Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic

TAKAAKI TATEISHI, IBM Research - Tokyo
MARCO PISTOIA, IBM T. J. Waton Research Center
OMER TRIPP, IBM Software Group and Tel Aviv University

We propose a novel technique for statically verifying the strings generated by a program. The verification is conducted by encoding the program in Monadic Second-Order Logic (M2L). We use M2L to describe constraints among program variables and to abstract built-in string operations. Once we encode a program in M2L, a theorem prover for M2L, such as MONA, can automatically check if a string generated by the program satisfies a given specification, and if not, exhibit a counterexample. With this approach, we can naturally encode relationships among strings, accounting also for cases in which a program manipulates strings using indices. In addition, our string analysis is path sensitive in that it accounts for the effects of string and Boolean comparisons, as well as regular-expression matches.

We have implemented our string-analysis algorithm, and used it to augment an industrial security analysis for Web applications by automatically detecting and verifying *sanitizers*—methods that eliminate malicious patterns from untrusted strings, making those strings safe to use in security-sensitive operations. On the 8 benchmarks we analyzed, our string analyzer discovered 128 previously unknown sanitizers, compared to 71 sanitizers detected by a previously presented string analysis.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification;
D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Languages, Security, Verification

Additional Key Words and Phrases: String Analysis, Static Program Analysis, Web Security

ACM Reference Format:
Tateishi, T., Pistoia, M., Tripp, O. 2012. Path- and index-sensitive string analysis based on monadic second-order logic. ACM Trans. Softw. Eng. Methodol. 9, 4, Article 39 (March 2010), 30 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

String analysis [Christensen et al. 2009; Geay et al. 2009; Christensen et al. 2003; Hooimeijer and Weimer 2009; Kieżun et al. 2009a; Minamide 2005; Wassermann and Su 2007] is a particular form of program analysis whose purpose is to infer string values arising at run time. It is often used in the verification of server-side Web applications, where string values used in security-sensitive computations are compared to safe and/or unsafe string patterns to detect potential security vulnerabilities, such as cross-site scripting (XSS), HTTP response splitting (HRS) and Structured Query Language (SQL) injection (SQLi) [Web].

1.1. String Analysis for Security

A common way of conducting string analysis is by constructing context-free grammars or regular grammars to approximate strings [Christensen et al. 2003; Minamide 2005; Wassermann and Su

This paper is a revised and extended version of a paper presented at ISSTA 2011 in Toronto, Canada.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
String clean(String v1){
  String v2 = "<";
  if (v1.contains(v2)) {
    int v3 = v1.indexOf(v2);
    String v4 = v1.substring(0, v3);
    return v4;
  }
  return v1;
}
```

Fig. 1. Sanitization against XSS

2007]. With this approach, each built-in string operation is modeled by a grammar transducer. This form of string analysis is suitable for analyzing code that *sanitizes* strings using string-manipulation operations such as Java's `replace` method. Many Web applications fall in this category because they sanitize their inputs by *removing* potentially malicious string patterns or *replacing* them with safe ones. Sanitizers often perform validation against certain patterns, and process the inputs only if validation succeeds. For these validation-based sanitizers, a path-sensitive string analysis is necessary. Conversely, path-insensitive string analyses will conservatively report violations even when proper validation takes place.

In addition, a very large number of Web applications perform sanitization by extracting substrings from input strings, starting at specific indices. Grammar-based string analyses are unable to precisely verify strings that are constructed in this way, and will have to report violations conservatively even when proper index-based sanitization has taken place. Consequently, path-sensitivity and the ability to model index-based string manipulation are essential features when verifying Web applications for security.

1.2. Motivating Example

The Java method `clean` in Figure 1 can be used to prevent an XSS attack. The input to the method can be any possible value, including values potentially under the control of an attacker. In XSS, an attacker typically wraps JavaScript code into a (`<script>`, `</script>`) tag pair, and embeds it into text that becomes part of an online encyclopedia, blog, or social network. Once the text is rendered on other people's browsers, the embedded code is automatically executed on the victims' computers. For this example, we consider the output safe if it does not contain character <.

In order to verify that the program is immune to XSS attacks, we need to prove that no string generated by the program contains <. According to this specification, `clean` is considered a valid XSS sanitizer; when condition `v1.contains(v2)` holds, < is effectively removed from the input string by combining `indexOf` and `substring`, and when that condition does not hold, the string value returned by the method is the same as the input string, which does not contain <. However, a path-insensitive grammar-based string analysis cannot follow this line of reasoning, since it would fail to capture the relationship between `v2` and `v3`, and thus the effect of the ensuing `substring` operation. As a consequence, it produces a resulting grammar that conservatively contains <.

1.3. Our Approach

To abstract string values, we use M2L(Str) (Monadic Second-order Logic on strings) [Henriksen et al. 1995]. The effect of branch conditions and dependencies among program variables is abstracted and encoded as M2L formulae. Built-in string operations are also abstracted by M2L formulae, with each formula representing relationships among input and output parameters. In particular, a string operation using an index can be represented naturally by a M2L formula, since M2L(Str) is capable of explicitly mentioning *positions* in a given finite string and can deal with variables ranging over positions (*position variables*) or variables ranging over sets of positions (*position set variables*) on the finite string.

The use of M2L(Str) has the following advantages in addition to enabling index sensitivity combined with path sensitivity:

— *Conservativeness.* M2L(Str) captures not only fixed-size strings but also finite strings (regular languages). This feature is necessary for guaranteeing that our string analysis is conservative, which implies that sanitization code verified by our string analysis can be safely used as a sanitizer, and for conservatively modeling several important built-in string operations such as Java's `replace` method in a manner similar to finite-state transducers that cannot be captured by fixed-size representation.
— *Efficient and effective automaton representation.* We can exploit an automatic theorem prover MONA [Klarlund and Møller 2001] to implement our string analysis algorithm, where MONA uses the BDD-based automaton representation of M2L formulae. Furthermore, the use of MONA has potential to advance the string analysis implementation in the future, since MONA enables performing separate compilation and generating constraints on input strings (like vulnerability signatures [Brumley et al. 2007; Yu et al. 2009]) including counterexamples.

Our analysis consists of the following two automated processes: (i) encoding a string-manipulating method as an M2L formula ϕ_1 that represents possible strings returned by the method, and (ii) encoding a regular expression indicating unsafe strings as an M2L formula ϕ_2, and checking the satisfiability of $\phi_1 \wedge \phi_2$ to verify that the possible strings returned by the method never contain any of unsafe strings, where the method is reported as a sanitizer iff the formula is unsatisfiable. In the first process, the effect of branch conditions and index-based string manipulations are also encoded in M2L, and reflected to the M2L formula ϕ_1. Therefore, our string analysis is both path-sensitive and index-sensitive, and thus we call it PISA (Path- and Index-sensitive String Analysis) in this paper.

1.4. Contributions

This paper makes the following contributions:

— *Novel features enabled by M2L.* Our encoding method goes beyond that for regular expressions [Klarlund and Møller 2001]. Compared to existing string analyses based on bit-vector logic and/or word equation [Bjørner et al. 2009; Kieżun et al. 2009a; Saxena et al. 2010], our M2L-based approach can model more string transformations such as replacement and upper-case transformations. Furthermore, to the best of our knowledge, PISA is the first purely static string analysis that simultaneously handles index sensitivity, path sensitivity, and string-replacement operations.
— *Sanitizer detection by path- and index-sensitive string analysis.* String analysis has already been used for sanitizer detection [Balzarotti et al. 2008]. However, [Balzarotti et al. 2008] uses an imprecise string analysis, which is neither index nor path sensitive, and compensates for this loss in precision by relying on a complementary dynamic analysis. PISA, on the other hand, is much more precise, which obviates the need for an accompanying dynamic analysis. This also enables scanning applications during the development phase, where they cannot yet be deployed (and thus dynamic analysis cannot be used), which is the optimal stage for detecting security vulnerabilities.
— *Implementation and evaluation.* PISA is fully implemented and is featured in a commercial security product [App]. We evaluated PISA's precision by comparing it with the technique of [Minamide 2005; Geay et al. 2009] on 8 open-source benchmarks. We further examined PISA's effectiveness by integrating it into a commercial taint-analysis algorithm. The results show PISA to be far more precise than the previous technique, and also effective in boosting the precision of its client taint analysis.

1.5. Organization

The rest of the paper is organized as follows: In Section 2, we present the overview of our string analysis algorithm. The core string-analysis algorithm is described in Section 3. Then, in Section 4, we extend the core algorithm with index sensitivity and path sensitivity. In Section 5, we further extend our analysis with features for interprocedural analysis and for handling regular expression operations. Section 6 discusses our implementation of the algorithm, as well as experimental results. Section 7 surveys related work, and Section 8 concludes this paper.

$$
\begin{array}{llcl}
\text{(Position variable)} & p & \in & \mathsf{Var1} \\
\text{(Position-set variable)} & P & \in & \mathsf{Var2} \\
\text{(Position term)} & t & ::= & p \mid t+i \mid t-i \mid \$ \mid 0 \\
\text{(Position-set term)} & T & ::= & \emptyset \mid \{t,\ldots,t\} \mid \mathsf{all} \mid P \mid T\cup T \mid T\cap T \\
 & & \mid & T\setminus T \mid T^{-1} \\
\text{(Formula)} & \phi & ::= & \text{`}a\text{'}(t) \mid t=t \mid t<t \mid t\leq t \\
 & & \mid & T=T \mid T\subset T \mid T\subseteq T \mid t\in T \\
 & & \mid & \neg\phi \mid \phi\wedge\phi \mid \phi\vee\phi \mid \phi\Rightarrow\phi \mid \phi\Leftrightarrow\phi \\
 & & \mid & \exists p.\phi \mid \forall p.\phi \mid \exists P.\phi \mid \forall P.\phi
\end{array}
$$

Fig. 2. Syntax of M2L(Str)

2. OVERVIEW

Our string analysis verifies a program by encoding it in M2L(Str) and then checking the satisfiability of an M2L formula. Therefore, we first present the definition of M2L(Str), and then briefly describe how to encode strings and programs in M2L(Str), followed by describing how to verify programs using MONA.

2.1. Monadic Second-order Logic on Strings

M2L(Str) is a widely used vehicle for a variety of verification problems [Henriksen et al. 1995]. Intuitively, an M2L(Str) formula is interpreted relative to a number $n \geq 0$, which is thought of as a set of positions $\{0,\ldots,n-1\}$ in a finite string w of length n. The syntax of M2L(Str) is defined in Figure 2, where $\mathsf{Var1}$ denotes a set of position variables and $\mathsf{Var2}$ a set of position-set variables. The formula 'a'(t) holds if a_i in (finite) string $w = a_0 \cdots a_{n-1}$ is 'a', where i is the interpretation of t. Constants 0 and \$ represent the first and last positions in a string, respectively. The addition $t+i$ of position-term t and natural number i is interpreted as $t+i = j+i$ mod n, where j is the interpretation of t, and n is the length of string w. $T+i$, where T is a position-set term, results in position set $\{t+i \mid t \in T\}$. $t-i$ and $T-i$ are interpreted similarly.

Its semantics is determined by checking whether an M2L formula ϕ holds on a finite string $w \in \Sigma^*$ and an assignment $\mathcal{I} \in (\mathcal{P} \to 2^{\mathsf{Pos}})$, where $\mathcal{P}$ is the set of free position set variables[1], and 2^{Pos} is the power set of the position set Pos. When the formula holds, we write $w,\mathcal{I} \models \phi$. Satisfiability of an M2L formula ϕ is checked by finding a finite string w and an assignment $\mathcal{I}$ that make $w,\mathcal{I} \models \phi$ hold. If there exist w and $\mathcal{I}$ that make $w,\mathcal{I} \models \phi$ hold, ϕ is satisfiable. For example, the M2L formula '`a`'$(0) \wedge$ '`b`'$(1) \wedge$ '`c`'$(2) \wedge$ '`a`'(3) holds on the finite string $w =$ "`abca`", which states that character '`a`' is located at positions 0 and 3, while characters '`b`' and '`c`' are located at positions 1 and 2, respectively. Thus, '`a`'$(0) \wedge$ '`b`'$(1) \wedge$ '`c`'$(2) \wedge$ '`a`'(3) is satisfiable. In contract, '`a`'$(0) \wedge$ '`b`'(0) is unsatisfiable, since we cannot find any finite string w in which both '`a`' and '`b`' are located at the same position 0.

2.2. Encoding Programs in M2L(Str)

Our encoding method treats string values using position sets without loss of the order of characters. For example, given the formula '`a`'$(0) \wedge$ '`b`'$(1) \wedge$ '`c`'$(2) \wedge$ '`a`'$(3) \wedge P = \{0\} \wedge Q = \{1,3\} \wedge R = \{0,1,3\}$ holds on $w =$ "`abca`" and $\mathcal{I} = \{P \mapsto \{0\}, Q \mapsto \{1,3\}, R \mapsto \{0,1,3\}\}$, the position set variables P,Q, and R can be considered to be the strings "`a`", "`ba`" and "`aba`", respectively. In this representation of strings, the concatenation of the two strings represented by P and Q is captured by $P \cup Q$, which is equal to R, without loss of the order of characters, since all of the positions in P are less than any position in Q. Thus, this concatenation relationship can be represented by the predicate:

$$
\begin{array}{l}
\mathsf{concat}(R,P,Q) \equiv \\
(R = P \cup Q) \wedge (\forall p,q \,.\, p \in P \wedge q \in Q \Rightarrow p < q),
\end{array}
$$

[1]Position variables are handled by using singleton position set variables.

$$\begin{array}{lcl}
\mathsf{prog}_{\mathtt{v1}}(V_1) & \equiv & \mathsf{true} \\
\mathsf{prog}_{\mathtt{v2}}(V_2) & \equiv & \text{“<”}(V_2) \\
\mathsf{prog}_{\mathtt{v3}}(v_3, V_1) & \equiv & [\![\mathtt{indexOf}]\!](v_3, V_1, \mathsf{prog}_{\mathtt{v1}}, \mathsf{prog}_{\mathtt{v2}}) \\
\mathsf{prog}_{0}(v_0, V_1) & \equiv & \mathsf{min}(v_0, V_1) \\
\mathsf{prog}_{\mathtt{v1'}}(V_1) & \equiv & \mathsf{prog}_{\mathtt{v1}}(V_1) \wedge [\![\mathtt{contains("<")}]\!](V_1) \\
\mathsf{prog}_{\mathtt{v4}}(V_4) & \equiv & [\![\mathtt{substring}]\!](V_4, \mathsf{prog}_{\mathtt{v1'}}, \mathsf{prog}_{0}, \mathsf{prog}_{\mathtt{v3}}) \\
\mathsf{prog}_{\mathtt{v1''}}(V_1) & \equiv & \mathsf{prog}_{\mathtt{v1}}(V_1) \wedge \neg [\![\mathtt{contains("<")}]\!](V_1),
\end{array}$$

Fig. 3. An example of predicate declarations

where $p < q$ ensures the order of the characters in P and Q. If we do not have the constraint $p < q$, ‘b’$(0) \wedge$ ‘a’$(1) \wedge$ ‘a’$(2) \wedge P = \{1\} \wedge Q = \{0, 2\} \wedge \mathsf{concat}(R, P, Q)$ holds on $w =$ “baa” and $\mathcal{I} = \{P \mapsto \{1\}, Q \mapsto \{0, 2\}, R \mapsto \{0, 1, 2\}\}$, where P and Q represents “a” and “ba”, respectively, and R represents string “baa” that is not the concatenation of strings represented by P and Q. Based on this predicate, we introduce the notation “s”(S) which means that position-set variable S represents string s.

With this encoding of strings, a program is encoded as a set of M2L predicate declarations, where each M2L predicate is declared corresponding to the definition of a program variable. We also use a pre-defined predicate for every string operation to represent each constraint among the return value and the parameters, where any constraint can be abstracted. For example, let us consider the following two-line program.

```
String v1 = "a";
String v2 = v1.concat(v1);
```

Here is the set of predicate declarations when this program is encoded into M2L.

$$\begin{array}{l}
\mathsf{prog}_{\mathbf{v1}}(V_1) \equiv \text{“a”}(V_1) \\
\mathsf{prog}_{\mathbf{v2}}(V_2) \equiv [\![\mathtt{concat}]\!](V_2, \mathsf{prog}_{\mathtt{v1}}, \mathsf{prog}_{\mathtt{v1}}) \\
\text{where} \quad [\![\mathtt{concat}]\!](R, \mathcal{P}_1, \mathcal{P}_2) \equiv \\
\qquad \exists P_1, P_2 \,.\, \mathcal{P}_1(P_1) \wedge \mathcal{P}_2(P_2) \wedge \mathsf{concat}(R, P_1, P_2)
\end{array}$$

Each predicate prog_{v_i} represents the post-condition for the assignment to program variable v_i, where the parameters V_1 and V_2 of the predicates are associated with the program variables `v1` and `v2`, respectively. $[\![\mathtt{concat}]\!]$ is a pre-defined predicate representing an abstraction of the string concatenation operation, where the return value is represented by R, and the parameters are represented by $\mathcal{P}_1$ and $\mathcal{P}_2$ which are instantiated by the predicates associated with the program variables.

With this encoding of programs, the `clean` method of Figure 1 is encoded as the set of predicate declarations of Figure 3, where we introduce the program variables `v1'` and `v1''` to distinguish the program variable `v1` used in the “then” block of the if-statement from that returned at the end of the method. The variables V_1, V_2, v_3, and V_4 are also M2L variables associated with the program variables `v1`,`v2`,`v3`, and `v4`, respectively, where upper-case variables V_1,V_2, and V_4 are position set variables each of which represents a string, and the lower-case variable v_3 is a position variable which represents an index. Note that the effect of the branch condition is encoded as constraints on these M2L variables in the declarations of the predicates $\mathsf{prog}_{\mathtt{v1'}}$ and $\mathsf{prog}_{\mathtt{v1''}}$, where $[\![\mathtt{contains("\langle")}]\!](V_1)$ [2] is the abstraction of the condition `contains("⟨")`, which means that the string represented by V_1 contains the string “<”. The predicate prog_0 and $\mathsf{prog}_{\mathtt{v3}}$ means index 0 and an index assigned to the program variable `v3`, where each of the predicates takes two parameters: a position variable and a position set variables, since we represents an index of a string using the pair of a position and a position set. $[\![\mathtt{indexOf}]\!]$ and $[\![\mathtt{substring}]\!]$ are the abstractions of string operations `indexOf` and `substring` as in the case of $[\![\mathtt{concat}]\!]$, respectively. The details of these abstractions are discussed later in Section 3 and 4.

[2]The reason why we do not use $[\![\mathtt{contains}]\!](V_1, V_2)$ is that M2L as well as our encoding method cannot treat equality of strings. Therefore, our encoding method relies on constant propagation analysis to obtain concrete strings such as “<” to avoid this limitation.

(Assignment)	$x = v$	The value v is assigned to program variable x.
(Function call)	$x = f(x_1, \cdots, x_n)$	The result of invoking function f with parameters $x_1, \cdots, x_n$ is assigned to program variable x. f is either a built-in function or a user-defined function.
(ϕ function)	$x = \texttt{phi}(b_1 : x_1, \ldots, b_n : x_n)$	When an immediate predecessor of the current basic block is b_i, program-variable x is assigned the value of x_i. Basic-block numbers are omitted for brevity when possible.
(Conditional jump)	`jump` $x,\ b$	The program jumps to the basic block numbered b if the value of x is true.
(Goto)	`goto` b	The program jumps to the basic block numbered b.
(Return)	`return` x	The value of the program variable x is returned.

Fig. 4. Instructions of the target language

2.3. Verifying Programs Using MONA

If our only concern is that the string "<" is unsafe, an unsafe specification Unsafe for the program is defined as

$$\mathsf{Unsafe}(V) \equiv \exists R' \,.\, \mathsf{substr}(R', V) \wedge \text{“<”}(R'),$$

where $\mathsf{substr}(R', V)$ means that the string R' is a substring of V. Alternatively, we can use the regular expression "`.*<.*`" which is equivalent to the specification, since we can encode the regular expression into M2L. We then verify that the program never returns a string containing the unsafe string by confirming the unsatisfiability of this formula using MONA:

$$(\mathsf{prog}_{\texttt{v1''}}(V) \vee \mathsf{prog}_{\texttt{v4}}(V)) \wedge \mathsf{Unsafe}(V),$$

where V is a free position-set variable.

If we suppose that both the strings "<" and ">" are unsafe, a corresponding unsafe specification Unsafe$'$ is defined as

$$\mathsf{Unsafe}'(V) \equiv \exists R' \,.\, \mathsf{substr}(R', V) \wedge (\text{“<”}(R') \vee \text{“>”}(R')).$$

We then construct the following formula, and check its satisfiability using MONA:

$$(\mathsf{prog}_{\texttt{v1''}}(V) \vee \mathsf{prog}_{\texttt{v4}}(V)) \wedge \mathsf{Unsafe}'(V),$$

where V is a free position-set variable. MONA reports that this formula is satisfiable, since finite string $w =$ ">" and position set $\mathcal{I} = \{V \mapsto \{0\}\}$ make the above formula hold, where V represents the string ">". Thus, we find that the program may return a string containing the string ">".

3. CORE ALGORITHM

In this section, we first describe our target language, and then describe the method of encoding strings and regular expressions followed by the method of encoding a program as a set of M2L predicate declarations. We also present a formal discussion of our encoding method, accompanied by a soundness theorem.

3.1. Target Language

We assume that the target program is translated to Static Single Assignment (SSA) form [Rosen et al. 1988; Cytron et al. 1991]. An SSA program comprises numbered basic blocks, each of which consists of the instructions in Figure 4. In addition, we use the notation $\texttt{op}(h)$ for built-in operator h. For example, `x=op(+)(1,2)` assigns the result of `1+2` to program variable `x`. Note that the return instruction and the instruction of calling a user-defined function are introduced only for reflecting actual program languages such as Java, as the core of our string analysis algorithm is intraprocedural. We omit the details of the method of encoding those types of instructions in Section 3.

$$
\begin{array}{lcl}
\text{“}a_1 \cdots a_n\text{”}(P) & \equiv & \exists t_1, \cdots, t_n \,.\, \text{‘}a_1\text{’}(t_1) \wedge \cdots \wedge \text{‘}a_n\text{’}(t_n) \\
 & & \wedge\, t_1 < t_2 \wedge t_2 < t_3 \wedge \cdots \wedge t_{n-1} < t_n \wedge\ P = \{t_1, \cdots t_n\} \\
\mathsf{concat}(R, P, Q) & \equiv & (R = P \cup Q) \wedge (\forall p, q \,.\, p \in P \wedge q \in Q \Rightarrow p < q) \\
\mathsf{strr}(R, P, p, q) & \equiv & p \leq q \wedge R \subseteq P \wedge (\forall r \,.\, r \in P \Rightarrow (r \in R \Leftrightarrow p \leq r \wedge r < q)) \\
\mathsf{substrr}(R, P, p, q) & \equiv & \exists p', q' \,.\, p \leq p' \wedge p' \leq q' \wedge q' \leq q \wedge \mathsf{strr}(R, P, p', q') \\
\mathsf{substr}(R, P) & \equiv & \mathsf{substrr}(R, P, \mathsf{min}(P), \mathsf{max}(P) + 1) \\
\mathsf{consecutive}(p, q, R) & \equiv & p < q \wedge p \in R \wedge q \in R \wedge (\forall r \,.\, p < r \wedge r < q \Rightarrow r \notin R)
\end{array}
$$

Fig. 5. Utility predicates

$$
\begin{array}{lcl}
\langle\!\langle T \rangle\!\rangle & \rightarrow & \lambda S \,.\, \text{‘}T\text{’}(S) \\
\langle\!\langle T_x \rangle\!\rangle & \rightarrow & \lambda S \,.\, \mathsf{prog}_x(S) \\
\langle\!\langle N_1 N_2 \rangle\!\rangle & \rightarrow & \lambda S \,.\, \exists S_1, S_2 \,.\, \langle\!\langle N_1 \rangle\!\rangle (S_1) \wedge \langle\!\langle N_2 \rangle\!\rangle (S_2) \wedge \mathsf{concat}(S, S_1, S_2) \\
\langle\!\langle N_1 \mid N_2 \rangle\!\rangle & \rightarrow & \lambda S \,.\, \langle\!\langle N_1 \rangle\!\rangle (S) \vee \langle\!\langle N_2 \rangle\!\rangle (S) \\
\langle\!\langle N^\star \rangle\!\rangle & \rightarrow & \lambda S \,.\, \exists P \,.\, \mathsf{min}(S) \in P \wedge \mathsf{max}(S) + 1 \in P \\
 & & \wedge \forall r, r' \,.\, \mathsf{consecutive}(r, r', P) \Rightarrow \exists Q \,.\, \mathsf{strr}(Q, S, r, r') \wedge \langle\!\langle N \rangle\!\rangle (Q)
\end{array}
$$

Fig. 6. Encoding regular expressions

However, this does not restrict us from conducting an interprocedural analysis, since our algorithm can be simply extended so as to conduct the interprocedural analysis by treating assignment relationships between caller's program variables and callee's program variables as described in Section 5.1. Note that the core analysis algorithm itself cannot be extended so as to control context sensitivity, since context sensitivity of the interprocedural version of our analysis relies on an underlying callgraph construction algorithm.

The following SSA program represents the `clean` method introduced in Section 1, where `1:`, `2:`, and `3:` represent the basic blocks numbered 1,2, and 3.

```
1:v0 = 0;                        2:return v1;
  v2 = "<";                      3:v3 = indexOf(v1,v2);
  b1 = v1.contains(v2);            v4 = substring(v1,v0,v3);
  jump b1, 3                       return v4;
```

3.2. Encoding Strings

Our encoding method treats a string value of size m as a position set P of the same size. The positions in P are taken from a "global" position set, $\{0, \ldots, n-1\}$, that represents a word w. Formally, if $w = a_0 \cdots a_{n-1}$ and $P = \{p_0, \cdots, p_{m-1}\}$ is a sorted position set, then P represents the string value s (of size m) iff P satisfies $s = a_{p_0} a_{p_1} \cdots a_{p_{m-1}}$. Given $w =$ "`abca`", the sets of positions $\{0, 1\}$ and $\{2, 3\}$ represent the strings "`ab`" and "`ca`", respectively.

Figure 5 lists utility predicates used for the encoding, where "a"(P) and $\mathsf{concat}(R, P, Q)$ are the same as those introduced in Section 2.2. Intuitively, $\mathsf{strr}(R, P, p, q)$ denotes that a string represented by R is the substring represented by P containing all the characters in the range $[p, q)$. $\mathsf{substrr}(R, P, p, q)$ is similar to strr, the difference being that R may be any substring. $\mathsf{min}(P)$ and $\mathsf{max}(P)$ return the minimum and maximum positions in P, respectively. Finally, predicate $\mathsf{consecutive}(p, q, R)$ denotes that positions p and q are consecutive in position set R. This predicate is used to encode the Kleene closure of a regular language (*cf.* consecutive_in_set, as described in [Klarlund and Møller 2001]).

Our algorithm for encoding regular expressions is the same as that of [Klarlund and Møller 2001], except that we accept program variables as terminal symbols. The basic idea of this algorithm is to recursively encode a regular expression as an M2L formula according to the structure of the regular expression.

Figure 6 shows our entire algorithm of encoding a set of strings represented by a regular expression r as a predicate denoted by $\langle\!\langle r \rangle\!\rangle$. T represents a terminal symbol (*i.e.*, a character), and T_x represents a terminal symbol associated with program variable x, where prog_x denotes a property of strings possibly assigned to x. N,N_1, and N_2 represent nonterminal symbols. In addition, we use the notation $\lambda S.\phi$, instead of explicitly declaring a new predicate ψ such that $\psi(S) = \phi$.

$$\llbracket \texttt{replace} \rrbracket (R, \mathcal{P}_s, \mathcal{P}_x, \mathcal{P}_y) \equiv \bigvee_{v \in V} (\exists S, X, Y \,.\, \mathcal{P}_s(S) \wedge (\forall S' \,.\, \mathsf{substr}(S', S \setminus X) \Rightarrow \neg \text{“}v\text{”}(S'))$$
$$\wedge \langle\langle v^\star \rangle\rangle' (X, S) \wedge \langle\langle y^\star \rangle\rangle (Y) \wedge \langle\langle (vy)^\star \rangle\rangle (X \cup Y) \wedge S \cap Y = \emptyset$$
$$\wedge R = ((S \setminus X) \cup Y))$$

where

$$\langle\langle v^\star \rangle\rangle' (X, S) \equiv \exists P \,.\, \mathsf{min}(X) \in P \wedge \mathsf{max}(X) + 1 \in P$$
$$\wedge \forall r, r' \,.\, \mathsf{consecutive}(r, r', P)$$
$$\Rightarrow \exists Q \,.\, \mathsf{strr}(Q, X, r, r') \wedge \mathsf{substr}(Q, S) \wedge \langle\langle v \rangle\rangle (Q)$$

$$\llbracket \texttt{indexOf} \rrbracket (p, P, \mathcal{P}_1, \mathcal{P}_2) \equiv \mathcal{P}_1(P) \wedge \bigvee_{v \in V} ((\exists P_2 \,.\, \text{“}v\text{”}(P_2) \wedge \mathsf{indexOf}(p, P, P_2))$$
$$\wedge (\mathsf{min}(P) \leq p \Rightarrow (\forall P_2, p' \,.\, \text{“}v\text{”}(P_2) \wedge \mathsf{indexOf}(p', P, P_2)$$
$$\wedge \mathsf{min}(P) \leq p' \Rightarrow p \leq p')))$$

where

$$\mathsf{indexOf}(p, P, Q) \equiv \mathsf{substr}(Q, P) \Rightarrow ((Q \neq \emptyset \Rightarrow \mathsf{min}(Q) = p) \wedge (Q = \emptyset \Rightarrow \mathsf{min}(P) = p))$$
$$\wedge (\neg \mathsf{substr}(Q, P) \Rightarrow p < \mathsf{min}(P)).$$

$$\llbracket \texttt{substring} \rrbracket (R, \mathcal{P}_s, \mathcal{P}_n, \mathcal{P}_m) \equiv \exists S, n, m \,.\, \mathcal{P}(S) \wedge \mathcal{P}_n(n, S) \wedge \mathcal{P}_m(m, S) \wedge \mathsf{strr}(R, S, n, m)$$

$$\llbracket \texttt{contains}, 1 \rrbracket (c, R, \mathcal{P}_1, \mathcal{P}_2) \equiv \begin{cases} \exists P \,.\, \mathcal{P}_2(P) \wedge \mathsf{substr}(P, R) & \text{when } c = \mathsf{true} \\ \bigvee_{s \in S} \neg (\exists P \,.\, \text{“}s\text{”}(P) \wedge \mathsf{substr}(P, R)) & \text{when } c = \mathsf{false} \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Fig. 7. Abstractions of the built-in functions

Encoding terminals T, T_x, concatenations $N_1 N_2$, and choice operations $N_1 | N_2$ is straightforward. For encoding a Kleene closure $N^\star$, we use $\mathsf{consecutive}(r, r', P)$ to break down a string matched by regular expression $N^\star$ into a consecutive set of substrings each of which is located between positions r and r' and matched by regular expression N, where r and r' are consecutive positions in P.

3.3. Abstracting Built-in Functions

We denote the abstraction of a built-in function f by $\llbracket f \rrbracket$. The parameters of a built-in function are implicitly represented by *higher-order variables*, each of which is instantiated by a predicate representing a property of the relevant actual parameter. Thus, all higher-order variables are instantiated by the end of the encoding process. For example, the higher-order variables $\mathcal{P}_1$ and $\mathcal{P}_2$ used in the encoding of the string-concatenation program in Section 2.2 are instantiated by predicate $\mathsf{prog}_{\mathtt{v1}}$, thus yielding:

$$\mathsf{prog}_{\mathtt{v2}}(V_2) \equiv \exists P_1, P_2 \,.\, \mathsf{prog}_{\mathtt{v1}}(P_1) \wedge \mathsf{prog}_{\mathtt{v1}}(P_2) \wedge \mathsf{concat}(V_2, P_1, P_2).$$

The examples of the abstractions we developed for the built-in functions are listed in Figure 7. Here, we focus only on $\llbracket \texttt{replace} \rrbracket$, which abstracts the Java method `replace`, where `s.replace(x,y)` substitutes all occurrences of `x` in `s` with `y`. Discussion of the other functions is deferred to Section 4, where extensions of the core algorithm needed by the corresponding abstractions are introduced. The abstraction of `replace` is defined based on the idea that string replacement is considered the iteration of removing and inserting characters. V is the set of concrete strings possibly assigned to `x`, X represents the set of positions to be removed from position-set S, and Y represents a set of positions to be inserted. Predicate $\langle\langle y^\star \rangle\rangle$ encodes regular expression $y^\star$, where y is the program variable corresponding to $\mathcal{P}_y$. Predicate $\langle\langle (vy)^\star \rangle\rangle$ constrains X and Y to guarantee that each pair of removed and inserted positions is consecutive, and predicate $\langle\langle v^\star \rangle\rangle'$ encodes regular expression $v^\star$, and constrains X to contain only positions removed from S. The reason why we need to compute the set V of the concrete strings is that our analysis is designed to be conservative. (As will be discussed later, `indexOf` and `contains` pose a similar requirement.) Consider the following Java code fragment:

```
String t = some_condition ? "a" : "b";
String u = "ab".replace(t,"z");
```

$$\begin{array}{ll}
[\![x := v]\!] & \rightarrow \mathsf{prog}_x(R) \equiv \text{“}v\text{”}(R) \\
[\![x := f(x_1,\cdots,x_n)]\!] & \rightarrow \mathsf{prog}_x(R) \equiv [\![f]\!]\,(R, \mathsf{prog}_{x_1},\cdots,\mathsf{prog}_{x_n}) \\
[\![x := \mathtt{phi}(x_1,\ldots,x_n)]\!] & \rightarrow \mathsf{prog}_x(R) \equiv \mathsf{prog}_{x_1}(R) \vee\cdots\vee \mathsf{prog}_{x_n}(R)
\end{array}$$

Fig. 8. Encoding Instructions

where `t`'s value is either “a” or “b”. However, replacing “a” and “b” in the string “ab” with “z” yields “zz”, while the actual result should be either “az” or “zb”.

In practice, a separate analysis technique (*e.g.*, constant propagation [Wegman and Zadeck 1991]) can be used to obtain the set of concrete values corresponding to the relevant strings, which are then reflected in the abstraction. We emphasize that other string-analysis techniques also require this information. For example, both Minamide's algorithm [Minamide 2005] and JSA [Christensen et al. 2003] approximate the `replace("a","b")` operation with an automaton, but not the function `replace`.

Having reviewed our abstraction of `replace`, we note that in some cases, defining the abstraction of a built-in function using a finite-state transducer, as described in [Minamide 2005], is easier than directly constructing an M2L formula. We use a simple approach to translate a string transducer into an M2L predicate: Since a transducer can be viewed as a finite-state automaton with output characters, we can represent it using the regular-expression notation, while denoting tuples of input characters and output characters as described in [Kay and Kaplan 1994]. For example, a transducer for `replace("a","b")` can be represented by $((\hat{\mathrm{a}}\mathrm{b})|(\hat{A}A))^{\star}$, where A matches any character, and the notation $\hat{c}o_1\ldots o_n$ means that c is an input character followed by the output characters $o_1,\cdots,o_n$. We can then encode this regular expression using the algorithm in Figure 6.

3.4. Encoding Instructions

Encoding an instruction amounts to producing a set of M2L predicate declarations. Similar to the abstraction of a built-in function, we denote the encoding of instruction I by $[\![I]\!]$.

We say that a left-hand variable is *cyclic* if it is defined depending on itself. Such a cyclic variable can only appear in a program with loops (and recursions). If there are no cyclic variable definitions, then the abstractions in Figure 8 apply. The abstractions of return instructions, calls to user-defined functions, goto instructions, and jump instructions are omitted, since this section assumes an intraprocedural and path-insensitive analysis for simplicity.

When a cyclic variable is affected only by string concatenations, we use the approach of [Christensen et al. 2003], which consists of the following two steps: We first construct a Context-Free Grammar (CFG) for the cyclic variable in the same way as Appendix D, where cyclic variables are considered nonterminal symbols, and then approximate the resulting CFG with a regular expression. Finally, we encode the regular expression in M2L using the algorithm shown in Figure 6, and declare predicate $\mathsf{prog}_{\mathrm{v}}(R) \equiv \langle\!\langle r\rangle\!\rangle\,(R)$, where `v` and r are the cyclic variable and the corresponding regular expression.

As an example, consider the following loop program and its corresponding SSA program containing cyclic variables `v2` and `v3`:

```
String v0 = "ab";                1:v0 = "ab"
String v1 =                        v1 = toUpperCase(v0)
    v0.toUpperCase();            2:v2 = phi(1:v0,3:v3)
String v2 = v0;                    i0 = length(v2)
while (v2.length()<10) {           b0 = op(<)(i0,10)
  v2 = v2.concat(v1);              b1 = op(neg)(b0)
}                                  jump b1, 4
 ...                             3:v3 = concat(v2,v1)
                                   goto 2
                                 4: ...
```

The possible set of strings assigned to `v2` can be represented by CFG $v_2 \rightarrow \mathtt{v0} \mid v_3, v_3 \rightarrow v_2\ \mathtt{v1}$, where `v0` and `v1` are considered terminal symbols (being non-cyclic). This CFG is overapproximated by the regular expression `v0 v1*`, which is encoded in M2L as $\langle\!\langle \texttt{v0 v1}^\star \rangle\!\rangle$. Here are the resulting predicate declarations:

$$\begin{aligned}
&\mathsf{prog}_{\texttt{v0}}(R) \equiv \text{“}\texttt{ab}\text{”}(R)\\
&\mathsf{prog}_{\texttt{v1}}(R) \equiv [\![\texttt{toUpperCase}]\!]\,(R, \mathsf{prog}_{\texttt{v0}})\\
&\mathsf{prog}_{\texttt{v2}}(R) \equiv \langle\!\langle \texttt{v0 v1}^\star \rangle\!\rangle\,(R)\\
&\mathsf{prog}_{\texttt{v3}}(R) \equiv \exists V_1, V_2 . \mathsf{prog}_{\texttt{v2}}(V_2) \wedge \mathsf{prog}_{\texttt{v1}}(V_1) \wedge \mathsf{concat}(R, V_2, V_1)
\end{aligned}$$

When cyclic variable x is affected by other string operations, we use the naïve abstraction $\mathsf{prog}_x(R) \equiv \mathsf{true}$, which holds for any string value.

Alternatively, as we describe in Appendix C, we could automatically find a loop invariant inv_x, for x, using the character-set abstraction [Christensen et al. 2003], where a position set variable is used as a set of characters by ignoring the order of characters in a string. This predicate can be used instead of prog_x. Note that the *strongest* loop invariant on character sets, in the sense of the *smallest* character set that satisfies cyclic string constraints, is necessary, since our purpose is to check the existence of an unsafe string. Otherwise, the naïve abstraction is allowed to be a loop invariant. In addition, finding the smallest character set requires the subset relation between character sets, whereby the equality of characters is also required. Our experience suggests, however, that computing the smallest character set, which involves checking the equality of characters, is an expensive process whose advantage over the naïve approach is negligible.

3.5. Soundness of the Encoding Method

Here, we describe soundness of our encoding method. Note that only loop-free programs are addressed, since we rely on the grammar-based abstraction by [Christensen et al. 2003; Minamide 2005] and the trivial widening operation that always yields the naïve abstraction $\mathsf{prog}_x(S) = \mathsf{true}$ to handle loops.

We first introduce the notation $L_{w,\mathcal{I}}(\psi)$ to denote the set of strings represented by R that satisfy $\psi(R)$ given finite string w and assignment $\mathcal{I}$. [3]

Definition 3.1 (*Generated Language*).

$$L_{w,\mathcal{I}}(\psi) \ \equiv\ \{s \mid w, \mathcal{I} \models \forall R\ .\ \text{“}s\text{”}(R) \Rightarrow \psi(R)\}$$

The set $[\![P]\!]$ of predicate declarations, which is obtained by encoding the program P, is sound if for every program variable x, there exists a finite string w and an assignment $\mathcal{I}$, such that $v \in L_{w,\mathcal{I}}(\mathsf{prog}_x)$, where v is a string value assigned to variable x and $[\![P]\!] = \{\mathsf{prog}_{x_1}, \cdots, \mathsf{prog}_{x_n}\}$. This soundness property is formulated as follows:

THEOREM 3.2 (SOUNDNESS OF THE ENCODING METHOD).

$$\exists w, \mathcal{I}\ .\ \forall x \in \mathit{dom}(\sigma)\ .\ \sigma(x) \in L_{w,\mathcal{I}}(\mathsf{prog}_x)$$

where σ represents an arbitrary program state (interpreted as a mapping from program variables to values) that is obtained in an actual program execution.

The above soundness criterion holds as long as the abstraction $[\![f]\!]$ is sound for every string operations f, where the condition of the soundness of $[\![f]\!]$ is as follows:

Definition 3.3 (*Sound Abstraction of Function f*).

$$\begin{aligned}
&\forall r, p_1, \cdots, p_n\ .\ r = f(p_1, \cdots, p_n)\\
&\Rightarrow \forall w, \mathcal{I}, \psi_1, \cdots, \psi_n\ .\ p_1 \in L_{w,\mathcal{I}}(\psi_1) \wedge \cdots \wedge p_n \in L_{w,\mathcal{I}}(\psi_n)\\
&\qquad\qquad \Rightarrow \exists w'\ .\ r \in L_{ww',\mathcal{I}}(\lambda R. [\![f]\!]\,(R, \psi_1, \cdots, \psi_n))
\end{aligned}$$

[3]This definition can be viewed as a concretization function in abstract interpretation, where there is no best abstraction as in the case of regular languages [Cousot and Cousot 1995], which is not a complete partial order.

$$\begin{array}{ll} [\![x := n]\!] & \rightarrow \mathsf{prog}_x(p,S) \equiv \mathsf{pos}_n(p,S) \\ [\![x := f(x_1,\cdots,x_n)]\!] & \rightarrow \mathsf{prog}_x(p,S) \equiv [\![f]\!]\,(p,S,\mathsf{prog}_{x_1},\cdots,\mathsf{prog}_{x_n}) \\ [\![x := \mathtt{phi}(x_1,\ldots,x_n)]\!] & \rightarrow \mathsf{prog}_x(p,S) \equiv \mathsf{prog}_{x_1}(p,S) \vee \cdots \vee \mathsf{prog}_{x_n}(p,S) \end{array}$$

Fig. 9. Additional abstraction of instructions for indices

where ww' is the concatenation of finite strings w and w', and $L_{w,\mathcal{I}}(\lambda R.\psi(R))$ is short for $\{s \mid w,\mathcal{I} \models \forall R.\text{“}s\text{”}(R) \Rightarrow \psi(R)\}$.

Intuitively, the soundness of $[\![f]\!]$ means that, for any actual arguments $p_1,\ldots,p_n$ and any value r returned by $f(p_1,\cdots,p_n)$, if string sets represented by $\psi_1,\ldots,\psi_n$ contain $p_1,\ldots,p_n$, respectively, and R represents the return value r, $[\![f]\!]\,(R,\psi_1,\cdots,\psi_n)$ is satisfiable.

We prove Theorem 3.2 in Appendix B. The proof is by induction on a transition system that defines the semantics of the SSA program described in Appendix A.

4. INDEX- AND PATH-SENSITIVITY

This section describes how to augment the core algorithm with the index sensitivity and the path sensitivity.

4.1. Handling String Indices

An index is encoded as a position and position-set pair. For example, if string “ace” is encoded as position set $\{0,2,4\}$ in M2L, then index 1 into it is encoded as the pair $(2,\{0,2,4\})$. More generally, we introduce the following M2L predicates to represent indices: $\mathsf{pos}_0(p,S) \equiv (p = \mathsf{min}(S)),\cdots,\mathsf{pos}_n(p,S) \equiv \mathsf{pos}_{n-1}(p,S \setminus \mathsf{min}(S))$, where $\mathsf{pos}_n(p,S)$ means that position p in position-set S represents index n into a string represented by S.

When encoding instructions, PISA accounts for indices following the rules in Figure 9, where a predicate prog_x takes a position and position-set pair as its arguments. Those rules apply when the left-hand-side variable in an instruction assumes a value representing an index into a string. Note that any numerical expression of the form $n + N$ can be encoded, where n is a variable and N is a constant. However, since M2L cannot directly encode numerical expressions of the form $n + m$, where m is also a variable, for such expressions PISA over-approximate it by $\mathsf{pos}_{\mathsf{any}}(p,S) = p \in S$, which represents an arbitrary index into a string represented by S. This same encoding is also used for a cyclic variable.

With index sensitivity at its disposal, PISA can model string operations such as `indexOf` and `substring`, where `indexOf(s1,s2)` returns the first index in `s1` at which `s2` occurs, whereas `substring(s,n,m)` extract from `s` the substring ranging between indices `n` and `m`. These methods are abstracted as shown in Figure 7. In the definition of $[\![\mathtt{substring}]\!]$, we choose the range $[n,m)$ of the substring of a string represented by S using $\mathcal{P}_n(n,S) \wedge \mathcal{P}_m(m,S)$, and constrain the substring relationship between R and S using $\mathsf{strr}(R,S,n,m)$. In the definition of $[\![\mathtt{indexOf}]\!]$, the first and second parameters represent a position and a string containing it, respectively, while V is the set of concrete values possibly assigned to `s2` in `indexOf(s1,s2)`. The definition uses $\mathsf{indexOf}(p,P,Q)$ that consists of the following three constrains: (1) p is a minimal index of Q when Q is a non-empty substring of P, where emptyness of the string is denoted by $Q = \emptyset$, (2) p is a minimal index of P when empty string is given, and (3) if any string represented by P does not contain any substring represented by Q, p is less than the minimal index of P. By requiring $p \leq p'$, we choose the minimal index among the candidates that satisfy $\mathsf{indexOf}(p,P,Q)$. In addition, due to the restriction about the minimal index, we need a set V of concrete string values possibly assigned to `s2` of `indexOf(s1,s2)`. It should be obtained by another analysis as in the case of $[\![\mathtt{replace}]\!]$ [4]. Otherwise, $[\![\mathtt{indexOf}]\!]$ involves the same problem as $[\![\mathtt{replace}]\!]$.

[4]If the minimal index is not required, the abstraction becomes simpler so as not to require the concrete strings, but makes the analysis too conservative to check the existence of unsafe characters.

As an example, consider the following SSA program fragment:

```
v0 = 0;  v1 = "a<b";  v2 = "<";
v3 = indexOf(v1,v2);  v4 = substring(v1,v0,v3);
```

We obtain the following set of predicates after expanding the definitions of $[\![$`indexOf`$]\!]$ and $[\![$`substring`$]\!]$:

$$
\begin{array}{lcl}
\mathsf{prog}_{\mathtt{v0}}(n,S) & \equiv & \mathsf{pos}_0(n,S) \\
\mathsf{prog}_{\mathtt{v1}}(R) & \equiv & \text{“}\mathtt{a<b}\text{”}(R) \\
\mathsf{prog}_{\mathtt{v2}}(R) & \equiv & \text{“}\mathtt{<}\text{”}(R) \\
\mathsf{prog}_{\mathtt{v3}}(p,P) & \equiv & \mathsf{prog}_{\mathtt{v1}}(P) \wedge (\exists P_2.\text{“}\mathtt{<}\text{”}(P_2) \wedge \mathsf{indexOf}(p,P,P_2)) \\
 & & \wedge(\mathsf{min}(P) \le p \Rightarrow (\forall P_2,p'.\text{“}\mathtt{<}\text{”}(P_2) \wedge \mathsf{indexOf}(p',P,P_2) \\
 & & \wedge \mathsf{min}(P) \le p' \Rightarrow p \le p')) \\
\mathsf{prog}_{\mathtt{v4}}(R) & \equiv & \exists V_1, v_0, v_3 \,.\, \mathsf{prog}_{\mathtt{v1}}(V_1) \\
 & & \wedge \mathsf{prog}_{\mathtt{v0}}(v_0,V_1) \wedge \mathsf{prog}_{\mathtt{v3}}(v_3,V_1) \wedge \mathsf{substrr}(R,V_1,v_0,v_3)
\end{array}
$$

4.2. Handling Branch Conditions

PISA employs a simple form of path sensitivity, which provides the ability to record the effects of branch conditions on a specific variable by encoding them as M2L predicates. For example, branch-condition `v.equals("a")` constrains variable `v`. Thus, if the test succeeds, we encode the relevant constraint as M2L-predicate $\phi_v(R) = \text{“a”}(R)$. When encoding the true branch of a condition as a set of predicate declarations, we use predicate $\mathsf{prog}'_v(R) = \mathsf{prog}_v(R) \wedge \phi_v(R)$, rather than $\mathsf{prog}_v(R)$, to represent the values possibly assigned to program-variable v.

Figures 10 and 11 present the encoding for path-sensitive analysis, where the notation $[\![I]\!]^b$ represents encoding instruction I in basic block b. Boolean operators $\wedge$ and $\vee$ are used for notational brevity: For predicates ψ_1 and ψ_2, $\psi_1 \wedge \psi_2$ [$\psi_1 \vee \psi_2$] represents a predicate ψ such that $\psi(R) = \psi_1(R) \wedge \psi_2(R)$ [$\psi(R) = \psi_1(R) \vee \psi_2(R)$]. In addition, we lift the lambda notation $\lambda R.\psi'$ to represent a predicate ψ such that $\psi(R) = \psi'$, without explicitly declaring predicate ψ.

Path-condition [Snelting 1996; Hammer et al. 2008] $PC(b_0, b')$ represents a necessary condition for flow from basic block b_0 to b'. We use the notation $PC'(b_0, b, b')$ to distinguish between ϕ-induced assignments, and thus represent a necessary condition for flow from b_0 to b' through b, which is an immediate predecessor of b'. Conditions are formed using Boolean program variables and logical operators.

Figure 11 describes our encoding of the effects of path conditions. Given variable v and basic blocks b and b', such that b is an immediate predecessor of b', we define $C(v, b, b')$. M2L predicate $C(v, b, b')$ represents a necessary condition for v to cause the transition from b to b'. The definition uses function C', which syntactically and recursively transforms the path condition into an M2L predicate. In the figure, $[\![f, m]\!]\,(t, R, \mathsf{prog}_{v_1}, \ldots, \mathsf{prog}_{v_n})$ represents a predicate restricting R *via* a necessary condition on the m-th parameter of Boolean method f, when f returns t. Such an abstraction should be predefined for each Boolean method, as in the case of built-in functions. In the absence of an abstraction, we default to true. $\mathsf{def}(v')$ represents an instruction, $v' = f(v_1, \cdots, v_n)$, which defines program variable v'.

Figure 10 uses $C(v, b, b')$ to encode ϕ instructions. The variables used by the ϕ statement are each constrained by taking the relevant immediate predecessor of b' into account. For the other instructions, we do not need to consider the immediate predecessor. Thus, we simply use the notation $C(x, b')$, which is equivalent to $\bigvee_{b \in \mathsf{pred}(b')} C(x, b, b')$, where $\mathsf{pred}(b')$ represents a set of immediate predecessors of basic block b'.

With path sensitivity, we can abstract `contains`, as described in Figure 7. $[\![$`contains,1`$]\!]\,(c, R, \mathcal{P}_1, \mathcal{P}_2)$ is the abstraction of `v1.contains(v2)` and constrains the variable `v1`, where $\mathcal{P}_1$ and $\mathcal{P}_2$ are corresponding to the parameters `v1` and `v2`, respectively. In the definition, S is the set of concrete strings assigned to v_2. Note that $\neg(\exists P \,.\, \mathsf{prog}_{v_2}(P) \wedge \mathsf{substr}(P, R))$ cannot be used when $c = \mathsf{false}$ since the analysis is conservative. As an example, consider the

$$\begin{array}{ll}
[\![x := v]\!]^b & \rightarrow \mathsf{prog}_x(R) \equiv \text{“}v\text{”}(R) \\
[\![x := f(x_1,\cdots,x_n)]\!]^b & \rightarrow \mathsf{prog}_x(R) \equiv [\![f]\!]\,(R, \mathsf{prog}_{x_1} \wedge C(x_1,b),\cdots,\mathsf{prog}_{x_n} \wedge C(x_n,b)) \\
[\![x := \mathtt{phi}(b_1 : x_1,\ldots,b_n : x_n)]\!]^b & \rightarrow \mathsf{prog}_x(R) \equiv \bigvee_{i\in\{1,\cdots,n\}} \mathsf{prog}_{x_i}(R) \wedge C(x_i,b_i,b)
\end{array}$$

Fig. 10. Abstraction for path-sensitive string analysis

$$C(v,b,b') \equiv C'(\mathsf{true}, v, PC'(b_0,b,b'))$$

$$C'(t,v,c) \equiv \begin{cases}
\lambda R\,.\ [\![f,m]\!]\,(t,R,\mathsf{prog}_{v_1},\ldots,\mathsf{prog}_{v_n}) & \text{when } c = f(v_1,\ldots,v_n), \text{ and } v = v_m \\
C'(t,v,\mathsf{def}(v')) & \text{when } c = v', \text{ where } v' \text{ is a program variable} \\
C'(\mathsf{true},v,c') & \text{when } c = \neg c' \text{ and } t = \mathsf{false} \\
C'(\mathsf{false},v,c') & \text{when } c = \neg c' \text{ and } t = \mathsf{true} \\
C'(\mathsf{true},v,c_1) \vee C'(\mathsf{true},v,c_2) & \text{when } c = c_1 \vee c_2 \text{ and } t = \mathsf{true} \\
C'(\mathsf{true},v,c_1) \wedge C'(\mathsf{true},v,c_2) & \text{when } c = c_1 \wedge c_2 \text{ and } t = \mathsf{true} \\
C'(\mathsf{false},v,c_1) \wedge C'(\mathsf{false},v,c_2) & \text{when } c = c_1 \vee c_2 \text{ and } t = \mathsf{false} \\
C'(\mathsf{false},v,c_1) \vee C'(\mathsf{false},v,c_2) & \text{when } c = c_1 \wedge c_2 \text{ and } t = \mathsf{false} \\
\lambda R\,.\ \mathsf{true} & \text{otherwise}
\end{cases}$$

Fig. 11. Constraint on program-variable v when the execution transitions from basic block b to basic block b'

following Java method, along with its corresponding SSA representation, where `op(or)` and `op(neg)` represent logical disjunction and negation, respectively:

```
String clean(String s) {              1: v1 = "<"; v2 = ">";
 if (s.contains("<") ||                  b1 = contains(s,v1);
     s.contains(">")) {                  b2 = contains(s,v2);
   s = "x";                              z0 = op(or)(b1,b2);
 }                                       z1 = op(neg)(z0);
 return s;                               jump z1,3;
}                                     2: v4 = "x";
                                      3: v5 = phi(1:s,2:v4);
                                         return v5;
```

For this method, we obtain the following predicates from the SSA program using the abstraction described in Figure 10:

$$\begin{array}{lcl}
\mathsf{prog}_{\mathtt{v1}}(R) & \equiv & \text{“<”}(R) \\
\mathsf{prog}_{\mathtt{v2}}(R) & \equiv & \text{“>”}(R) \\
\mathsf{prog}_{\mathtt{v4}}(R) & \equiv & \text{“x”}(R) \\
\mathsf{prog}_{\mathtt{v5}}(R) & \equiv & (\mathsf{prog}_{\mathtt{s}}(R) \wedge C(\mathtt{v4},2,3)(R)) \vee (\mathsf{prog}_{\mathtt{v4}}(R) \wedge C(\mathtt{s},1,3)(R))
\end{array}$$

Based on path conditions $PC'(1,1,2) = \mathtt{b1} \vee \mathtt{b2}$, $PC'(1,1,3) = \neg(\mathtt{b1} \vee \mathtt{b2})$, and $PC'(1,2,3) = \mathtt{b1} \vee \mathtt{b2}$, $C(\mathtt{v4},2,3)$ and $C(\mathtt{s},1,3)$ are calculated as follows:

$$\begin{array}{lcl}
C(\mathtt{v4},2,3) & = & C'(\mathsf{true},\mathtt{v4},PC'(1,2,3)) = \lambda R\,.\ \mathsf{true} \\
C(\mathtt{s},1,3) & = & C'(\mathsf{true},\mathtt{s},PC'(1,1,3)) \\
 & = & \lambda R\,.\ [\![\mathtt{contains},1]\!]\,(\mathsf{false},R,\mathsf{prog}_{\mathtt{s}},\mathsf{prog}_{\mathtt{v1}}) \\
 & & \quad \wedge \lambda R\,.\ [\![\mathtt{contains},1]\!]\,(\mathsf{false},R,\mathsf{prog}_{\mathtt{s}},\mathsf{prog}_{\mathtt{v2}}) \\
 & = & \lambda R\,.\ \neg(\exists P\,.\ \text{“<”}(P) \wedge \mathsf{substr}(P,R)) \\
 & & \quad \wedge \lambda R\,.\ \neg(\exists P\,.\ \text{“>”}(P) \wedge \mathsf{substr}(P,R)).
\end{array}$$

5. PRACTICAL EXTENSIONS

This section describes extensions of the core algorithms needed to support real-world Java programs. These extensions include handling of user-defined methods, and operations involving regular expressions.

5.1. Interprocedural Analysis

```
void   f1(String v1) { String s1 = f3(v1); }
void   f2(String v2) { String s2 = f3(v2); }
String f3(String v3) { return v3; }
```

Fig. 12. Sample Java Program for Interprocedural Analysis

$$\begin{array}{lcl} m & \in & \mathcal{M} \\ N & ::= & \{(m, g)\} \\ E & ::= & \{((m, g), (m, g))\} \\ D & ::= & \{((x, m), \{(x, m)\})\} \\ cg & ::= & \{(N, E, D)\} \end{array}$$

Fig. 13. Representation of callgraph

$$\begin{array}{lcl} \llbracket E \rrbracket & \rightarrow & \bigcup_{((m_1,g_1),(m_2,g_2))\in N'} \llbracket g_1 \rrbracket^{m_1} \cup \llbracket g_2 \rrbracket^{m_2} \\ \llbracket D \rrbracket & \rightarrow & \bigcup_{((x,m),\{(x_1,m_1),\cdots,(x_n,m_n)\})\in D} \\ & & \quad \{\mathsf{prog}_{x,m}(R) \equiv \mathsf{prog}_{x_1,m_1}(R) \\ & & \qquad\qquad \vee \cdots \vee \mathsf{prog}_{x_n,m_n}(R)\} \\ \llbracket (N, E, D) \rrbracket & \rightarrow & \llbracket E \rrbracket \cup \llbracket D \rrbracket \end{array}$$

Fig. 14. Abstraction for interprocedural analysis

Our interprocedural version of the string analysis relies on a callgraph, where each node of the callgraph contains a set of instructions in the SSA form and these instructions are translated into a set of M2L predicate declarations using our encoding method. The relationships among callgraph nodes are used to obtain possible assignment relationships between caller's program variables and callee's program variables (parameters and return variables). The assignment relationships are encoded in M2L as if those are assignment instructions.

Let us consider a context-insensitive callgraph for three Java's methods shown in Figure 12, where the callgraph has three nodes n_1, n_2, and n_3 for the methods `f1`, `f2`, and `f3`, respectively. We obtain the four assignment relationships `v3=v1`, `v3=v2`, `s1=v3`, `s2=v3`. These assignment relationships are then encoded in M2L using the encoding method described in Section 3.4, where, due to two possible assignment to `v3`, the first two relationships can be encoded in the same way to encode the ϕ-instruction `v3=phi(v1,v2)`. Note that this approach simply ignores the call stack. Therefore, context-sensitivity of the callgraph affects the precision of the string analysis.

To formulate the method of encoding the callgraph, we first introduce the syntax of callgraphs as described in Figure 13. $\mathcal{M}$ is a set of names of callgraph nodes. cg is a callgraph, where N is a set of callgraph nodes which consists of pairs of the node names and control-flow graphs, E is a set of edges of a callgraph, and D is a set of direct data dependencies of parameters and return values due to function calls. $((x, m), \{(x_1, m_1), \cdots, (x_n, m_n)\}) \in D$ represents that the values of variables $x_1, \cdots, x_n$ in respective callgraph nodes $m_1, \cdots, m_n$ flow to variable x in node m.

With the above representation of the callgraph, we use the encoding method described in Figure 14, where the node name is propagated to the abstraction of instructions to annotate predicates generated from a callgraph node.

5.2. Operations Involving Regular Expressions

Regular-expression matching is often used by real applications, both in string operations and in branch conditions. Typically, the regular expression is given as a constant string value. Thus, when we encode a string literal that is used as a regular expression, we use the method described in Figure 6.

```
void detectSanitizers(
       Set<Method> M, Set<Pattern> P,  // input
       Set<Pair<CGNode,Pattern>> R) {  // output
  CallGraph cg = callgraphOf(M);
  for (Method m : M) {
    Set<CGNode> N = nodes(cg, m);
    for (CGNode n : N) {
      Set<Instruction> I = instructionsOf(n, cg);
      Set<Variable> V = returnVariablesOf(n);
      for (Pattern p : P) {
        boolean r = doStringAnalysis(I, V, p);
        if (r) R.add(new Pair(n, p)); } } } }
```

Fig. 15. Outline of the Sanitizer-detection Algorithm

For example, the following Java program, along with its corresponding SSA representation, replaces all substrings matched by the regular expression "`(ab)*`" with the letter "z".

```
void foo(String s) {
  String v3 =
    s.replaceAll("(ab)*","z")
}
```

```
v1 = "(ab)+";
v2 = "z";
v3 = replaceAll(s,v1,v2);
```

The SSA representation of the program, shown above, is encoded as the following predicate declarations, where $\langle\!\langle(\mathrm{ab})^+\rangle\!\rangle$ is an M2L predicate corresponding to regular expression $(\mathrm{ab})^+$:

$$\begin{aligned}
\mathsf{prog}_{\mathtt{v1}}(R) &\equiv \langle\!\langle(\mathrm{ab})^+\rangle\!\rangle\,(R)\\
\mathsf{prog}_{\mathtt{v2}}(R) &\equiv \text{“z”}(R)\\
\mathsf{prog}_{\mathtt{v3}}(R) &\equiv [\![\mathtt{replaceAll}]\!]\,(R, \mathsf{prog}_{\mathtt{s}}, \mathsf{prog}_{\mathtt{v1}}, \mathsf{prog}_{\mathtt{v2}})
\end{aligned}$$

6. IMPLEMENTATION AND EVALUATION

We applied PISA to a production-level taint-analysis engine for the purpose of automatic detection of *user-defined sanitizers* (sanitizers defined in application code). In this section, we describe our sanitizer-detection algorithm, then present the implementation of that algorithm, followed by an experimental evaluation of our approach.

6.1. Sanitizer Detection

Our algorithm for detecting sanitizers consists of two steps. The first step is to find sanitizer candidates based on a syntactic check: An input pattern ranging over method signatures is used to focus the analysis on methods that are likely to act as sanitizers (*e.g.*, methods accepting a single `String` argument and returning a `String` object). Figure 15 shows the algorithm of the second step. This phase consumes the set `M` of candidate methods and the set `P` of unsafe string patterns, and outputs the set `R` of the pairs of the form `(n,p)` such that the method corresponding to callgraph node `n` is a sanitizer for unsafe pattern `p`, where each pattern is used to categorize detected sanitizers. Note that sanitizer categorization is essential for taint analysis, since a method is typically a sanitizer only for certain types of attack. The procedure comprises the following steps:

(1) `callgraphOf(M)` builds the callgraph `cg` rooted at the set `M` of sanitizer candidates. Any callgraph-construction algorithm can be chosen to build `cg` (*e.g.*, context insensitive or context sensitive, with various levels of context sensitivity [Grove et al. 1997; Grove and Chambers 2001]).
(2) For each sanitizer candidate, we obtain the set of callgraph nodes representing it.
(3) For each node `n`, a set `I` of instructions is then computed using function `instructionsOf (n)`, which returns the set of instructions in n and in all the nodes transitively called by `n`. In addition, for each caller/callee pair, assignment relationships between actual parameters in the caller node and formal parameters in the callee node are included in `I`. This allows us to perform the interprocedural analysis described in Section 5.1.

(4) For each unsafe pattern `p`, return variables of node `n` are extracted by `returnVariablesOf(n)`, and then verified by the string analysis, `doStringAnalysis(I,V,p)`, where the string analysis returns `true` iff the set of potential string values taken by the return variables V of node n never contains the unsafe pattern p. If the string analysis returns `true`, then `n` is reported as a sanitizer for the unsafe string pattern `p`.

6.2. Implementation

We implemented the algorithm described in Section 6.1 as a Java program using the T. J. Watson Libraries for Analysis (WALA) framework [WALA], which calls MONA [Henriksen et al. 1995] as a command-line program to check the satisfiability of M2L formulae. In addition, the implementation embodies the following *sound* optimizations:

— String constants whose length exceeds fixed-size n are over-approximated by a disjunctive regular expression that ranges over a partitioning of the original string, where each partition (except, maybe, the suffix) is of size n. Our experiments used $n = 5$. For example, "`longstringvalue`" is over-approximated by regular expression "`(longs|tring|value)+`".
— The verification of each method has a 30-second time limit, since verifying all of the methods within a reasonable time is more important than verifying a particular method for a long time. If the limit is reached without having concluded the analysis, we conservatively over-approximate the behavior of the method by concluding that it is not a sanitizer.
— We did not define the last-position term , \$, of M2L(Str), when simulating M2L(Str) in MONA, since it was not required by our encoding method.
— The path-condition analysis is run only if the SSA form of the target method consists of less than 50 basic blocks. Otherwise, the analysis is still sound because it conservatively becomes path-insensitive.

6.3. Evaluation

We ran our experiments on top of the Sun Java Runtime Environment (JRE) V1.6.0_06 with 1 GB of maximum heap size using a Lenovo ThinkPad T61p with a Core2 Duo T7800 2.6GHz CPU and 3 GB of RAM. Statistics on the benchmark applications we used are shown in Table I. The Candidates column reflects the number of methods in application code (*i.e.*, ignoring imported libraries and JUnit test classes) that accept a single parameter of type `String` and return a `String` value. This is the criterion we used for identifying sanitizer candidates. (Note that in some cases, methods that were not intended to be used as sanitizers may be accepted by both the criterion we just described and the ensuing analysis.)

Our study focused on the following four types of attack: XSS, HRS, Log Forging (LOG) and Path Traversal (PATH) [Web]. The corresponding regular-expression patterns used in the specification are "`.*[<>].*`" (HTML tags), "`.*[\r\n].*`" (strings of multiple lines), "`.*[\r\n\x08].*`" (strings of multiple lines and backspace) and "`.*\.\./.*`" (strings representing relative paths), respectively.

With the above environment, we conducted two sets of experiments. In the first set, we investigated how many user-defined sanitizers in the application code of 8 open-source benchmarks PISA was able to detect, and how effectively the path sensitivity and the index sensitivity were able to improve the accuracy of the sanitizer detection by comparing PISA against the CFG-based string analysis [Geay et al. 2009], which is described in Appendix D. Based on the result of this first set of evaulations, we proved that PISA detected more sanitizers than the CFG-based string analysis, and both the path sensitivity and the index sensitivity contributed to the improvement. In the second set, we examined the overall impact of PISA on a production-level taint-analysis algorithm, and proved that the detected sanitizers improved the result of the taint analysis in that we successfully reduced false reports.

6.3.1. Sanitizer Detection. Our experiment on sanitizer detection comprised four configurations: The first two, PISA/CI and PISA/CS, both embody the PISA algorithm, the difference between them being that the former is based on a context-insensitive (0-CFA) callgraph, whereas the latter relies on

Table I. Statistics on benchmark applications

App. name	Version	Classes	LOC	Candidates
SBM	1.08	143	5,541	15
Blojsom	3.1	255	13,967	51
PersonalBlog	1.2.6	69	5,317	7
Roller	0.9.9	283	41,589	56
SnipSnap	1.0-BETA-1	614	46,962	52
Webgoat	5.1	193	33,906	43
JSPWiki	2.6	503	81,301	91
MVNForum	1.0.2	820	142,954	56
Total				371

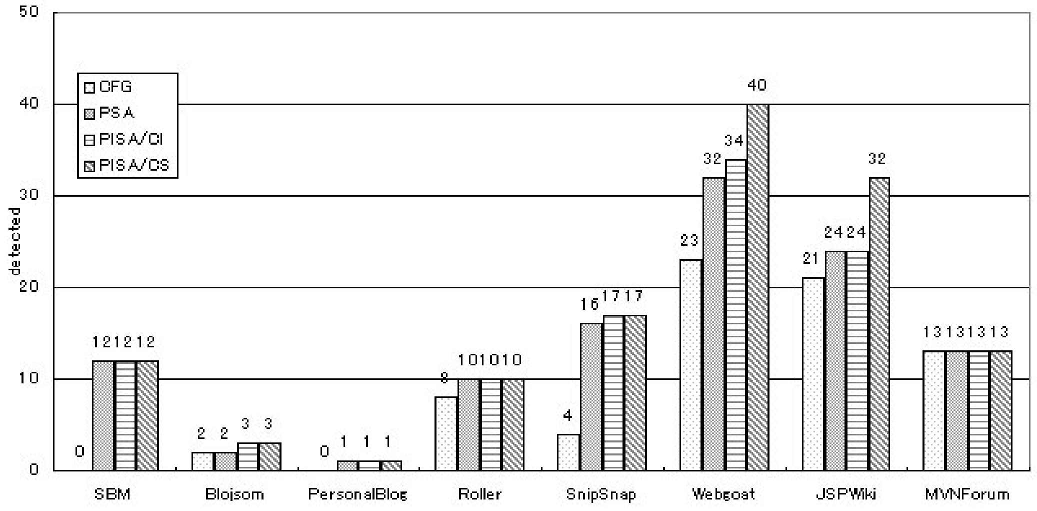

Fig. 16. Numbers of detected sanitizers

a context-sensitive (1-CFA) callgraph [Grove et al. 1997; Grove and Chambers 2001]. Recall from Section 5.1 that PISA depends on its underlying callgraph in various ways, and thus the overall accuracy of PISA derives, in part, from the precision of its supporting callgraph. The two remaining candidates are PSA/CI, which is a variant of PISA/CI employing only path sensitivity (and thus lacking the abstract models for `indexOf` and `lastIndexOf`), and the CFG-based string analysis of [Geay et al. 2009], which is based on [Minamide 2005]; this analysis is neither path- nor index-sensitive, but can handle cyclic variables by computing invariants. This computation is done by iterating the grammar transduction and by using the character-set approximation [Christensen et al. 2003; Minamide 2005] as a widening operation.

Figure 16 summarizes how many true sanitizers were detected by PISA on the 8 benchmark applications in comparison with the CFG-based string analyzer. More details are reported in Table II, which shows the number of true positives (TP) and false negatives (FN) for each of the four types of attack, as well as the accuracy scores, which are calculated as $100 \times \mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$. If a method automatically detected by PISA is a true sanitizer, it is counted as a true positive (TP). Otherwise, it is counted as a false positive (FP). If PISA fails to detect a true sanitizer, that result is counted as a false negative (FN). Note that the accuracy scores and the false negatives are counted only for the relatively small 6 benchmark applications, since detecting false negatives requires manually reviewing all of the candidates to find the true sanitizers that are not detected by PISA—a very time-consuming and error-prone operation. In addition, our sanitizer detection reported no false positives

Table II. Accuracy results of the sanitizer-detection experiment

		XSS			HRS			LOG			PATH			Total		
		TP	FN	Score	TP	FN	Score	TP	FN	Score	TP	FN	Score	TP	FN	Score
SBM	CFG	0	3	0	0	3	0	0	3	0	0	3	0	0	12	0
	PSA/CI	3	0	100	3	0	100	3	0	100	3	0	100	12	0	100
	PISA/CI	3	0	100	3	0	100	3	0	100	3	0	100	12	0	100
	PISA/CS	3	0	100	3	0	100	3	0	100	3	0	100	12	0	100
Blojsom	CFG	0	8	0	1	2	33	0	2	0	1	4	20	2	16	11
	PSA/CI	0	8	0	1	2	33	0	2	0	1	4	20	2	16	11
	PISA/CI	0	8	0	1	2	33	0	2	0	2	3	40	3	15	17
	PISA/CS	0	8	0	1	2	33	0	2	0	2	3	40	3	15	17
PersonalBlog	CFG	0	4	0	0	3	0	0	3	0	0	3	0	0	13	0
	PSA/CI	1	3	25	0	3	0	0	3	0	0	3	0	1	12	8
	PISA/CI	1	3	25	0	3	0	0	3	0	0	3	0	1	12	8
	PISA/CS	1	3	25	0	3	0	0	3	0	0	3	0	1	12	8
Roller	CFG	2	10	17	2	3	40	2	3	40	2	3	40	8	19	30
	PSA/CI	4	8	33	2	3	40	2	3	40	2	3	40	10	17	37
	PISA/CI	4	8	33	2	3	40	2	3	40	2	3	40	10	17	37
	PISA/CS	4	8	33	2	3	40	2	3	40	2	3	40	10	17	37
SnipSnap	CFG	1	3	25	1	3	25	1	3	25	1	7	13	4	16	20
	PSA/CI	4	0	100	4	0	100	4	0	100	4	4	50	16	4	80
	PISA/CI	4	0	100	4	0	100	4	0	100	5	3	63	17	3	85
	PISA/CS	4	0	100	4	0	100	4	0	100	5	3	63	17	3	85
Webgoat	CFG	5	16	24	7	9	44	5	8	38	6	9	40	23	42	35
	PSA/CI	14	7	67	7	9	44	5	8	38	6	9	40	32	33	49
	PISA/CI	14	7	67	7	9	44	5	8	38	8	7	53	34	31	52
	PISA/CS	14	7	67	9	7	56	7	6	54	10	5	67	40	25	62
JSPWiki	CFG	5			5			5			6			21		
	PSA/CI	6			6			6			6			24		
	PISA/CI	6			6			6			6			24		
	PISA/CS	8			8			8			8			32		
MVNForum	CFG	3			4			3			3			13		
	PSA/CI	3			4			3			3			13		
	PISA/CI	3			4			3			3			13		
	PISA/CS	3			4			3			3			13		
Total	CFG	16			20			16			19			71		
	PSA/CI	35			27			23			25			110		
	PISA/CI	35			27			23			29			114		
	PISA/CS	37			31			27			33			128		

thanks to the conservativeness of our string analysis, where false positives in the sanitizer detection are caused only when the string analysis is not conservative and fails to infer unsafe strings that arise at runtime.

Table III shows running time and the statistics reported by MONA. The Abort column shows the total time spent analyzing candidates that were aborted due to either the time limit we set in our analysis or a size limit in MONA. The number of aborted candidates is shown in parentheses. For PISA, the Enc and Ver columns show the time spent on translating the callgraph into a MONA program and the running time of MONA, respectively. For the CFG-based analysis, the Enc column shows the time spent on translating the callgraph into a set of production rules, and the Ver column shows the total time spent on inference and containment checks. The MONA Statistics column shows summaries of the statistics reported by MONA. The Pred and DAG columns show the average/maximum numbers of generated predicates and DAGs (graph representations of formulae), respectively. The State and BDD columns show the average and maximum numbers of elements in the largest sets of states and Binary Decision Diagram (BDD) nodes of minimized automata, respectively. Note that the number of the predicates is almost equivalent to the number of instructions analyzed by PISA.

Table III. Running Times and MONA Statistics

		Time(sec)				MONA Statistics							
						Average				Maximum			
		Abort	Enc	Ver	Total	Pred	DAG	State	BDD	Pred	DAG	State	BDD
SBM	CFG	0.0(0)	0.1	0.3	0.4								
	PSA/CI	0.0(0)	2.0	6.9	8.9	32	278	187	2,316	134	748	923	11,352
	PISA/CI	0.0(0)	2.0	6.7	8.7	32	278	187	2,316	134	748	923	11,352
	PISA/CS	0.0(0)	2.0	6.8	8.8	32	278	187	2,316	134	748	923	11,352
Blojsom	CFG	0.0(0)	1.8	7.9	9.7								
	PSA/CI	30.3(1)	4.1	17.9	52.3	18	197	49	481	54	615	923	10,679
	PISA/CI	30.2(1)	4.2	18.3	52.7	18	204	49	473	71	841	922	10,647
	PISA/CS	120.3(4)	89.1	19.6	229.0	35	255	55	542	370	1,119	922	10,647
Personal-Blog	CFG	0.0(0)	0.3	0.2	0.5								
	PSA/CI	0.0(0)	0.9	1.9	2.8	23	215	32	235	77	519	129	1,182
	PISA/CI	0.0(0)	0.9	1.9	2.8	23	215	32	235	77	519	129	1,182
	PISA/CS	0.0(0)	0.7	2.0	2.7	23	215	32	235	77	519	129	1,182
Roller	CFG	0.0(0)	4.1	4.5	8.6								
	PSA/CI	0.0(0)	13.2	32.4	45.6	17	222	205	2,912	68	983	8,193	124,926
	PISA/CI	0.0(0)	13.1	33.0	46.1	17	224	205	2,912	68	983	8,193	124,926
	PISA/CS	0.0(0)	18.0	31.4	49.4	17	221	206	2,956	59	594	8,193	124,926
SnipSnap	CFG	0.0(0)	1.8	3.3	5.1								
	PSA/CI	0.0(0)	9.7	20.5	30.2	25	234	129	1,411	126	679	1,908	22,571
	PISA/CI	0.0(0)	9.9	21.0	30.9	25	247	128	1,403	129	679	1,908	22,571
	PISA/CS	0.0(0)	21.2	22.6	43.8	30	262	145	1,601	241	865	1,908	22,571
Webgoat	CFG	0.0(0)	1.4	17.1	18.5								
	PSA/CI	0.0(0)	8.5	33.7	42.2	91	420	75	814	683	1,893	633	8,002
	PISA/CI	0.0(0)	9.2	34.3	43.5	92	434	75	821	684	1,893	633	8,002
	PISA/CS	0.0(0)	8.4	32.7	41.1	72	407	73	803	596	2,401	633	8,002
JSPWiki	CFG	0.0(0)	2.0	23.5	25.5								
	PSA/CI	80.5(10)	396.0	36.7	513.2	41	271	97	1,317	493	1,398	1,539	36,079
	PISA/CI	80.7(10)	394.8	36.7	512.2	41	272	97	1,318	493	1,398	1,539	36,079
	PISA/CS	152.5(9)	388.3	43.8	584.6	43	286	122	1,595	487	1,876	2,064	36,079
MVN-Forum	CFG	30.6(1)	1.3	92.0	123.9								
	PSA/CI	44.7(1)	11.4	59.6	115.7	25	229	89	1,423	124	980	1,406	50,894
	PISA/CI	64.7(2)	11.2	23.9	99.8	25	234	70	698	124	980	552	6,305
	PISA/CS	47.2(1)	18.5	59.5	125.2	29	253	94	1,525	144	1,058	1,406	50,894
Total	CFG	30.6(1)	12.8	148.8	192.2								
	PSA/CI	155.5(12)	445.8	209.6	810.9								
	PISA/CI	175.6(13)	445.3	175.8	796.7								
	PISA/CS	320.0(14)	546.2	218.4	1084.6								

PISA/CI versus CFG-based String Analyzer. We summarize the results for the 8 Web applications in the Total row, showing that PISA/CI detected and categorized 114 sanitizers compared to 72 sanitizers detected by the CFG-based string analyzer.

Figure 17 shows several true sanitizers that were successfully detected by PISA. Of these sanitizers, only method `normalize`, defined in class `BlojsomUtils` of Blojsom, was detected by both the CFG-based string analyzer and PISA. The other methods were not detected by the CFG-based string analyzer.

Here, we consider method `cleanLink`, defined in class `MarkupParser` of JSPWiki, and method `getFileName`, defined in the class `Course` of Webgoat, mentioned above. The purpose of `cleanLink` is to keep only legal characters (letters, numbers, and characters specified by `PUNCTUATION_CHARS_ALLOWED`) in the link using the branch condition. Method `getFileName` checks for the existence of the illegal characters, and extracts the substring between “`/`” and “`.`” using the methods `substring`, `indexOf`, and `lastIndexOf`.

In terms of efficiency, in our experiments PISA/CI was almost 4.2 times slower than the CFG-based string analyzer. According to our observations, the reason is that PISA/CI has to deal with branch conditions as well as integer and string values. However, when only the verification time was considered, PISA/CI was only 1.4 times slower than the CFG-based string analyzer. Also, the

```
static final String PUNCTUATION_CHARS_ALLOWED = " ()&+,-=._$";
static String cleanLink(String link){
  return cleanLink(link, PUNCTUATION_CHARS_ALLOWED);
}
static String cleanLink(String link, String allowedChars){
  if (link == null) return null;
  link = link.trim();
  StringBuffer clean=new StringBuffer(link.length());
  boolean isWord = true; boolean wasSpace = false;
  for (int i = 0; i < link.length(); i++){
    char ch = link.charAt(i);
    if (Character.isWhitespace(ch)) {
      if (wasSpace) continue;
      wasSpace = true;
    } else {
      wasSpace = false;
    }
    if (Character.isLetterOrDigit(ch)|| allowedChars.indexOf(ch) != -1) {
      if (isWord) ch = Character.toUpperCase(ch);
      clean.append(ch); isWord = false;
    } else {
      isWord = true;
    }
  }
  return clean.toString();
}


private static String getFileName(String s) {
  String fileName = new File(s).getName();
  if (fileName.indexOf("/") != -1) {
    fileName =
      fileName.substring(fileName.lastIndexOf("/"),fileName.length());
  }
  if (fileName.indexOf(".") != -1) {
    fileName = fileName.substring(0,fileName.indexOf("."));
  }
  return fileName;
}


public String getCcnParameter(String name){
  return getRegexParameter(name, "\\d{16}");
}
public String getPhoneParameter(String name){
  return getRegexParameter(str, "[\\d\\s-]+");
}
private String getRegexParameter(String name, String regexp) {
  String param = getStringParameter(name,regexp);
  if (Pattern.matches(str,regexp)) return str;
  else return "";
}


public static String normalize(String path) {
  if (path == null) { return null; }
  String value = path;
  value = value.replaceAll("\\.*", "");
  value = value.replaceAll("/{2,}", "");
  return value;
}
```

Fig. 17. Sanitizers detected by PISA

CFG-string analyzer directly uses the automata given as the specification for checking the inferred CFGs, but PISA/CI generates M2L predicate declarations from the regular expressions, and MONA interprets those predicates each time PISA/CI verifies a sanitizer candidate.

PISA/CI versus PSA/CI. In Blojsom, SnipSnap, and Webgoat, PISA/CI detected 4 sanitizers , which include `getFileName` of Figure 17, for PATH vulnerabilities that were not detected by

```
final String entities[] = {"<", ">"};
final String refs[] = {"&lt;", "&gt;"};
String cleanByLoop(String s) {
  for (int i = 0; i < entities.length; i++)
    s = s.replaceAll(entities[i], refs[i]);
  return s;
}


String removeNonLetter(String str) {
  String ret = "";
  char[] cs = str.toCharArray();
  for (int i = 0; i < cs.length; i++)
    if (Character.isLetter(cs[i]))
      ret = ret + cs[i];
  return ret;
}


static String escapeHTML( String s ) {
  if ( s==null ) return "";
  else return Utilities.escapeHTML(s);
}
// the following two escapeHTML methods
// are defined in class Utilities.
static String escapeHTML(String s) {
  return escapeHTML(s, true);
}
static String escapeHTML(
    String s, boolean escapeAmpersand){
  if (escapeAmpersand) {
    s = stringReplace(s, "&", "&amp;"); }
  s = stringReplace(s, "&nbsp;", " ");
  s = stringReplace(s, "\"", "&quot;");
  s = stringReplace(s, "<", "&lt;");
  s = stringReplace(s, ">", "&gt;");
  return s;
}
static String stringReplace(
    String str, String str1, String str2){
  String ret = StringUtils.replace(str,str1,str2);
  return ret;
}
```

Fig. 18. Sanitizers not detected by PISA

PSA/CI, compared to 6 sanitizers for PATH vulnerability that were not detected by the CFG-based string analyzer but detected by PSA/CI. We observed that index-based string operations were used for replacing or removing unsafe substrings for XSS and HRS, but the indices were calculated by loops and/or numerical expressions (*e.g.*, `n + m`, where `n` and `m` are variables) that cannot be encoded in M2L.

PISA/CI versus PISA/CS. For Webgoat and JSPWiki, some true-positive sanitizers detected by PISA/CS were not detected by PISA/CI. Overall, PISA/CS detected 128 sanitizers in the 8 applications. The method `getCcnParameter` of Figure 17 is the simplified version of `ParameterParser.getCcnParameter` of Webgoat, which was detected as a sanitizer by PISA/CS, but which was not be detected by PISA/CI. The context-sensitive callgraph can distinguish among the callers of `getRegexParameter`. Thus, PISA/CS can determine that the return values of `getRegexParameter` called by `getCcnParameter` matched only the regular expression ”`\\d16`”. This allowed PISA/CS to detect `getCcnParameter` as a sanitizer for HRS. In contrast, PISA/CI could not detect `getCcnParameter` as a sanitizer for HRS since `getRegexParameter` is called by both `getCcnParameter` and `getPhoneParameter` and PISA/CI determined that the return values of `getRegexParameter` matched either ”`\\d16`” or ”`[\\d\\s]+`”. For the same reason, we had other true sanitizers that were not detected by PISA/CI, but were detected by PISA/CS.

```
String escapeLine(String s) {
  String ret =
    s.replace("<","&lt;").replace(">","&gt;")
  ret += "<br/>";
  return ret;
}
```

Fig. 19. A method considered safe

Limitations and False Negatives. The false negatives, which we found manually, were mainly caused by these limitations.

— The method `cleanByLoop` on Figure 18 should be a sanitizer for XSS since it never returns a string value containing < or >. However, neither PISA nor the CFG-based string analyzer can detect it as a sanitizer, even though the CFG-based string analyzer can handle the loop. This is because the resulting CFG inferred by the CFG-based string analyzer contains a string value that comes directly from the value of the parameter `s`. To solve this problem, we might have to unroll the loops while propagating the constant string values. Other examples of the same problem include these Webgoat's methods that were not detected as sanitizers: `ParameterParser.htmlEncode` of Webgoat, `Screen.convertMetachars`, and `HtmlEncoder.encode`.

— PISA's path-sensitivity relies on constraints on individual local variables that are directly checked by built-in Boolean functions. Due to this limitation, method `removeNonLetter` in Figure 18, which is similar to `Macros.removeNonAlphanumeric` in Roller, could not be detected as a sanitizer since PISA cannot determine that `cs[i]` used in the condition is the same as `cs[i]` used in the `then` block. Other examples of the same problem include these Roller's methods: `SmileysPlugin.htmlEscape`, `Macros.replaceNonAlphanumeric`, `Macros.removeNonAlphanumeric`, `Utilities.replaceNonAlphanumeric`, `Utilities.removeNonAlphanumeric`, and the Webgoat's method: `ParameterParser.getIPParameter`.

— We experimented only with 0-CFA and 1-CFA. However, we would need n-CFA ($n > 1$) to detect some sanitizers in the benchmark applications. For example, we need 3-CFA to detect method `escapeHTML` in Figure 18, which is defined in class Macros of Roller.

— Both PISA and the CFG-based string analyzer did not have a complete set of abstractions for built-in string operations (E.g.: `SimpleDateFormat.format` called by methods in PersonalBlog and Roller, and `StringTokenizer.nextToken` called by methods in Webgoat).

Aside from the false negatives, we found following considerable limitations during the manual review of sanitizers.

— Our implementation did not handle the length of strings due to the limiation from M2L, hence branch conditions checking the length of strings cannot be encoded in M2L. However, the emptiness checks, which are written like `s.length()==0`, are encoded as in the case of `s.isEmpty()`.

— Method `escapeLine` in Figure 19, returns a string containing `"<br/>"`, though '¡' and '¿' are correctly escaped, hence the method is not considered a sanitizer under our specification. However, strings returned by the method can be considered safe in that `"<br/>"` is not tainted (not input by a user). To detect this method as a sanitizer, we need more expressive specification language, with which we distinguish untainted strings from tainted strings as in the case of the string analysis combined with the taint-label propagation proposed by [Geay et al. 2009; Wang et al. 2009]. For implementing the approach [Geay et al. 2009] with PISA, only additional 1-bit is needed to represent taintedness of characters.

6.3.2. Integration with Taint Analysis. To gain insight on the impact of PISA on the overall precision of the security scanner, we integrated PISA/CS into the taint-analysis algorithm used by the IBM Rational AppScan [App]. Table IV lists the results we obtained in terms of the number of vulnerable locations (call sites of the sinks) reported by the scanner.

For this study, we weakened our criterion for identifying sanitizer candidates by lifting the requirement that the method's input be a single string argument. (We used this more liberal criterion

Table IV. Results of the taint-analysis-integration experiment

	w/o pre-defined sanitizers								w/ predefined sanitizers							
	w/o PISA (Configuration A)				w/ PISA (Configuration B)				w/o PISA (Configuration C)				w/ PISA (Configuration D)			
	XSS	HRS	LOG	PATH	XSS	HRS	LOG	PATH	XSS	HRS	LOG	PATH	XSS	HRS	LOG	PATH
SBM	118	4	0	4	115	4	0	4	118	1	0	4	115	1	0	4
Blojsom	1	5	97	14	1	4	94	10	1	4	84	14	1	4	81	10
PersonalBlog	1	0	3	0	1	0	3	0	1	0	3	0	1	0	3	0
Roller	5	0	9	0	4	0	9	0	3	0	9	0	2	0	9	0
SnipSnap	50	6	10	8	50	6	10	7	50	6	10	3	50	6	10	2
Webgoat	8	1	7	4	8	1	7	4	8	1	5	4	8	1	5	4
JSPWiki	27	35	91	19	27	15	88	19	25	14	78	10	25	14	75	7
MVNForum	108	10	2	4	102	10	2	4	70	9	2	2	64	9	2	2
Total	651				609				539				515			
Relative # of FPs	136				94				24				0			

to guarantee that the candidate filter does not eliminate too many real sanitizers due to their signature.) Consequently, PISA/CS detected 2423 methods as sanitizers. We evaluated four different configurations, each corresponding to a particular combination of whether or not PISA is used and whether or not the set of predefined sanitizers provided as part of the AppScan algorithm is used. The four configurations we defined thus allow us to appreciate the effect of using only an automatically generated specification (configuration B) compared to the other three alternatives of (a) using no specification at all (configuration A), (b) using only a manual specification as authored by a team of security experts (configuration C), and (c) using the most complete specification we can obtain (configuration D).

The numbers in Table IV confirm that the effect of the sanitizer detection was significant, where we counted the number of vulnerability locations reported for each pair of benchmark application and vulnerability type, and calculated the total of those numbers. Note that the richer the sanitizer specification is, the more accurate the taint analysis becomes, since we can reduce false reports by increasing sanitizers. Thus, configuration D is the most accurate, and so we counted false positives (FPs) relative to configuration D.

As the result, configuration A yielded 136 false positives relative to configuration D, while B and C ruled out 42 false positives and 112 false positives, respectively. This shows that PISA (configuration B) can account for 38% of the benefit from a manual specification of sanitizers (configuration C) prepared by a security team with considerable effort. In addition, configuration C yielded 24 false positives relative to configuration D. This means that, thanks to PISA, the user can still improve the accuracy of the taint analysis tool in a fully automated fashion, without requiring any time or expertise from the development team.

7. RELATED WORK

Many string-analysis algorithms have been presented to date. Java String Analyzer (JSA) was first introduced by Christensen, *et al.* [Christensen et al. 2003]. JSA approximates a string value by a regular language, allowing for statically checking errors in dynamically generated SQL queries. According to the online manual [Christensen et al. 2009], the latest version of JSA has a form of path sensitivity through assertions, which is similar to ours. However, any experimental results of the path sensitivity has not yet been presented.

Minamide [Minamide 2005] proposed approximating string values with a CFG and modeling built-in string operations using transducers to check the well-formedness of dynamically generated HTML documents. Wassermann and Su [Wassermann and Su 2007] extended Minamide's algorithm to syntactically isolate unsafe substrings from safe substrings in PHP programs. Their string analyzer can also account for regular-expression matches, but their paper does not mention how to deal with branch conditions that consist of string comparisons and Boolean operators as well as regular-expression matches.

Yu, *et al.* [Yu et al. 2008] proposed another string-analysis algorithm for PHP, in which built-in string operations are modeled by the standard operations and a newly introduced *replacement* operation on automata. They also used the MONA's automaton package to implement these operations, and their approach was augmented with a backward analysis [Yu et al. 2009].

Kieżun, *et al.* [Kieżun et al. 2009a] presented HAMPI, a string-constraint solver based on quantifier-free bit-vector logic. Their subsequent paper [Ganesh et al. 2011] mentioned indices of strings in their core language model. However, HAMPI is designed to treat the bounded model of string constraints, thus it treats only fixed-size grammars. Hooimeijer and Weimer [Hooimeijer and Weimer 2009] presented a decision procedure for solving constraints on regular languages, and applied the proposed decision procedure to infer input parameters that create SQL injection vulnerabilities. Fu, *et al.* [Fu and Li 2010] proposed another constraint solver based on a variation of the word equation, in which string-replacement operations were modeled using finite-state transducers.

Hooimeijer *et al.* [Hooimeijer et al. 2011] presented BEK, a language for string manipulation. The BEK semantics is defined via a symbolic finite transducer; hence, it can reason about strings of unbonded length. The decision procedure of symbolic finite transducers was further studied by Veanes *et al.* [Veanes et al. 2012]. As a whole, only HAMPI addressed the need for index sensitivity, and only JSA and BEK have a form of path sensitivity.

Our string analysis approach can be used for generating input strings, and can be combined with several symbolic execution techniques. Wasserman, *et al.* [Wassermann et al. 2008] presented a dynamic test input generation that uses finite-state transducers. Kieżun, *et al.* [Kieżun et al. 2009b] presented Ardilla, a dynamic analysis tool for creating SQL injection attacks, and mentioned only potential of using a string analysis for generating attacks instead of the custom-made attack generator. However, their subsequent paper [Ganesh et al. 2011] demonstrated the use of HAMPI for generating attacks.

There are a few other papers that combine symbolic execution and string analysis with index sensitivity. Bjørner, *et al.* [Bjørner et al. 2009] proposed to use word equations for checking the feasibility of paths generated by a dynamic symbolic-execution engine Pex [Tillmann and Halleux 2008], while handling the same kind of index sensitivity. In this work, the path constraints are checked by the SMT solver Z3 [Z3], which is also used by Rex [Veanes et al. 2010]. Saxena, *et al.* [Saxena et al. 2010] recently proposed Kudzu that is a symbolic execution engine combined with a string constraint solver covering bit-vector logic and word equation. However, in the core language models used by the above two techniques, no string-replacement functions, which is essential for making Web applications secure, were not statically modeled. Shannon, *et al.* [Shannon et al. 2009] proposed to treat methods like `indexOf` in their symbolic execution engine by modeling convertion between symbolic strings and symbolic integers. Compared to their approach, our approach could be more precise, since it simultaneously treats string constraints and index constraints without any conversions.

Checking satisfiability of an M2L formula could be implemented by exploiting other solvers such as a SAT solver or a SMT solver through constructing a bounded model of M2L(Str) [Ayari and Basin 2000] , or symbolic representations of automata [Veanes et al. 2010]. In addition, the idea of representing strings as subsequences of a larger sequence was also used in [Maier 2009]. Engelfriet and Hoogeboom [Engelfriet and Hoogeboom 2001] have shown that M2L-definable string transductions can be implemented by deterministic two-way finite-state transducers, and are closed under composition. We could leverage their results to compose string operations.

String analysis, as well as PISA, can be integrated with various static taint analyses, and can contribute to finding and categorizing user-defined sanitizers in application code. Tripp, *et al.* [Tripp et al. 2009] presented a Taint Analysis for Java (TAJ), designed to efficiently analyze large Web applications. Livshits and Lam [Livshits and Lam 2005] characterized a static taint analysis with *source*, *sink*, and *derivation* descriptors, where the derivation descriptors specify how data is propagated by a given method. Their analysis leverages a context-sensitive points-to analysis based on BDDs [Whaley and Lam 2004]. Livshits, *et al.* [Livshits et al. 2009] presented Merline, a tool for automated and probabilistic inference of explicit information-flow specifications consisting of sets of sources, sinks and sanitizers.

These works did not make any use of string analysis, and relied on specifications of sanitizers without any guarantee that the sanitizers configured into the static taint analyzers were correct or that the specifications themselves were complete.

Balzarotti, *et al.* [Balzarotti et al. 2008] examined a string analysis for improving the accuracy of their sanitizer detection algorithm, but did not mention how to improve the string analysis algorithm.

To the best of our knowledge, only our paper presented how effectively the accuracy improvement of the string analysis can contribute to the accuracy improvement of the taint analysis.

8. CONCLUSION

In this paper, we presented a novel string-analysis technique, which enables unprecedented precision when modeling string operations, thanks to the combination of path and index sensitivity. Our string analysis is conducted by encoding the program in M2L, and relies on the satisfiability checking of an M2L formula. Our technique is motivated by the need for effective security analysis of Web applications, where a robust procedure for detecting and verifying sanitizers is essential. Our evaluation of the proposed approach shows it to compare favorably to the CFG-based string analysis of [Geay et al. 2009] (discovering 128 *vs.* 71 sanitizers), and have a significant impact on its client taint analysis' precision.

Acknowledgements

We thank Anders Møller and Yinnon Haviv for their invaluable comments and advice on a preliminary version of this paper, Naoshi Tabuchi for his help on our earlier work, the developers of MONA, WALA, and IBM AppScan for providing some of the building blocks for this work, and the anonymous reviewers of TOSEM for their invaluable comments.

REFERENCES

IBM Rational AppScan Source Edition. `http://www.ibm.com/software/rational/products/appscan/source`.

Open Web Application Security Project (OWASP). `http://www.owasp.org/index.php/Category:Attack`.

AYARI, A. AND BASIN, D. 2000. Bounded model construction for monadic second-order logics. In *Proceedings of Iternational Conference on Computer-aided Verification (CAV).*

BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland).*

BJØRNER, N., TILLMANN, N., AND VORONKOV, A. 2009. Path feasibility analysis for string-manipulating programs. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS).*

BRUMLEY, D., WANG, H., JHA, S., AND SONG, D. 2007. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the IEEE Computer Security Foundations Symposium.*

CHRISTENSEN, A. S., FELDTHAUS, A., AND MØLLER, A. 2009. JSA – the Java String Analyzer. `http://www.brics.dk/JSA`.

CHRISTENSEN, A. S., MØLLER, A., AND SCHWARTZBACH, M. I. 2003. Precise analysis of string expressions. In *Proceedings of International Static Analysis Symposium (SAS).*

COUSOT, P. AND COUSOT, R. 1995. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA).*

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13,* 4.

ENGELFRIET, J. AND HOOGEBOOM, H. J. 2001. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL) 2,* 2.

FU, X. AND LI, C.-C. 2010. A string constraint solver for detecting web application vulnerability. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE).*

GANESH, V., KIEŻUN, A., ARTZI, S., GUO, P. J., HOOIMEIJER, P., AND ERNST, M. 2011. HAMPI: a string solver for testing, analysis and vulnerability detection. In *Proceedings of the 23rd International Conference of Computer Aided Verification (CAV).*

GEAY, E., PISTOIA, M., TATEISHI, T., RYDER, B., AND DOLBY, J. 2009. Modular string-sensitive permission analysis with demand-driven precision. In *Proceedings of the 31th International Conference on Software Engineering (ICSE).*

GROVE, D. AND CHAMBERS, C. 2001. A Framework for Call Graph Construction Algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23,* 6.

GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. 1997. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA).*

HAMMER, C., SCHAADE, R., AND SNELTING, G. 2008. Static path conditions for java. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security (PLAS).*

HENRIKSEN, J. G., JENSEN, J. L., JØRGENSEN, M. E., KLARLUND, N., PAIGE, R., RAUHE, T., AND SANDHOLM, A. 1995. MONA: Monadic second-order logic in practice. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS).*

HOOIMEIJER, P., LIVSHITS, B., MOLNAR, D., SAXENA, P., AND VEANES, M. 2011. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*. SEC'11. USENIX Association, Berkeley, CA, USA, 1–1.

HOOIMEIJER, P. AND WEIMER, W. 2009. A decision procedure for subset constraints over regular languages. In *Proceedings of Programming Language Design and Implementation (PLDI).*

KAY, M. AND KAPLAN, R. M. 1994. Regular models of phonological rule systems. *Computational Linguistics, 20(3).*

KIEŻUN, A., GANESH, V., GUO, P. J., HOOIMEIJER, P., AND ERNST, M. D. 2009a. HAMPI: A solver for string constraints. In *Proceedings of the ACM International Symposium on Testing and Analysis (ISSTA).*

KIEŻUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. 2009b. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE.*

KLARLUND, N. AND MØLLER, A. 2001. *MONA Version 1.4 User Manual*. BRICS. Notes Series NS-01-1. `http://www.brics.dk/mona`.

LIVSHITS, B., NORI, A. V., RAJAMANI, S. K., AND BANERJEE, A. 2009. Merline: Specification inference for explicit information flow problems. In *Proceedings of Programming Language Design and Implementation (PLDI).*

LIVSHITS, V. B. AND LAM, M. S. 2005. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium.*

MAIER, P. 2009. Deciding extensions of the theories of vectors and bags. In *Verification, Model Checking, and Abstract Interpretation (VMCAI).*

MINAMIDE, Y. 2005. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web (WWW).*

REPS, T. 1997. Program analysis via graph reachability. In *ILPS.*

ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL).*

SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. 2010. A symbolic execution framework for javascript. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland).*

SHANNON, D., GHOSH, I., RAJAN, S., AND KHURSHID, S. 2009. Efficient symbolic execution of strings for validating web applications. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems (DEFECTS).*

$$
\begin{array}{rcll}
x & \in & \mathcal{X} & \\
b, p & \in & \mathbf{N} & \\
s & \in & \Sigma^* & \\
\mathit{bool} & \in & \{\mathsf{true}, \mathsf{false}\} & \\
v & ::= & s \mid p \mid \mathit{bool} & \\
I & ::= & x = v & \text{Assignment} \\
 & \mid & x = f(\vec{x}) & \text{Function call} \\
 & \mid & x = \mathtt{phi}(b : x, \ldots, b : x) & \Phi\text{-instruction} \\
 & \mid & x = \mathtt{jump}\ x, b & \text{Conditional jump} \\
 & \mid & x = \mathtt{goto}\ b & \text{Goto} \\
B & ::= & (b, \vec{I}) & \text{Basic block} \\
N & ::= & \{B\} & \text{Set of basic blocks}
\end{array}
$$

Fig. 20. Syntax of the target language

SNELTING, G. 1996. Combining slicing and constraint solving for validation of measurement software. In *Proceedings of the Third International Symposium on Static Analysis*.

TILLMANN, N. AND HALLEUX, J. D. 2008. Pex: white box test generation for .NET. In *Proceedings of Tests and Proofs (TAP)*.

TRIPP, O., PISTOIA, M., FINK, S., SRIDHARAN, M., AND WEISMAN, O. 2009. TAJ: Effective taint analysis of web applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

VEANES, M., DE HALLEUX, P., AND TILLMANN, N. 2010. Rex: Symbolic regular expression explorer. In *Proceedings of the Third International Conference on Software Testing, Verification, and Validation (ICST)*. Vol. 0. IEEE Computer Society, Los Alamitos, CA, USA, 498–507.

VEANES, M., HOOIMEIJER, P., LIVSHITS, B., MOLNAR, D., AND BJORNER, N. 2012. Symbolic finite state transducers: algorithms and applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '12. ACM, New York, NY, USA, 137–150.

WALA. T. J. Watson Libraries for Analysis, `http://wala.sf.net/`.

WANG, X., ZHANG, L., XIE, T., MEI, H., AND SUN, J. 2009. Locating need-to-translate constant strings for software internationalization. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*.

WASSERMANN, G. AND SU, Z. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of Programming Language Design and Implementation (PLDI)*.

WASSERMANN, G., YU, D., CHANDER, A., DHURJATI, D., INAMURA, H., AND SU, Z. 2008. Dynamic test input generation for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

WEGMAN, M. N. AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13,* 2.

WHALEY, J. AND LAM, M. S. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of Programming Language Design and Implementation (PLDI)*. Vol. 39.

YU, F., ALKHALAF, M., AND BULTAN, T. 2009. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

YU, F., BULTAN, T., COVA, M., AND IBARRA, O. 2008. Symbolic string verification: An automata-based approach. In *Proceedings of the 15th International SPIN Workshop on Model Checking of Software*.

Z3. Z3, `http://research.microsoft.com/projects/z3`.

A. TARGET LANGUAGE

Figure 20 shows the formal syntax of our target language. A program consists of a set of basic blocks N. The meta-variables x and v represent a program variable and a value, respectively, where the value v is a string s, an index p, or a Boolean value *bool*. The guard condition c represents

(CONST)
$\sigma \vdash (b, b', x = v; \vec{I}) \to \sigma[v/x] \vdash (b, b', \vec{I})$
(CALL)
$\sigma \vdash (b, b', x = f(x_1, \cdots, x_n); \vec{I}) \to \sigma[v/x] \vdash (b, b', \vec{I})$
 where $v = f(\sigma(x_1), \cdots, \sigma(x_i))$
(PHI)
$\sigma \vdash (b, b_i, x = \texttt{phi}(b_1 : x_1, \cdots, b_n : x_n); \vec{I}) \to \sigma[v_i/x] \vdash (b, b_i, \vec{I})$
 where $v_i = \sigma(x_i)$
(JUMP)

$$\sigma \vdash (b, b', \texttt{jump}\ x, b'') \to \begin{cases} \sigma \vdash (b'', b, \vec{I}) \\ \quad \text{when } \sigma(x) = \mathsf{true}, \text{ where } (b'', \vec{I}) \in N. \\ \sigma \vdash (b+1, b, \vec{I}) \\ \quad \text{when } \sigma(x) = \mathsf{false}, \text{ where } (b+1, \vec{I}) \in N. \end{cases}$$

(GOTO)
$\sigma \vdash (b, b', \texttt{goto}\ x, b'') \to \sigma \vdash (b'', b, \vec{I})$ where $(b'', \vec{I}) \in N$
(BB)
$\sigma \vdash (b, b', \epsilon) \to \sigma \vdash (b+1, b, \vec{I})$ where $(b+1, \vec{I}) \in N$

Fig. 21. Operational semantics

consists of Boolean values, the program variables, and Boolean operators. The meta-variable B is a basic block which is numbered by b and contains the sequence of instructions $\vec{I}$, where we denote an empty sequence by ϵ and a delimiter by ";". We omit the `return` instructions, since we are discussing intraprocedural string analysis.

The semantics of the target language is depicted as transition rules in Figure 21. $\sigma \vdash (b, b', \vec{s})$ denotes a program state, where σ is a mapping from variables to values, b is a current basic block number, b' is the immediate predecessor of the current basic block b, and $\vec{I}$ is a sequence of instructions. In addition, we denotes the transitive closure of $\to$ by $\to^*$.

B. PROOF OF SOUNDNESS

The proof is done by induction on the rules of the operational semantics with the following invariant inv on the program states σ:

$$\begin{array}{l} \mathsf{inv}(\sigma) \equiv \exists w, \mathcal{I} \ .\ \mathsf{inv}'(\sigma, w, \mathcal{I}) \\ \text{where } \mathsf{inv}'(\sigma, w, \mathcal{I}) \equiv \forall x \in \mathit{dom}(\sigma) \ .\ \sigma(x) \in L_{w,\mathcal{I}}(\mathsf{prog}_x) \end{array}.$$

Note that the formula is exactly equivalent to Theorem 3.2 if we expand the definition of inv'.

Obviously, at the initial state σ_0, $\mathsf{inv}(\sigma_0)$ holds, since $\mathit{dom}(\sigma) = \emptyset$ holds.

Next, for every rule, we prove that, if $\mathsf{inv}(\sigma)$ holds for the program state σ, $\mathsf{inv}(\sigma')$ also holds for a program state σ' that is obtained by a one-step transition from the program state σ.

CONST. If $\mathsf{inv}(\sigma)$ holds, there exists a finite string w and an assignment $\mathcal{I}$ that satisfy $\mathsf{inv}'(\sigma, w, \mathcal{I})$. In addition, there exists a finite string w' such that $v \in L_{w',\mathcal{I}}(\text{“}v\text{”})$. Therefore, $v \in L_{ww',\mathcal{I}}(\text{“}v\text{”})$ is also holds, where ww' is the concatenation of the finite strings w and w'. Since $\mathsf{prog}_x(R) \equiv \text{“}v\text{”}(R)$ is the predicate declaration encoded from the CONST rule, $v \in L_{ww',\mathcal{I}}(\mathsf{prog}_x)$ holds. Therefore, taking $\sigma' = \sigma[v/x]$ into account, (since the invariant $\mathsf{inv}'(\sigma', ww', \mathcal{I})$ holds), $\mathsf{inv}(\sigma')$ also holds.

CALL. If $\mathsf{inv}(\sigma)$ holds, there exists a finite string w and an assignment $\mathcal{I}$ that satisfy $\mathsf{inv}'(\sigma, w, \mathcal{I})$. Thus, for every parameter $x_i (i = 0, \cdots, n)$ of the function call, $\sigma(x_i) \in L_{w,\mathcal{I}}(\mathsf{prog}_{x_i})$ holds. Here, from Definition 3.3, there exists a finite string w' that satisfies

$$v \in L_{ww',\mathcal{I}}(\lambda R. \llbracket f \rrbracket (R, \mathsf{prog}_{x_1}, \cdots, \mathsf{prog}_{x_n}))$$

, where v is a return value of the function call. Accordingly, we can obtain $v \in L_{ww',\mathcal{I}}(\mathsf{prog}_x)$ from the fact that a corresponding predicate declaration is

$$\mathsf{prog}_x(R) \equiv \llbracket f \rrbracket (R, \mathsf{prog}_{x_1}, \cdots, \mathsf{prog}_{x_n}).$$

Therefore, at the program state $\sigma' = \sigma[v/x]$, $\mathsf{inv}(\sigma')$ holds.

PHI. $\sigma(x_i) \in L_{w,\mathcal{I}}(prog_{x_i})$ follows from $\mathsf{inv}(\sigma)$, where $x_j (i = 0, \cdots, n)$ is the parameter of the phi-function. Therefore, for the return value v_i, this formula holds:

$$v_i \in L_{w,\mathcal{I}}(\mathsf{prog}_{x_1}) \vee \cdots \vee L_{w,\mathcal{I}}(\mathsf{prog}_{x_n})$$

Since the corresponding predicate declaration is

$$\mathsf{prog}_x(R) \equiv \mathsf{prog}_{x_1}(R) \vee \cdots \vee \mathsf{prog}_{x_n}(R),$$

we obtain the formula $v_i \in L_{w,\mathcal{I}}(\mathsf{prog}_x)$. Thus, at the program state $\sigma' = \sigma[v_i/x]$, $\mathsf{inv}(\sigma')$ holds.

JUMP,GOTO,BB. There are no updates on the program state. Therefore, the invariant is preserved.

C. FINDING LOOP INVARIANT

A simple solution to find a loop invariant is to use the idea of character-set approximation. We first introduce a free position set variable CS_x that represents a set of characters consisting of possible string values to be assigned to the program variable x. We then define the invariant inv_x as

$$\mathsf{inv}_x(R) \equiv \forall r.r \in R \Rightarrow r \in \mathsf{CS}_x$$

, where $r \in R$ means that r is a character in the string represented by R, and $r \in \mathsf{CS}_x$ means that a character represented by r is in the character set represented by CS_x. The position set CS_x can be constrained by

$$\forall R.\mathsf{prog}_x(R) \Rightarrow \mathsf{inv}_x(R)$$

, since inv_x is the loop invariant. However, this solution is not useful to check for the existence of an unsafe string, since the constraint defines the lower bound of CS_x, but does not define the upper bound. We can find the least solution of CS_x by introducing the equality of characters, which is different from the equality of strings, and the equality of position sets. However, we would need a further abstraction of the character set, since checking the equality of two characters is expensive. Therefore, we decided to use the declaration $\mathsf{prog}_x(R) \equiv \mathsf{true}$.

D. CFG-BASED STRING ANALYSIS

Our CFG-based string analysis approximates the possible strings arising at runtime using a CFG as in the case of Minamide's string analysis [Minamide 2005], and checks if the obtained CFG is contained by a regular expression that is specified as a specification. The CFG is deduced by translating assignment relationships to production rules of the CFG. Let us consider the following Java program, which append "a" to the string assigned to the variable a three times after initializing it with "xxa".

```
String a = "xxa";
for (int i = 0; i < 3; i++) a = a + "a";
```

The corresponding CFG is obtained by translating every program variable v to a nonterminal S_v and = to $\rightarrow$ as in production rules.

$$S_a \rightarrow \mathtt{x\ x\ a}$$
$$S_a \rightarrow S_a\ \mathtt{a}$$

The CFG with start symbol S_a represents a possible set of strings assigned to program variable a, which yields the set of strings {"xxa", "xxaa", "xxaaa", $\cdots$}.

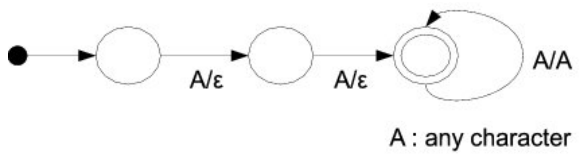

Fig. 22. An transducer $[\![$`substring(2)`$]\!]$

When a program contains a predefined string operation f, we first construct a set of production rules with the abstraction $[\![f]\!]$ of the string operation f. The abstraction is defined by a transducer, which is an automaton with outputs, and can translate a CFG to another CFG. For example, let us consider the following program, in which the first two lines are the same as the previous program and string operation `substring(2)` is called at the last line.

```
String a = "xxa";
for (int i=0; i<3; i++) a = a + "a";
String r = a.substring(2);
```

This program is translated to the following production rules with the abstraction $[\![$`substring(2)`$]\!]$.

$$\begin{array}{l} S_a \to \mathtt{x\ x\ a} \\ S_a \to S_a\ \mathtt{a} \\ S_r \to [\![\mathtt{substring(2)}]\!]\,(S_a) \end{array}$$

The abstraction $[\![$`substring(2)`$]\!]$ is defined by the transducer shown in Figure 22. By applying that transducer to the CFG with the start symbol S_a, we can obtain the following CFG with the start symbol S_r, which represents the string set $\{$“`a`”, “`aa`”, “`aaa`”, $\cdots\}$.

$$\begin{array}{l} S'_a \to \mathtt{a} \\ S'_a \to S'_a\ \mathtt{a} \\ S_r \to S'_a \end{array}$$

Note that, when we have a cyclic definition that is affected by string operations, we compute an invariant by iterating the grammar transduction and by using the character-set approximation [Christensen et al. 2003; Minamide 2005] as a widening operation.

After deducing the above grammar, given a regular expression r as specification that represents unsafe strings, we check the emptiness of $L(r) \cap L(G)$, where $L(r)$ is a set of strings matched by the regular expression r, and $L(G)$ is a set of strings yielded by the above grammar. This check can be done by CFL reachability algorithm [Reps 1997]. If $L(r) \cap L(G)$ is empty, we can say that the program does not produce any unsafe strings. Otherwise, the program may produce an unsafe string.