String Analysis

Fang Yu Marco Cova

Abstract

String analysis is a static analysis technique that determines the string values that a variable can hold at specific points in a program. This information is often useful to help program understanding, to detect and fix programming errors and security vulnerabilities, and to solve certain program verification problems. We present a novel approach to perform string analysis on real-world programs.

1 Introduction

To detect software vulnerabilities and perform sanitization functions in PHP programs, it is essential to know which values may occur as a result of string expressions. The exact answer is unknown due to the theoretical result that the family of context free languages is not closed under intersection and complementation. String operations for context free languages may yield a language which is not in the context free family. As a result, except that the emptiness checking is decidable, all other problems of context-free languages, e.g., inclusion and equivalence checking, are undecidable.

On the other hand, the family of regular languages is closed under union, intersection, complementation, star closure, homomorphism and right quotient. Since all problems that we are interested in are decidable when we deal with regular languages, this family is widely used for string analysis.

Christensen et al. [2] summarize each string expression as a regular language that includes all possible values of that string expression. This over approximation makes it feasible to statically predict values of string expressions in general programs. We extend their method to analyze PHP programs. Particularly we are interested in context string replacement, which is a widely-adopted string operation to secure PHP programs but had not been addressed in [2]. Context string replacement had been discussed in the context of natural language processing [16, 11, 4, 14]. All these works are based on the composition of finite state transducers. Vaillette [14] formulates regular expressions and relations in monadic second order logics, so that the corresponding DFA generated by MONA accepting the replaced languages. Vaillette's method suffers from the requirement of the same length regular relation. It is also not clear how to generate the specified monadic second order logics automatically.

Here we propose a new automata based algorithm. We construct several internal deterministic finite automata (DFA), and take the intersection of them to yield the one accepting the replaced language. For unconditional context string replacement, all internal DFAs have liner size to the given problem.

Our work can be divided into two parts: a front-end that translates the given PHP program into a flow graph, and a back-end that analyzes the flow graph and generates finite state automata. The front-hand parser accepts the full PHP language and then yields the control flow graph which illustrates an abstract description of a program performing string manipulations. The back-end analyzer generates a finite automaton for each string expression. The automaton is updated according to the manipulation along the control flow, by accepting the regular language summarizing all possible values of the string expression. Finally, for each string expression we intersect the set of accepted strings with the set of attacking strings. Once the intersection is an empty set, the PHP program is proven to be vulnerable free, Otherwise, a witness to attack the PHP program is automatically generated.

2 Related Work

String Analysis. Christensen, Møller and Schwartzbach [2] first propose automata-based analysis (called JSA) to statically determine the values of string expressions in Java programs. All possible values of string expressions are generated upon regular expression manipulations. The starting point is the flow graph, which is then translated into a context free grammar with one nonterminal for each node. Each edge and its operation is encoded as a production rule. The size of the resulting grammar is linear in the size of the flow graph. The context free grammar is then approximated with a regular grammar. This approximation is achieved by Mohri-Nederhof algorithm. The idea is eliminating production cycles by replacing one operation production with $X \rightarrow r$, where X is a nonterminal and r denotes the regular language Σ^*. They further transform the regular grammars into multilevel finite automata M, which can be then used to generate a DFA. The size of the final DFA is worst case doubly exponential to the size of M. They have successfully applied this method to various real applications like analyzing dynamic generated XML in the JWIG and static syntax checking of SQL. However, they did not address context string replacement in this work.

Kirkegaard et al. apply JSA to statically analyze the XML transformations in Java programs [7]: by using DTD schemas as types and modeling the effect of XML transformation operations, they statically verify that the analyzed program transform valid input data into valid output data.

Gould et al. [5] use string analysis to check for errors in dynamically generated SQL query strings in Java-based web applications. Their analysis is also based on the JSA analysis [2].

Christodorescu et al. [3] present an implementation of string analysis for executable programs for the x86 architecture. Their technique recovers semantic information from binary code (e.g., they perform string inference, alias analysis) and leverages the JSA infrastructure of [2] to perform the string analysis.

Minamide [9] describes an application of string analysis to statically detect cross-site scripting vulnerabilities and to validate pages generated by web applications written in the PHP language. Similarly to JSA, he first extracts a grammar from a PHP program considering assignments as production rules, the grammar is then transformed into a context-free grammar, and, finally, this is used to validate the desired properties. He claims to support most of the string operations available in PHP, including regular expression-based replacement.

Context String Replacement Context string replacement had been discussed through in Natural Language Processing [6, 11, 16, 4]. Karttunen [6] first proposes the replace operator of regular expressions by applying phonological rewrite rules. All these works based on the same strategy that is first to decompose the complex relation into a set of independent components, and then define the whole operation as a composition. Mohri and Sproat[11] implement the rewrite rules with finite state transducers. They also generalize the rewrite rules as $\phi \rightarrow \psi/\lambda \ldots \rho$, which is interpreted in the following way: ϕ is to be replaced by ψ whenever it is preceded by λ and followed ρ. $\psi, \phi, \lambda, \rho$ are regular languages in general. The finite state transducer corresponding to the left to right obligatory rule $\phi \rightarrow \psi/\lambda \ldots \rho$ can be obtained by the composition of the following five transducers: $r \circ f \circ \mathit{repalce} \circ l_1 \circ l_2$.

- r: $\Sigma^* \rho \rightarrow \Sigma^* > \rho$.
- f: $(\Sigma \cup \{>\})^* \phi > \rightarrow (\Sigma \cup \{>\})^* \{<_1, <_2\} \phi >$.
- *replace*: replace ϕ with ψ in the context $<_1\ \phi\ >$ using the cross product transducer $\phi \times \psi$.
- l_1: $\Sigma^* \lambda <_1 \rightarrow \Sigma^* \lambda$.
- l_2: $\Sigma^* \overline{\lambda} <_2 \rightarrow \Sigma^* \overline{\lambda}$.

The transducer r inserts a marker $>$ before every instance of ρ. The transducer f inserts markers $<_1$ and $<_2$ before each instance of ϕ that is followed by $>$. The *replace* transducer replaces ϕ with ψ in the context $<_1\ \phi\ >$, and simultaneously deletes $>$. The transducer l_1 admits only those strings in which occurrences of $<_1$ are preceded by λ and deletes $<_1$ then. The transducer l_1 admits only those strings in which occurrences of $<_2$ are not preceded by λ and deletes $<_2$ then.

In [16, 4], Noord and Gerdemann further propose several transducers to specify delicate replacement operations, like the left-most, the longest match and the first-match semantics. An disadvantage is that the finite state transducer of regular expression is easily becoming quite complicated and hard to prove its correctness. Unlike using finite state transducers, Vaillette[14] solves the context string replacement by representing regular language with monadic second order logics. They represent regular languages and constraints in monadic second order logics and let MONA handle the rest. The left-most longest match replacement can be done in one pass. This is the only work with the complete proof of its correctness. However, their algorithm only works on the same length regular relation. Recall that finite state transducer is not closed under intersection and complementarity. This restricts the application of their method since their algorithm requires to use intersection and complementarity of finite transducers. For the same length regular expression, one may treat pair as an alphabet and as a result boolean closure follows then. To address this problem, Vaillette induces some specific symbol, e.g., "0" as an empty string, to characterize different length regular relation. (using 0 to pick up the slack). As Vaillette mentioned in the paper, ambiguous packed strings, e.g., $00ab$ and $ab00$, may yield different replaced strings in several cases. Vaillette further proposed the weaken version algorithm to address this problem, in which a string is replaced if some of its packed string matches the regular expression.

Tools. All the analyzed approaches to string analysis are based on the modeling of string values and operations in terms of finite state automata. There exists a number of tools and libraries to perform manipulation and analysis of finite state automata. Here we present a brief overview of the most commonly used tools.

Finite State Automata utilities (FSA) [15] is a collection of utilities to manipulate regular expressions, finite-state automata and finite-state transducers. The supported manipulations include the standard operations (e.g., minimization, composition, complementation, intersection, Kleene closure), and an implementation of the Mohri and Sproat's compiler of rewrite rules [11]. FSA is implemented in Prolog and is released under the GPL license.

The AT&T FSM library [10] is a set of general-purpose software tools available for Unix, for building, combining, optimizing, and searching weighted finite-state acceptors and transducers. It was originally designed to provide algorithms and representations for phonetic, lexical, and language-modeling components of large-vocabulary speech recognition systems. The mathematical foundation of FSM is the theory of rational power series.

MONA is a tool that translates formulas to finite-state automata [1]. MONA is based on the Weak Second-order Theory of One or Two successors (WS1S/WS2S).

3 Approach

In the rest of the paper, we will refer to the following example to explain our technique.

```
$www = $_GET["www"];
if (strpos($www, "http://") === false) {
    $www = "http://" . $www;
}
$clean_www = ereg_replace("<script", "&lt;script", $www);
echo "www parameter is: " . $clean_www;
```

This code fragment presents a series of string manipulation operations that can be commonly found in web applications written in the PHP language. The `www` variable is set to a value provided by the user (through the request parameter _GET["www"]). If the provided string does not start with the string `http://`, the protocol specifier is prepended to the string. Then, all '<' characters in the string used to start a `script` tag are replaced with their corresponding HTML entity "&lt;" (a common defense against cross-site scripting attacks), and, finally, the string is echoed back to the user.

Most of the string manipulation operations performed in real-world applications can be reduced to three operations, which we call the *basis*:

- *assignment*: assigns the current string value of a variable to another variable (the assignment operator in PHP is `=`);
- *concatenation*: concatenates two string variables, containing either user-defined or static string values (the concatenation operation in PHP is `.`);
- *replacement*: replaces the parts of a string that match a specified pattern with the given replacement string. Depending on the specific replacement function, the manipulation can be modeled as a homomorphism (e.g., the PHP functions `htmlspecialchars`, `tolower`, `toupper`), transducer (e.g., `str_replace`, `trim`), or context replacement (e.g., `ereg_replace`).

Our approach can be divided in two parts. A back-end component models the value of string variables in an input program as finite state automata. It also implements the manipulation operations supported by our tool, i.e., assignment, concatenation and replacement. The front-end component parses the input program, transforms the program's string manipulation operations in corresponding basis operations, and, by using standard data-flow techniques and the back-end component, determines (an over-approximation of) the possible values of string operations in all program points. Note that the back-end component is language-independent and can be reused to analyze programs written in different programming language. The front-end supports the PHP language.

3.1 Back-End

3.1.1 String Automaton Generator

Our backend is a BDD-based string automaton generator. We associate each string variable a DFA, which is manipulated along with the control flow graph, so that the associated DFA accepts all possible values of the string variable at its current state. To support basic string manipulations and checking, we identify the following DFA functions.

- Construct(char $*e$): e is a regular expression over the ASCII alphabet. This function returns a dfa M such that $L(M) = \{w | w \in L(e)\}$.

- Concatenate(dfa M_1, dfa M_2): This function returns a dfa M such that $L(M) = w_1w_2|w_1 \in L(M_1), w_2 \in L(M_2)$ Replace (dfa M_1, dfa M_2, dfa M_3): This function returns a dfa M, such that $L(M) = \{w_1c_1w_2c_2w_kc_kw_{k+1}|\exists k, w_1x_1w_2x_2w_kx_kw_{k+1} \in L(M_1),\forall_i, x_i \in L(M2), w_i \notin L(M_2), c_i \in L(M_3)\}$.
- Union (dfa M_1, dfa M_2) : This function returns a dfa M, such that $L(M) = L(M_1) \cup L(M2)$
- Intersect(dfa M_1, dfa M_2): This function returns a dfa M, such that $L(M) = L(M_1) \cap L(M_2)$

We also support the following check functions.

- EmpCheck(dfa M_1): Check whether $L(M_1)$ is empty, and generate an example if it is not empty.
- ItrCheck(dfa M_1, dfa M_2): Check whether $L(M_1)$ and $L(M_2)$ are intersected, and generate an example if they are intersected.
- IclCheck(dfa M_1, dfa M_2): Check whether $L(M_1) \subseteq L(M_2)$.
- EquCheck(dfa M_1, dfa M_2): Check whether M_1 and M_2 accept the same language.

Emptiness and intersection checking are used to detect whether a string variable may have a vulnerable value; Inclusion and equivalence checking are used to detect whether a fixed point is reached.

We use the DFA packages of MONA [1] to implement these functions. The DFAs in MONA are encoded as Binary Decision Diagrams(BDDs). BDD has a canonical form and can be manipulated efficiently: constant time to emptiness checking and polynomial time to union, intersection, negation, and reduction to canonical form.

Two main challenges to achieve our goal is DFA construction and replacement. We discuss more details in the following sub sections.

3.1.2 Encode ASCII code

MONA stands on *mona*dic second order logics, which are the second order logics over three kinds of variables: Boolean, Integer and Integer Set. The idea to encode string in MONA is to declare a bounded integer set to denote the positions of a string. We then recursively construct predicates, e.g., $is_E(p,q)$, such that the predicate $is_E(p,q)$ holds if and only if the substring from position p to q (including p_{th} letter but not the q_{th} letter) belongs to the language defined by E.

We first declare eight subsets for ASCII character.

```
var2
bit0 where bit0 sub $,
bit1 where bit1 sub $,
bit2 where bit2 sub $,
bit3 where bit3 sub $,
bit4 where bit4 sub $,
bit5 where bit5 sub $,
bit6 where bit6 sub $,
bit7 where bit7 sub $;
```

Each subset represents a set of strings, so that if p is in the subset, the specified bit of the character in the p_{th} position of the strings is true. Based on this idea, one can encode the p_{th} position of a string is some ASCII character in the following way. Recall that ASCII 'b' is 98, which is 01100010.

```
//the pth position is b
macro is_b(var1 p, var1 q) =
q = p + 1 &
p notin bit0 & p in bit1
& p notin bit2 & p notin bit3
& p notin bit4 & p in bit5
& p in bit6 & p notin bit7;
```

One can specify that the p_{th} position of a string is any ASCII character by simply removing the constraints on p.

```
//the pth position is anything in S
macro is_S(var1 p, var1 q) =
q = p + 1;
```

To concatenate two strings, we first define consecutive positions in an integer set. We say p and q are two consecutive integers in P, if there does not exist r in P and p¡r¡q, which can be specified as the following predicate in MONA.

```
macro consecutive_in_set(var1 p, var1 q, var2 P) =
    p < q & p in P & q in P &
    all1 r: p < r & r < q => r notin P;
```

Then one can concatenate two regular expressions, e.g., Ex and Ey, in the following way.

```
pred is_ExEy(var1 p, var1 q) =
ex1 r where p<=r & r<=q: is_Ex(p,r) & is_Ey(r, q);
```

A star closure, e.g., E*, can be specified as a predicate that there exists an integer set P such that for any two consecutive positions r and r', the substring between them is in E. The size of P indicates the repeated times of E. Since the size of P can be arbitrary, the following predicate specifies the star closure of E.

```
pred is_E_star(var1 p, var1 q) =
ex2 P: p in P & q in P &
all1 r,r': consecutive_in_set(r, r', P) => is_E(r, r');
```

In sum, any regular expression can be specified as a MONA predicate, which can then be used by mona to generate a DFA encoded in BDDs.

Each time we encounter a new regular expression in PHP programs, we output the corresponding DFA by first generating a mona predicate accordingly. After we get the DFA, we can manipulate the DFA with previous defined functions.

3.1.3 Context String Replacement

In this section, we formally define context string replacement and propose a novel automata-based algorithm to manipulate this operation. We give a running example in the next section.

Definition Unconditional Context String Replacement: Given three DFAs, M_1, M_2, and M_3. Construct a FA M, such that $L(M) = \{y | \exists k \geq 0, w_1 x_1 w_2 \ldots w_k x_k w_{k+1} \in L(M_1), y = w_1 c_1 w_2 ... w_k c_k w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), c_i \in L(M_3), \forall 1 \leq i \leq k+1, w_i \notin L(M_2)\}$.

Our algorithm consists of constructing several internal DFAs. Each of them aims to satisfy a specified condition to construct the final DFA. Formally speaking, a DFA M is a tuple $< Q, Q_i, \Sigma, T, Q_f >$. Q is a finite set of states. Q_i is the initial state. Σ is a set of alphabet and Q_f is the final state. Let $\bar{x}$ denote a new string in which we add bar to each character in x. Assume $\sharp_1$,$\sharp_2 \notin \Sigma$, and $\forall x \in \Sigma, \bar{x} \notin \Sigma$.

- $\bar{M_1}$, such that $L(\bar{M_1}) = \{\bar{w} | \exists k \geq 0, w = w_1 x_1 w_2 \ldots w_k x_k w_{k+1} \in L(M_1), \bar{w} = w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \ldots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1}\}$.
- $\bar{M_2}$, such that $L(\bar{M_2}) = \{\bar{w} | \bar{w} = w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \ldots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1}, \forall 1 \leq i \leq k, x_k \in L(M_2), \forall 1 \leq i \leq k+1, w_k \notin L(M_2)\}$
- $\bar{M_3}$, $L(\bar{M_3}) = \{\bar{w} | \exists k \geq 0, w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \ldots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1} \in L(\bar{M_1}) \cap L(\bar{M_2}), \bar{w} = w_1 \sharp_1 c_1 \bar{x_1} \sharp_2 w_2 \ldots w_k \sharp_1 c_k \bar{x_k} \sharp_2 w_{k+1}, \forall 0 \leq i \leq k, c_k \in L(M_3)\}$.
- $M = \bar{M_3}|_\Sigma$.

$M|_\Sigma$ denotes *projection* associated with Σ. This projection function replaces all edges labelled with $\{\sharp_1, \sharp_2\} \cup \bar{\Sigma}$ to an edge labelled with an empty string. These ϵ transitions induce the nondeterminism and hence, the projection of a DFA may result in a NFA (Non-determinstic Finite Automaton). Though a NFA can be converted to a DFA, in general, the number of states of the corresponding DFA is exponential to the number of states of the given NFA.

3.1.4 A running example

We demonstrate the DFAs of our running example in Figure 1.

3.2 Front-End

The front-end parses programs written in the PHP language, transforms them into a three address code format, and builds the corresponding Control Flow Graph (CFG). In these phases of the analysis, most of the PHP language feature are correctly handled, but there currently is limited support for code that uses object oriented features.

String analysis is built on top of our implementation of the interprocedural data flow framework described in [13]. The framework approaches interprocedural analysis using a “functional approach”: each procedure is treated as a structure of blocks and establishes relations between attribute data at its entry and related data at any of its nodes. Using these relations, attribute

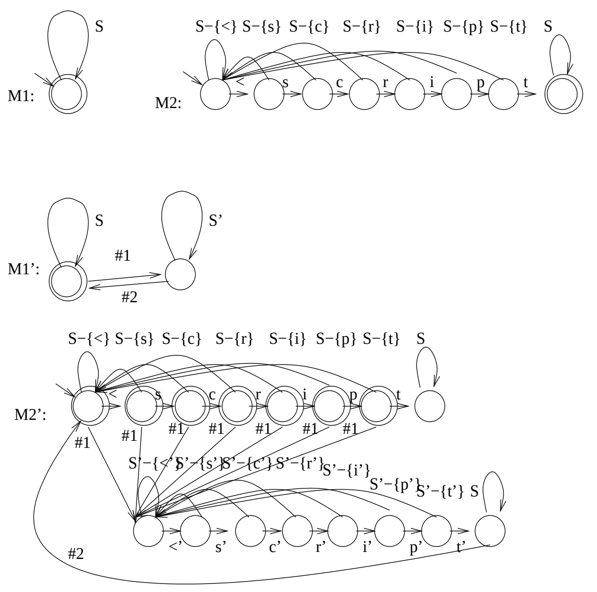
M1:
S
M2:
S−{<} S−{s} S−{c} S−{r} S−{i} S−{p} S−{t} S
< s c r i p t
M1':
S
S'
#1
#2
M2':
S−{<} S−{s} S−{c} S−{r} S−{i} S−{p} S−{t} S
< s c r i p t
#1
S'−{<'} S'−{s'} S'−{c'} S'−{r'} S'−{i'} S'−{p'} S'−{t'} S
<' s' c' r' i' p' t'
#2

Figure 1: The internal DFAs of the running example

data is propagated through each procedure call in a program. Unlike other algorithms for interprocedural data flow analysis, e.g., [12], the framework of [13] does not require the set of data flow facts to be finite or the data flow functions to be distributive, and, therefore, it can be applied to implement a larger number and type of analyses.

In string analysis, the set of data flow facts consists of strings in the alphabet of ASCII characters. We use FAs to represent sets of strings: a FA represents the set of strings that correspond to the language accepted by the automaton. In particular, we encode finite automata using the DFA format specified in the MONA package. The meet operator for our analysis is the union operation on FAs.

The analysis is performed in two steps. First, transfer functions are assigned to each node in the CFG of the program. Besides the standard *id*, *top* and *bottom* functions, we have implemented functions to propagate string information along CFG nodes where string assignment, concatenation, or replacement is performed. We also define functions to model the effect of specific library functions, which, e.g., return string values by reading the contents of external data sources, such as databases and files. This step of the analysis simply requires an inspection of each node of the CFG and, thus, is linear in the size of the analyzed program.

The second step of the analysis is the actual computation of the data flow facts. This step is performed leveraging the iterative, workpile-driven algorithm described in [13]. When the meet operator or one of the basis operations needs to be applied (e.g., in a transfer function), the front-end invokes the implementation of the corresponding operation provided by the MONA-based back-end. Currently, an external program (written in the C language) provides the interface through which the front-end and the back-end communicate. All data required to execute a string operation is exchanged using the DFA format specified by MONA.

For example, consider the statement `$www = "http://" . $www;` in our sample program. During the first phase of the analysis, it is assigned the *concatenate* transfer function, which models the concatenation of two strings. During the second phase, when information is propagated through the node corresponding to this statement, the transfer function is applied. The transfer function retrieves the DFAs corresponding to the values associated with the string `http://` and the variable `$www` at current point of the analysis. Then, it invokes the interface program to execute the concatenation of the two DFAs. The concatenation operation is performed by the back-end, and the result, a DFA whose language consists of the concatenation of the languages of the input DFAs, is sent back to the front-end. Finally, the front-end updates the known facts about the variable `$www` storing the returned DFA.

Depending on the specific problem at hand, the results of the analysis can be used in different ways. Consider, for example, the problem of finding cross-site scripting vulnerabilities in web applications. An application is vulnerable to cross-site scripting attacks if it can be tricked into storing malicious code (typically JavaScript code) from an attacker and then presenting the malicious code to users [8]. In this case, victim users will execute the code under the assumption that it originates from the (trusted) application, rather than from the attacker. A well-understood method to check for the presence of cross-site scripting vulnerabilities consists of determining if an application can send back to a user data containing malicious JavaScript code. This approach can be easily implemented using string analysis. In fact, it is sufficient to determine all the program points where the application sends attacker-controlled data to the user and, for each of these points, determine if the set of possible string values of the transmitted data (as determined by our analysis) intersects the language of malicious code. Note that intersection and emptiness check are standard operations on DFAs and therefore are easily implemented by our back-end component.

Finally, we implemented a taint analysis for PHP applications. Taint analysis is a data flow analysis that determines the set of program variables whose value is directly or indirectly under control of a user of a program. We use taint analysis to perform the first step in the detection of cross site scripting vulnerabilities: the identification of program points where attack-controlled data is sent to a user. We specify a set of *taint sources*, i.e., sources of possibly attacker-controller data (e.g., request parameters, database content), and a set of transfer functions that model how taint is propagated through the application. Then, by reusing our implementation of the interprocedural data flow framework, we can determine the taintedness of any variable in the program.

4 Current status and Future Work

We have implemented a prototype tool, called *STRANGER* (standing on *STR*ing *A*utomato*N* *GE*nerato*R*). The front end is implemented in PHP, which can parse general PHP programs, analyze control flows, and proceed taint analysis. The backend incorporated with MONA DFA packages is implemented in C. STRANGER can proceed string analysis by combining the front end and the back end as we described in the previous section. For basic operations, STRANGER shows some promise in performance.

However, our current version is incomplete in several folds: a) the input files of mona predicates are generated manually, and b) the implementation of concatenate and replace operations of our backend is not exactly as we specify. In our future work, we will first fix these defects, and then apply our method to real applications.

References

[1] BRICS. The MONA project. `http://www.brics.dk/mona/`.

[2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[3] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. String analysis for x86 binaries. In *Proceedings of the 6th ACM SIGPLAN-*

SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005). ACM Press, September 2005.

[4] Dale Gerdemann and Gertjan van Noord. Transducers from rewrite rules with backreferences. In *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics*, pages 126–133, 1999.

[5] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, 2004.

[6] Lauri Karttunen. The replace operator. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 16–23, 1995.

[7] Christian Kirkegaard, Anders Mller, and Michael I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Transactions on Software Engineering*, 30(3), March 2004.

[8] A. Klein. Cross site scripting explained. Technical report, Sanctum Inc., 2002.

[9] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441, 2005.

[10] Mehryar Mohri, Fernando C. N. Pereira, and Michael D. Riley. AT&T FSM library – finite-state machine library. `http://www.research.att.com/~fsmtools/fsm/`.

[11] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 231–238. Association for Computational Linguistics, 1996.

[12] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[13] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, 1981.

[14] Nathan Vaillette. Logical specification of regular relations for NLP. *Natural Language Engineering*, 9(1):65–85, 2003.

[15] Gertjan van Noord. FSA utilities toolbox. `http://odur.let.rug.nl/~vannoord/Fsa/`.

[16] Gertjan van Noord and Dale Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Proc. of the 4th International Workshop on Implementing Automata (WIA)*, pages 122–139. Springer-Verlag, July 1999.