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

Takaaki Tateishi, Marco Pistoia, Omer Tripp

IBM Research - Tokyo
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan
Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic

Takaaki Tateishi
IBM Research - Tokyo
tate@jp.ibm.com

Marco Pistoia
IBM Research - T. J. Watson Research Center
pistoia@us.ibm.com

Omer Tripp
IBM Software Group and
Tel Aviv University
omert@il.ibm.com

ABSTRACT

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

Keywords
String Analysis, Static Program Analysis, Web Security

1. INTRODUCTION

String analysis [7, 8, 12, 17, 19, 23, 32] 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) [2].

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 [8, 23, 32]. 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 \(<script>\) 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 &lt;.

In order to verify that the program is immune to XSS attacks, we need to prove that no string generated by the program contains &lt;. According to this specification, clean is considered a valid XSS sanitizer; when condition v1.contains(v2) holds, &lt; 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 &lt;. However, a path-insensitive grammar-based string analysis cannot follow this line of reasoning, since it would...
fail to capture the relationship between \( v_2 \) and \( v_3 \), 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\(\text{Str} \) (Monadic Second-order Logic on strings) [16]. 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\(\text{Str} \) 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\(\text{Str} \) has the following advantages in addition to enabling index sensitivity combined with path sensitivity:

- **Conservativeness.** M2L\(\text{Str} \) 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 algorithm is unsatisfiable. In the first process, the effect of branch conditions and index-based string manipulations are also encoded that cannot be captured by fixed-size representation.

- **Efficient and effective automaton representation.** We can exploit an automatic theorem prover MONA [20] 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 [6, 34]) including counterexamples.

Our analysis consists of the following two automated processes: (i) encoding a string-manipulating method as an M2L formula \( \phi_1 \) that represents possible strings returned by the method, and (ii) encoding a regular expression indicating unsafe strings as an M2L formula \( \phi_2 \), 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 resulting 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 \( \phi_1 \). 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 [20]. Compared to existing string analyses based on bit-vector logic and/or word equation [5, 19, 25], 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 [4]. However, [4] 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 [1]. We evaluated PISA’s precision by comparing it with the technique of [23, 12] 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 extend our analysis to become interprocedural. Section 6 discusses our implementation of the algorithm, as well as experimental results. Section 7 surveys related work, and Section 8 concludes this paper.

## 2. OVERVIEW

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

### 2.1 Monadic Second-order Logic on Strings

M2L\(\text{Str} \) is a widely used vehicle for a variety of verification problems [16]. The syntax of M2L\(\text{Str} \) is defined in Figure 2, where \( \mathit{Var1} \) denotes a set of position variables and \( \mathit{Var2} \) a set of position-set variables. The formula ‘\( a’ \) holds if \( a_i \) in (finite)

\[
\text{String clean(String v1)} \{
\text{String v2 = "";} \\
\text{if (v1.contains(v2)) } \{ \\
\text{int v3 = v1.indexOf(v2);} \\
\text{String v4 = v1.substring(0, v3);} \\
\text{return v4;} \\
\text{\} return v1;} \\
\}
\]

**Figure 1:** Sanitization against XSS

**Figure 2:** Syntax of M2L\(\text{Str} \)


\[ \text{prog}_1(V_1) \equiv \text{true} \]

\[ \text{prog}_2(V_2) \equiv \langle \langle V_2 \rangle \rangle \]

\[ \text{prog}_3(V_2, V_1) \equiv \langle \langle \text{indexOff}(V_2, V_1, \text{prog}_1, \text{prog}_2) \rangle \rangle \]

\[ \text{prog}_4(V_1) \equiv \langle \langle \text{min}(V_1) \rangle \rangle \]

\[ \text{prog}_5(V_1) \equiv \langle \langle \text{contains}(<>) \rangle \rangle (V_1) \]

\[ \text{prog}_6(V_4) \equiv \langle \langle \text{substring}(V_4, \text{prog}_1, \text{prog}_2, \text{prog}_3) \rangle \rangle \]

\[ \text{prog}_7(V_1) \equiv \langle \langle \text{contains}(<>) \rangle \rangle (V_1) \]

Figure 3: An example of predicate declarations

\((P \rightarrow 2^{\text{Pos}})\), where \(P\) is the set of free position set variables, and \(2^{\text{Pos}}\) is the power set of the position set \(\text{Pos}\). When the formula holds, we write \(w, I \models \phi\). For example, the M2L formula \(\langle \langle a'(0) \rangle \rangle \land \langle \langle b'(1) \rangle \rangle \land \langle \langle c'(2) \rangle \rangle \land \langle \langle a'(3) \rangle \rangle \) holds on the finite string \(w = \text{"abca"}\), which states that character \(a\)'s position set \(0\) is located at positions 0 and 3, while characters \(b\) and \(c\)'s position set variables \(1\) and \(2\), respectively. Thus, we can write \(w, I \models \langle \langle a'(0) \rangle \rangle \land \langle \langle b'(1) \rangle \rangle \land \langle \langle c'(2) \rangle \rangle \land \langle \langle a'(3) \rangle \rangle \), where \(I = \{\}\).

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, the M2L formula \(\langle \langle a'(0) \rangle \rangle \land \langle \langle b'(1) \rangle \rangle \land \langle \langle c'(2) \rangle \rangle \land \langle \langle a'(3) \rangle \rangle \land \langle \langle P \rangle \rangle \land \langle \langle Q \rangle \rangle = \{0\} \land Q = \{1, 3\} \land R = \{0, 1, 3\} \land w = \text{"abca"}\) and \(I = \{\}\) is satisfied.

In this representation of strings, the concatenation of the two strings \(a\) and \(b\), denoted by \(a | b\), can be considered to be the strings \(a\) and \(b\). Thus, this concatenation relationship can be represented by the predicate:

\[ \text{concat}(R, P, Q) \equiv \langle \langle R = (P \cup Q) \rangle \rangle \land \forall \langle p, q. p \in P \land q \in Q \Rightarrow p < q \rangle \]

The predicate \(\text{concat}(R, P, Q)\) holds on \(w = \text{"abca"}\) and \(I = \{\}\) is satisfied.

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:

\[
\begin{align*}
\text{String } v1 & = \text{"a"}; \\
\text{String } v2 & = v1.\text{concat}(v1); \\
\end{align*}
\]

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

\[ \text{prog}_1(V_1) \equiv \langle \langle \text{a}'(V_1) \rangle \rangle \]

\[ \text{prog}_2(V_2) \equiv \langle \langle \text{concat} \rangle \rangle (V_2, \text{prog}_1, \text{prog}_1) \]

\[ \langle \langle \text{concat} \rangle \rangle (R, P_1, P_2) \equiv \langle \langle \exists P_3, P_4, P_5(P_1) \land P_6(P_2) \land \text{concat}(R, P_1, P_2) \rangle \rangle \]

Each predicate \(\text{prog}_1\) represents the post-condition for the assignment to program variable \(v_1\), where the parameters \(V_1\) and \(V_2\) of the predicates are associated with the program variables \(v1\) and \(v2\), respectively. \(\text{concat}\) 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 \(P_1\) and \(P_2\) 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 \(v^1\) and \(v^2\) 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 \(V_1\), \(V_2\), \(v_0\), and \(V_3\) are also M2L variables associated with the program variables \(v1,v2,v3\), and \(v4\), respectively, where upper-case variables \(V_1\), \(V_2\), and \(V_3\) are position set variables each of which represents a string, and the lowercase variable \(v_0\) 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 \(\text{prog}_1\) and \(\text{prog}_2\) where \(\langle \langle \text{contains}(<>) \rangle \rangle (V_1)\) is the abstraction of the condition \(\text{contains}(<>)\), which means that the string represented by \(V_1\) contains the string \('<\)". The predicate \(\text{prog}_2\) 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. \(\text{concat}\) and \(\text{substring}\) are the abstractions of string operations \(\text{indexOf}\) and \(\text{substring}\) as in the case of \(\text{concat}\), respectively. The details of these abstractions are discussed later in Section 3 and 4.

If our only concern is that the string \('<\)" is unsafe, an unsafe specification \(\text{Unsafe}\) for the program is defined as \(\exists R'. \text{substring}(R', V) \land \langle \langle \text{contains}(<>) \rangle \rangle (R')\), where \(\text{substring}(R', V)\) means that the string \(R'\) is a substring of \(V\). Alternatively, we can use the regular expression \(\langle \langle \text{contains}(<>) \rangle \rangle\) which is equivalent to the specification, since we can encode the regular expression into M2L. We then verify that the program never return a string containing the unsafe string by confirming the unsatisfiability of this formula:

\(\exists V. (\text{prog}_1(V) \lor \text{prog}_2(V)) \land \text{Unsafe}(V)\).

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 [24, 10]. An SSA program comprises numbered basic blocks, each of which consists of the instructions in Figure 4. In addition, we use the notation \(\text{op}(h)\) for built-in operator \(h\). For example, \(x = \text{op}(+) (1, 2)\) assigns the result of \(x + 2\) 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. 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.

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

\[
\begin{align*}
1:v0 & = 0; \\
v2 & = < >; \\
v3 & = \text{indexOf}(v1,v2); \\
1:v1 & = v0.\text{contains}(v2); \\
v4 & = \text{substring}(v1,v0,v3); \\
\text{jump} & b1; \\
1:b1 & = 3; \\
\text{return} & v4; \\
\end{align*}
\]

The reason why we do not use \(\text{contains}(V_1, V_2)\) 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) \( x = f(x_1, \ldots, x_n) \) The result of invoking function \( f \) with parameters \( x_1, \ldots, x_n \) is assigned to program variable \( x \).
(Conditional jump) \( x = \phi_i(b_1 : x_1, \ldots, b_n : x_n) \) When an immediate predecessor of the current basic block is \( b_i \), program-variable \( x \) is assigned the value of \( x_j \). Basic-block numbers are omitted for brevity when possible.
(Goto) \( jum p \ x, b \) The program jumps to the basic block numbered \( b \) if the value of \( x \) is true.
(Return) \( return \ x \) The value of the program variable \( x \) is returned.

**Figure 4: Instructions of the target language**

\[
\text{“a1 ⋯ a_n”(P)} \equiv \exists \text{t}_1, \ldots, \text{t}_n .
\text{“a1”(t}_1\text{) ∧ ⋯ ∧ “a_n”(t}_n\text{)} \land \text{t}_1 < \text{t}_2 < \text{t}_3 < \cdots < \text{t}_{n-1} < \text{t}_n
\land \text{P = } \{\text{t}_1, \ldots, \text{t}_n\}.
\]

\[
\text{concat}(R, P, Q) \equiv \text{R = P } \cup \text{Q}.
\]

\[
\text{strr}(R, P, p, q) \equiv p < q < r < q \Rightarrow \exists \text{p’, q’.} \text{p’ < p } \land \text{p’ < q } \land \text{q’ < q}.
\]

\[
\text{substr}(R, P, p, q) \equiv \text{substr}(R, P, \text{min}(P), \text{max}(P) + 1).
\]

\[
\text{consecutive}(p, q, R) \equiv p < q < r < q \Rightarrow r \not\in \text{R}.
\]

\[
\begin{align*}
\mathcal{T} & \rightarrow \text{AS} \cdot \mathcal{Z}(S) \\
\mathcal{T}_2 & \rightarrow \text{AS} \cdot \text{prog}(_2\text{)}(S) \\
\mathcal{N}_1 \mathcal{N}_2 & \rightarrow \text{AS} \cdot \text{N}_1 \cdot \text{S}_2 \cdot \langle \mathcal{N}_1\rangle(S_1) \land \langle \mathcal{N}_2\rangle(S_2) \land \text{concat}(\mathcal{S}, \mathcal{S}_2) \\
\langle \mathcal{N}_1\rangle \mathcal{N}_2 & \rightarrow \text{AS} \cdot \langle \mathcal{N}_1\rangle(S) \land \langle \mathcal{N}_2\rangle(S) \land \text{concat}(\mathcal{S}, \mathcal{S}_2) \\
\mathcal{N}^* & \rightarrow \text{AS} \cdot \exists \text{P} \cdot \text{min}(\text{P}) \in \text{P} \land \text{max}(\text{P}) + 1 \in \text{P} \land \forall \text{r, r’}. \text{\lambda}(r, r’, \text{P}) \Rightarrow \exists \text{Q} \cdot \text{strr}(\text{Q}, \text{S}, r, r’) \land \langle \mathcal{N}\rangle(Q)
\end{align*}
\]

**Figure 5: Utility predicates**

**Figure 6: Encoding regular expressions**

### 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, 1, \ldots, n - 1\}, that represents a word \( w \). Formally, if \( w = a_0 \ldots a_{n-1} \) and \( P = \{p_0, \ldots, p_{m-1}\} \) is a sorted position set, then \( P \) represents the string value \( s \) (of size \( m \)) iff \( P \) satisfies \( s = a_{p_0}a_{p_1}\ldots a_{p_{m-1}} \). Given \( w = \text{abca} \), 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 concat\( (R, P, Q) \) are the same as those introduced in Section 2.2. Intuitively, substr\( (R, P, p, q) \) denotes a string represented by \( R \) is the substring represented by \( P \) containing all the characters in the range \( [p, q] \). subtr\( (R, P, q) \) is similar to substr, the difference being that \( R \) may be any substring \( \text{min}(P) \) and \( \text{max}(P) \) return the minimum and maximum positions in \( P \), respectively. Finally, predicate 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 [20]).

Our algorithm for encoding regular expressions is the same as that of [20], except that we accept program variables as terminal symbols. Figure 6 shows how to encode a set of strings represented by a regular expression \( r \) as a predicate denoted by \( \langle r \rangle \). \( T \) represents a terminal symbol (i.e., a character), and \( T_2 \) represents a terminal symbol associated with program variable \( x \), where prog\_x denotes a property of strings possibly assigned to \( x \). \( N_1, \text{N}_2 \) represent nonterminal symbols. In addition, we use the notation \( \text{AS}_\phi \), instead of explicitly declaring a new predicate \( \psi \) such that \( \psi(S) = \phi \).

### 3.3 Abstracting Built-In Functions

We denote the abstraction of a built-in function \( f \) by [\( f \)]. 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 \( P_1 \) and \( P_2 \) used in the encoding of the string-concatenation program in Section 2.2 are instantiated by predicate prog\_\text{\textunderscore}v, thus yielding:

\[
\text{prog\_\text{\textunderscore}v}_2(P_2) \equiv
\exists P_1 \cdot \text{prog\_\text{\textunderscore}v}_1(P_1) \land \text{prog\_\text{\textunderscore}v}_2(P_2) \land \text{concat}(V_2, P_2, P_1).
\]

The examples of the abstractions we developed for the built-in functions are listed in Figure 7. Here, we focus only on \text{replace}, which abstracts the Java method \text{replace}, where \text{replace}(x, y) substitutes all occurrences of \( x \) in \( y \) 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. In our abstraction of \text{replace}, \( V \) 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 \( \langle y' \rangle \) encodes regular expression \( y' \), where \( y \) is the program variable corresponding to \( P_2 \). Predicate \( \langle \langle y' \rangle \rangle \) constrains \( X \) and \( Y \) to guarantee that each pair of removed and inserted positions is consecutive, and predicate \( \langle \langle y' \rangle \rangle \) encodes regular expression \( v' \), and constrains \( X \) to contain only
Figure 8: Encoding Instructions

The possible set of strings assigned to v2 can be represented by CFG \( v_2 \to v_0 \mid v_3 \to v_2 \mid v_1 \), where \( v_0 \) and \( v_1 \) are considered terminal symbols (being non-cyclic). This CFG is overapproximated by the regular expression \( v_0 v_1^* \), which is encoded in M2L as \( \{v_0 v_1^*\} \). Here are the resulting predicate declarations:

\[
\begin{align*}
\text{prog}_0(R) & \equiv "ab"(R) \\
\text{prog}_1(R) & \equiv \text{toUpperCase}(R, \text{prog}_0) \\
\text{prog}_2(R) & \equiv \{v_0 v_1^*\}(R) \\
\text{prog}_3(R) & \equiv \exists v_1, v_2. \text{prog}_2(v_2) \land \text{prog}_0(v_1) \\
& \land \text{cond}(R, v_2, v_1)
\end{align*}
\]

When cyclic variable \( x \) is affected by other string operations, we use the naïve abstraction \( \text{prog}_0(R) \equiv \text{true} \), which holds for any string value.

Alternatively, we could automatically find a loop invariant \( \text{inv} \), for \( x \), using the character-set abstraction [8], where a position set variable is used as a set of characters by ignoring the order of characters. Hence, this predicate can be used instead of \( \text{prog}_2 \).

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 [8, 23] and the trivial widening operation that always yields the naïve abstraction \( \text{prog}_0(S) = \text{true} \) to handle loops.

We first introduce the notation \( L_w.x(\psi) \) to denote the set of strings represented by \( R \) that satisfy \( \psi(\bar{R}) \) given infinite string \( w \) and assignment \( x \).

**Definition 1 (Generated Language).**

\[ L_w.x(\psi) \equiv \{ s \mid w, x \models \psi \} \]

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 \( x \), such that \( \psi(\bar{R}) \) holds, and \( \{P\} = \{\text{prog}_1, \ldots, \text{prog}_n\} \).

**Theorem 1 (Soundness of the Encoding Method).**

\[ \exists w, x \models \text{dom}(\sigma) \cdot \sigma(x) \in L_w.x(\text{prog}) \]

where \( \sigma \) represents a program state (interpreted as a mapping from program variables to values).

The above soundness criterion holds as long as for every string operation \( f \), the abstraction \( \text{af} \) satisfies the following condition:

**Definition 2 (Sound Abstraction of Function f).**

\[ \forall r, p_1, \ldots, p_n. r = f(p_1, \ldots, p_n) \Rightarrow \forall w, \sigma, \psi_1, \ldots, \psi_n. p_1 \in L_w.x(\psi_1) \land \cdots \land p_n \in L_w.x(\psi_n) \Rightarrow \exists w'. r \in L_w.x(\lambda \Sigma. f(\Sigma, \psi_1, \ldots, \psi_n)) \]

where \( w.w' \) is the concatenation of finite strings \( w \) and \( w' \), and \( L_w.x(\lambda R.\psi(R)) \) is short for \( \{s \mid w, x \models \psi(\bar{R}, s(\bar{R}) \Rightarrow \psi(\bar{R})) \} \).

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 [9], which is not a complete partial order.
This same encoding is also used for a cyclic variable.

which represents an arbitrary index into a string represented by

the form

index into a string. Note that any numerical expression of

any numerical expression of

position-set

When encoding instructions, PISA accounts for indices following the rules in Figure 9, where a predicate \( \text{prog}_a \) 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 \( \text{pos}_{s\gamma}(p, S) \), which represents an arbitrary index into a string represented by \( S \).

With index sensitivity at its disposal, PISA can model string operations such as indexOf and substring, where indexOf \( f(s, a, b) \) returns the first index in \( s \) at which \( b \) occurs, whereas substring \( s(m, n) \) extract from \( s \) the substring ranging between indices \( n \) and \( m \). These methods are abstracted as shown in Figure 7. The first and second parameters in the \( \text{indexOf} \) formula represent a position and a string containing it, respectively, while \( V \) is the set of concrete values possibly assigned to \( s \) in \( \text{indexOf}(s, a, b) \). Intuitively, \( \text{indexOf}(p, Q) \) holds if \( Q \) is a substring of \( P \) starting at the index represented by \( (p, P) \). By requiring \( p \leq p' \), we choose the minimal index among the candidates that satisfy \( \text{indexOf}(p, Q) \). In addition, due to the restriction about the minimal index, we need a set \( V \) of concrete string values possibly assigned to \( s \) of \( \text{indexOf}(s, a, b) \). It should be obtained by another analysis as in the case of \( \text{replace} \).

Otherwise, \( \text{indexOf} \) involves the same problem as \( \text{replace} \).

As an example, consider the following SSA program fragment:

\[
\begin{align*}
\forall v = 0; \\
v_1 &= "a\&b"; \\
v_2 &= "c"; \\
v_3 &= \text{indexOf}(v_1, v_2); \\
v_4 &= \text{substring}(v_1, v_0, v_3);
\end{align*}
\]

We obtain the following set of predicates after expanding the definitions of \( \text{indexOf} \) and \( \text{substring} \):

\[
\begin{align*}
\forall \text{prog}_a(n, S) \equiv \text{pos}_{s\gamma}(n, S) \\
\forall \text{prog}_a(R) \equiv "a\&b"(R) \\
\forall \text{prog}_a(p, S) \equiv \text{pos}_{s\gamma}(p, S) \\
\forall \text{prog}_a(p, S) \equiv \text{pos}_{s\gamma}(p, S) \land \min(P) \leq \min(p) \\
\end{align*}
\]

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.

4. 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:

\[
\begin{align*}
\forall \text{pos}_{s\gamma}(p, S) \equiv (p = \min(S)) \cdot \cdots \cdot \text{pos}_{s\gamma}(p, S) \equiv \text{pos}_{s\gamma}(p, S) \land \min(S), \\
\end{align*}
\]

where \( \text{pos}_{s\gamma}(p, S) \) means that position \( p \) in position-set \( S \) represents index \( n \) 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 \( f(s, a, b) \) returns the first index in \( s \) at which \( b \) occurs, whereas substring \( s(m, n) \) extract from \( s \) the substring ranging between indices \( n \) and \( m \). These methods are abstracted as shown in Figure 7. The first and second parameters in the \( \text{indexOf} \) formula represent a position and a string containing it, respectively, while \( V \) is the set of concrete values possibly assigned to \( s \) in \( \text{indexOf}(s, a, b) \). Intuitively, \( \text{indexOf}(p, Q) \) holds if \( Q \) is a substring of \( P \) starting at the index represented by \( (p, P) \). By requiring \( p \leq p' \), we choose the minimal index among the candidates that satisfy \( \text{indexOf}(p, Q) \). In addition, due to the restriction about the minimal index, we need a set \( V \) of concrete string values possibly assigned to \( s \) of \( \text{indexOf}(s, a, b) \). It should be obtained by another analysis as in the case of \( \text{replace} \).

Otherwise, \( \text{indexOf} \) involves the same problem as \( \text{replace} \).

As an example, consider the following SSA program fragment:

\[
\begin{align*}
\forall v = 0; \\
v_1 &= "a\&b"; \\
v_2 &= "c"; \\
v_3 &= \text{indexOf}(v_1, v_2); \\
v_4 &= \text{substring}(v_1, v_0, v_3);
\end{align*}
\]

We obtain the following set of predicates after expanding the definitions of \( \text{indexOf} \) and \( \text{substring} \):

\[
\begin{align*}
\forall \text{prog}_a(n, S) \equiv \text{pos}_{s\gamma}(n, S) \\
\forall \text{prog}_a(R) \equiv "a\&b"(R) \\
\forall \text{prog}_a(p, S) \equiv \text{pos}_{s\gamma}(p, S) \\
\forall \text{prog}_a(p, S) \equiv \text{pos}_{s\gamma}(p, S) \land \min(P) \leq \min(p) \\
\end{align*}
\]

4If 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.
void f1(String v1) { String s1 = f3(v1); } 
void f2(String v2) { String s2 = f3(v2); } 
String f3(String v3) { return v3; } 

Figure 12: Sample Java Program for Interprocedural Analysis

Figure 10 uses \( C(v, b, b') \) to encode \( \phi \) instructions. The variables used by the \( \phi \) 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 \( C(x, b, b') \), which is equivalent to \( \bigvee_{b \in \text{pred}(b')} C(x, b, b') \), where \( \text{pred}(b') \) represents a set of immediate predecessors of basic block \( b' \).

With path-sensitivity, we can abstract \( \text{contains} \), as described in Figure 7. In \( \{\text{contains}\} \), \( S \) is the set of concrete strings assigned to \( v \). Note that \( \neg (3P . \text{proj} _{R} . \lambda R . \text{false} + \text{subj} (P, R)) \) cannot be used when \( c = \text{false} \) since the analysis is conservative. As an example, consider the following Java method, along with its corresponding SSA representation, where \( \text{op} \) (or \( \text{op} \) (neg)) represent logical disjunction and negation, respectively:

```java
String clean(String s) {
    if (s.contains("<") || s.contains(">") { s = "x";
    b1 = contains(s,v1); s = "x";
    z0 = op(or)(b1,b2);
    z1 = op(neg)(z0);
    return s;
}
    jump z1,3;
}
```

For this method, we obtain the path conditions: \( PC'(1,1,2) = b1b2b2. PC'(1,1,3) = b1b2b2 \), and \( PC'(1,2,3) = b1b2b2 \). The resulting predicates are:

\[
C(v4,2,3) = C'(true, v4, PC'(1,2,3)) = \lambda R . \text{true}
\]

\[
C(s,1,3) = C'(\text{true}, s, PC'(1,1,3)) = \lambda R . \text{false} \land \text{proj} _{R} . \lambda R . \text{false} \lor \text{proj} _{R} . \lambda R . \text{true}
\]

Note that this approach simply ignores the call stack. Therefore, context-sensitivity of the call graph affects the precision of the string analysis.

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, and we then present the implementation of that algorithm. We then discuss two sets of evaluations: In the first set, we investigate how many user-defined sanitizers in the application code of 8 open-source benchmarks PISA was able to detect. In this first set of evaluations, we used two variants of PISA, and we compared PISA to alternative algorithms. In the second set, we examine the overall impact of PISA on a production-level taint-analysis algorithm.

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 13 shows the algorithm of the second step. This phase consumes the set \( P \) of candidate methods and the set \( \mathcal{P} \) 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. \text{callgraphOf}(M) builds the call graph \( cg \) rooted at the set \( M \) of sanitizer candidates. Any call-graph-construction algorithm can be chosen to build \( cg \) (e.g., context insensitive or context sensitive, with various levels of context sensitivity [14, 13]).
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 \text{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.
4. For each unsafe pattern \( p \), return variables of node \( n \) are extracted by \text{returnVariablesOf}(n), and then verified by the string analysis, \text{doStringAnalysis}(I,V,p), where the string analysis returns \text{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 \text{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 [31], which calls MONA [16] 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 “(long|strin|g|val|ue)”.  

• 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 completed 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, $S$, 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 1. The Candidates column reflects the number of methods in application code (strings representing relative paths), respectively. The sanitizer detection reported no false positives. Note that 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, Table 2 shows only true positives and false negatives, since the sanitizer detection reported no false positives. Note that 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 1: Statistics on benchmark applications

<table>
<thead>
<tr>
<th>App. name</th>
<th>Version</th>
<th>Classes</th>
<th>LOC</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>1.08</td>
<td>143</td>
<td>5,541</td>
<td>15</td>
</tr>
<tr>
<td>Bluslion</td>
<td>3.1</td>
<td>255</td>
<td>15,965</td>
<td>51</td>
</tr>
<tr>
<td>PersonalBlog</td>
<td>1.2.6</td>
<td>69</td>
<td>5,317</td>
<td>7</td>
</tr>
<tr>
<td>Roller</td>
<td>0.9.9</td>
<td>283</td>
<td>41,859</td>
<td>86</td>
</tr>
<tr>
<td>SnipSnap</td>
<td>1.0-RETA-1</td>
<td>614</td>
<td>46,962</td>
<td>52</td>
</tr>
<tr>
<td>Weboat</td>
<td>5.1</td>
<td>193</td>
<td>33,906</td>
<td>43</td>
</tr>
<tr>
<td>JSPWiki</td>
<td>2.6</td>
<td>503</td>
<td>81,301</td>
<td>90</td>
</tr>
<tr>
<td>MVNForum</td>
<td>1.2.7</td>
<td>820</td>
<td>142,954</td>
<td>56</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>571</td>
</tr>
</tbody>
</table>

Table 2: Accuracy results of the sanitizer-detection experiment

<table>
<thead>
<tr>
<th>App. name</th>
<th>Version</th>
<th>XSS</th>
<th>TP</th>
<th>FP</th>
<th>TP</th>
<th>Score</th>
<th>XSS</th>
<th>FP</th>
<th>TP</th>
<th>FP</th>
<th>Total</th>
<th>Score</th>
<th>Total</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PISA/CI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PISA/CS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSA/CI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSA/CS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>876</td>
<td>100</td>
<td>63</td>
<td>876</td>
<td>100</td>
<td>876</td>
<td>100</td>
<td>876</td>
<td>100</td>
<td></td>
<td>100</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

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 call graph 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 call graph 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 inputting invariants. This computation is done by iterating the grammar transduction and by using the character-set approximation [8, 23] as a widening operation.
Figure 14: Sanitizers detected by PISA

<table>
<thead>
<tr>
<th>Table 3: Running Times and MONA Statistics</th>
<th>Methods</th>
<th>Aborted</th>
<th>Executed</th>
<th>Total</th>
<th>Average</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>PISA/CS</td>
<td>0.00</td>
<td>0.1</td>
<td>0.3</td>
<td>0.4</td>
<td>679.1</td>
<td>1,907</td>
<td>679.1</td>
</tr>
<tr>
<td>PISA/CI</td>
<td>0.00</td>
<td>2.0</td>
<td>6.7</td>
<td>8.7</td>
<td>187</td>
<td>1,406</td>
<td>187</td>
</tr>
<tr>
<td>PISA/CS</td>
<td>0.00</td>
<td>2.0</td>
<td>6.8</td>
<td>8.8</td>
<td>228</td>
<td>1,278</td>
<td>228</td>
</tr>
<tr>
<td>Rigon/C</td>
<td>0.00</td>
<td>2.0</td>
<td>6.7</td>
<td>8.7</td>
<td>228</td>
<td>1,278</td>
<td>228</td>
</tr>
<tr>
<td>Rigon/C</td>
<td>0.00</td>
<td>2.0</td>
<td>6.8</td>
<td>8.8</td>
<td>228</td>
<td>1,277</td>
<td>228</td>
</tr>
<tr>
<td>SnipSnap</td>
<td>0.00</td>
<td>0.9</td>
<td>2.0</td>
<td>2.9</td>
<td>106</td>
<td>630</td>
<td>106</td>
</tr>
<tr>
<td>SnipSnap</td>
<td>0.00</td>
<td>0.7</td>
<td>2.0</td>
<td>2.7</td>
<td>213</td>
<td>1,146</td>
<td>213</td>
</tr>
<tr>
<td>SnipSnap</td>
<td>0.00</td>
<td>0.7</td>
<td>2.0</td>
<td>2.7</td>
<td>213</td>
<td>1,146</td>
<td>213</td>
</tr>
<tr>
<td>Webgoat</td>
<td>0.00</td>
<td>2.5</td>
<td>5.6</td>
<td>8.1</td>
<td>119</td>
<td>1,264</td>
<td>119</td>
</tr>
<tr>
<td>Webgoat</td>
<td>0.00</td>
<td>2.5</td>
<td>5.7</td>
<td>8.2</td>
<td>119</td>
<td>1,264</td>
<td>119</td>
</tr>
<tr>
<td>Webgoat</td>
<td>0.00</td>
<td>2.5</td>
<td>5.7</td>
<td>8.2</td>
<td>119</td>
<td>1,264</td>
<td>119</td>
</tr>
<tr>
<td>Webgoat</td>
<td>0.00</td>
<td>2.6</td>
<td>5.8</td>
<td>8.4</td>
<td>119</td>
<td>1,264</td>
<td>119</td>
</tr>
<tr>
<td>Webgoat</td>
<td>0.00</td>
<td>2.6</td>
<td>5.8</td>
<td>8.4</td>
<td>119</td>
<td>1,264</td>
<td>119</td>
</tr>
<tr>
<td>Webgoat</td>
<td>0.00</td>
<td>2.6</td>
<td>5.8</td>
<td>8.4</td>
<td>119</td>
<td>1,264</td>
<td>119</td>
</tr>
</tbody>
</table>

private static String getFileName(String s) { 
    String fileName = new File(s).getName();
    if (fileName.indexOf("/")) return fileName;
    return fileName.substring(fileName.lastIndexOfIndexOf("/"));
}

public String getWebParameter(String name) { 
    return getRegexParameter(name, "[^\d\s-]+"STRING);
}

public String getPhoneParameter(String name) { 
    return getRegexParameter(name, "[^\d]+"STRING);
}

private String getRegexParameter(String name, String regexp) { 
    String param = getParameter(name, regexp);
    if (Pattern.matches(str,regexp)) return param;
    else return "";
}

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 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 PISA/CS.** In Blojsom, SnipSnap, and Webgoat, PISA/CI detected 4 sanitizers for PATH vulnerabilities that were not detected by PISA/CI, compared to 6 sanitizers for PATH vulnerability that were not detected by the CFG-based string analyzer but detected by PISA/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 getWebParameter of Figure 14 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 calligraph 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.
final String entities[] = {"<", ">"};
final String refs[] = {"1lt","1gt");
String cleanByLoop(String str) {
    for (int i = 0; i < entities.length; i++)
        s = s.replaceAll(entities[i], refs[i]);
    return s;
}
String removeNonLetter(String str) {
    String cleanByLoop(String s) {
        char[] cs = str.toCharArray();
        for (int i = 0; i < cs.length; i++)
            if (Character.isLetter(cs[i]))
                ret = ret + cs[i];
        return ret;
    }
    for (int i = 0; i < entities.length; i++)
        s = s.replaceAll(entities[i], refs[i]);
    return s;
}

Figure 15: Sanitizers not detected by PISA

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

- The method cleanByLoop on Figure 15 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 methods that were not detected as sanitizers: ParameterParser.htmlEncode, Screen.convertWe
tachers, 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 15, 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.

- 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. (E.g., Macros.escapeHTML in Roller).

- Both PISA and the CFG-based string analyzer did not have a complete set of abstractions for built-in string operations (E.g.: Macros.escapeHTML in Roller).

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 augmented with a backward analysis [34]. Kiežun, ton package to implement these operations, and their approach was modeled by the standard operations and a newly introduced operation on automata. They also used the MONA’s automa-

There are a few papers that combine string analysis with sym-

7. RELATED WORK

Many string-analysis algorithms have been presented to date. Java String Analyzer (JSA) was first introduced by Christensen, et al. [8]. JSA approximates a string value by a regular language, al-

Table 4: Results of the taint-analysis-integration experiment

<table>
<thead>
<tr>
<th></th>
<th>w/o PISA</th>
<th>w/ PISA</th>
<th>w/o predefined sanitizers</th>
<th>w/ predefined sanitizers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Configuration A</td>
<td>Configuration B</td>
<td>Configuration C</td>
<td>Configuration D</td>
</tr>
<tr>
<td>XSS</td>
<td>50</td>
<td>70</td>
<td>50</td>
<td>70</td>
</tr>
<tr>
<td>MVNForum</td>
<td>90</td>
<td>95</td>
<td>85</td>
<td>90</td>
</tr>
<tr>
<td>LOG</td>
<td>70</td>
<td>75</td>
<td>70</td>
<td>75</td>
</tr>
<tr>
<td>LOG</td>
<td>50</td>
<td>55</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>LOG</td>
<td>30</td>
<td>35</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>LOG</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

a team of security experts), and (c) using the most complete speci-

affect the overall precision of the security tool considerably and in a

fully automated fashion, without requiring any time or expertise

in particular, PISA can account for close to 40% of the benefit from the considerable effort invested by a

security team devising a manual specification.

<table>
<thead>
<tr>
<th></th>
<th>w/o PISA</th>
<th>w/ PISA</th>
<th>w/o predefined sanitizers</th>
<th>w/ predefined sanitizers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Configuration A</td>
<td>Configuration B</td>
<td>Configuration C</td>
<td>Configuration D</td>
</tr>
<tr>
<td>XSS</td>
<td>50</td>
<td>70</td>
<td>50</td>
<td>70</td>
</tr>
<tr>
<td>MVNForum</td>
<td>90</td>
<td>95</td>
<td>85</td>
<td>90</td>
</tr>
<tr>
<td>LOG</td>
<td>70</td>
<td>75</td>
<td>70</td>
<td>75</td>
</tr>
<tr>
<td>LOG</td>
<td>50</td>
<td>55</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>LOG</td>
<td>30</td>
<td>35</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>LOG</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
symbolic-execution engine Pex [28], while handling the same kind of index sensitivity. In this work, the path constraints are checked by the SMT solver Z3 [36], which is also used by Rex [30]. Saxena, et al. [25] 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. [26] proposed to treat methods like indexOf in their symbolic execution engine by modeling conversion between symbolic strings and symbolic integers. Com-

ods like indexOf checks for this work, and the ISSTA reviewers for their helpful com-

Tabuchi for his help on our earlier work, the developers of MONA, We thank Anders Møller and Yinnon Haviv for their invaluable

vs.

approach shows it to compare favorably to the CFG-based string

of Web applications, where a robust procedure for detecting and

contribute to finding application-specific sanitizers.

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) [3], or sym-

bolic representations of automata [30].

8. CONCLUSION

In this paper, we presented a novel string-analysis technique, which enables unprecedented precision when modeling string op-

erations, 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 [12] (discovering 128 vs. 71 sanitizers), and have a sig-

nificant 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 ISSTA reviewers for their helpful com-

ments.

9. REFERENCES


[31] T. J. Watson Libraries for Analysis, wala.3.net/.


**APPENDIX**

### A. TARGET LANGUAGE

Figure 16 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 $\text{bool}$. The guard condition $e$ represents consists of Boolean values, the program variables, and Boolean operators. The meta-variable $B$ is a basic block which is determined by $b$ and contains the sequence of instructions $I$, where we denote an empty sequence by $\epsilon$, 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 17. $\sigma \vdash (b, b', \vec{s})$ denotes a program state, where $\sigma$ 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 $I$ is a sequence of instructions. In addition, we denote the transitive closure of $\rightarrow$ by $\rightarrow^*$.

### B. PROOF OF SOUNDNESS

The proof is done by induction on the rules of the operational semantics with the following invariant $\mathsf{inv}$ on the program states $\sigma$:

\[
\mathsf{inv}(\sigma) \equiv \exists w, I. \mathsf{inv}'(\sigma, w, I),
\]

where $\mathsf{inv}'(\sigma, w, I) \equiv \forall x \in \mathsf{dom}(\sigma). (\sigma(x) \in L_w, I(\mathsf{prog}_s))$.

Note that the formula is exactly equivalent to Theorem 1 if we expand the definition of $\mathsf{inv}'$.

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