Abstract Interpretation Framework

- Associate each string variable at each program point with an automaton that accepts an over approximation of its possible values.
- Use these automata to perform symbolic executions on string variables.
- Iteratively
  - Compute the next state of current automata against string operations and
  - Update automata by joining the result to the automata at the next statement
- Terminate the execution upon reaching a fixed point.
Challenges

- **Precision**: Need to deal with sanitization routines having decent PHP functions, e.g., `ereg` replacement.

- **Complexity**: Need to face the fact that the problem itself is undecidable. The fixed point may not exist and even if it exists the computation itself may not converge.

- **Performance**: Need to perform efficient automata manipulations in terms of both time and memory.
Features of Our Approach

We propose:

- A Language-based Replacement: to model decent string operations in PHP programs.
- An Automata Widening Operator: to accelerate fixed point computation.
- A Symbolic Encoding: using Multi-terminal Binary Decision Diagrams (MBDDs) from MONA DFA packages.
A Language-based Replacement

\[ M = \text{REPLACE}(M_1, M_2, M_3) \]

- \( M_1, M_2, \) and \( M_3 \) are DFAs.
  - \( M_1 \) accepts the set of original strings,
  - \( M_2 \) accepts the set of match strings, and
  - \( M_3 \) accepts the set of replacement strings
- Let \( s \in L(M_1), x \in L(M_2), \) and \( c \in L(M_3) \):
  - Replaces all parts of any \( s \) that match any \( x \) with any \( c \).
  - Outputs a DFA that accepts the result to \( M \).
Some examples:

<table>
<thead>
<tr>
<th>$L(M_1)$</th>
<th>$L(M_2)$</th>
<th>$L(M_3)$</th>
<th>$L(M)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ baaabaa}</td>
<td>{aa}</td>
<td>{c}</td>
<td>{bacbc, bcabc}</td>
</tr>
<tr>
<td>{ baaabaa}</td>
<td>$a^+$</td>
<td>$\varepsilon$</td>
<td>{bb}</td>
</tr>
<tr>
<td>{ baaabaa}</td>
<td>$a^+ b$</td>
<td>{c}</td>
<td>{bcaa}</td>
</tr>
<tr>
<td>{ baaabaa}</td>
<td>$a^+$</td>
<td>{c}</td>
<td>{bccb, bccbc, bccbcc, bccbcc, bccbcc}</td>
</tr>
<tr>
<td>$ba^+ b$</td>
<td>$a^+$</td>
<td>{c}</td>
<td>$bc^+ b$</td>
</tr>
</tbody>
</table>
An over approximation with respect to the leftmost/longest(first) constraints

Many string functions in PHP can be converted to this form:
- htmlspecialchars, tolower, toupper, str_replace, trim, and
- preg_replace and ereg_replace that have regular expressions as their arguments.
Implementation of $\text{REPLACE}(M_1, M_2, M_3)$:

- Mark matching sub-strings
  - Insert marks to $M_1$
  - Insert marks to $M_2$
- Replace matching sub-strings
  - Identify marked paths
  - Insert replacement automata

In the following, we use two marks: $<$ and $>$ (not in $\Sigma$), and a duplicate set of alphabet: $\Sigma' = \{\alpha' | \alpha \in \Sigma\}$. We use an example to illustrate our approach.
An Example

Construct $M = \text{REPLACE}(M_1, M_2, M_3)$.

- $L(M_1) = \{baab\}$
- $L(M_2) = a^+ = \{a, aa, aaa, \ldots\}$
- $L(M_3) = \{c\}$
Step 1

Construct $M'_1$ from $M_1$:

- Duplicate $M_1$ using $\Sigma'$
- Connect the original and duplicated states with $<$ and $>$

For instance, $M'_1$ accepts $b < a'a' > b$, $b < a' > ab$. 

![Diagram of automaton]
Step 2

Construct $M'_2$ from $M_2$:

- Construct $M_2$ that accepts strings do not contain any substring in $L(M_2)$. (a)
- Duplicate $M_2$ using $\Sigma'$. (b)
- Connect (a) and (b) with marks. (c)

For instance, $M'_2$ accepts $b < a'a' > b$, $b < a' > bc < a' >$.

(a)  
(b)  
(c)
Intersect $M'_1$ and $M'_2$.

- The matched substrings are marked in $\Sigma'$.
- Identify $(s, s')$, so that $s \rightarrow < \ldots \rightarrow > s'$.

In the example, we identify three pairs: $(i, j)$, $(i, k)$, $(j, k)$.
Step 4

Construct $M$:

- Insert $M_3$ for each identified pair. (d)
- Determinize and minimize the result. (e)

$L(M) = \{bcb, bccb\}$. 

(d)  

(e)
The operator was originally proposed by Bartzis and Bultan [BB, CAV04]. Intuitively, we

- Identify equivalence classes, and
- Merge states in an equivalence class
State Equivalence

$q, q'$ are equivalent if one of the following condition holds:

- $\forall w \in \Sigma^*, w$ is accepted by $M$ from $q$ then $w$ is accepted by $M'$ from $q'$, and vice versa.
- $\exists w \in \Sigma^*, M$ reaches state $q$ and $M'$ reaches state $q'$ after consuming $w$ from its initial state respectively.
- $\exists q'', q$ and $q''$ are equivalent, and $q'$ and $q''$ are equivalent.
An Example for $M \sqcup M'$

- $L(M) = \{\epsilon, ab\}$ and $L(M') = \{\epsilon, ab, abab\}$.
- The set of equivalence classes: $C = \{q''_0, q''_1\}$, where $q''_0 = \{q_0, q'_0, q_2, q'_2, q'_4\}$ and $q''_1 = \{q_1, q'_1, q'_3\}$.

![Diagram](image-url)

Figure: Widening automata
Recall that we want to compute the least fixpoint that corresponds to the reachable values of string expressions.

- The fixpoint computation will compute a sequence $M_0, M_1, \ldots, M_i, \ldots$, where $M_0 = I$ and $M_i = M_{i-1} \cup \text{post}(M_{i-1})$. 
A Fixed Point Computation

Consider a simple example:

- Start from an empty string and concatenate $ab$ at each iteration.
- The exact computation sequence $M_0, M_1, ..., M_i, ...$ will never converge, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 1 \leq k \leq i\} \cup \{\epsilon\}$. 
Accelerate The Fixed Point Computation

Use the widening operator $\nabla$.

- Compute an over-approximate sequence instead: $M'_0$, $M'_1$, ..., $M'_i$, ...
- $M'_0 = M_0$, and for $i > 0$, $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup \text{post}(M'_{i-1}))$.

An over-approximate sequence for the simple example:

(a) $M'_0$
(b) $M'_1$
(c) $M'_2$
(d) $M'_3$