Type-Based Enforcement of Secure Programming Guidelines — Code Injection Prevention at SAP

Robert Grabowski[1], Martin Hofmann[1], and Keqin Li[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität,
Oettingenstrasse 67, D-80538 München, Germany
{robert.grabowski,martin.hofmann}@ifi.lmu.de
[2] SAP Research, France
805 Avenue du Dr Maurice Donat, 06254 Mougins Cedex, France
keqin.li@sap.com

Abstract. Code injection and cross-site scripting belong to the most common security vulnerabilities in modern software, usually caused by incorrect string processing. These exploits are often addressed by formulating programming guidelines or "best practices".

In this paper, we study the concrete example of a guideline used at SAP for the handling of untrusted, potentially executable strings that are embedded in the output of a Java servlet. To verify adherence to the guideline, we present a type system for a Java-like language that is extended with refined string types, output effects, and polymorphic method types.

The practical suitability of the system is demonstrated by an implementation of a corresponding string type verifier and context-sensitive inference for real Java programs.

1 Introduction

Modern software typically must be able to interact with the whole world. For example, almost all business software provides access via a web interface, thereby exposing it to worldwide security threats. At the same time, one can no longer rely on the high skill and experience of specialist programmers.

To address this issue, programming guidelines and "best practices" have been developed [1] that summarize and condense the expert knowledge and make it available to a larger community. The extent to which such guidelines are correctly applied, however, is left to the responsibility of the programmer. It is thus desirable to use automatic methods to check that programming guidelines have been correctly and reasonably applied without compromising the flexibility of writing code. We propose to use a type-based approach for this purpose.

As a proof-of-concept, in this paper, we concentrate on a guideline used at SAP [2,3] to counter the particular security threat posed by *code injection*, where a malicious user inputs strings containing code fragments that may potentially be executed — assuming there exists a corresponding vulnerability on the server

G. Barthe, A. Datta, and S. Etalle (Eds.): FAST 2011, LNCS 7140, pp. 182–197, 2012.

side. This scenario is surprisingly common, and indeed in the top ten list of most critical web application security risks published by the Open Web Application Security Project (OWASP), the top two positions are related to code injection [4].

1.1 Code Injection

For a simple example, consider a wiki web service that allows the creation of a new page by sending arbitrary text `contents` to the server, which is then displayed as a HTML page to a visitor using the following code on the server:

```
output("<body>" + contents + "</body>");
```

If `contents` is the malicious string

```
<script src="http://attacker.com/evil.js" />
```

then loading the generated HTML page will automatically execute a script from a different server. There are of course numerous more sophisticated attacks [3,5,6], for example such that an attacker can spy on authentication cookies.

All these attacks share a common pattern: they usually arise whenever untrusted input, typically a string, is combined to form a piece of executable code, e.g. an SQL query, an HTML page, or a script. The vulnerability is caused by part of the user input not being processed in the intended way. The attack can be countered by preprocessing the input prior to concatenation with code fragments. In our example, the input in `contents` could be preprocessed ("sanitized") by replacing `<` and `>` by the HTML entities `&lt;` and `&gt;`, respectively.

In general, there are different forms of code injection, the most popular being cross-site scripting (XSS), SQL injection, XPath injection, and dynamic evaluation. In this work, we will focus on XSS attacks like the one presented above, though the results could be applied to similar code injection attacks as well.

1.2 Programming Guidelines

A number of program analysis techniques have been proposed that directly address code injection and related attacks [7,8,9,10,5]. In our view, it is usually hard to specify code injection attacks exactly and to define what programs are subject to those attacks. As a result, such tools are useful in bug hunting, but may lack a rigorously specified and predictable behaviour.

In contrast, a programming guideline or practice can be formalized exactly, thus providing the semantic foundation for a sound procedure that can be shown to enforce the guideline. While strict compliance does not necessarily prevent all attacks, we argue that by separating the somewhat imprecise task of preventing an attack from the strict enforcement of a policy, the overall security can be improved, and the entire analysis can be simplified. As a side effect, programming guidelines help to prevent attacks already at the design time of the program. A comprehensive archive of programming guidelines used in industry is maintained by the OWASP Application Security Verification Standard Project [1].

Although one might argue that programming guidelines constitute just a special class of security properties, they are the easiest to handle in software development and quality assurance, and possibly also when it comes to legal aspects. Despite their ubiquity and importance in practice, such guidelines have rarely been an application target in works on formal sound program analysis.

In the end, the focus on safe coding practices entails a more fundamental shift in the overall security model: Instead of assuming an adversary with varying attack capabilities, we primarily target a well-intentioned, generally trustworthy programmer who occasionally makes mistakes that enable these attacks.

Enforcing a guideline with automatic methods always constitutes a delicate navigation between efficiency, accuracy, and intuitiveness. Typically, programming guidelines, while focusing on syntax, carry some semantic component and therefore are in general undecidable. Now, if the automatic method raises too many false alarms then programmers will ignore the results of the method. The tool should run efficiently to be useful during coding. Finally, the method should be predictable, i.e., there should be a well-described approximation of the guideline in question which is then decided accurately by the automatic method.

In this work we focus on a particular programming guideline used at SAP to counter code injection attacks [2,3]. Basically, this guideline requires the use of "sanitizing" functions that quote or escape characters that could otherwise cause the interpretation of parts of the strings as executable code. There is, however, no single sanitizing function that should be used for arbitrary user input; rather must one out of four such functions be selected according to the string context into which the user input is to be embedded. This makes static enforcement of the guideline a nontrivial task because we need to explore the possible string contexts as accurately as possible merely by analysing the program text.

1.3 Benefits of Type Systems

This paper demonstrates that a type system is an appropriate and sufficiently accurate analysis technique for the SAP sanitization guideline, as it is a syntactic framework that classifies phrases according to categories such as the values they compute, or their origin. Beyond the specific scenario, we argue that type systems bring a number of general advantages for the verification of coding guidelines.

As guidelines are meant to be understood by the programmer, it is natural to enforce them using a technique that builds on the familiar Java type system. From a theoretical point of view, type systems can be used to draw a clear distinction between the declarative statement of a program property with a typability relation, and its automatic verification using an inference algorithm [11]. The declarative definition of valid typing judgements simplifies the formulation of a rigorous soundness proof [12], and type derivations can act as proof certificates.

Type systems have been successfully used not only to prove data type safety, but also to enforce security requirements such as noninterference properties used in information flow security. To our knowledge, however, type systems have not yet been used specifically to implement *programming guidelines* to prevent code injection vulnerabilities.

1.4 Contributions

The goal of this paper is to provide a type system that ensures that a Java programmer follows a given programming guideline to prevent code injection attacks. The main contributions are:

1. the identification of a new subfield: using type systems for the automatic enforcement of programming guidelines;
2. the development of an expressive type system for a particular programming guideline used at SAP for the prevention of code injection, this includes the formalization of the guideline with finite state machines;
3. the development of an accompanying implementation;
4. enhancing the accuracy of type-based string analysis to come close to that of black-box analyses without certification.

Note that for a strict formalization of the security property, the type system is defined on a theoretic core language "FJEUS" in the style of Featherweight Java. The implementation, however, works on actual Java source code.

We proceed as follows: we show how a particular class of XSS programming guidelines can be formalized as a finite state machine (Section 2). In Section 3, the core language is defined, followed by the type and effect system (Section 4). In Section 5, we describe how the types can be automatically inferred. Section 6 details some highlights of the Type-Based Java String Analyzer implementation.

2 The Programming Guideline

In the SAP NetWeaver Platform, the SAP Output Encoding Framework provides XSS prevention facilities for programs that generate HTML code and have access to certain untrusted "user input" strings, like information coming from a GET request. By encoding or "sanitizing" such user-supplied input before rendering it, any inserted scripts are prevented from being transmitted in executable form. To prevent XSS attacks, The following programming guideline for a correct framework usage is specified, in which different cases need to be distinguished.

1. When a user string is output within HTML tags, a function `escapeToHtml` should be applied for output encoding.
2. When a user string is output in a JavaScript context, a function `escapeToJs` should be applied for output encoding.

The functions are provided by the framework; concrete implementations could for instance remove all HTML tags or all quotation marks from the strings. Due to limited space, we leave away two other embedding cases that apply to HTML attribute values, and the fact that the methods come in overloaded versions for different use cases. For more detailed information about the guideline with respect to the usage of the SAP Output Encoding Framework, please refer to its documentation [13].

```
public void doGet(HttpRequest request, SecureAPI api) {
  String input = request.getInputParameter();
  // -- case 1: HTML embedding --
  String s = "<body>" + api.escapeToHtml(input) + "</body>";
  api.output(s);
  // -- case 2: JavaScript embedding --
  if (showAlert) {
    api.output("<script>");
    api.output(" alert('" + api.escapeToJs(input) + "');");
    api.output("</script>");
  }
}
```

Fig. 1. Example program

2.1 Formalization of the Programming Guideline

We now make the above guideline more precise, and illustrate this with the program in Figure 1, which shall also serve as a running example for this paper.

We assume that untrusted user strings originate in the return value of a method `getInputParameter`. All strings that are derived from these return values by string operations are also considered unsafe. The only string operation we consider is concatenation with the + operator. We assume an interface `SecureAPI` that models the framework and provides the two sanitization functions `escapeToHtml` and `escapeToJs`, as well as an output function `output`.

Before being passed to `output`, any unsafe string must be passed to one of the two sanitization functions: when the string is embedded somewhere between "<script>" and "</script>", `escapeToJs` must be used, otherwise, one shall use `escapeToHtml`. The example program satisfies the guideline, because the correct sanitization function for `input` is applied depending on where it is embedded.

Although the guideline may appear relatively simple, it already imposes a number of requirements for the analysis. It is not sufficient to approximate possible string values, as the trustworthiness of a string cannot be solely derived from its value: The same value can be either a trusted literal or a malicious piece of injected code. On the other hand, a pure dataflow analysis is not enough, as the choice of the sanitization depends on triggers like <script> literals.

2.2 Output Traces and Policy Automaton

We classify the strings in the program according to their contents and origin:

- all strings coming from `getInputParameter` are assigned the class `Input`;
- `escapeToHtml` returns `C1`-classified strings, whereas `escapeToJs` returns `C2`-classified strings;
- literals have the classification `Lit`, except for the strings "<script>" and "</script>", which are classified `Script` and `/Script`, respectively.

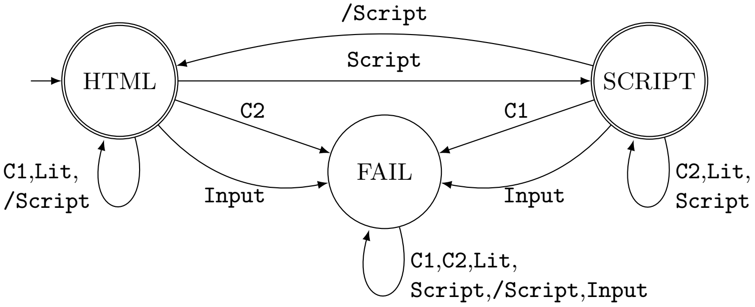

Fig. 2. Sample policy automaton

Concatenating these strings and passing them to the `output` function define the *output trace* of the program, which are words over the alphabet of classifications $\Sigma = \{\texttt{Lit}, \texttt{C1}, \texttt{C2}, \texttt{Script}, \texttt{/Script}, \texttt{Input}\}$. Our program generates two traces:

1. $\texttt{Lit} \cdot \texttt{C1} \cdot \texttt{Lit} \cdot \texttt{Script} \cdot \texttt{Lit} \cdot \texttt{C2} \cdot \texttt{Lit} \cdot \texttt{/Script}$ if $\texttt{showAlert} = \texttt{true}$
2. $\texttt{Lit} \cdot \texttt{C1} \cdot \texttt{Lit}$ if $\texttt{showAlert} = \texttt{false}$

The formalized guideline now requires that all output traces are accepted by the finite state machine (*policy automaton*) given in Figure 2. The machine contains accepting states for two modes, such that "normal mode" accepts `C1`, but not `C2` strings, while "script mode" does the reverse. A switch to "script mode" occurs whenever a `Script` string is encountered, and back to "normal mode" at `/Script`. All other cases lead to a special inescapable fail state, e.g. whenever an `Input` string is encountered. A trace is accepted if it leads to an accepting state. The machine thus accepts the traces of the example program above, but not e.g. $\texttt{Script} \cdot \texttt{C1} \cdot \texttt{/Script}$, which means the wrong sanitization function is used, or `Input`, which means a string from `getInputParameter` has been directly output, and hence has not been sanitized at all.

The example machine is kept rather simple for presentation purposes. In practice, one could use extended machines, e.g. to detect mode switches for `output("<scr"+"ipt>")`, or to handle the mentioned other sanitization cases. We could also allow a nondeterministic automaton, which would then be determinized using powerset construction. However, the policy itself, i.e. the decision whether a given trace is accepted, will always be deterministic.

We now factor the infinite set of traces by behavioural equivalence with respect to the policy automaton, and in this way obtain a finite set of equivalence classes carrying a monoid structure. Formally, let $G = (Q, q_0, \delta, F)$ be the automaton with a set of states Q, an initial state q_0, accepting states F, and a transition function δ. Two words w_1 and w_2 are equivalent if they have the same effect in each state: $w_1 \cong w_2 \iff \forall q \in Q.\ \delta(q, w_1) = \delta(q, w_2)$. The equivalence class to which a word w belongs is denoted by $[w]$. Concatenation is defined on classes by $[w_1] \cdot [w_2] = [w_1 \cdot w_2]$. Note that $[\epsilon]$ is the neutral element. The subset $\mathsf{Allowed}$ denotes those equivalence classes that contain words accepted by G.

The example automaton shown above has the following associated monoid: $\mathit{Mon} = \{[\texttt{Lit}], [\texttt{C1}], [\texttt{C2}], [\texttt{Script}], [\texttt{/Script}], [\texttt{Input}], [\texttt{C1}\cdot\texttt{Script}], [\texttt{C2}\cdot\texttt{/Script}]\}$. All of these eight classes have a different effect on the automaton, and there are no more classes. The neutral element is $[\varepsilon] = [\texttt{Lit}]$; the set of accepted classes is $\mathsf{Allowed} = \{[\texttt{Lit}], [\texttt{C1}], [\texttt{Script}], [\texttt{/Script}], [\texttt{C1} \cdot \texttt{Script}]\}$.

We also define a function $\mathsf{litword}$ that specifies the word $w \in \Sigma^*$ for a given string literal. For our example program, we have $\mathsf{litword}$("<script>")=$\texttt{Script}$, $\mathsf{litword}$("</script>")=$\texttt{/Script}$, and $\mathsf{litword}(\mathit{str})$=$\texttt{Lit}$ for all other literals str.

We assume the designer of the security guideline formalizes their requirements in the form of a finite state machine, and computes the associated monoid. Our type system is parametric with respect to a given monoid.

3 The FJEUS Language

FJEUS is a formalized and downsized object-oriented language that captures those aspects of Java that are interesting for our analysis: objects with imperative field updates, and strings. The language is an extension of FJEU [14] with strings, which itself extends Featherweight Java (FJ) [15] with side effects on a heap.

3.1 Syntax

The following table summarizes the (infinite) abstract identifier sets in FJEUS, the meta-variables we use to range over them, and the syntax of expressions:

variables: $x, y \in \mathit{Var}$	fields: $f \in \mathit{Fld}$	string literals: $\mathit{str} \in \mathit{Str}$
classes: $C, D \in \mathit{Cls}$	methods: $m \in \mathit{Mtd}$	

$$\begin{aligned}\mathit{Expr} \ni e ::= {} & x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ x_1 = x_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\ & \texttt{null} \mid \texttt{new}\ C \mid x.f \mid x_1.f{:=}x_2 \mid x.m(\overline{x}) \mid {}''\mathit{str}'' \mid x_1 + x_2\end{aligned}$$

For the sake of simplicity we omit other primitive data types and type casts, and require programs to be in let normal form. The somewhat unusual equality conditional construct is included to have reasonable if-then-else expressions without relying on booleans. The language features string literals, and a concatenation + as the only string operation. An overlined term $\overline{x}$ stands for an ordered sequence.

An FJEUS program $P = (\preceq, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ defines the following: $\preceq \in \mathcal{P}(\mathit{Cls} \times \mathit{Cls})$ is the subclass relation; $D \preceq C$ means D is a subclass of C. The functions $\mathit{fields} \in \mathit{Cls} \to \mathcal{P}(\mathit{Fld})$, $\mathit{methods} \in \mathit{Cls} \to \mathcal{P}(\mathit{Mtd})$ specify for each class C its fields and methods. A method table $\mathit{mtable} \in \mathit{Cls} \times \mathit{Mtd} \rightharpoonup \mathit{Expr}$ gives for each method of a class its implementation, i.e. the FJEUS expression that forms the method's body. We assume the formal argument variables of a method m are named x_1^m, x_2^m, etc., besides the implicit and reserved variable *this*. Only these variables may occur freely in the body of m. Alternatively, the implementation may be given directly in form of a big-step semantic relation; we call such methods *external*. A number of well-formedness conditions are imposed on these functions to ensure the usual class inheritance properties; details are given in the extended version of the paper which can be found on the first author's homepage. From now on, we assume a fixed well-formed program P.

$$\frac{(s,h) \vdash e_1 \Downarrow v_1, h_1 \;\&\; w_1 \qquad (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \;\&\; w_2}{(s,h) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, h_2 \;\&\; w_1 \cdot w_2}
\qquad
\frac{\begin{array}{c} s(x) = l \quad h(l) = (C, \_) \quad |\overline{x^m}| = |\overline{y}| = n \\ s' = [\mathit{this} \mapsto l] \cup [x_i^m \mapsto s(y_i)]_{1 \le i \le n} \\ (s', h) \vdash \mathit{mtable}(C, m) \Downarrow v, h' \;\&\; w \end{array}}{(s,h) \vdash x.m(\overline{y}) \Downarrow v, h' \;\&\; w}$$

$$\frac{\begin{array}{c} l \notin \mathit{dom}(h) \qquad w = \mathsf{litword}(\mathit{str}) \\ h' = h[l \mapsto (w, \mathit{str})] \end{array}}{(s,h) \vdash \text{"}\mathit{str}\text{"} \Downarrow l, h' \;\&\; \epsilon}
\qquad
\frac{\begin{array}{c} h(s(x_1)) = (w_1, \mathit{str}_1) \qquad h(s(x_2)) = (w_2, \mathit{str}_2) \\ l \notin \mathit{dom}(h) \qquad h' = h[l \mapsto (w_1 \cdot w_2, \mathit{str}_1 \cdot \mathit{str}_2)] \end{array}}{(s,h) \vdash x_1 + x_2 \Downarrow l, h' \;\&\; \epsilon}$$

Fig. 3. Operational semantics of FJEU

3.2 Instrumented String Semantics

A state consists of a store (variable environment or stack) and a heap (memory). Stores map variables to values, while heaps map locations to objects. The only kinds of values in FJEUS are object locations and *null*. We distinguish two kinds of objects: *ordinary objects* contain a class identifier and a valuation of the fields, while *string objects* are immutable character sequences tagged with a word w over the alphabet Σ. The state model is summarized by the following table:

locations:	$l \in \mathit{Loc}$	stores:	$s \in \mathit{Var} \rightharpoonup \mathit{Val}$
values:	$v \in \mathit{Val} = \mathit{Loc} \uplus \{\mathit{null}\}$	heaps:	$h \in \mathit{Loc} \rightharpoonup \mathit{Obj} \uplus \mathit{SObj}$
string objects:	$\mathit{SObj} = \Sigma^* \times \mathit{Str}$	objects:	$\mathit{Obj} = \mathit{Cls} \times (\mathit{Fld} \rightharpoonup \mathit{Val})$

The FJEUS semantics is defined as a big-step relation $(s,h) \vdash e \Downarrow v, h' \;\&\; w$. It means that the expression e evaluates in store s and heap h to the value v and modifies the heap to h', generating an output trace (word) $w \in \Sigma^*$.

Figure 3 shows some of the rules that define the operational semantics. We only discuss the parts that are related to strings or traces. For `let` constructs, the output traces of the subexpressions are simply concatenated. The trace of a method body execution is also the trace of the method call. String literals cause the creation of a new string object in the heap, tagged with the word given by litword. Since a literal does not produce any output, we have the empty trace ϵ here. A concatenated string $x_1 + x_2$ is tagged by concatenating the tags of the original strings. Additional functionality like string sanitization and output is provided by external methods. Implementations for these external methods, along with the full rule system, can be found in the extended version of the paper.

We call the semantics "instrumented", because the tags attached to the string objects are imaginary and do not exist during the actual program execution. Rather, they are used here for a rigorous definition of the programming guideline. The tags have an intensional meaning which is defined either by litword in the case of literals, or by the semantics of external methods. We assume these methods use the "correct" tags, e.g. `getInputParameter`() returns an `Input`-tagged string.

4 Type and Effect System

Our analysis is a type and effect system that is parametric with respect to a given policy automaton. Whenever a program is typable, it means the programmer has followed the security guideline that the automaton describes. Untypable programs either violate the guideline, or the type system is not expressive enough to show that the guideline has been followed.

4.1 Refined String Types and Class Tables

The distinction of ordinary and string objects is mirrored in the type system:

$$\mathit{Typ} \ni \tau, \sigma \ ::= \ C \mid \mathtt{String}_U \qquad \text{where} \quad U \subseteq \mathit{Mon}$$

A value typed with $\mathtt{String}_U$ intuitively means that it is a location that refers to a string object that is tagged with a word w such that $[w] \in U$. We use subsets of *Mon* rather than single monoid elements to account for joining branches of conditionals (including the conditionals implicit in dynamic dispatch).

A class table (A, M) models Java's class member types. The *field typing* $A : (\mathit{Cls} \times \mathit{Fld}) \rightharpoonup \mathit{Typ}$ assigns to each class C and field $f \in \mathit{fields}(C)$ the type of the field. The type is required to be invariant with respect to subclasses of C.

The *method typing* $M : (\mathit{Cls} \times \mathit{Mtd}) \rightharpoonup \mathcal{P}(\mathit{Typ}^* \times \mathit{Typ} \times \mathcal{P}(\mathit{Mon}))$ assigns to each class C and each method $m \in \mathit{methods}(C)$ an unbounded number of *method types* $(\overline{\sigma}, \tau, U)$, from now on written $\overline{\sigma} \xrightarrow{U} \tau$, which specify the types of the formal argument variables and of the result value, as well as the possible effects of the method (explained below). All method types assigned to a method must have the same underlying unannotated Java signature, but the string type refinements as well as the method effect may differ. This enables infinite polymorphic method types, as far as the refinements to the Java type systems are concerned. For every method type in $M(C, m)$ and each subclass $C' \preceq C$, there must be an improved method type in $M(C', m)$, where improved means it is contravariant in the argument types, covariant in the result class, and has a smaller effect set.

The polymorphism makes it possible to use a different type at different invocation sites of the same method, or even at the same invocation site in different type derivations. The extended paper contains an example program where polymorphic method types improve the precision of the analysis of the method.

4.2 Typing Rules

The declarative typing judgement takes the form $\Gamma \vdash e : \tau \,\&\, U$ where e is an expression, Γ maps variables (at least those in e) to types, τ is a type, and U is a subset of *Mon*. The meaning is that if the values of the variables comply with Γ and the evaluation of e terminates successfully then the result complies with τ, and the output written during this evaluation will belong to one of the classes in U. In particular, if $U \subseteq \mathsf{Allowed}$ then e adheres to the guideline. It suffices to perform this latter check for an entry point such as the "main" method.

$$\frac{\Gamma \vdash e_1 : \tau \,\&\, U \qquad \Gamma, x : \tau \vdash e_2 : \tau' \,\&\, U'}{\Gamma \vdash \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 : \tau' \,\&\, UU'} \qquad \frac{\overline{\sigma} \xrightarrow{U} \tau \in M(C, m)}{\Gamma, x : C, \overline{y} : \overline{\sigma} \vdash x.m(\overline{y}) : \tau \,\&\, U}$$

$$\frac{\mathsf{litword}(str) = w}{\Gamma \vdash \text{"}str\text{"} : \texttt{String}_{\{[w]\}} \,\&\, \{[\varepsilon]\}} \qquad \frac{\Gamma(x_1) = \texttt{String}_U \qquad \Gamma(x_2) = \texttt{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \texttt{String}_{UU'} \,\&\, \{[\varepsilon]\}}$$

Fig. 4. FJEUS Type System (extract)

Figure 4 only shows some of the typing rules; the full type system can be found in the extended paper. The `let` rule takes into account that first the effects of expression e_1 take place, and then the effects of expression e_2. For the concatenation of the subeffects we define $UU' = \{[w \cdot w'] \mid [w] \in U, [w'] \in U'\}$. For method calls, it suffices to choose one method type from $M(C, m)$. The type annotation for string literals relies on $\mathsf{litword}$. The type of a concatenated string is defined by concatenating the monoid elements of the two initial string types.

An FJEUS program $P = (\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ is *well-typed* if for all classes C, methods m, and method types $\overline{\sigma} \xrightarrow{U} \tau \in M(C, m)$, one can derive the typing judgement $[\mathit{this} \mapsto C] \cup [x_i^m \mapsto \sigma_i]_{i \in \{1,\ldots,|\overline{x^m}|\}} \vdash \mathit{mtable}(C, m) : \tau \,\&\, U$.

The polymorphic method types make the type system very expressive in terms of possible analyses of a given program. Each method may have many types, each corresponding to a derivation of the respective typing judgment. In different derivations, different type annotations may be chosen for new string objects and for called methods. The inference algorithm later uses context-sensitive restrictions to determine the (finite) set of suitable types for each method.

4.3 External Methods

As previously mentioned, external methods are not defined syntactically by *mtable*, but by providing the semantics directly. We can nevertheless assign method types for them, which then act as *trusted signatures*, such that the methods are considered well-typed even though no type derivation is provided. This enables the specification of trusted types for methods of the security API.

For our running example, we assume there are two classes `SecureAPI` and `HttpRequest` that contain external methods with the signatures shown below. In particular, `output` has exactly the effect given by the refinement of the string argument; exploiting polymorphism, we assign a method type for each $U \subseteq \mathit{Mon}$.

$$\begin{aligned} M(\texttt{HttpRequest}, \texttt{getInputParameter}) &= \{() \xrightarrow{\{[\texttt{Lit}]\}} \texttt{String}_{\{[\texttt{Input}]\}}\} \\ M(\texttt{SecureAPI}, \texttt{escapeToHtml}) &= \{\texttt{String}_{\{[\texttt{Input}]\}} \xrightarrow{\{[\texttt{Lit}]\}} \texttt{String}_{\{[\texttt{C1}]\}}\} \\ M(\texttt{SecureAPI}, \texttt{escapeToJs}) &= \{\texttt{String}_{\{[\texttt{Input}]\}} \xrightarrow{\{[\texttt{Lit}]\}} \texttt{String}_{\{[\texttt{C2}]\}}\} \\ M(\texttt{SecureAPI}, \texttt{output}) &= \{\texttt{String}_U \xrightarrow{U} \texttt{Void} \mid U \subseteq \mathit{Mon}\} \end{aligned}$$

4.4 Interpretation of the Typing Judgement

We now give a formal interpretation of the typing judgement in form of a soundness theorem. It relies on a *heap typing* $\Sigma : Loc \rightharpoonup Cls \uplus Mon$ that assigns to each heap location l an upper bound of the actual class found at l for ordinary objects, or a monoid element that matches the tag for string objects. Heap typings are a standard practice in type systems [11] to avoid the need for a co-inductive well-typedness definitions in the presence of cyclic heap structures.

We just briefly describe how heap typings are used for the soundness statement here; the extended paper contains a complete definition. We define a typing judgment $\Sigma \vdash v : \tau$, which means that according to heap typing Σ, the value v may be typed with τ. The judgment is lifted point-wise to stores and variable contexts: $\Sigma \vdash s : \Gamma$. The relation $\Sigma \vdash h$ establishes the connection to the heap: it asserts that for all locations l, the type $\Sigma(l)$ actually describes the object $h(l)$.

The interpretation of the judgement $\Gamma \vdash e : \tau \,\&\, U$ states that whenever a well-typed program is executed on a heap that is well-typed with respect to some typing Σ, then the final heap after the execution is well-typed with respect to some heap typing $\Sigma' \sqsupseteq \Sigma$ that is possibly larger to account for new objects that may have been allocated during the program execution.

Theorem 1 (Soundness). *Fix a well-typed program P. For all $\Sigma, \Gamma, \tau, s, h, e, v, h', w$ such that $\Gamma \vdash e : \tau \,\&\, U$ and $\Sigma \vdash s : \Gamma$ and $(s, h) \vdash e \Downarrow v, h' \,\&\, w$ and $\Sigma \vdash h$, there exists some $\Sigma' \sqsupseteq \Sigma$ such that $\Sigma' \vdash v : \tau$ and $\Sigma' \vdash h'$ and $[w] \in U$.*

The proof of the theorem can be found in the extended paper. It follows that the typability relation proves adherence to the programming guideline:

Corollary 1. *Let P be a well-typed FJEUS program. Let* `main` *be a method which takes no arguments, serves as the entry point of P, and has the implementation e. If $\vdash e : \tau \,\&\, U$ can be derived and $U \subseteq$ Allowed, then any output trace of the program is described by Allowed. By definition of Allowed, the trace is accepted by the policy automaton, thus the program follows the programming guideline.*

5 Automatic Type Inference

Since FJEUS formalizes the core of Java, we consider programs that are completely annotated with basic class type information, as is standard in Java programs. We now present an inference algorithm that automatically computes the refinements, i.e. the annotations for the `String` types as well as the effects.

5.1 Algorithmic Type Checking

The type system from Section 4 is transformed into a syntax-directed version with typing judgements $\Gamma \,;\, z \vdash e \rightrightarrows \tau \,\&\, U$. It suggests an algorithm that takes a type environment Γ, a context z (explained below) and an expression e, and computes the type τ and the effect U. The full system is given in the extended paper. It is a specialization of the declarative system and is thus sound.

```
Void doGet(HttpRequest request, SecureAPI api)
  let input = request.getInputParameter() in
  let s = "<body>" + api.escapeToHtml(input) + "</body>" in
  api.output(s)
```

Fig. 5. Example program in FJEUS

To infer the annotated class table, we create a set variable $U \subseteq \mathit{Mon}$ for each `String` field and method argument, as well as for each method effect. Side conditions on these variables U in the type system are collected as *set constraints*, which can then be solved by an external set constraint solver.

Since we are interested in inferring polymorphic method types, the question arises how many different types should be computed for a method. We propose a *context-sensitive* analysis where types are distinguished according to a call context from a finite set Cxt. Methods are analysed for a given context $z \in \mathit{Cxt}$. Whenever a submethod is called, a context transfer function ϕ is used to obtain a new context z' for which a type for called method is to be derived, if not already done. As Cxt is finite, the analysis will eventually terminate. In a sense, both Cxt and ϕ are a finite abstraction of the control flow graph of the execution.

Choosing a context is a trade-off between precision and efficiency. Following our earlier work [16], we leave the system parametric in Cxt and ϕ, so that it can be flexibly instantiated with different kinds of context sensitivity [17,18,19].

5.2 Typing the Example Program

We now show how the types of the example program (Figure 1) would be inferred, thereby showing that the program indeed adheres to the programming guideline. The FJEUS version of the first half of the program is shown in Figure 5.

The algorithm infers possible types and effects for all methods, and one can then check that the inferred effect of a top-level method, e.g. `doGet` or `main`, only contains classes from the set Allowed. The inference works as follows: the external type for `getInputParameter` gives $\texttt{String}_{\{[\texttt{Input}]\}}$ as the type for `input`. Thus, we find a matching method type for the call to `escapeToHtml`, returning a string of type $\texttt{String}_{\{[\texttt{C1}]\}}$. For the literals "<body>" and "</body>", the function litword gives the class [`Lit`]. Therefore, the concatenation produces for `s` a string type refined with $[\texttt{Lit}] \cdot [\texttt{C1}] \cdot [\texttt{Lit}] = [\texttt{Lit} \cdot \texttt{C1} \cdot \texttt{Lit}] = [\texttt{C1}]$. We can choose the respective type for `output` and get [`C1`] as the output effect of `doGet`. This is a subeffect of Allowed, hence the method indeed follows the guideline. For other typable and untypable programs, as well as a program that requires context-sensitivity, please refer to the examples provided with the implementation.

6 Implementation

We have transferred string type refinements and effects to the Java language, extended the Java type system accordingly, and implemented the context-sensitive

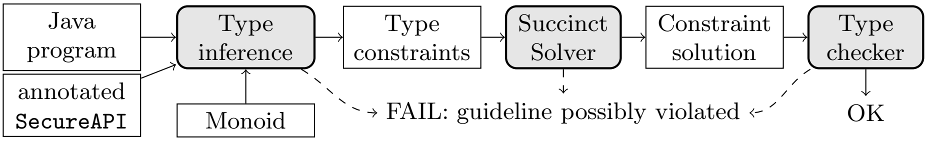

Fig. 6. Overview of the TJSA tool

type inference in a tool called *Type-Based Java String Analyzer* (TJSA). A live demo and a documentation can be found on our website [20], where the analyzer can be tried out on several provided example programs, or on custom Java code.

The tool is based on fjavac [21], a Java compiler implemented in OCaml. We have extended the standard Java type checker with refined `String` types and method output effects. In the syntax, the programmer may specify this extended type information using certain Java annotations. The `SecureAPI` class from Section 4.3 can thus be given as an annotated Java interface.

Figure 6 gives a brief overview of the way TJSA works. Given the `SecureAPI` signature, the *unannotated* example Java program from Figure 1, and the monoid from Section 2, TJSA can fully automatically infer the missing string type refinements and output effects. It generates a variable for each string type or method effect whenever no annotation can be found in the program. The analyzer collects all set constraints for these variables according to the algorithmic type system, and solves them with the Succinct Solver tool [22]. TJSA finally checks the validity of the derivation with the type variable solutions embedded. The result and the meaning of (un)typability is clearly communicated to the user.

One may add annotations by hand to support the inference or to enforce type checks, but this is generally not required. This leaves us confident that existing code using (an annotated version of) the SAP Output Encoding Framework can be verified without modifications for compliance with the guideline.

The typing algorithm works linearly on the program structure and collects constraints on type variables parametrized by contexts, therefore the main complexity aspect lies in solving the constraints. We observe that the number of type variables is bounded by the size of the program and the context set *Cxt*, and the possible values for each variable is exponential in the size of the monoid *Mon*. Apart from that, we have not yet performed an extensive tool evaluation, as our main goal was to develop the key ideas of the analysis, focusing on correctness.

7 Conclusion and Related Work

We have shown that programming guidelines are a type of security policy that addresses security vulnerabilities at the level of coding: expert knowledge on the prevention of attacks is condensed into simple principles that are easy to implement for the programmer. We have argued that type systems are a suitable form to enforce programming guidelines, as the programmer is familiar with types, their behaviour is predictable, and the correctness is easy to maintain.

In particular, we have focused on a concrete programming guideline for the correct use of the SAP Output Encoding Framework to prevent cross-site scripting attacks. The guideline has been formalized by defining valid output traces, and we have given a type string system that computes and verifies such traces. The type system is parametric in the policy automaton, and can thus readily verify other string-related guidelines that can be formalized with such automata.

Indeed, it would be interesting to identify existing sanitization frameworks that are suitable for a verification with our system. In this regard, a recent formal study of common XSS sanitization mechanisms [23] complements our work.

Once the guideline has been formalized with the instrumented semantics, its enforcement is also within the reach of other string analyses such as [24,25,26,8]. While the analysis in [24] enables a more precise approximation of string contents using context-free languages, our analysis incorporates interprocedural aspects via polymorphism and context sensitivity. Most importantly, our analysis is type-based with the advantages described in Section 1.3.

Finite automata have been proposed to express policies for resource usage events, and type-and-effect systems have been used to approximate events generated by a program [27,28]. However, the validity of inferred event histories with respect to an automaton has not been verified directly with type systems before. Nevertheless, it seems promising to elaborate to what extent the mentioned approaches can be used to formalize and verify guidelines for secure coding.

The precision of the analysis could be improved by refining class types with *regions*, as presented in our previous work [16]. Such an object-sensitive extension enables the use of different field types to different objects of the same class. Also, the connection between the sound FJEUS theory and the implemented tool for Java programs could be made more formal.

Our main medium-term goal, however, is to look at programming guidelines for security in general, and to enforce them with a type-based analysis. This may involve an even tighter integration of techniques from static analysis with type systems. The underlying principles will still be correctness of the analysis, and an implementation that is easy to use and that supports the programmer.

References

1. Open Web Application Security Project: The OWASP Application Security Verification Standard Project, `http://www.owasp.org/index.php/ASVS`
2. Wiegenstein, A.: A short story about Cross Site Scripting SAP Blog, `http://www.sdn.sap.com/irj/scn/weblogs?blog=/pub/wlg/2422`
3. Hildenbrand, P.: Guard your web applications against XSS attacks: Output encoding functionality from SAP. SAP Insider 8(2) (2007)
4. Open Web Application Security Project: The OWASP ten most critical web application security risks, `http://owasptop10.googlecode.com/`
5. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: 2006 IEEE Symp. on Security and Privacy (SP 2006), pp. 258–263. IEEE Computer Society, Washington, DC, USA (2006)

6. Wikipedia: Cross-site scripting (2011), `http://en.wikipedia.org/w/index.php?title=Cross-site_scripting\&oldid=417581017` (online accessed March 14, 2011)
7. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: 33rd Symposium on Principles of Programming Languages (POPL 2006), Charleston, SC, pp. 372–382. ACM Press, New York (2006)
8. Crégut, P., Alvarado, C.: Improving the Security of Downloadable Java Applications With Static Analysis. Electr. Notes Theor. Comp. Sci. 141(1), 129–144 (2005)
9. Wassermann, G., Su, Z.: Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In: Conf. on Prog. Lang. Design and Implementation (PLDI 2007), San Diego, CA. ACM Press, New York (2007)
10. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: 14th USENIX Security Symposium (SSYM 2005), p. 18. USENIX Association, Berkeley (2005)
11. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
13. SAP AG: SAP NetWeaver 7.0 Knowledge Center, `http://help.sap.com/content/documentation/netweaver/`
14. Hofmann, M.O., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
15. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: 1999 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999). ACM (1999)
16. Beringer, L., Grabowski, R., Hofmann, M.: Verifying Pointer and String Analyses with Region Type Systems. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 82–102. Springer, Heidelberg (2010)
17. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. PhD thesis. Carnegie Mellon University, Pittsburgh, PA, USA (1991)
18. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Conf. on Programming language design and implementation (PLDI 1994), pp. 242–256. ACM, New York (1994)
19. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. SIGPLAN Not. 39(6), 131–144 (2004)
20. Grabowski, R.: Type-Based Java String Analysis (2011), `http://jsa.tcs.ifi.lmu.de/`
21. Tse, S., Zdancewic, S.: Fjavac: a functional Java compile (2006), `http://www.cis.upenn.edu/~stevez/stse-work/javac/index.html`
22. Nielson, F., Nielson, H.R., Seidl, H.: A succinct solver for alfp. Nordic J. of Computing 9, 335–372 (2002)
23. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., Song, D.: A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 150–171. Springer, Heidelberg (2011)
24. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)

25. Annamaa, A., Breslav, A., Kabanov, J., Vene, V.: An Interactive Tool for Analyzing Embedded SQL Queries. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 131–138. Springer, Heidelberg (2010)
26. Tabuchi, N., Sumii, E., Yonezawa, A.: Regular expression types for strings in a text processing language. Electr. Notes Theor. Comput. Sci. 75 (2002)
27. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31, 23:1–23:43 (2009)
28. Skalka, C., Smith, S.: History effects and verification. In: Asian Programming Languages Symposium (November 2004)