IEICE TRANS. INF. & SYST., VOL.E96–D, NO.6 JUNE 2013

PAPER Special Section on Formal Approach

More Precise Analysis of Dynamically Generated String Expressions in Web Applications with Input Validation

Seikoh NISHITA[†a)], ***Member***

SUMMARY The string analysis is a static analysis of dynamically generated strings in a target program, which is applied to check well-formed string construction in web applications. The string analysis constructs a finite state automaton that approximates a set of possible strings generated for a particular string variable at a program location at runtime. A drawback in the string analysis is imprecision in the analysis result, leading to false positives in the well-formedness checkers. To address the imprecision, this paper proposes an improvement technique of the string analysis to make it perform more precise analysis with respect to input validation in web applications. This paper presents the improvement by annotations representing screening of a set of possible strings, and empirical evaluation with experiments of the improved analyzer on real-world web applications.

key words: *static string analysis, input validation, web applications*

1. Introduction

String analysis [1] is a static analysis targeting at dynamically generated strings in a program. The string analysis constructs a finite state automaton that approximates a set of possible strings generated for a particular string variable at a particular program location of interest; these locations are called *hotspots*.

Several applications of the string analysis have been proposed for the purpose of checking well-formedness of dynamically generated strings in web applications [2]–[6]. Since these checkers perform conservative analyses, the results of the analyses contain false positives in general: because of the imprecision in the string analysis, the checkers possibly report on an error in a hotspot where no invalid string is generated at runtime. For example, the string analysis is often ineffective in a web application with *input validation*, that is a common coding technique to check validity of an untrusted input string.

To address the imprecision, this paper proposes an improvement of the string analysis to make it perform more precise analysis in respect of the input validation. This paper makes the following contributions:

- A proposal of a condition-expression analysis and an annotation insertion. The former obtains annotations that indicate possible strings in then/else condition branches. Our condition-expression analysis supports built-in operations and functions including the equality comparison operations with type conversion.
- A presentation of the improvement of the string analysis to compose more precise automata for a hotspot with input validation by the condition-expression analysis and the annotation insertion.
- An implementation of the improved string analysis with small modification in an original string analyzer.
- An empirical evaluation of our implementation on real-world web applications.

This paper uses PHP language as the target for the string analysis. We use a PHP program in Fig. 1 throughout this paper to demonstrate our technique. The program, `login.php`, performs login authentication in a web application. The input validation of the user name and the password is performed at lines 2–7: a valid user name and a valid password are a non-empty string composed of lower case alphabetic characters and any string, respectively. The equality comparison at line 7 may involve type conversion from the string in `$name` to a boolean.

We note that this paper uses regular languages as the first class data structure instead of finite state automata for convenience of presentation of our technique.

Section 2 describes the string analysis. Sections 3 and 4 show our improvement technique and its implementation, respectively. Section 5 reports an empirical evaluation. Sections 6 and 7 discuss future works and related works, respectively. Section 8 concludes the paper.

2. The Static String Analysis

The string analysis [1] produces a regular language for a hotspot in a program. The regular language is regarded as an approximation of a set of dynamically generated strings at the hotspot. Procedure of the string analysis is as follows:

```
 1: <?php
 2:   if (preg_match('/^[a-z]*$/', $_GET['nm']))
 3:     $name = $_GET['nm'];
 4:   else
 5:     $name = false;
 6:   $pass = $_GET['ps'];
 7:   if ($name && isset($pass)) {
 8:     $pass = md5($pass);
 9:     $sql = "SELECT * FROM users WHERE
            name='$name' AND pass='$pass'";
10:     $result = mysql_query($sql);
11:   }
12:   else {  error();  }
13: ?>
```

Fig. 1 An example program, `login.php`.

Manuscript received July 9, 2012.
Manuscript revised November 10, 2012.

[†]The author is with the Department of Computer Science, Faculty of Engineering, Takushoku University, Hachioji-shi, 193–0985 Japan.
a) E-mail: snishita@cs.takushoku-u.ac.jp

DOI: 10.1587/transinf.E96.D.1278

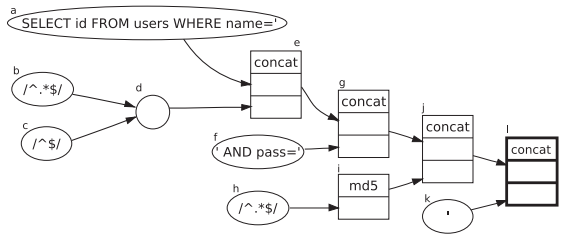

Fig. 2 A string flow graph of the string analysis.

1. Control flow analysis and hotspot detection for a target program. We simply mark, in the control flow, a hotspot as every node for a location of particular arguments of an output function call. We choose the argument `$sql` of `mysql_query()` at line 10 in `login.php` as a hotspot in this paper.
2. Construction of a string flow graph for the hotspot. The flow graph captures the data flow of strings and string operations in the program[†]. Figure 2 illustrates a string flow graph for the hotspot in `login.php`. Edges of the graph are directed def-use edges that represent the possible data flow. Nodes are divided in three groups: (1) string constants and set of strings expressed as oval nodes with regular expressions, (2) assignment operations and other join locations as round nodes, (3) string concatenations and the other string operations as rectangle nodes with operation names.
3. Regular language generation. This step comprises several sub-steps. The regular language generated from the string flow graph in Fig. 2 is represented by following Perl Compatible Regular Expression (PCRE): `/^SELECT FROM users WHERE name='.*' AND pass='[0-9a-f]{16}'$/`. We note that the analysis internally generates a regular language for each node in a string flow graph. A regular language for a string operation node is obtained by *abstraction*, which is a function on regular languages; it transforms a language for the argument of the string operation into another language for the return value.

In order for string flow graph construction at the second step, the string analysis leverages a data flow analysis that ignores condition expressions in the target program. This is a reason why the string analysis produces an imprecise regular language for a hotspot with input validations.

3. The Improvement of the String Analysis

This paper regards a condition expression as screening of possibly generated strings obtained by the string analysis: (1) a condition expression for input validation of a variable makes a smaller set of possible strings for the variable at then- (else-) branch than a set of strings for the variable at the entry point of the condition statement, and (2) when the condition expression is under non-string type, the condition expression is mapped onto string type.

As the first step of the improvement, we introduce an annotation named *sieve* which indicates the screening of a set of strings. Then, we weave tasks on the sieves into the original string analysis as follows:

1' Control flow analysis for a target program. After the control-flow graph construction, a condition-expression analysis produces sieves. Then, they are inserted into the control flow (Sects. 3.2 and 3.3)
2' Construction of a string flow graph for a hotspot. This step is performed without any modification.
3' Regular language generation. The sieves are handled as pseudo string operations, that is, we introduce extra abstractions of the sieves (Sect. 3.4)

3.1 Definitions

This section defines domain of values, a type conversion based on the section on "type juggling" in PHP manual [7] and a control flow graph as a small language.

Definition 1 (Values): $\mathcal{B}$, $\mathcal{I}$, $\mathcal{S}$ and $\mathcal{V}$ denote the universal set of booleans, integers, strings and values respectively; $\mathcal{V} \equiv \mathcal{B} \cup \mathcal{I} \cup \mathcal{S} \cup \{\text{null}\}$. L_r denotes a regular language specified by a regular expression r[††]. □

Definition 2 (Type conversion): A textual representation of integer i, and an integer represented by string s are denoted by $str(i)$ and $int(s)$ respectively. For example, $str(1)$ = `'1'`, int(`'12ab'`) = 12, and int(`'abc'`) = 0. Let T and T' be $\mathcal{B}$, $\mathcal{I}$ or $\mathcal{S}$. The function $\gamma^T : \mathcal{V} \to T$ converts a given value to a T-value. The function $\Gamma^T_{T'}$ is a lifting function of γ^T to the set T'. We abbreviate $\Gamma^T_{\mathcal{V}}$ as Γ^T. The function ${\Gamma^T_{T'}}^{-1}$ is a right inverse of $\Gamma^T_{T'}$. The function N^T corresponds to a complement function: if a given set is singleton, it makes the complement set. Otherwise it makes the universal set T.

$$\gamma^{\mathcal{S}}(a) \equiv \begin{cases} a & \text{if } a \text{ is a string} \\ \texttt{''} & \text{if } a \text{ is null or false} \\ \texttt{'1'} & \text{if } a \text{ is true} \\ str(a) & \text{if } a \text{ is an integer} \end{cases}$$

$$\gamma^{\mathcal{B}}(a) \equiv \begin{cases} a & \text{if } a \text{ is a boolean} \\ \text{false} & \text{if } a \text{ is 0, } \texttt{'0'}\text{, } \texttt{''} \text{ or null} \\ \text{true} & \text{otherwise} \end{cases}$$

$$\gamma^{\mathcal{I}}(a) \equiv \begin{cases} a & \text{if } a \text{ is an integer} \\ 0 & \text{if } a \text{ is false or null} \\ 1 & \text{if } a \text{ is true} \\ int(a) & \text{if } a \text{ is a string} \end{cases}$$

$$\Gamma^T_{T'}(A) \equiv \{\gamma^T(a) \mid a \in A\} \quad (\text{if } A \subseteq T')$$

$${\Gamma^T_{T'}}^{-1}(B) \equiv \{a \in T' \mid \gamma^T(a) \in B\} \quad (\text{if } B \subseteq T)$$

$$N^T(A) \equiv \cup_{a \in A}(T \backslash \{a\}) \quad (\text{if } A \subseteq T)$$ □

[†]In order to construct a string flow graph for a PHP program, our string analyzer obtains a dependency graph of a hotspot, then converts the graph to a string flow graph by replacing every non-string constant and non-string operation to an oval node according to its (return) type and the type conversion scheme.

[††]This paper uses PCRE as notation of regular expressions.

Definition 3 (The type conversion scheme for equation): For a given comparison operation e_1==e_2 and e_1!=e_2, the type conversion is performed by the following two steps.

step 1: converts the null comparand(s):
$e_1 \leftarrow$ '' (if e_1 is null)
$e_2 \leftarrow$ '' (if e_2 is null)
step 2: follows one of the six cases:

case 1: $e_2 \leftarrow \gamma^{\mathcal{B}}(e_2)$ (if $e_1 \in \mathcal{B}$)
case 2: $e_1 \leftarrow \gamma^{\mathcal{B}}(e_1)$ (else if $e_2 \in \mathcal{B}$)
case 3: $e_2 \leftarrow \gamma^{\mathcal{I}}(e_2)$ (else if $e_1 \in \mathcal{I}$)
case 4: $e_1 \leftarrow \gamma^{\mathcal{I}}(e_1)$ (else if $e_2 \in \mathcal{I}$)
case 5: $e_1 \leftarrow \gamma^{\mathcal{I}}(e_1); e_2 \leftarrow \gamma^{\mathcal{I}}(e_2)$ (else if e_1 and e_2 are both textual representation of integers)
case 6: otherwise, no type conversion is performed. □

Definition 4 (Control flow graph): A control flow graph (CFG) is (N, Lbl, l_i, l_f): N, Lbl, l_i and l_f are a set of nodes, a set of labels, labels of the initial node and the final node of the graph respectively. Node $n \in N$ is defined as follows:

$$\begin{aligned} n &::= l{:}v{=}rhs;\texttt{goto}\, l \mid l{:}\texttt{if}\, v\, l\, l \mid l{:}\texttt{exit} \\ rhs &::= e \mid op(e, \ldots, e) \\ e &::= v \mid c \end{aligned}$$

The symbols l, v, e, op and c denote a label, a variable, an expression, a built-in operation and a constant in $\mathcal{V}$ respectively. A node is an assignment with a goto statement (assignment-node), a condition branch (if-node) or the exit. A right-hand side of an assignment statement is an expression or a result of built-in operation application. An expression is a variable or a constant.

$N[l]$ denotes a node with label l. $prev(l)$ and $next(l)$ denote sets of labels of the previous and the next nodes of $N[l]$, respectively. Var and $Var[l]$ denote a finite set of variables appearing in a given CFG and a node $N[l]$, respectively. □

Figure 3 shows the CFG$(N, \{1, \ldots, 13\}, 1, 13)$ of the program `login.php`. The variable `$x` in the CFG is a temporary variable for conditions.

Definition 5: *Builtins* and *Filters*, respectively, denote the set of the built-in operations and ones used for the input validation, which we call a *filter operation*.

```
 1: $x = preg_match('/^[a-z]*$/',$_GET['nm']);goto 2
 2: if $x 3 4
 3: $name = $_GET['nm']; goto 5
 4: $name = false; goto 5;
 5: $pass = $_GET['ps']; goto 6
 6: if $name 7 12
 7: $x = isset($pass); goto 8
 8: if $x 9 12
 9: $pass = md5($pass); goto 10
10: $sql="SELECT id FROM users WHERE
    name='$name' AND pass='$pass'"; goto 11
11: $result = mysql_query($sql); goto 13
12: error(); goto 13
13: exit
```

Fig. 3 CFG for the example program `login.php`.

$Builtins \equiv \{\texttt{md5}, \texttt{mysql\_query}, \texttt{error}, \texttt{!}, \texttt{.}\} \cup Filters$
$Filters \equiv \{\texttt{preg\_match}, \texttt{isset}, ==\}$ □

We note that the binary operation `.` is the string concatenation. The actual definition of *Filters* in our implementation is composed of 29 filter operations [8].

3.2 The Condition-Expression Analysis

First, we define data structures utilized in our technique.

Definition 6: For an expression e and a variable v in a given CFG, a filter-operation application a is defined as follows:

$$a ::= \texttt{preg\_match}(e, v) \mid \texttt{isset}(v) \mid v{==}e \mid e{==}v$$

Fa is defined as the set of filter-operation applications appearing in a given CFG. For a given filter-operation application a, if a variable w appears at an argument of a that corresponds the position of v in the above definition, we say that w is a *target* of a. □

For example, w is the target of $\texttt{preg\_match}(v, w)$. (However, v is not.)

Definition 7: Let $a = op(e_1, \ldots, e_n)$ be a filter-operation application. We call $\underline{op}(e_1, \ldots, e_n)$ and $\overline{op}(e_1, \ldots, e_n)$ *condition expressions* of a, which mean conditions that make a true and false, respectively. Cnd is the set of all condition expressions of filter-operation applications appearing in a given CFG. If a variable w is a target of a and k is a condition expression of a, we say that w is a *target* of k. □

Though meaning of a condition expression $\underline{op}(e_1, \ldots, e_n)$ is same as one of $op(e_1, \ldots, e_n)$, we distinguish the two expressions in their usage: the former is used as an annotation.

Definition 8: The power set $Sieves$ is defined as follows:

$$\begin{aligned} &Sieves \\ &\equiv \{\top\} \cup \{K \subseteq Cnd \mid \exists w \in Var\, \forall k \in K : w \text{ is a target of } k\} \end{aligned}$$

We call $K \in Sieves$ *a sieve*. If a variable w is a target of every condition expression in a sieve K, we say that w is a *target* of K. The set $tv(K)$ is composed of variables each of which is a target of K. In addition, $var(K)$ is the set of variables appearing in K. The partial order $\sqsubseteq$, the least upper bound operation $\sqcup$ on $Sieve$ and the negation $\overline{K}$ are defined as follows:

$$\begin{aligned} &K \sqsubseteq \top \\ &K_1 \sqsubseteq K_2 \Leftrightarrow K_1 \neq \top \wedge K_2 \neq \top \wedge K_1 \subseteq K_2 \\ &K_1 \sqcup K_2 \equiv \begin{cases} K_1 \cup K_2 & \text{if } K_1 \neq \top \wedge K_2 \neq \top \\ & \quad \wedge K_1 \cup K_2 \in Sieve \\ \top & \text{otherwise} \end{cases} \\ &\overline{\top} \equiv \top \\ &\overline{K} \equiv \{\overline{op}(e_1, \ldots, e_n) \mid \underline{op}(e_1, \ldots, e_n) \in K\} \\ &\qquad \cup \{\underline{op}(e_1, \ldots, e_n) \mid \overline{op}(e_1, \ldots, e_n) \in K\} \end{aligned}$$ □

Secondly, we define three data structures obtained by

$$Cv[l] = \{\} \quad \text{if } l = l_f$$
$$Cv[l] = \bigcup_{l' \in next(l)} (Cv[l'] \backslash kill_{CV}(N[l]) \cup gen_{CV}(N[l])) \quad \text{otherwise}$$
$$kill_{CV}(l : v{=}rhs\texttt{;goto } l') = \{v\}$$
$$kill_{CV}(n) = \{\}$$
$$gen_{CV}(l : \texttt{if } x\ l_1\ l_2) = \{x\}$$
$$gen_{CV}(l : v{=}rhs\texttt{;goto } l') = \{v'\} \quad \text{if } (rhs{=}v' \text{ or } rhs{=}!v') \text{ and } v \in Cv[l']$$
$$gen_{CV}(n) = \{\} \quad \text{otherwise}$$

$$Tv[l][w] = \{\} \quad \text{if } l = l_i$$
$$Tv[l][w] = \bigcup_{l' \in prev(l)} (Tv[l'][w] \backslash kill_{TV}(w, N[l']) \cup gen_{TV}(w, N[l'])) \quad \text{otherwise}$$
$$kill_{TV}(w, l : v{=}rhs\texttt{;goto } l') = Lbl \quad \text{if } w \in Var[l]$$
$$kill_{TV}(w, n) = \{\} \quad \text{otherwise}$$
$$gen_{TV}(w, l : \texttt{if } x\ l_1\ l_2) = \{l\} \quad \text{if } w \in tv(Sc[l][x])$$
$$gen_{TV}(w, n) = \{\} \quad \text{otherwise}$$

$$\{\} \sqsubseteq Sc[l][v] \quad \text{for all } l \in Lbl,\ v \in Var$$
$$Sc[l][x] \sqsubseteq Sc[l_1][x] \quad \text{if } x \in Cv[l_1] \wedge (l : \texttt{if } y\ l_1\ l_2) \in N$$
$$Sc[l][x] \sqsubseteq Sc[l_2][x] \quad \text{if } x \in Cv[l_2] \wedge (l : \texttt{if } y\ l_1\ l_2) \in N$$
$$Sc[l][y] \sqsubseteq Sc[l'][x] \quad \text{if } x \in Cv[l'] \wedge y \in Cv[l] \wedge (l : x{=}y\texttt{;goto } l') \in N$$
$$\overline{Sc[l][y]} \sqsubseteq Sc[l'][x] \quad \text{if } x \in Cv[l'] \wedge y \in Cv[l] \wedge (l : x{=}!y\texttt{;goto } l') \in N$$
$$\{\underline{op}(e_1, \ldots, e_n)\} \sqsubseteq Sc[l'][x] \quad \text{if } x \in Cv[l'] \wedge op(e_1, \ldots, e_n) \in Fa \wedge (l : x{=}op(e_1, \ldots, e_n)\texttt{;goto } l') \in N$$
$$\{x \underline{==} \texttt{true}\} \sqsubseteq Sc[l'][x] \quad \text{if } x \in Cv[l'] \wedge rhs \notin Fa \wedge (l : x{=}rhs\texttt{;goto } l') \in N$$
$$\top \sqsubseteq Sc[l'][x] \quad \text{if } x \in Cv[l'] \wedge v \notin Cv[l] \wedge v \in var(Sc[l][x]) \wedge (l : v{=}rhs\texttt{;goto } l') \in N$$
$$Sc[l][x] \sqsubseteq Sc[l'][x] \quad \text{if } x \in Cv[l'] \wedge v \notin Cv[l] \wedge v \notin var(Sc[l][x]) \wedge (l : v{=}rhs\texttt{;goto } l') \in N$$

Fig. 4 Dataflow equations and releations.

Table 1 Result of the condition-expression analysis for `login.php`.

Node l	$Cv[l]$	$Sc[l][..]$ `$x`	`$name`	$Tv[l][..]$ `$_GET{'nm'}`	`$name`	`$pass`	$Pc[l]$
1	{}	ϕ	ϕ	ϕ	ϕ	ϕ	**true**
2	{`$x`}	{preg_match('/^[a-z]*$/',$_GET['nm'])}	ϕ	{2}	ϕ	ϕ	**true**
3	{}	ϕ	ϕ	{2}	ϕ	ϕ	$\texttt{\$x@2}$
4	{}	ϕ	ϕ	{2}	ϕ	ϕ	$\overline{\texttt{\$x@2}}$
5	{`$name`}	ϕ	{`$name`==true}	{2}	ϕ	ϕ	**true**
6	{`$name`}	ϕ	{`$name`==true}	{2}	ϕ	ϕ	**true**
7	{}	ϕ	ϕ	{2}	{6}	ϕ	`$name`@6
8	{`$x`}	{isset(`$pass`)}	ϕ	{2}	{6}	ϕ	`$name`@6
9	{}	ϕ	ϕ	{2}	{6}	{8}	`$name`@6 $\wedge$ `$x`@8
10	{}	ϕ	ϕ	{2}	{6}	ϕ	`$name`@6 $\wedge$ `$x`@8
11	{}	ϕ	ϕ	{2}	ϕ	ϕ	`$name`@6 $\wedge$ `$x`@8
12	{}	ϕ	ϕ	{2}	{6}	{8}	$\overline{\texttt{\$name@6}} \vee \overline{\texttt{\$x@8}}$
13	{}	ϕ	ϕ	{2}	{6}	{8}	**true**

standard (constraint based) data flow analyses. Figure 4 shows their data flow equations and relations. In addition, Table 1 illustrates the result of the analyses of these data structures for the CFG in Fig. 3.

Definition 9: We call a variable $x \in Var$ a condition variable at the entry point of a node n, if x is already defined at the entry point and there exists a path from n to a use of x as a condition in an if-node. $Cv[l]$ is a set of condition variables at the entry point of node $N[l]$.

For a label l and a condition variable x, $Sc[l][x]$ is a sieve of condition expressions each of which may be stored in x at the entry point of the node $N[l]$; exceptionally, if $Sc[l][x]$ is $\top$, the sieve for x is unknown.

Let a variable w be a target of a filter-operation application that is stored in a variable x at an if-node $m = (l : \texttt{if } x\ l_1,\ l_2)$. If there exists a path from m to a CFG node n, and if w is not referred to nor updated at all intermediate nodes on the path, the variable w is a *target variable from node m to node n*. For a label l and a variable w, a set of labels $Tv[l][w]$ is defined as follows: $Tv[l][w] \equiv \{l' \mid w \text{ is a target variable from node } N[l'] \text{ to node } N[l]\}$. □

Definition 10 (Pre-conditions): Pre-condition p at a node n is a sufficient condition under which the control reaches to n at runtime. Pre-condition p is defined as follows:

$$p ::= x@l \mid \overline{x@l} \mid p \wedge p \mid p \vee p \mid \textbf{true} \mid \textbf{false}$$

The literal $x@l$ and $\overline{x@l}$ mean a condition variable x at node $N[l]$ and its negation respectively. $Lit(p)$ is a set of all literals appearing in a pre-condition p. $Pc[l]$ denotes a pre-condition at the entry point of node $N[l]$. □

In our implementation, the pre-condition analysis is based on a data flow analysis. A pre-condition is recorded in the conjunction normal form, and it supports AND/OR operations, implication, and simplification with a few axioms [8].

3.3 The Annotation Insertion

In order to introduce annotations as CFG nodes, we extend the set of built-in operations, *Builtins*, with new pseudo built-in operations that work only for the string analysis:

$$Builtins' \equiv \{\underline{op} \mid op \in Filters\} \cup \{\overline{op} \mid op \in Filters\} \cup \{\texttt{union}\} \cup Builtins$$

An annotation is an assignment-node with the pseudo operations: $w = \texttt{union}(k_1, k_2, \ldots, k_n)$, which means that a variable w is a target of a sieve$\{k_1, k_2, \ldots, k_n\}$ at a CFG node, and at least a condition expression k_i in the sieve is true. We note that an annotation is simplified as $w = k_1$ in the case where $n = 1$.

Figure 5 illustrates an algorithm for the annotation insertion. The algorithm Weave, first, obtains the data structures by the condition-expression analysis (lines 2–5). Sec-

```
WEAVE()
 1    inout : CFG(N, Lbl, l_i, l_f)
 2   obtains Cv[l] for each label l ∈ Lbl
 3   obtains Sc[l][x] for each l ∈ Lbl, x ∈ Cv[l]
 4   obtains Tv[l][w] for each l ∈ Lbl, w ∈ Var
 5   obtains Pc[l] for each label l ∈ Lbl
 6  for each l ∈ Lbl
 7  do for each w referred at N[l]
 8     do for each l' ∈ Tv[l][w]
 9        do for each x@l' ∈ Lit(Pc[l])
10           do INSERTTHENANOT(w, l', Sc[l'][x])
11           for each x̄@l' ∈ Lit(Pc[l])
12           do INSERTELSEANOT(w, l', Sc[l'][x])

INSERTTHENANOT()
 1    input : w, l', sv
 2  let {k_1, ..., k_n} = sv
 3  for each k_i
 4  do if k_i == (e==w)
 5        then k_i ← (w==e)
 6   insert w = union(k_1, ..., k_n) at the exit point to then clause of N[l']
```

Fig. 5 The sieve analysis.

ondly, the algorithm inserts every annotation for a sieve $Sc[l'][x]$ and/or its negation into the exit point to then/else branch of an if-node $N[l']$ under a condition, which avoids inserting a part of redundant annotations (lines 6–12). The condition is derived from following two observations: (1) if a target variable w does not appear at then/else branch of an if-node n, no annotation for w is required at n. (2) if a target variable w is updated before use of w in then/else branch of an if-node n, no annotation for w is required either. The former reflects the test of existence of a literal in the pre-condition in the algorithm. On the other hand, the latter is reflected by $Tv[..][..]$ for target variables.

INSERTTHENANOT, first, normalizes a given sieve so that a target variable of each equality condition in the sieve appears as the first argument. Second, it makes an annotation from the sieve. Third, it inserts the annotation at the exit point to the then clause of a node. We note that the annotation insertion at the line 6 is abbreviated: the additional nodes are generally required according to the definition of CFG. On the other hand, INSERTELSEANOT similarly inserts an annotation at the exit point to the else clause.

With focus on possible execution paths and their conditions in a CFG, the technical report [8] proves the correctness of WEAVE: if an annotation $w = \texttt{union}(k_1, \ldots, k_n)$ is inserted at a branch of an if-node, and if the control reaches to the branch, then at least one condition expression k_i in the annotation is true at runtime.

Accuracy of the pre-conditions for $Pc[l]$ leads to precision and overhead of the improved string analysis. If a pre-condition is necessary and sufficient condition to reach control at a node, and if the pre-condition is simple with respect to the number of literals, WEAVE inserts requisite minimum annotations. On the other hand, if a pre-condition is not necessary, but a sufficient condition, the algorithm may miss inserting a necessary annotation, leading to imprecise string analysis. If a pre-condition has a redundant literal,

```
1: $x = preg_match('/^[a-z]*$/',$_GET['nm']); goto 2
2: if $x a 4
a: $_GET['nm'] = preg_match('/^[a-z]*$/', $_GET['nm']);
    goto 3
3: $name = $_GET['nm']; goto 5
4: $name = false; goto 5;
5: $pass = $_GET['ps']; goto 6
6: if $name b 12
b: $name = $name==true; goto 7
7: $x = isset($pass); goto 8
8: if $x c 12
c: $pass = isset($pass); goto 9
9: $pass = md5($pass); goto 10
10: $sql="SELECT id FROM users WHERE
    name='$name' AND pass='$pass'"; goto 11
11: $result=mysql_query($sql); goto 13
12: error(); goto 13
13: exit
```

Fig. 6 Annotated CFG for the example program `login.php`.

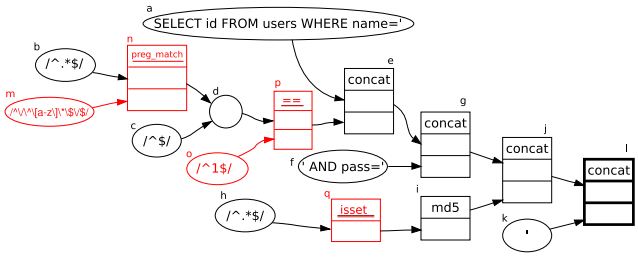

Fig. 7 The string flow graph of the annotated CFG.

WEAVE may insert unnecessary annotation, which does not affect to the improvement of the string analysis, but makes additional overhead in the string analysis.

Figures 6 and 7 illustrate annotated CFG from the original CFG in Fig. 3 and a string flow graph for the annotated CFG, respectively. The annotation insertion reflects a string flow graph generated by the string analysis: the additional nodes `m`, `n`, `o` and `p` correspond to the input validation to the variable `$name`. On the other hand, the node `q` corresponds to the variable `$pass`. The node `m` represents the singleton language of the string `'/^[a-z]*$/'`. The node `o` represents a singleton language of `'1'` that is converted from the boolean true at line 7 in `login.php`.

3.4 Abstractions of the Extended Operations

This section describes each abstraction $(\!|op|\!)$ of every extended operation op that is a part of an annotation inserted by the algorithm WEAVE.

Definition 11: For given languages $L, L' \subseteq \mathcal{S}$, abstractions of extended operations are defined as following expressions.

$$(\!|\texttt{union}|\!)(L, L') \equiv L \cup L'$$
$$(\!|\underline{\texttt{preg\_match}}|\!)(L, L') \equiv L' \cap \cup_{r \in L}(L_r)$$
$$(\!|\texttt{preg\_match}|\!)(L, L') \equiv L' \cap \cup_{r \in L}(\overline{L_r})$$
$$(\!|\underline{\texttt{isset}}|\!)(L) \equiv L, \quad (\!|\overline{\texttt{isset}}|\!)(L) \equiv L \cap \{\texttt{''}\}$$
$$(\!|\underline{\texttt{==}}|\!)(L, L') \equiv L \cap \cup_{i=1}^{6} \texttt{EQ}_i(L')$$

$$(\!|\underline{==}|\!)(L, L') \equiv L \cap \cup_{i=1}^{6} \overline{\mathtt{EQ}_i}(L')$$

where

$$\mathtt{EQ}_1 \equiv \Gamma_{\mathcal{B}}^{\mathcal{S}} \circ \Gamma^{\mathcal{B}} \circ \Gamma^{\mathcal{S}^{-1}}, \quad \overline{\mathtt{EQ}_1} \equiv \Gamma_{\mathcal{B}}^{\mathcal{S}} \circ N^{\mathcal{B}} \circ \Gamma^{\mathcal{B}} \circ \Gamma^{\mathcal{S}^{-1}}$$
$$\mathtt{EQ}_2 \equiv \Gamma^{\mathcal{S}} \circ \Gamma^{\mathcal{B}^{-1}} \circ \Gamma_{\mathcal{B}}^{\mathcal{S}^{-1}}, \overline{\mathtt{EQ}_2} \equiv \Gamma^{\mathcal{S}} \circ \Gamma^{\mathcal{B}^{-1}} \circ N^{\mathcal{B}} \circ \Gamma_{\mathcal{B}}^{\mathcal{S}^{-1}}$$
$$\mathtt{EQ}_3 \equiv \Gamma_{\mathcal{I}}^{\mathcal{S}} \circ \Gamma^{\mathcal{I}} \circ \Gamma^{\mathcal{S}^{-1}}, \quad \overline{\mathtt{EQ}_3} \equiv \Gamma_{\mathcal{I}}^{\mathcal{S}} \circ N^{\mathcal{I}} \circ \Gamma^{\mathcal{I}} \circ \Gamma^{\mathcal{S}^{-1}}$$
$$\mathtt{EQ}_4 \equiv \Gamma^{\mathcal{S}} \circ \Gamma^{\mathcal{I}^{-1}} \circ \Gamma_{\mathcal{I}}^{\mathcal{S}^{-1}}, \overline{\mathtt{EQ}_4} \equiv \Gamma^{\mathcal{S}} \circ \Gamma^{\mathcal{I}^{-1}} \circ N^{\mathcal{I}} \circ \Gamma_{\mathcal{I}}^{\mathcal{S}^{-1}}$$
$$\mathtt{EQ}_5 \equiv id, \qquad \overline{\mathtt{EQ}_5} \equiv N^{\mathcal{S}}$$
$$\mathtt{EQ}_6 \equiv id, \qquad \overline{\mathtt{EQ}_6} \equiv N^{\mathcal{S}} \qquad \square$$

The abstraction (|preg_match|) regards the first argument L as a set of PCREs, and obtains the set union of regular languages for all expressions in L. The abstraction (|isset|) is the identity function, since all non-null values are mapped to the universal set of strings by the type conversion. On the other hand, the abstraction (|$\overline{\texttt{isset}}$|) obtains set-intersections of L and the singleton set of the empty string, because the null converted to the empty string.

The abstraction of the equation comparison regards the second argument L' as a screening of possible strings L according to the type conversion scheme of Definition 3, because the annotation insertion normalizes each annotation of the equation comparison to the form of $w = (w{==}e)$. The step 1 in the type conversion scheme is performed by the string analysis itself, i.e. the analysis converts the null value to the empty string. Though all cases in step 2 of the type conversion scheme have the precedence of its application, we give approximation of the abstraction by a simple integration of the results in all cases, which correspond to the auxiliary functions $\mathtt{EQ}_n()$ $(i = 1, 2, \ldots, 6)$.

The function $\mathtt{EQ}_1()$ is derived from the case 1: according to the type conversion scheme, e_2 is converted to a boolean at runtime. First, $\mathtt{EQ}_1()$ obtains original values from a given set of strings by $\Gamma^{\mathcal{S}^{-1}}$. Second, it converts the original values to booleans by $\Gamma^{\mathcal{B}}$. Third, it converts the booleans to strings by $\Gamma_{\mathcal{B}}^{\mathcal{S}}$. On the other hand, the function $\mathtt{EQ}_2()$ obtains strings for e_1 that can be converted to the boolean. First, $\mathtt{EQ}_2()$ restores booleans for e_2 by $\Gamma_{\mathcal{B}}^{\mathcal{S}^{-1}}$. Second, it obtains values that can be converted to the booleans by $\Gamma^{\mathcal{B}^{-1}}$. Third, it converts the values to strings by $\Gamma^{\mathcal{S}}$.

The functions $\overline{\mathtt{EQ}_i}$ correspond with cases where an equation is not satisfied. The definitions of $\mathtt{EQ}_i$ and $\overline{\mathtt{EQ}_i}$ are similar each other, but the latter involves the function N^T.

We note that the auxiliary functions and the abstraction of the (not-) equality comparison are simplified by set equation with Definition 2.

3.5 The String Analysis of the Sample Program

The string analysis produces a regular language for each node in the string flow graph. Table 2 shows regular expressions that denote the regular languages produced by the string analysis for nodes `n`, `d`, `p`, `q` and `i`.

The possible strings of the variable `$name` in line 10 of `login.php` correspond to the node `d` and `p` in the original and improved string flow graphs respectively. The additional nodes in the improved string flow graph enhance precision of the analysis results. The regular expression for node `p` is obtained by the following set equations:

$$\begin{aligned}
&(\!|\underline{==}|\!)(L_r, \{\texttt{'1'}\}) && \text{(where } r = \texttt{/\^{}[a-z]*\$/}\text{)}\\
&= L_r \cap \cup_{i=1}^{6} \mathtt{EQ}_i(\{\texttt{'1'}\}) &&\\
&= L_r \cap (\{\texttt{'1'}\} \cup \overline{\{\texttt{''}, \texttt{'0'}\}} \cup \{s' \mid \texttt{'1'} = str(int(s'))\}) &&\\
&= L_{r'} && \text{(where } r' = \texttt{/\^{}[a-z]+\$/}\text{)}
\end{aligned}$$

On the other hand, the possible strings of the variable `$pass` correspond to the node `i` in both of the string flow graphs. The additional nodes do not work as screening of the possible strings in this case.

Table 2 Regular expressions for nodes in the string flow graphs.

Node n	Languages for nodes in the original graph	Languages for nodes in the improved graph
n		`/^[a-z]*$/`
d	`/^.*$`	`/^[a-z]*$/`
p		`/^[a-z]+$/`
q		`/^.*$/`
i	`/^[0-9a-z]+$/`	`/^[0-9a-z]+$/`

3.6 Soundness Issues

Let L_1 and L_2 be possible strings for a variable obtained by the original and improved string analysis respectively. We expect the inclusion relation $L_2 \subseteq L_1$, because the idea of the improvement is screening of L_1. However, the inclusion relation is not satisfied, when the annotation insertion draws additional cycle with string operations in a string flow graph; our technique leverages the string analysis, which performs the character-set approximation [1] for such cycles as originally proposed. As a result, the additional cycles and the character-set approximation break the inclusion relation [8].

4. Implementation

In order to implement our technique, we utilized an original string analyzer, which we had developed in a former work. The original string analyzer is composed of Pixy [9], [10] as the front-end and Java String Analyzer (JSA) [11] as the back-end. Pixy performs dependency analysis for PHP programs. On the other hand, JSA executes the string analysis with string flow graphs transformed from dependent graphs.

We have implemented the improved string analyzer as follows: (1) we retrofitted inline expansion, redundant CFG node elimination, the condition-expression analysis and the annotation insertion into Pixy, and (2) we installed the new abstractions into JSA. We note that the inline expansion is limitedly applied, when input validation is coded as user-defined functions in a given CFG. On the other hand, the redundant CFG node elimination [8] simplifies complex condition branches in a CFG produced by Pixy for a condition expression with boolean AND/OR operations. The implementation required no drastic change for source codes in

Table 3 The target web applications and the experimental result.

Name	Version	Main Files	Lines	Hotspots	Improvement status			CFG of Analyses		Time of Analyses (s)	
					same	included	corrupt	original	improved	original	improved
Ajchat	0.10	16	3,412	28	23	5	0	3,296	3,327	98.5	106.8
aphpkb	0.95.4	20	8,964	39	29	10	0	8,878	9,145	225.7	241.7
BaconMap	0.7.0	32	33,066	195	153	42	0	47,349	46,585	531.7	727.4
Cacti	0.8.7a	28	70,947	1,955	919	884	152	290,033	294,257	202,433.8	239,985.1
Chipmunk Pwngame	1.0	13	887	22	22	0	0	714	714	58.0	60.7
Easy Banner Free		4	648	21	17	4	0	2,054	2,440	44.2	57.1
Jetbox CMS	2.1	12	42,738	204	155	33	16	44,211	42,845	9,315.5	25,983.0
jurpopage	0.2.0	27	97,430	158	0	158	0	21,167	21,164	248.3	282.1
MyEasyMarket	4.1	10	2,565	148	54	94	0	2,539	3,762	111.4	176.2
PHP Bible Search	0.99	1	314	4	2	2	0	148	148	7.7	9.9
phpCheckZ	1.1.0	20	25,788	42	6	32	4	30,176	29,105	428.1	647.3
RiotPix	0.61	7	1,331	25	19	6	0	1,046	989	62.6	64.5
sendcard	3.4.1	2	11,172	9	9	0	0	4,537	4,801	78.3	83.3
smbind	0.4.8	14	1,963	259	239	11	0	8,046	8,227	188.8	212.3
TigerPhpNewsSystem	1.0b	3	6,846	157	132	25	0	11,564	11,010	11,312.0	13,283.6
TinyBB	1.2.4	2	4,089	72	65	7	0	1,105	1,060	52.9	60.6
Utopia Newspro	1.4.0	18	5,953	206	132	74	0	15,487	19,648	221.2	231.5
WSN Guest	1.24	20	7,656	39	34	4	1	68,973	70,064	1,384.5	1,707.3

the original string analyzer, since our technique keeps the framework of the original string analysis.

Our original and improved string analyzers are limited by Pixy in respect of the syntax of PHP language. First, Pixy supports only the syntax of PHP4. Second, the dependency analysis in Pixy does not support a part of PHP syntax: class and object, dynamic definitions of user-defined functions, variable variables and so on. If the analysis failed, Pixy does not analyze the dependency analysis correctly. These limitation directly affects both of the string analyzers.

Though this paper uses regular languages as the first class data structure, JSA obtains finite state automata instead of regular languages. We leveraged the automaton package of JSA for the implementation of the new abstractions. In addition, we implemented *int*() in ⟪==⟫ with a finite state transducer in order to parse textual representation of integers. Our implementation of ⟪`preg_match`⟫ simply performs union of automata for each PCRE string accepted by a given automaton. If the given automaton accepts infinite strings, the abstraction makes the automaton accepting any string. Our PCRE parser is based on the regular expression parser provided by the automaton package that does not support almost all of the extension in PCRE syntax [8].

5. Empirical Evaluation

For an empirical evaluation, we selected 18 real-world web applications. These applications have several main program files as the entry points of web access. We set the analysis target to all of them except ones written in PHP5 syntax: we excluded a main file from the target, if our string analyzer terminated because of PHP5 syntax usage, since Pixy only supports PHP4 syntax. Moreover, because of the limitation of Pixy, our string analyzer often fails to identify an including file whose name is specified by an expression rather than a string constant in a target program. To address this problem, we carefully replaced such expression with a static string. The experiment was performed on a 2.56 GHz Xeon with 48 GB RAM running Linux. In the experiment, we focus upon function calls to send SQL queries as hotspots of the string analysis. We evaluate the original and improved string analyzers by the size of control flow graphs, the execution time and comparison of finite state automata in respect of the inclusion relation of accepted languages.

Table 3 shows the 18 web applications and the experimental result: the name and the version, the number of main files and total lines, the number of hotspots, the improvement status, the total number of nodes (size) in generated control flow graphs, and total execution time. The improvement status is divided in three parts: "same" means the number of hotspots where the two string analyzers obtain same automata, "included" means the number of hotspots where the improved string analyzer obtains more precise result than the original analyzer, and "corrupt" means the number of hotspots where the improve string analyzer obtains imprecise result because of the character set approximation. The sub-columns "original" and "improved" mean the size of CFGs and the total time of the original and the improved string analyzers, respectively.

The analysis result in respect of the improvement status is improved in 16 applications. On the other hand, our technique does not work fine in the two applications, Chipmunk Pwngame and sendcard. The improved string analyzer contrarily obtains worse results for 173 hotspots in four applications: the imprecision in the 169 hotspots arose from the character-set approximation with our improvement. The other four results involved broken dependent graphs generated by Pixy.

The CFG size increases and decreases among the experiment of each application, because of inline expansion and the redundant node elimination. The growth rate on the CFG size by the improvement is from 94.6% to 148.2%. The total execution time is increased by the improvement: the growth rate is from 103.1% to 278.9%. The technical

report [8] of this paper illustrates more information on execution time: it shows that the time of data flow generation occupies the total time of the analyses, while the time of the condition-expression analysis and annotation insertion do not occupy the total time in the experiment.

6. Future Works

Section 3.3 described that the improvement technique leverages a pre-condition analysis, which affects the precision and the overhead of the improvement. A study on a suitable pre-condition analysis is a future work.

The function *int*() in Definition 2 converts the longest number prefix of a given string to an integer. However, it does not support the textual representation of an integer with an exponent like "1e0", and one in the hexadecimal format like "0x1". For precise analysis of an equation like "1e0 == 0x1", (1) *int*() should support both of the textual representation format in PHP, and (2) $\mathtt{EQ}_5$ in Definition 11 should support the equation whose comparands are converted to integers. Both of them are future works.

Static single assignment (SSA) simplifies our technique, because the data structures *Cv*[], *Sc*[][] and *Tv*[][] would not depend on node labels. However, if an implementation of the string analysis requires SSA form of a given CFG, our technique has to restore a modified CFG into SSA form again after the annotation insertion [8]. The restoration of SSA form with our technique is a future work.

7. Related Works

Minamide proposed another string analysis that composes context free grammars [12]. Wassermann and Su applied his technique to a static analysis to find the SQL injection vulnerability [13]. In the paper, they touched upon an analysis of input validations by intersection of a regular expression and intermediate data structure. In contrast, Ono et al. proposed an analysis of a program including input validations [14]. Their analyzer constructs constraints of security labels for a target program, then obtains regular language of a hotspot by constraint solver.

Compared with their analyses, our technique is based on the framework of the string analysis, and enhances it by small modification. Our technique, moreover, supports the type conversion involved in the equality comparison operations. We note that the sample program `login.php` in this paper is a model of a snippet in paper [13], which Wassermann's technique did not work fine in their experiment.

8. Conclusions

This paper described an improvement of the string analysis in respect of input validations in web applications. We described the improvement and its implementation by small modifications of an original string analyzer with the condition-expression analysis and the annotation insertion, and reported an experiments with the real world web applications. The experimental results indicate that our improved analyzer mostly produces more precise automata in most case.

Acknowledgements

We thank the anonymous reviewers of Formal Approach 2012 for their valuable comments on earlier drafts of this paper.

References

[1] A.S. Christensen, A. Møller, and M.I. Schwartzbach, "Precise analysis of string expressions," Proc. SAS '03, pp.1–18, 2003.
[2] A.S. Christensen, A. Møller, and M.I. Schwartzbach, "Extending Java for high-level web service construction," ACM Trans. Prog. Lang. Syst., vol.25, no.6, pp.814–875, 2002.
[3] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," Proc. ICSE '04, pp.645–654, 2004.
[4] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," ACM Trans. Softw. Eng. Methodol., vol.16, no.4, p.14, 2007.
[5] W.G.J. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing SQL-injection attacks," Proc. ASE '05, pp.174–183, 2005.
[6] C. Kirkegaard and A. Møller, "Static analysis for Java Servlets and JSP," Proc. SAS '06, pp.6–10, 2006.
[7] The PHP Group, PHP Manual, Feb. 2012.
[8] S. Nishita, "Technical report on more precise analysis of dynamically generated string expressions in web applications with input validation," Tech. Rep., Takushoku Univ., http://www.cs.takushoku-u.ac.jp/Labo/snishita/papers/TR-snishita-2012-01-revised.pdf, Nov. 2012.
[9] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," Proc. 2006 IEEE Symposium on Security and Privacy, pp.258–263, 2006.
[10] International Secure Systems Lab, "Pixy: XSS and SQLI scanner for PHP programs," http://pixybox.seclab.tuwien.ac.at/pixy/documentation.php, accessed July 18, 2010.
[11] BRICS, "Java string analyzer," http://www.brics.dk/JSA/, 2003, accessed July 10, 2010.
[12] Y. Minamide, "Static approximation of dynamically generated web pages," Proc. WWW '05, pp.432–441, 2005.
[13] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," Proc. PLDI '07, pp.32–41, 2007.
[14] K. Ono, N. Tabuchi, and T. Tateichi, "Systems, methods and computer program products for string analysis with security labels for vulnerability detection," United States Patent 7530107B1, 2009.

Seikoh Nishita received a Dr. Eng. degree from Osaka University, Japan in 1999. Presently he is an associate professor of Faculty of Engineering of Takushoku University. He received IEICE Best Paper Award in 1998. His research interests include static and dynamic analysis of web applications, and countermeasures against web applications vulnerability. He is a member of JSSST, IPSJ, and ACM.