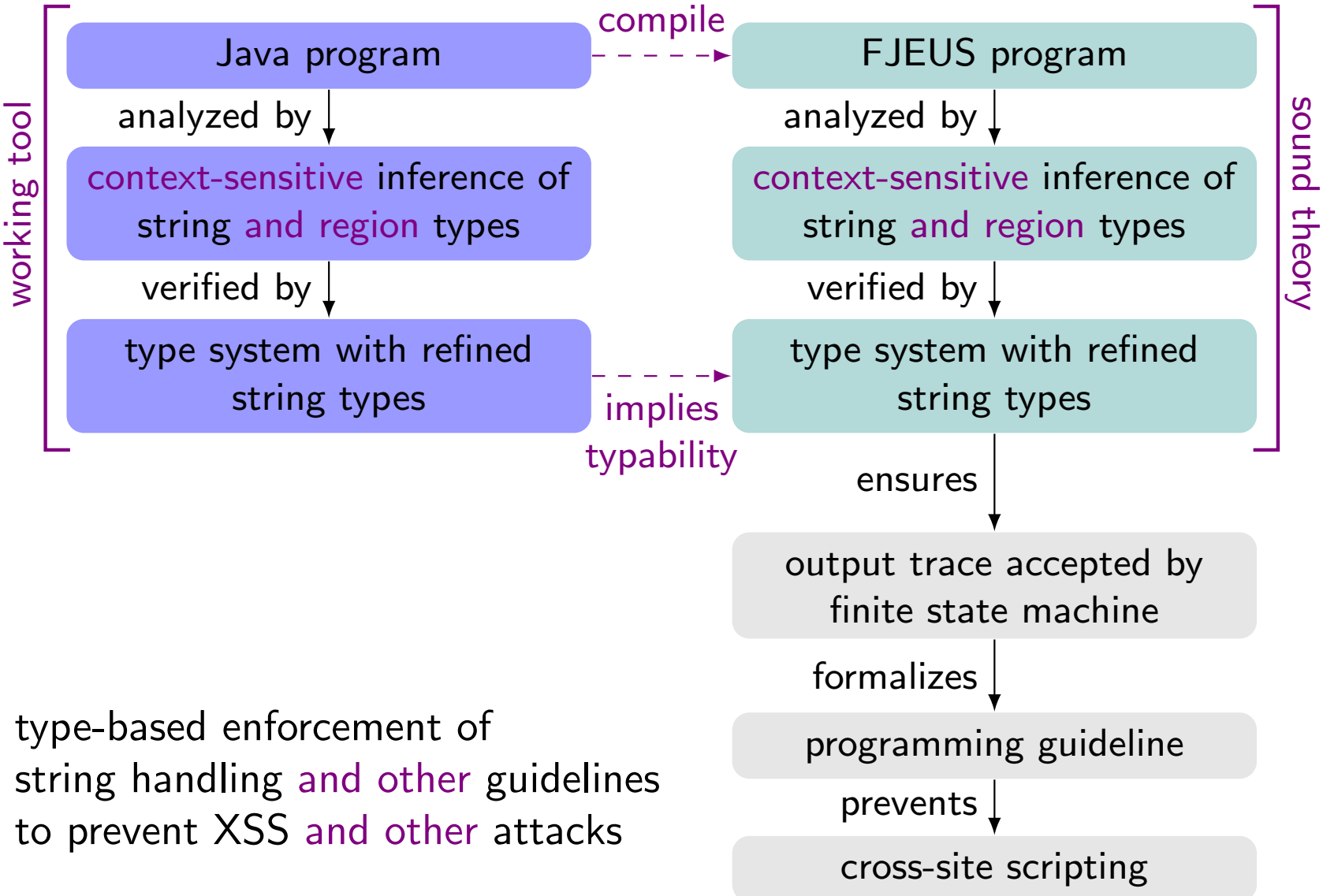
type-based enforcement of
string handling guidelines
to prevent XSS attacks

Summary and Future Work



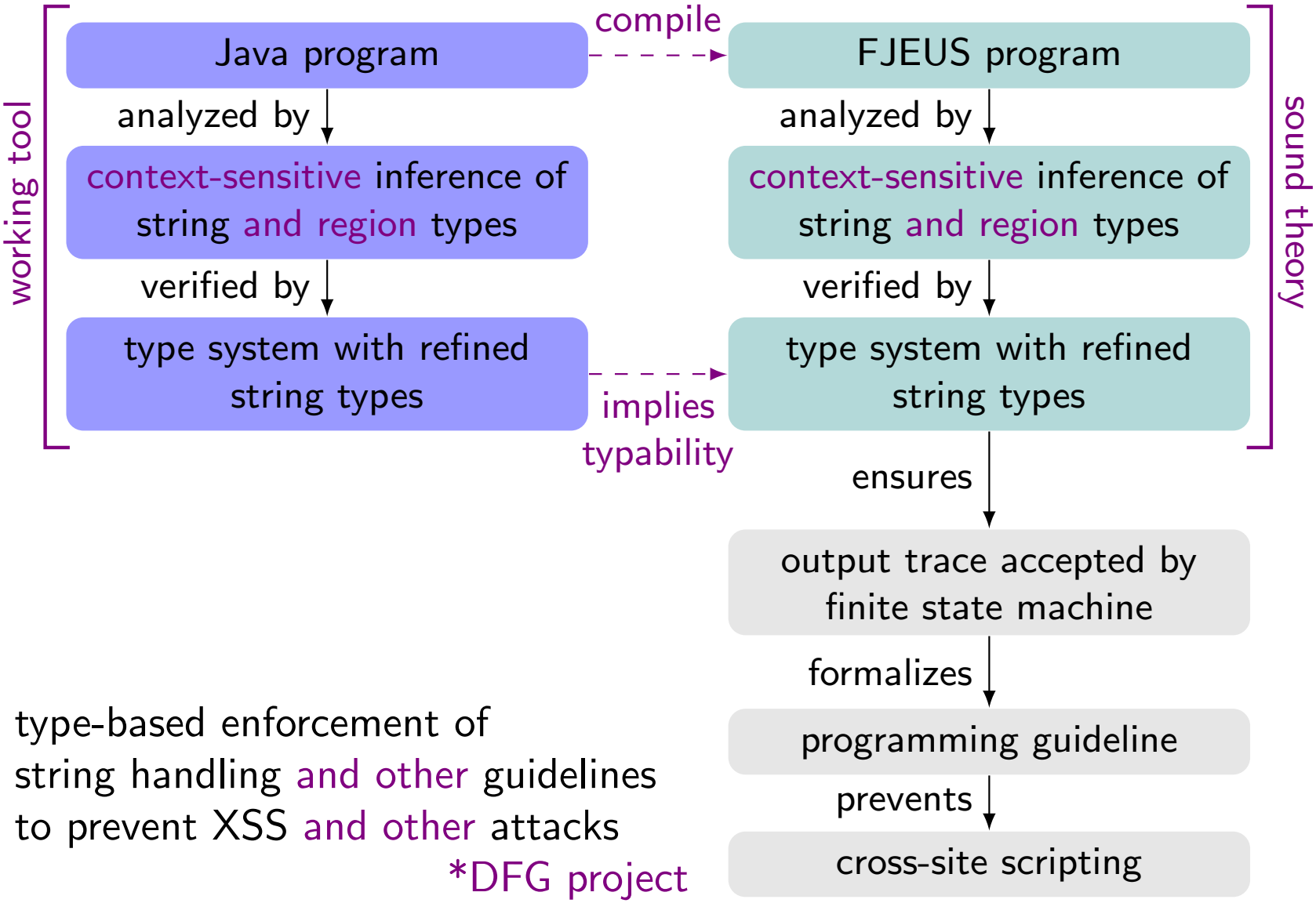
type-based enforcement of
string handling and other guidelines
to prevent XSS attacks

Summary and Future Work



type-based enforcement of
string handling and other guidelines
to prevent XSS and other attacks

Summary and Future Work



type-based enforcement of
string handling and other guidelines
to prevent XSS and other attacks

*DFG project

Thank you for your attention!

Backup slides: Actual use cases

1. embed between HTML tags:

```
output("<h1> Hello " + escapeToHtml(x) + "</h1>");
```

2. embed as attribute value:

```
output("<img alt='" + escapeToAttr(x) + "' />");
```

3. embed as URL attribute value:

```
output("<img src='http://" + escapeToUrl(x) + "' />");
```

4. embed within JavaScript:

```
output("<script>");
output("  alert('" + escapeToJs(x) + "');");
output("</script>");
```