Abstract parsing: static analysis of dynamically generated string output using LR-parsing technology

Kyung-Goo Doh[1], Hyunha Kim[1], and David A. Schmidt[2]

[1] Hanyang University, Ansan, South Korea
[2] Kansas State University, Manhattan, Kansas, USA

Abstract. We combine LR(k)-parsing technology and data-flow analysis to analyze, in advance of execution, the documents generated dynamically by a program. Based on the document language's context-free reference grammar and the program's control structure, the analysis predicts how the documents will be generated and parses the predicted documents. This simultaneous analyze-and-parse strategy gives better precision than an analyze-then-parse strategy because it computes *abstract parse stacks*, which encode a generated document's context-free structure. The technique is implemented in Objective Caml and has been tested to statically validate the syntax of HTML documents dynamically generated by PHP programs.

1 Introduction

Scripting languages like PHP, Perl, Ruby, and Python use strings as a "universal data structure" to communicate values, commands, and programs. For example, one might write a PHP script that assembles within a string variable an SQL query or an HTML page or an XML document. Typically, the well-formedness of the assembled string is verified when the string is supplied as input to its intended processor (database, web browser, or interpreter), and an incorrectly assembled string might cause processor failure. Worse still, a malicious user might deliberately supply misleading input that generates a document that attempts a cross-site-scripting or injection attack.

As a first step towards preventing failures and attacks, the well-formedness of a dynamically generated, "grammatically structured" string (document) should be checked with respect to the document's context-free *reference grammar* (for SQL or HTML or XML) before the document is supplied to its processor. Better still, the document generator program *itself* should be analyzed to validate that all its generated documents are well formed with respect to the reference grammar, much like an application program is type checked in advance of execution. Such an analysis should indicate the grammatical structure of the generated documents so that there is clear indication of those positions within the document where unsanitized data or potential attacks might be inserted. This level of precision goes further than what is provided by existing regular-expression-based analysis techniques.

In this paper, we employ LR(k)-parsing technology and data-flow analysis to *analyze* statically a program that dynamically generates documents as strings, and at the same time, *parse* the dynamically generated strings with the context-free reference grammar for the document language. The analyze-and-parse strategy computes *abstract parse stacks* that remember the context-free structure of the strings.

Our approach requires that the reference grammar is LR(k) and that the program to be analyzed is annotated with "hot spots" (those program points where critically important strings are generated). Starting from each hot spot, the static analysis conducts demand-driven *abstract parsing* of the string assembled at the hot-spot.

We have implemented an abstract-parsing analyzer and have applied it to PHP programs that dynamically generate strings of SQL queries and HTML documents.

The paper is organized as follows: The next section reviews research on string analysis, and Section 3 summarizes our contributions. Sections 4 and 5 present a motivating example and the key concepts behind abstract parsing. Section 6 surveys the worklist algorithm that implements the flow analysis, and Sections 7 and 8 discuss technical issues regarding input variables and string-update operations. Section 9 sketches our implementation, and Section 10 concludes.

2 Previous efforts

Because of the popularity of document generators and the dangers that they introduce, there exist a variety of approaches for validating document generators and their generated documents:

Parsing the generated strings: From the perspective of the document processor, it is important to protect oneself from malicious incoming queries. Along this line, Wassermann and Su [20] studied the format of command-injection attacks on SQL servers and devised an SQL reference grammar with annotations that identify in the grammar the positions where injection attacks might be inserted. A parser based on the grammar is inserted as a front-end filter to the SQL database — every incoming query must be parsed before it proceeds to the database.

Document-generation languages: One might limit malformed document generation by restricting the language used to write document-generator programs. XDuce [10, 11] is an ML-like language customized to building XML documents that are struct-like values statically typed with regular-expressions. The typing ensures that dynamically generated documents conform to "templates" defined by the document types. In a similar vein, <bigwig> [3] and JWIG [6] are domain-specific languages for XHTML-document generation. JWIG, an extension of Java, provides Java-encoded templates to generate documents. An accompanying static analyzer validates regular-expression well-formedness of the assembled documents.

Thiemann [18] studied the problem of inferring string-data types that are exactly the reference grammar's nonterminals: His extension of ML's type checker generates a set of typing constraints, expressed as grammar rules, for the strings generated by a program and then checks containment of the constraint-set language within the reference-grammar language with Early's parsing algorithm, searching for grammar nonterminals that are fixed points (solutions) to the constraint set.

Regular-expression-based static analysis: Checking context-free grammar inclusion is costly, so analyses based on regular expressions are typically employed. One example is Christensen, et al.'s string analyzer [5], which extracts from a Java program a set of data-flow equations for the generated strings, treating the equations as a context-free grammar. Rather than check for context-free language inclusion, the flow equations are overapproximated into a regular grammar, using a conversion due to Mohri and Nederhof. Queries about grammatical well-formedness are posed as regular expressions, and finite-state machinery decides the answers.

Using Christensen's string analyzer with a context-free-language reachability algorithm, Wasserman, et al.[19] devised a static analysis that type checks dynamically generated SQL queries in Java database applications. Kirkegaard and Møller [14] adapted Christensen's work and Knuth's balanced grammars to check whether the approximated regular grammar conforms to a balanced XML grammar, statically predicting generated XML documents to be well-formed and valid.

Minamide's analysis [15] also uses Christensen's string analyzer and extracts a flow-equation set for a string expression, treating the equation set as if it were a context-free grammar. But the analysis does not solve the equation set — instead, it generates instances of strings that act as examples and counter-examples. The novelty is the application of finite-state-automata transducers to revise the flow equations due to string-update operations embedded in the program. The transducers are also used to sanitize suspect user input before it is injected into a dynamically generated document. Minamide and Tozawa [16] developed exponential-time algorithms that validate a context-free grammar against a subclass of balanced context-free grammars, which can be used to validate dynamically generated XML documents.

Choi, et al. [4] used abstract-interpretation with heuristic widening to devise a string analyzer that handles heap variables and context sensitivity. However, its regular-expression-based machinery shares the same limitations with earlier efforts.

Flow-analysis techniques: When a user supplies malicious input data for inclusion into a dynamically generated document, a flow analyzer might track the input's flow and determine whether unsanitized input is injected into a dynamically generated document. Along these lines, Xie and Aiken [22] devised and applied an interprocedural flow analyzer that detects potential SQL injection errors in PHP programs. Jovanovich, et al., [13] implemented a tool with similar aims.

Combining the regular-expression and flow-analysis approaches are Wassermann and Su [21], who use Minamide's approach to extract data-flow equations from a program. They then annotate the flow equations as to which strings are untrustworthy so that solving the equations implements a data-flow analysis that tracks potential injection errors.

3 Our contribution

Our work means to complement these approaches by improving their precision:

1. We use the data-flow equations extracted from a program as a higher-order schema from which we generate first-order flow equations that *calculate the parse stacks generated when the dynamically generated strings are parsed (by the context-free reference grammar).* The solved equations convey context information more precise than that given by regular-expression techniques.
2. We cannot retain *all* parse information and also ensure termination, so we "fold" "repeating" parse stacks into single-entry, single-exit graphs (with cycles).
3. Rather than implement string-update operations as f.s.a.-transductions on the original flow equation set (cf. [15]), we use an *invariance property* for string updates, which means a string can be updated only if the outcome of the string's LR-parse is left unaltered.

It is easy to envision how our abstract parsing technique can be augmented by semantic-processing functions [2] so that a Xie-and-Aiken or Wassermann-and-Su tainting analysis can be conducted along with the abstract parse.

4 Motivating example

We can compare the approaches just surveyed with a small example. Say that a script must generate an output string that conforms to this grammar,

$$S \rightarrow \mathtt{a} \mid \mathtt{[}\, S \,\mathtt{]}$$

where S is the only nonterminal. (HTML, XML, and SQL are such bracket languages.) The grammar is LR(0), but it can be difficult to enforce even for simple programs, like the one in Figure 1, left column. Perhaps we require this program to print only well-formed S-phrases — the occurrence of `x` at "`print x`" is a "hot spot" and we must analyze `x`'s possible values.

An analysis based on type checking assigns types (reference-grammar nonterminals) to the program's variables. The occurrences of `x` can indeed be datatyped as S, but `r` has no data type that corresponds to a nonterminal.

An analysis based on regular expressions solves flow equations shown in Figure 1's right column in the domain of regular expressions, determining that the hot spot's ($X3$'s) values conform to the regular expression, $\mathtt{[}^* \cdot \mathtt{a} \cdot \mathtt{]}^*$, but this does not validate the assertion.

```
x = 'a'                  X0 = a
r = ']'                  R = ]
while ...                X1 = X0 ⊔ X2
  x = '[' . x . r        X2 = [ · X1 · R
print x                  X3 = X1
```

(Read . as an infix string-append operation.)

Fig. 1. Sample program and its data-flow equations

A grammar-based analysis does not solve the flow equations, but treats them instead as a set of grammar rules. The "type" of `x` at the hot spot is $X3$. Next, a semi-decidable language-inclusion check tries to prove that all $X3$-generated strings are S-generable.

Our approach solves the flow equations in the domain of *parse stacks* — $X3$'s meaning is the *set of LR-parses* of the strings that might be denoted by `x`.

Assume that the reference grammar is LR(k); we first calculate its LR-items and build its parse ("*goto*") controller; see Figure 2. (This example, and the others in this paper, are LR(0) for simplicity.) For review, the Figure displays an example parse.

We interpret the flow equations in Figure 1 as *functions* that map an input parse state to (a set of) output parse stacks. Figure 3 defines the collecting interpretation, but the informal development of the program in Figure 1 conveys well the intuitions:

The demand in Figure 1 to analyze the hot spot at $X3$ generates the function call, $X3(s_0)$, where s_0 is the start state for parsing an S-phrase. The flow equation, $X3 = X1$, generates the function,

$$X3(s_0) = X1(s_0)$$

which itself demands a parse of the string generated at point $X1$ starting from state s_0:

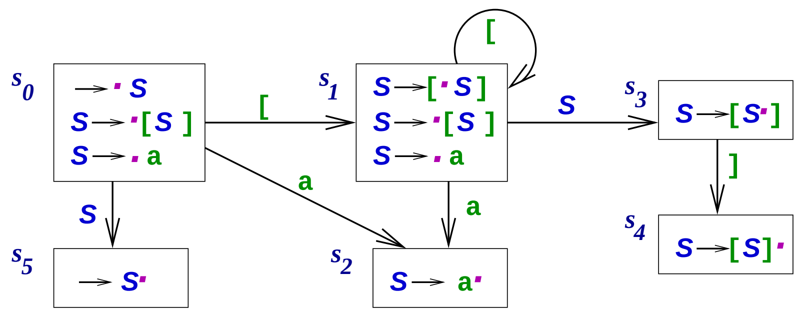
$$X1(s_0) = X0(s_0) \cup X2(s_0)$$

The union of the parses from $X0$ and $X2$ must be computed.[3] Consider $X0(s_0)$:

$$\begin{aligned} X0(s_0) &= goto(s_0, \mathtt{a}) = s_2 \quad (\text{reduce: } S \rightarrow \mathtt{a}) \\ &\Rightarrow goto(s_0, S) = s_5 \end{aligned}$$

showing that a parse of string `'a'` from state s_0 generates state s_2, a final state, that reduces to nonterminal S, which generates state s_5 — an S-phrase has been parsed. (The $\Rightarrow$ signifies when the parser makes a *reduce* step to a nonterminal.)

[3] As Figure 3 indicates, the functions compute sets of parse stacks; in this motivating example, all the sets are singletons.



parse stack (top lies at right)	input sequence (front lies at left)	
s_0	`[[a]]`	
$s_0 :: s_1$	`[a]]`	(because $goto(s_0, \texttt{[}) = s_1$)
$s_0 :: s_1 :: s_1$	`a]]`	
$s_0 :: s_1 :: s_1 :: s_2$	`]]`	(reduce: $S \to \texttt{a}$)
$s_0 :: s_1 :: s_1$	S `]]`	
$s_0 :: s_1 :: s_1 :: s_3$	`]]`	(because $goto(s_1, S) = s_3$)
$s_0 :: s_1 :: s_1 :: s_3 :: s_4$	`]`	(reduce: $S \to \texttt{[}S\texttt{]}$)
$s_0 :: s_1$	S `]`	
$s_0 :: s_1 :: s_3$	`]`	
$s_0 :: s_1 :: s_3 :: s_4$		(reduce: $S \to \texttt{[}S\texttt{]}$)
s_0	S	
$s_0 :: s_5$		(finished)

Fig. 2. *goto* controller for $S \to \texttt{[}S\texttt{]} \mid \texttt{a}$ and an example parse of `[[a]]`

The completed stack is therefore $s_0 :: s_5$. The remaining call, $X2(s_0)$, commences like this ($\oplus$ is explained two lines below):

$$X2(s_0) = (\texttt{[} \cdot X1 \cdot R)(s_0) = goto(s_0, \texttt{[}) \oplus (X1 \cdot R)$$
$$= s_1 \oplus (X1 \cdot R) = s_1 :: (X1(s_1) \oplus R)$$

The $\oplus$ operator sequences the parse steps: for parse stack, st, and function, E, $st \oplus E = st :: E(top(st))$, that is, the stack made by appending st to the stack returned by $E(top(st))$.

Next, $X1(s_1) = X0(s_1) \cup X2(s_1)$ computes to s_3, and we proceed:

$$X2(s_0) = s_1 :: (X1(s_1) \oplus R) = s_1 :: (s_3 \oplus R) = s_1 :: s_3 :: R(s_3)$$
$$= s_1 :: s_3 :: s_4 \quad (\text{reduce: } S \to \texttt{[}S\texttt{]})$$
$$\Rightarrow goto(s_0, S) = s_5$$

That is, $X2(s_0)$ built the stack, $s_1 :: s_3 :: s_4$, denoting a parse of $\texttt{[}S\texttt{]}$, which reduced to S, giving s_5.

Let Σ name the states in the parser's *goto*-controller. A flow equation, $Xi = E_i$, denotes the function, $\mathcal{X}_i : \Sigma \to \mathcal{P}(\Sigma^*)$, defined as $\mathcal{X}_i(s) = [\![E_i]\!](s)$, where

$[\![\mathtt{t}]\!]s = \{reduce(s, goto(s, \mathtt{t}))\}$, where $\mathtt{t}$ is a terminal symbol

$[\![E_1 \sqcup E_2]\!]s = [\![E_1]\!]s \cup [\![E_2]\!]s$

$[\![Xj]\!]s = [\![E_j]\!]s$, where $Xj = E_j$ is the flow equation for Xj

$[\![E_1 \cdot E_2]\!]s = \{reduce(s, p') \mid p' \in ([\![E_1]\!]s) \oplus [\![E_2]\!]\}$,

where $S \oplus g = \{p :: g(top(p)) \mid p \in S\}$

where $reduce(s, p)$ reduces the final states within parse stack, $s :: p$.

$reduce(s, p) =$
$\quad t := top(p)$
$\quad$ *if* $t = s_m$, the final state for item, $T \to U_1 U_2 \cdots U_m \cdot$,
$\quad$ *then* $p' := pop(m, p)$ # pop m states, corresponding to $U_1 U_2 \cdots U_m$
$\qquad p'' := p' :: goto(top(s :: p'), T)$
$\qquad$ *return* $reduce(s, p'')$ # repeat till finished
$\quad$ *else* *return* p # t was not a final state, so nothing to reduce

Fig. 3. Collecting interpretation: $\mathcal{X}_i(s) = [\![E_i]\!]s$ denotes the parse stacks generated by parsing the strings described by E_i, starting from s.

Here is the complete list of solved function calls:

$$
\begin{aligned}
&X3(s_0) = X1(s_0)\\
&X1(s_0) = X0(s_0) \cup X2(s_0) = \cdots = s_5 \cup s_5 = s_5\\
&X0(s_0) = goto(s_0, \mathtt{a}) = s_2 \Rightarrow goto(s_0, S) = s_5\\
&X2(s_0) = goto(s_0, \mathtt{[}) \oplus (X1 \cdot R) = s_1 :: X1(s_1) \oplus R\\
&\qquad\quad = \cdots = s_1 :: s_3 :: R(s_3) = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S) = s_5\\
&R(s_3) \quad = goto(s_3, \mathtt{]}) = s_4\\
&X1(s_1) = X0(s_1) \cup X2(s_1) = \cdots = s_3 \cup s_3 = s_3 \text{ (see comment below)}\\
&X0(s_1) = goto(s_1, \mathtt{a}) = s_2 \Rightarrow goto(s_1, S) = s_3\\
&X2(s_1) = goto(s_1, \mathtt{[}) \oplus (X1 \cdot R)\\
&\qquad\quad = s_1 :: (X1(s_1) \oplus R) = \cdots = s_1 :: s_3 :: R(s_3) \text{ (see comment below)}\\
&\qquad\quad = s_1 :: s_3 :: s_4 \Rightarrow goto(s_1, S) = s_3
\end{aligned}
$$

The solution is $X3(s_0) = s_5$, validating that the strings printed at the hot spot must be S-phrases.

Each equation instance, $X_i(s_j) = E_{ij}$, *a first-order data-flow equation*. In the example, $X1(s_1)$ and $X2(s_1)$ are mutually recursively defined, and their solutions are obtained by iteration-until-convergence. The flow-equation set is *generated dynamically while the equations are being solved*. This is a variant of demand-driven analysis [1, 8, 9], called *minimal function-graph semantics* [12], and is computed by an extension to the standard worklist algorithm; see the Appendix.

5 Abstract parse stacks

In the previous example, the result for each $X_i(s_j)$ was a single stack. In general, a set of parse stacks can result, e.g., for

`x = '['`	$X0 = \texttt{[}$
`while ...`	$X1 = X0 \sqcup X2$
`  x = x . '['`	$X2 = X1 \cdot \texttt{[}$
`x = x . 'a' . ']'`	$X3 = X1 \cdot \texttt{a} \cdot \texttt{]}$

at conclusion, `x` holds zero or more left brackets and an S-phrase; $X3(s_0)$ is the infinite set, $\{s_5,\ s_1 :: s_3,\ s_1 :: s_1 :: s_3,\ s_1 :: s_1 :: s_1 :: s_3,\ \cdots\}$.

To bound the set, we abstract it by "folding" its stacks so that no parse state repeats in a stack. Since Σ, the set of parse-state names, is finite, folding produces a finite set of finite-sized stacks (that contain cycles).

The abstract interpretation based on abstract, folded stacks is defined in Figure 4. Here is the intuition: A stack segment like $p = s_1 :: s_1$ is a linked list, a graph, $\leftarrow s_1 \leftarrow s_1 \leftarrow$, where the stack's top and bottom are marked by pointers; when we push a state, e.g., $p :: s_2$, we get $\leftarrow s_1 \leftarrow s_1 \leftarrow s_2 \leftarrow$. The folded stack is formed by merging same-state objects and retaining all links: $\leftarrow s_1 \leftarrow s_2 \leftarrow$. (This can be written as the regular expression, $s_1^+ :: s_2$.) Folding can apply to multiple states, e.g., $\leftarrow s_6 \leftarrow s_7 \leftarrow s_6 \leftarrow s_7 \leftarrow s_6 \leftarrow s_8 \leftarrow$ folds to $\leftarrow s_6 \leftarrow s_7 \quad s_8 \leftarrow$.

The abstract interpretation of the loop program that began this section is defined with abstract stacks in Figure 5. The result, $X3(s_0) = \{s_1^+ :: s_3,\ s_5\}$, asserts that the string at $X3$ might be a well-formed S phrase, or it might contain a surplus of unmatched left brackets.

At the end of the calculation in Figure 5, the reduction of $S \rightarrow [S]$ is done on the folded stack segment, $s_1^+ :: s_3 :: s_4$, that is, the complete stack is $s_0 \leftarrow s_1 \leftarrow s_3 \leftarrow s_4 \leftarrow$, meaning that three states must be popped: we traverse s_4, s_3, and s_1, and follow the links from the last state, s_1, to see what the remaining stack might be. There are two possibilities: $s_0 \leftarrow s_1 \leftarrow$ and $s_0 \leftarrow$. We compute the result for each case, as shown in the Figure.

6 Worklist algorithm

The algorithm that computes the solution to a hot-spot is a variation of the conventional worklist algorithm.

In the conventional worklist algorithm, there is a fixed flowgraph that indicates flows to nodes and a flow equation for each node. The initialization step builds the entire flowgraph and places demands on the worklist to calculate the value at every node in the graph. The algorithm then iterates, extracting a demand from the worklist, computing the value of that demand, and placing into

For label set, Σ, a *Σ-labelled graph*, g, is a tuple, $\langle nodes_g, edges_g, label_g\rangle$, where

- $nodes_g$ is a set of nodes,
- $edges_g \subseteq nodes_g \times nodes_g$ is a set of directed edges (at most one per source, target node pair),
- and $label_g : nodes_g \to \Sigma$ assigns a label to each node.

Let $Graph_\Sigma$ be the set of Σ-labelled graphs.

An *abstract stack* is a triple, (g, bot, top), such that $g \in Graph_\Sigma$ and $bot, top \in nodes_g$ mark the bottom and top nodes of the stack. Let $AbsStack_\Sigma$ be the set of abstract stacks labelled with Σ-values.

Example: the stack, $s_1 :: s_1 :: s_3$, is modeled as $(\langle\{a,b,c\},\ \{(c,b),(b,a)\},\ [a \mapsto s_1, b \mapsto s_1, c \mapsto s_3]\rangle,\ a,\ c)$.

An abstract stack, (g, bot, top), concretizes to the set of all finite sequences (paths) within g commencing at top and ending at bot.

Two abstract stacks, $G_1 = (g_1, bot_1, top_1)$ and $G_2 = (g_2, bot_2, top_2)$, are composed by :: into the disjoint union of g_1 and g_2 plus one new edge from bot_2 to top_1:

$$\begin{aligned} G_1 :: G_2 = (\langle & nodes_{g_1} \uplus nodes_{g_2}, \\ & edges_{g_1} \cup edges_{g_2} \cup \{(bot_2, top_1)\}, \\ & label_{g_1} + label_{g_2}\rangle, \\ & bot_1, top_2) \end{aligned}$$

An abstract stack is folded by merging all nodes that share the same label, in effect, equating the nodes with the labels:

$$\begin{aligned} fold(g, bot, top) = (\langle & \{s \in \Sigma \mid \exists n \in nodes_g, label_g(n) = s\}, \\ & \{(s, s') \mid \exists (n, n') \in edges_g, label_g(n) = s, label_g(n') = s'\}, \\ & \lambda s.s\rangle, \\ & label_g(bot),\ label_g(top)) \end{aligned}$$

The abstract interpretation of flow equation, $Xi = E_i$, is the function,
$\mathcal{X}_i : \Sigma \to \mathcal{P}_{fin}(AbsStack_\Sigma)$, defined as

$$\mathcal{X}_i(s) = \{fold(p) \mid p \in [\![E_i]\!](s)\}.$$

(See Figure 3.)

A set of folded stacks can be further abstracted into a *single* structure of the form, $Graph_\Sigma \times \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, by unioning the stacks' node sets, edge sets, *bot*-values and *top*-values. The resulting graph is a *subgraph* of the parser's *goto*-controller.

Fig. 4. Abstract interpretation defined in terms of abstract, folded, parse stacks

Flow equation set generated from demand, $X3(s_0)$:

$$X0(s_0) = \texttt{[}(s_0) \qquad X2(s_0) = X1(s_0) \oplus \texttt{[}$$
$$X1(s_0) = X0(s_0) \cup X2(s_0) \qquad X3(s_0) = X1(s_0) \oplus (\texttt{a.]})$$

Least fixed-point solution expressed with abstract parse stacks:

$X0(s_0) \;=\; \texttt{[}(s_0) \;=\; \{s_1\}$

Because $X1$ and $X2$ are mutually defined, we iterate to a solution, where Xi's value at iteration j is denoted Xi_j:

$X1_1(s_0) = \{s_1\} \cup \emptyset \;=\; \{s_1\}$

$X2_1(s_0) = X1_1(s_0) \oplus \texttt{[} \;=\; \mathit{fold}\{s_1 :: s_1\} \;=\; \{s_1^+\}$

$X1_2(s_0) = \{s_1\} \cup \{s_1^+\} \;=\; \{s_1, s_1^+\}$

$= \{s_1^+\}$. (We can merge the two stack segments since the first is a prefix of the second and has the same bottom and top states.)

$X2_2(s_0) = X1_2(s_0) \oplus \texttt{[} \;=\; \{s_1^+ :: \texttt{[}(s_1)\} \;=\; \mathit{fold}\{s_1^+ :: s_1\} \;=\; \{s_1^+\}$

$X1_3(s_0) = \{s_1\} \cup \{s_1^+\} \;=\; \{s_1, s_1^+\} \;=\; \{s_1^+\} \;=\; X1_2(s_0)$

$X2_3(s_0) = \{s_1^+\} \;=\; X2_2(s_0)$

$X3(s_0) \;=\; \{s_1^+ :: \texttt{a}(s_1) \oplus \texttt{]}\}$

First, $s_1^+ :: \texttt{a}(s_1) \;=\; s_1^+ :: s_2 \;\Rightarrow\; s_1^+ :: \mathit{goto}(s_1, S) \;=\; s_1^+ :: s_3$.

$= \{s_1^+ :: s_3 :: \texttt{]}(s_3)\} \;=\; \{s_1^+ :: s_3 :: s_4\}$

The reduction, $S \rightarrow [S]$, splits the stack into two cases:
(i) there are multiple s_1s within s_1^+; (ii) there is only one s_1:

$= (i)\{s_1^+ :: \mathit{goto}(s_1, S)\} \;\cup\; (ii)\{\mathit{goto}(s_0, S)\}$

$= \{s_1^+ :: s_3, \;\; s_5\}$

Fig. 5. Iterative solution with folded parse stacks, depicted as regular expressions

the worklist new demands to evaluate those nodes whose values are affected by the one just updated. Iteration terminates when the worklist is empty [17].

In our worklist algorithm, the flowgraph is constructed while iteration is undertaken. The algorithm uses three data structures: the worklist of unresolved calls, $Xi(s_j)$; a Cache that maps each call to its current (partial) solution (a set of abstract parse stacks); and the flowgraph of call dependencies, which is dynamically constructed.

The algorithm is defined in the Appendix, but here is an overview: The initialization step places the initial call, $X0(s_0)$, into the worklist and into the call graph and then assigns to the cache the partial solution, $\mathit{Cache}[X0(s_0)] := \emptyset$. The iteration step repeats the following until the worklist is empty:

1. Extract a call, $X(s)$, from the worklist, and for the corresponding flow equation, $X = E$, compute $E(s)$, folding abstract stacks as necessary. (In the Appendix, this is done by $\mathit{compute}_{X(s)}(s, E)$).
2. While computing $E(s)$, if a call, $X'(s')$ is encountered, *(i)* add the dependency, $X'(s') \rightarrow X(s)$, to the call graph (if it is not already present); *(ii)* if there is no entry for $X'(s')$ in the cache, then assign $\mathit{Cache}[X'(s')] := \emptyset$ and place $X'(s')$ on the worklist.

Worklist,
added and processed
from top to bottom:

$X3(s_0)$
$X1(s_0)$
$X0(s_0)$
$X2(s_0)$
$X1(s_0)$
$X1(s_1)$
$X3(s_0)$
$X0(s_1)$
$X2(s_1)$
$X1(s_1)$
$X2(s_0)$
$X2(s_1)$
$R(s_3)$
$X2(s_0)$
$X2(s_1)$
$X1(s_0)$
$X1(s_1)$

Cache updates, inserted from top to bottom,
where $X(s) \mapsto P$ abbreviates $Cache[X(s)] := P$

$$
\begin{array}{l}
X3(s_0) \mapsto \emptyset \\
X1(s_0) \mapsto \emptyset \\
X0(s_0) \mapsto \emptyset \\
X2(s_0) \mapsto \emptyset \\
X0(s_0) \mapsto reduce(s_0, goto(s_0, \mathtt{a})) = reduce(s_0, s_2) \\
\quad = reduce(s_0, goto(s_0, S)) = reduce(s_0, s_5) = \{s_5\} \\
X1(s_1) \mapsto \emptyset \\
X1(s_0) \mapsto \{s_5\} \\
X0(s_1) \mapsto \emptyset \\
X2(s_1) \mapsto \emptyset \\
X3(s_0) \mapsto \{s_5\} \\
X0(s_1) \mapsto reduce(s_1, goto(s_1, \mathtt{a})) = \{s_3\} \\
X1(s_1) \mapsto \{s_3\} \\
R(s_3) \mapsto \emptyset \\
R(s_3) \mapsto reduce(s_3, goto(s_3, \mathtt{]})) = \{s_4\} \\
X2(s_0) \mapsto (\mathtt{[} :: X1 :: R)(s_0) \\
\quad = s_1 \oplus (X1 :: R) = (s_1 :: X1(s_1)) \oplus R \\
\quad = s_1 :: s_3 :: R(s_3) = reduce(s_0, s_1 :: s_3 :: s_4) \\
\quad = reduce(s_0, goto(s_0, S)) = \{s_5\} \\
X2(s_1) \mapsto (\mathtt{[} :: X1 :: R)(s_1) = \{s_3\}
\end{array}
$$

Generated call graph:

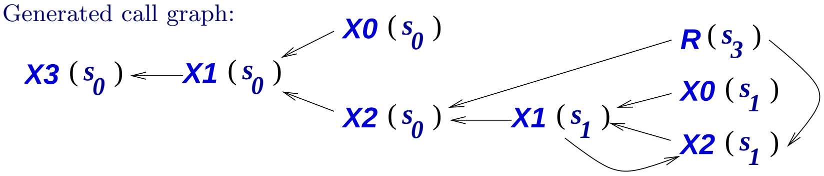


Fig. 6. Worklist-algorithm calculation of call, $X3(s_0)$, in Figure 1

3. When $E(s)$ computes to an answer set, P, and P contains an abstract parse stack not already listed in $Cache[X(s)]$, then assign $Cache[X(s)] := (Cache[X(s)] \cup P)$ and add to the worklist all $X''(s'')$ such that the dependency, $X(s) \rightarrow X''(s'')$, appears in the flowgraph.

Figure 6 shows the worklist calculation for $X3(s_0)$ in Figure 1.

7 Input variables

Input and nonlocal variables present the usual difficulties for a static analysis. If we require that such variables hold grammatically well-structured strings as their values, then we can use the nonterminal symbols of the reference grammar as “data types.” For example, we might set the type of input variable, `x`, to be nonterminal S and use Figure 2 to analyze

```
readS x                 X = S
y = '[' . x . ']'       Y = [ · X · ]
```

We solve the flow equations,

$$Y(s_0) = (\texttt{[} \cdot X \cdot \texttt{]})(s_0) = goto(s_0, \texttt{[}) \oplus (X \cdot \texttt{]}) = s_1 :: (X(s_1) \oplus \texttt{]})$$
$$X(s_1) = goto(s_1, S) = \{s_3\}$$

and compute that $Y(s_0) = s_1 :: s_3 :: goto(s_3, \texttt{]}) = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S) = \{s_5\}$, because we assumed that input variable `x` denotes a parsed `S`-phrase.

8 String-update operations

String-manipulating languages use operations like `replace` and `substring`, which can be employed foolishly or sensibly on strings that represent well-structured values. An example of the former is `x = '[[a]]'; replace('a', '[', x)`, which replaces occurrences of `'a'` in `x` by `'['`, changing `x`'s value to the grammatically ill-formed phrase, `'[[[]]'`. A more sensible replacement would be `replace('[a]', 'a', x)`, which preserves `x`'s grammatical structure.

To validate an operation, `replace(U,V,x)`, we require that `U` and `V` "parse the same" in every possible context where they might appear (within `x`): Say that `replace(U,V,x)` is *update-invariant for* `x` iff for all (nonfinal) parse states, $s \in \Sigma$, $U(s) = V(s)$. This means replacing `U` by `V` preserves `x`'s parse.

When we analyze a program, we may first *ignore* the `replace` operations, treating them as "no-ops." Once the flow equations are solved, we validate the invariance of each `replace(U,V,x)` by generating hot-spot requests for strings `U` and `V` for all possible parse states, building on the cached results of the worklist algorithm. Finally, we compare the results to see if `replace(U,V,x)` is update-invariant for `x`. Here is an example:

```
y = '[[[a]]]'           Y0 = [ · [ · [ · a · ] · ] · ]
x = 'a'                 X0 = a
while ...               X1 = X0 ∪ X2
  x = '['. x .']'       X2 = [ · X1 · ]
replace(x, 'a', y)      Y1 = replace(X1, a, Y0)
```

Say that the program must be analyzed for `y`'s final value: $Y1(s_0)$. We initially ignore the replacement operation at $Y1$ and solve the simpler equation, $Y1(s_0) = Y0(s_0)$, instead, which quickly computes to $\{s_5\}$. Next, we analyze the `replace` operation by generating these hot-spot requests for all the nonfinal parse states:

$$a(s_0),\ X1(s_0),\ a(s_1),\ X1(s_1),\ a(s_3),\ X1(s_3)$$

For example, the first request computes to

$$a(s_0) = goto(s_0, a) = s_2 \Rightarrow goto(s_0, S) = s_5$$

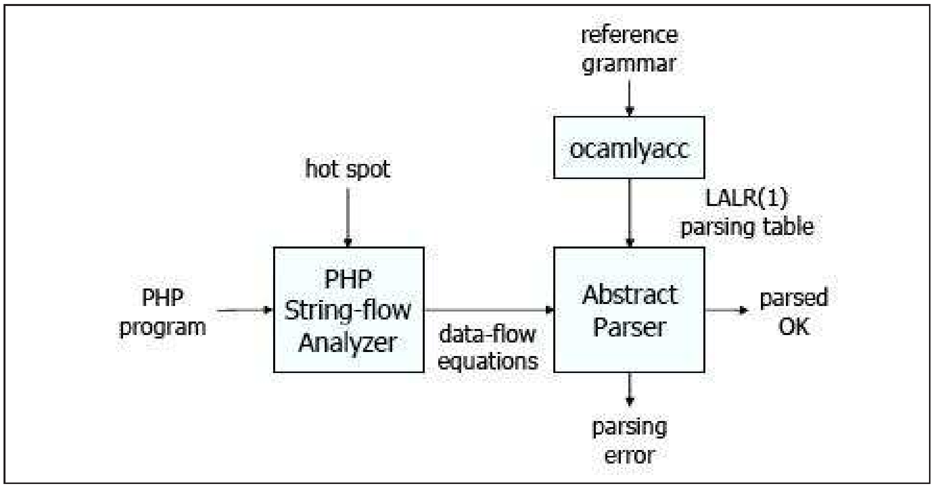


Fig. 7. Implementation

and the second repeats an earlier example,

$$X1(s_0) = X0(s_0) \cup X2(s_0)$$
$$X2(s_0) = \cdots = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S)) = s_5$$

showing that both strings compute to the same parse-stack segments in starting context s_0. The other hot spots compute this same way. Once all the hot spots are solved, we confirm that $X1$ and a have identical outcomes for all possible parse contexts. This validates the invariance of `replace(x,'a',y)` at $Y1$, preserving the original solution.

It is important that we validate update-invariance for *all* possible contexts. Consider the reference grammar,

$$N \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{[a]}$$

Although both `a` and `b` are N-phrases, `replace('a','b','[a]')` violates `[a]`'s grammatical structure.

9 Implementation and experiments

The abstract parser, essentially the worklist algorithm, is implemented in Objective Caml, structured as seen in Figure 7. The front end of Minamide's string analyzer for PHP [15] is modified to accept a PHP program with a hot-spot location and to return a set of data-flow equations with string operations for the hot spot. A parser generator, `ocamlyacc`, produces an LALR(1) parsing table for the reference grammar, and the abstract parser uses the data-flow equations and the parsing table to parse statically the strings dynamically generated by the PHP program. Since abstract parsing works directly on characters (and not tokens), the reference grammar is given at the same level, like a grammar for scannerless parsing. (Our experiment with HTML grammars extended this way did not show any degradation of performance.) The worklist algorithm in the Appendix is defined for LR(0) grammars, but its extension to LR(1) required only a minor modification.

We applied our abstract parser to publicly available PHP programs that dynamically generate HTML documents, the same suite of programs Minamide used in his paper [15]. Experiments were done on a MacOSX with an Intel Core 2 Duo Processor (2.56GHz) and 4 GByte memory. The table in Figure 8 summarizes our experiments.

	webchess	faqforge	phpwims	timeclock	schoolmate
files	21	11	30	6	54
lines	2918	1115	6606	1006	6822
total hot spots	6	14	30	7	1
hot spots parsed OK	5	1	19	0	1
hot spots parsed ERR	1	15	17	7	18
true ERRs	1	31	14	14	17
false-positive ERRs	0	0	3	0	3
total time(sec)	0.224	0.155	1.979	0.228	2.077

Fig. 8. Experimental results

We manually tagged the HTML-document-generating hot spots and ran our abstract parser for each hot spot. Since we do not yet have parse-error recovery in our implementation, each time a parse error was identified by our analyzer, we located the source of the error in the program, fixed it, and tried again until no parse errors were detected. The false-positive alarms that appeared were all caused by ignoring the tests within conditional commands. The parsing time shown in the table is the sum of all execution times needed to find all parsing errors for all hot spots. The reference grammar's parse table took 1.323 seconds to construct; this is not included in the analysis times. The nature of the parsing errors we found are classified in Figure 9.

Contrasted to Minamide's validation approach [15], our abstract parser works without limiting the nesting depth of tags, validates the syntax approximately 30∼3,500 times faster, and is guaranteed to find all parsing errors.

Minamide had to exclude one PHP application, named `tagit`, from his experiments, since `tagit` generates an arbitrary nesting depth of tags. In principle, our abstract parser should be able to validate `tagit`, but we also excluded `tagit` from our profiling studies because the current version of our abstract parser checks that string-update operations satisfy the update-invariance property (cf. Section 8). Unexpectedly (to us!), so many string updates in `tagit` violated update invariance that our abstract parser generated too many false-positives to be helpful.

classification	occurrences
open/close tag syntax error	6
open/close tag missing	45
superfluous tag	5
improperly nested	14
misplaced tag	5
escaped character syntax error	2

Fig. 9. Classification of parsing errors found

We can reduce false positives due to violation of update invariance by selectively employing Minamide's f.s.a.-transducer technique [15], where a string update is analyzed separately from the flow analysis with its own f.s.a. transducer. For example, the last flow equation in this program,

```
x = 'a'                        X0 = a
while ...                      X1 = X0 ∪ X2
  x = '[['. x .']'             X2 = [ · [ · X1 · ]
replace('[[', '[', x)          X3 = replace([[, [, X1)
```

could be replaced by just $X3 = X1$, and we would use a separate transducer to analyze $replace(\texttt{[[}, \texttt{[}, X1)$. We leave this as a future work.

On the other hand, one might argue that *any* string-update operator that violates update invariance is dubiously employed and deserves closer scrutiny. In this regard, the abstract parser's "false positives" are healthy warnings.

10 Conclusion

Injection and cross-site-scripting attacks can be reduced by analyzing the programs that dynamically generate documents [21]. In this paper, we have improved the precision of such analyses by employing LR-parsing technology to validate the context-free grammatical structure of generated documents.

A parse tree is but the first stage in calculating a string's meaning. The parsed string has a semantics (as enforced by its interpreter), and it is possible to encode this semantics with semantics-processing functions, like those written for use with a parser-generator. (Tainting analysis — tracking unsanitized data — is an example semantic property that can be encoded this way.) The semantics computation can then be approximated by the static analysis so that abstract parsing and abstract semantic processing proceed simultaneously. This is a future step in our work.

References

1. G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proc. Int'l. Conf. Software Maintenance, Oxford*, 1999.

2. A. Aho and J. Ullman. *Principles of Compiler Design*. Addison Wesley, 1977.
3. C. Brabrand, A. Møller, and M.I. Schwartzbach. The <bigwig> project. *ACM Trans. Internet Technology*, 2, 2002.
4. T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *Proc. Asian Symp. Prog. Lang. and Systems*, pages 374–388. Springer LNCS 4279, 2006.
5. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proc. PLAN-X-02*, 2002.
6. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Extending Java for high-level web service construction. *ACM Trans. Prog. Lang. and Systems*, 25, 2003.
7. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
8. E. Duesterwald, R. Gupta, and M.L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Prog. Lang. and Systems*, 19:992–1030, 1997.
9. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engg.*, 1995.
10. H. Hosoya. XDuce: A typed XML processing language. Technical Report `http://xduce.sourceforge.net/`, 2008.
11. H. Hosoya, J. Vouillon, and B.C. Pierce. Regular expression types for XML. *ACM Trans. Prog. Lang. and Systems*, 27:46–90, 2005.
12. N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th Symp. on Principles of Prog. Languages*, pages 296–306. ACM Press, 1986.
13. N. Jovanovich, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. IEEE Symp. on Security and Privacy*, pages 258–263, 2006.
14. C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proc. International Symp. Static Analysis*, pages 336–352. Springer LNCS 4134, 2006.
15. Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. 14th ACM Int'l Conf. on the World Wide Web*, pages 432–441, 2005.
16. Y. Minimide and A. Tozawa. XML validation for context-free grammars. In *Proc. Asian Symp. Prog. Lang. and Systems*, pages 357–373. Springer LNCS 4279, 2006.
17. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
18. P. Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM workshop Types in languages design and implementation*, pages 59–70, 2005.
19. G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dymanically generated queries in database applications. *ACM Trans. Software Engineering and Methodology*, 16(4):14:1–27, 2007.
20. G. Wassermann and Z. Su. The essence of command injection attacks in web applications. In *Proc. 33d ACM Conf. on Principles of Prog. Lang.*, pages 372–382, 2006.
21. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM Conf. on Prog. Lang. Design and Implementation*, pages 32–41, 2007.
22. Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Symp.*, 2006.

Appendix: Worklist algorithm

Input:

- controller (*goto* function) for parser;
- flow-equation schemes, $\{X_i = E_i\}_{0<i\leq n}$;
- initial demand, $X_0(s_0)$.

Data structures:

- $W \in Call^* =$ worklist of demands (calls) of form, $X_j(s)$, $s \in ParseState$;
- F: dynamically generated call graph, consisting of arcs of form, $X(s) \to X'(s')$, read as, "$X(s)$'s value flows to $X'(s')$";
- $Cache : Call \to \mathcal{P}(ParseStack)$: dynamic array mapping calls to sets of parse-stack segments, where

  $ParseStack =$ regular trees whose nodes are $ParseState$s, such that one node is marked the stack bottom and another the stack top.

  There is a unique entry, $Cache[X(s)] := P$, in the cache array iff the node, $X(s)$, appears in F.

Algorithm:

1. *Initialize:*
   $W := [X_0(s_0)]$;
   $F := \{X_0(s_0)\}$;
   $Cache[X_0(s_0)] := \emptyset$
2. *Iterate:*
   $while\ \ W \neq []\ do:$
   $\quad X(s) := head(W);\ W := tail(W);$
   $\quad let\ X = E$ be the flow equation that matches $X(s)$;
   $\quad P := compute_{X(s)}(s, E)$; (see below)
   $\quad if\ P \not\subseteq Cache[X(s)]$
   $\quad then\ Cache[X(s)] := Cache[X(s)] \cup P;$
   $\quad\quad forall\ X'(s')$ such that $X(s) \to X'(s') \in F$,
   $\quad\quad\quad W := W + [X'(s')];$

where
$compute_{Call} : ParseState \times FlowExpression \to \mathcal{P}(ParseStack)$
is defined

$compute_c\ (s, a) = return\ reduce(s, goto(s, a))$
$compute_c(s, E_1 \sqcup E_2) =\ return\ compute_c(s, E_1) \cup compute_c(s, E_2)$
$compute_c\ (s, X) =$
$\quad if\ Cache[X(s)]$ is undefined (has no entry),
$\quad then\ Cache[X(s)] := \emptyset;$
$\quad\quad$ add the edge, $X(s) \to c$, to F;
$\quad\quad W := W + [X(s)];$
$\quad if\ c < X$ (that is, $c \to X(s)$ is a program back-arc),
$\quad then\ return\ fold(Cache[X(s)])$
$\quad else\ return\ Cache[X(s)]$

$compute_c\ (s, E_1 \cdot E_2) =$

$P := \bigcup\{p \oplus E_2 \mid p \in compute_c(s, E_1)\}$

$where\ p \oplus E_2 = \{p :: p' \mid p' \in compute_c(top(p), E_2)\}$

$return\ \bigcup\{reduce(s, p'') \mid p'' \in P\}$

Auxiliary function $reduce(s,p)$ reduces parse stack, $s :: p$, as needed, never popping stack bottom, s. If the stack needs no reduction, $reduce(s,p) = \{p\}$:

$reduce : ParseState \times ParseStack \to \mathcal{P}(ParseStack)$

$reduce\ (s,p) =$

$t := top(p);$

$if\ t = s_m,$ a final state for item, $T \to U_1 U_2 \cdots U_m$,

and the top m states in p are $s_1 :: s_2 :: \cdots :: s_m$, matching the item,

$then$

$newTops := \{s' \mid s' \leftarrow s_1 \in p\}$ (the predecessor state(s) to s_1 in p)

$if\ newTops = \emptyset,$ (popped stack empty?)

$then$

$R := \{goto(s, T)\}$

$else$

$poppedStacks := \{p$ with s' marked as top $\mid s' \in newTops\}$

$R := \{p' :: goto(top(p'), T) \mid p' \in poppedStacks\}$

$return\ \bigcup\{reduce(s, p'') \mid p'' \in R\}$ (repeat till finished)

$else$ // t was not a final state, so nothing to reduce:

$return\ \{p\}$