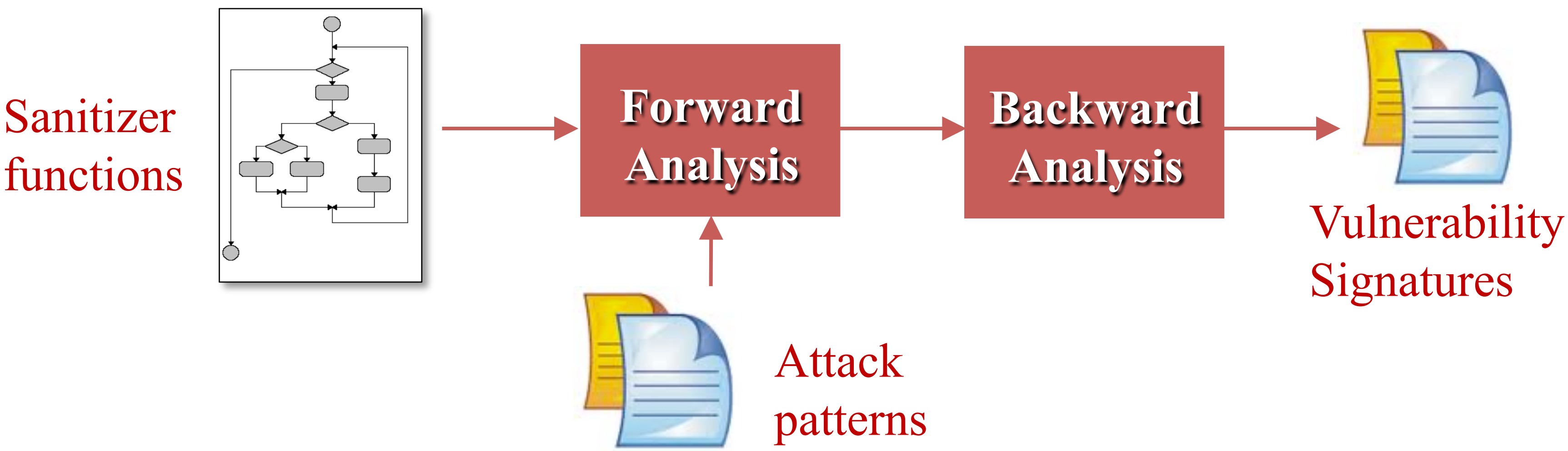
Automata Based String Analysis for Vulnerability Detection

Automata-based String Analysis

- Finite State Automata can be used to characterize sets of string values
- Automata based string analysis
  - Associate each string expression in the program with an automaton
  - The automaton accepts an over approximation of all possible values that the string expression can take during program execution
- Using this automata representation we symbolically execute the program, only paying attention to string manipulation operations

Forward & Backward Analyses

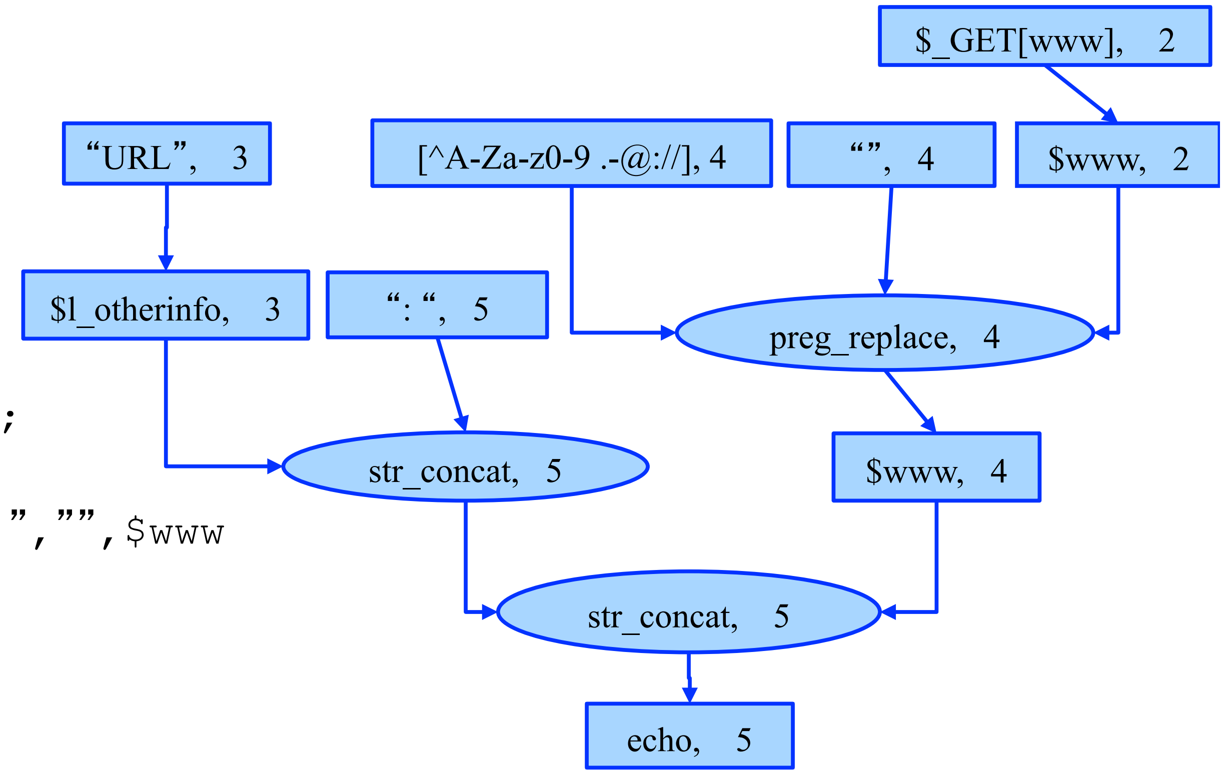
- First convert sanitizer functions to dependency graphs
- Combine symbolic forward and backward symbolic reachability analyses
- Forward analysis
  - Assume that the user input can be any string
  - Propagate this information on the dependency graph
  - When a sensitive function is reached, intersect with attack pattern
- Backward analysis
  - If the intersection is not empty, propagate the result backwards to identify which inputs can cause an attack



Dependency Graphs

Extract dependency graphs from sanitizer functions

```
1:<?php
2: $www = $ GET["www"];
3: $l_otherinfo = "URL";
4: $www = ereg_replace(
   "[^A-Za-z0-9 .-@://]","",$www
   );
5: echo $l_otherinfo .
   ": " .$www;
6:?>
```



Dependency Graph

Forward Analysis

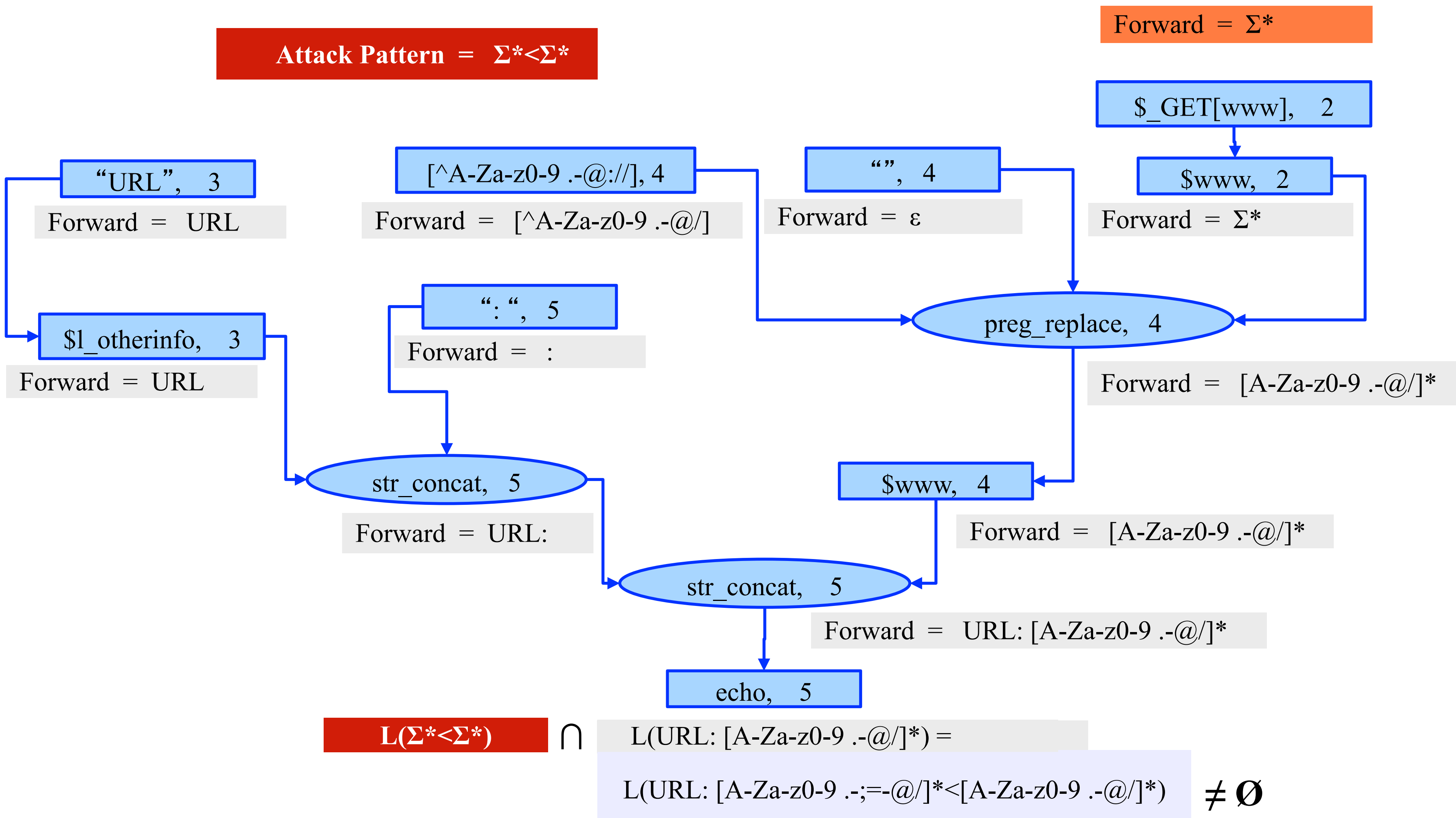
- Using the dependency graph conduct vulnerability analysis
- Automata-based forward symbolic analysis that identifies the possible values of each node
- Each node in the dependency graph is associated with a DFA
  - DFA accepts an over-approximation of the strings values that the string expression represented by that node can take at runtime
  - The DFAs for the input nodes accept Σ^*
- Intersecting the DFA for the sink nodes with the DFA for the attack pattern identifies the vulnerabilities

Forward Analysis

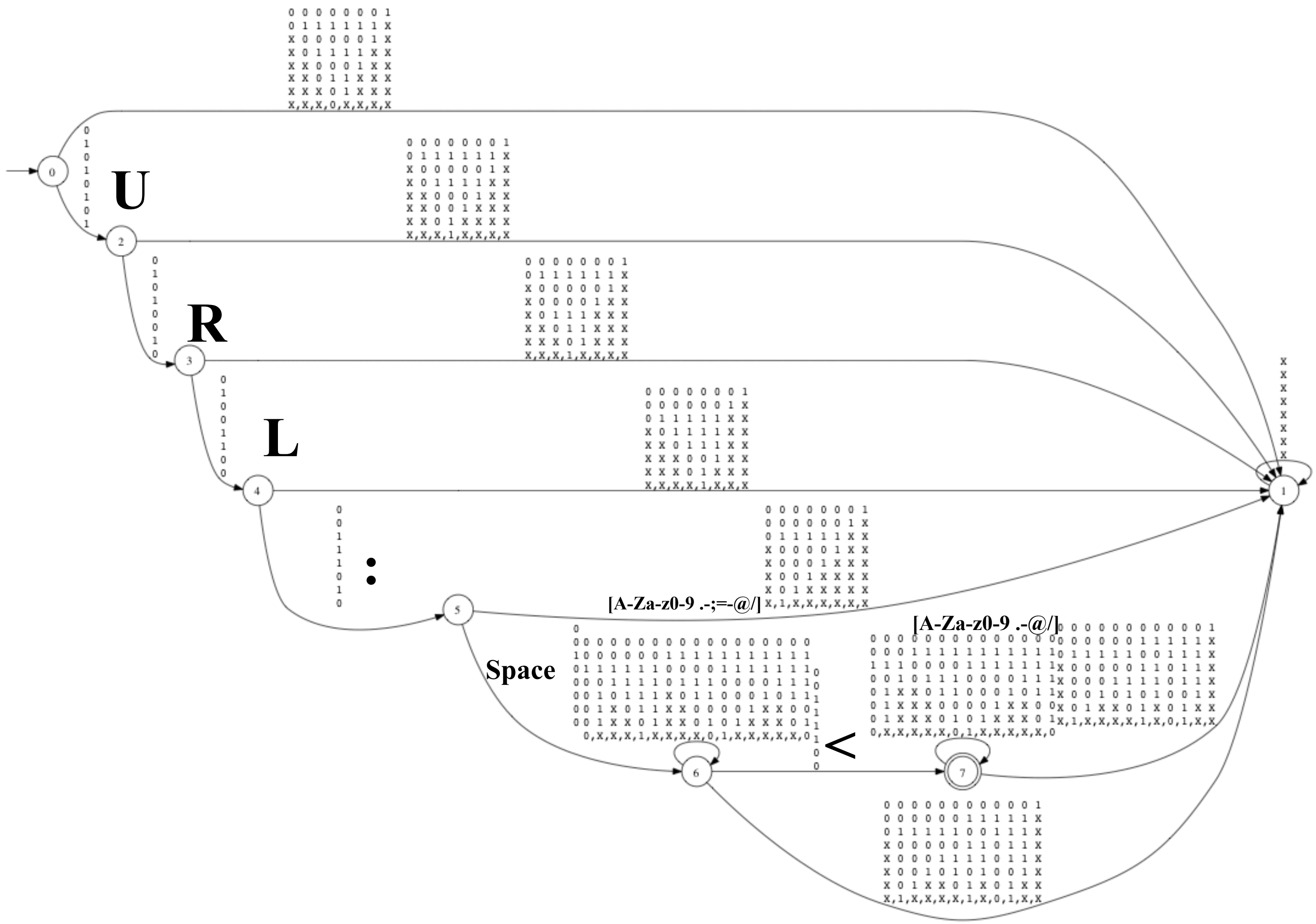
- Need to implement **post-image computations** for string operations:
  - **postConcat**(M1, M2)
    returns M, where M=M1.M2
  - **postReplace**(M1, M2, M3)
    returns M, where M=replace(M1, M2, M3)

- Need to handle many specialized string operations:
  - regmatch, substring, indexof, length, contains, trim, addslashes, htmlspecialchars, mysql_real_escape_string, tolower, toupper

Forward Analysis



Result Automaton



URL: [A-Za-z0-9 .-;=-@/]*<[A-Za-z0-9 .-@/]*

Automata Lattice

- Given an automaton A, let L(A) denote the set of string accepted by the automaton
- We use automata A to represent sets of string values in L(A)
- We can define partial order among automata based on the subset ordering among the languages they accept.
- If we have a program with a set of variables V and a set of statement labels L (assume that each statement is labeled), we can use $|L| \times |V|$ automata to represent value of each string variable at each program point.

Forward Reachability

Algorithm 5 FORWARDANALYSIS(L, F, V)

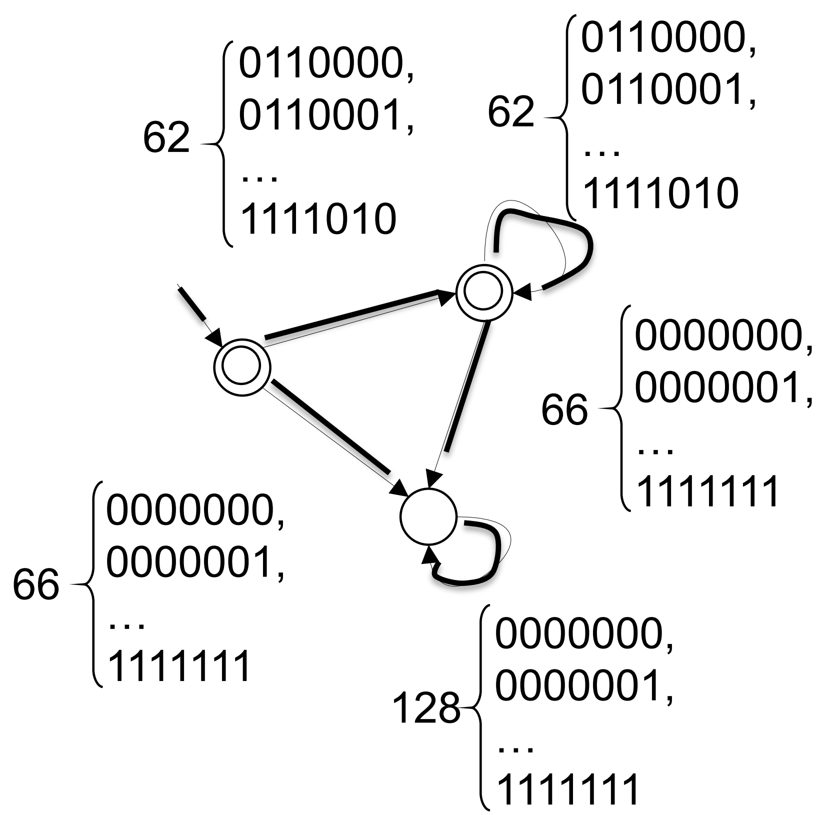
$I := \{l \mid \forall l'.(l', l) \notin F\}$;
for $l \in L \setminus I, v \in V$ **do**
    $\vec{A}[l, v] = A(\emptyset)$;
end for
for $l \in I, v \in V$ **do**
    $\vec{A}[l, v] = A_{init}(v)$;
end for
queue $WQ := NULL$;
WQ.enqueue(l_1);
while $WQ \neq NULL$ **do**
    $l := WQ$.dequeue();
    for $(l, l') \in F$ **do**
        if $\textsc{Post}(\vec{A}[l], (l, l')) \not\sqsubseteq \mathcal{L}(\vec{A}[l'])$ **then**
            $\vec{A}(l') = \vec{A}(l') \nabla (\vec{A}(l') \sqcup \textsc{Post}(\vec{A}(l), l))$;
            WQ.enqueue(l');
        end if
    end for
end while

Symbolic Automata Representation

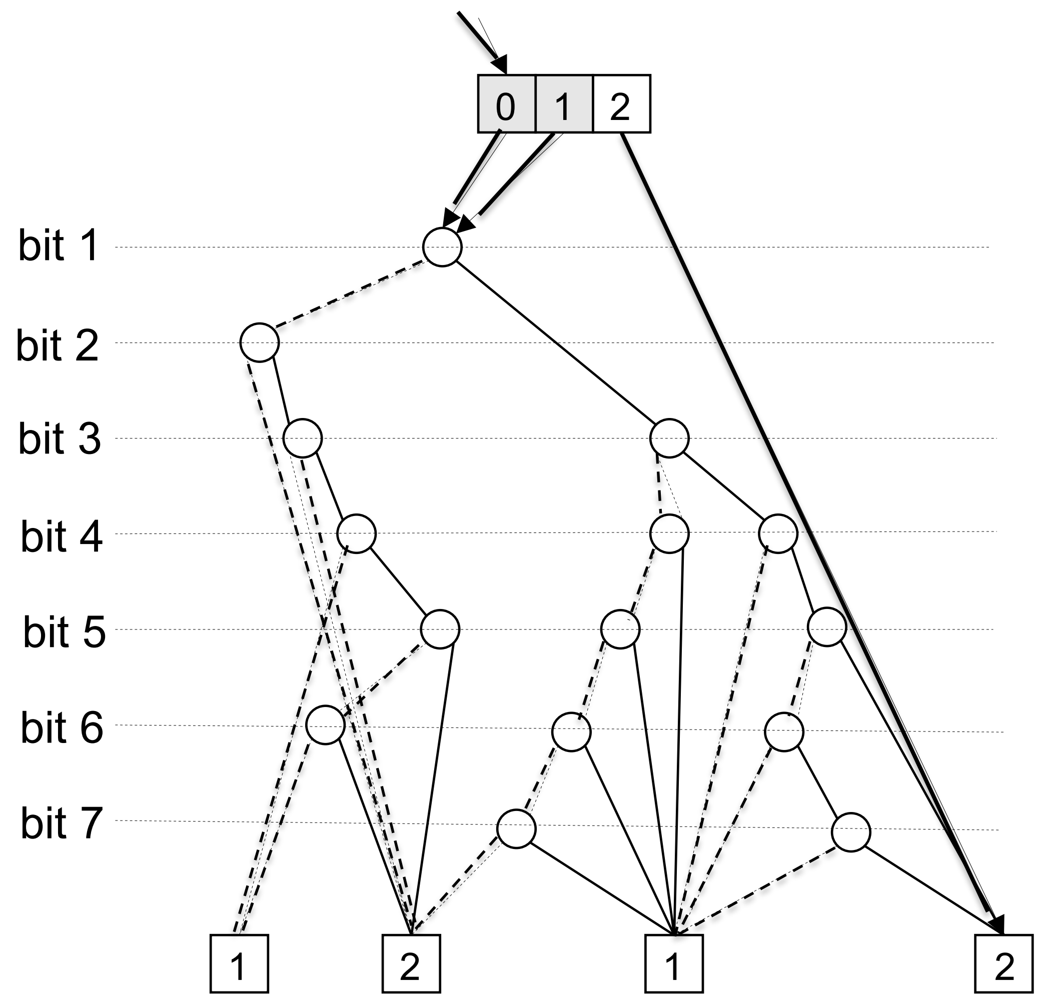
- MONA DFA Package for automata manipulation
  - [Klarlund and Møller, 2001]
- Compact Representation:
  - Canonical form and
  - Shared BDD nodes
- Efficient MBDD Manipulations:
  - Union, Intersection, and Emptiness Checking
  - Projection and Minimization
- Cannot Handle Nondeterminism:
  - Use dummy bits to encode nondeterminism

Symbolic Automata Representation

Explicit DFA representation

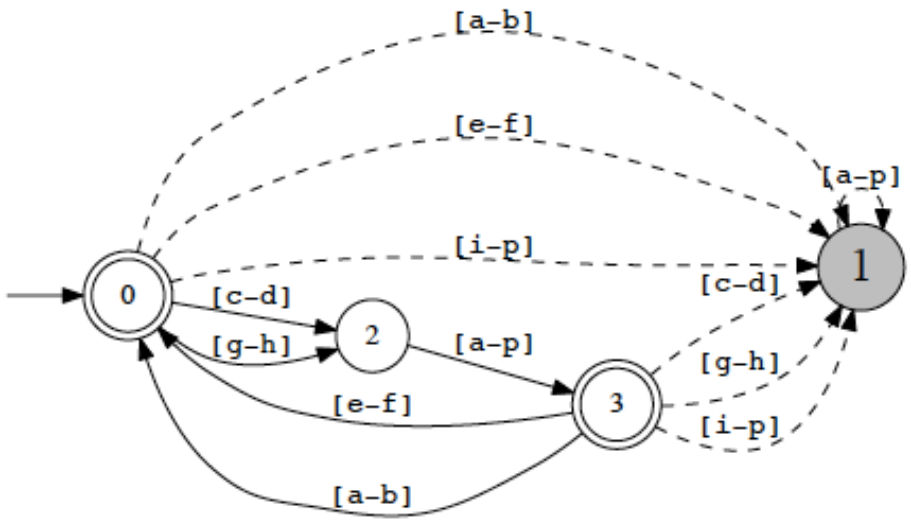


Symbolic DFA representation

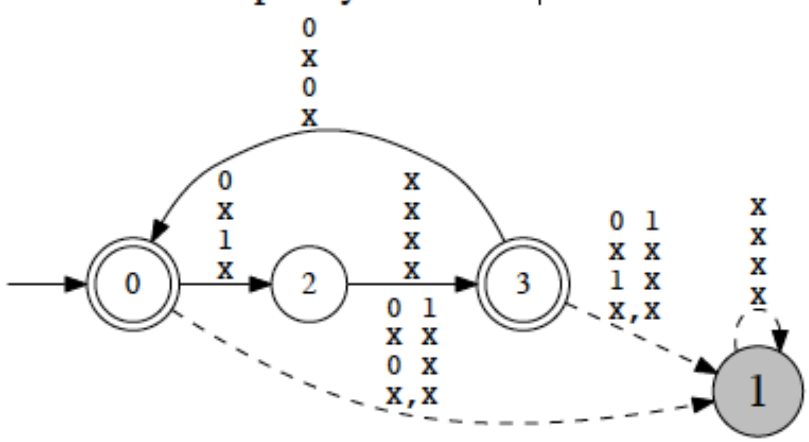


Symbolic Automata Representation

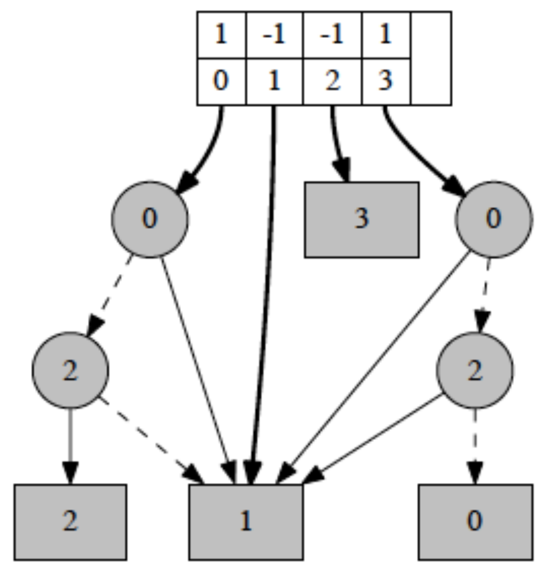
Sample Explicit DFA



Sample Symbolic DFA



Internal Representation of
Sample DFA as MBDD



Automata Widening

- String verification problem is undecidable

- The forward fixpoint computation is not guaranteed to converge in the presence of loops and recursion

- Compute a sound approximation
  - During fixpoint compute an over approximation of the least fixpoint that corresponds to the reachable states

- Use an automata based widening operation to over-approximate the fixpoint
  - Widening operation over-approximates the union operations and accelerates the convergence of the fixpoint computation

Automata Widening

Given a loop such as

```
1:<?php
2:  $var = “head”;
3:  while (...){
4:    $var = $var . “tail”;
5:  }
6:  echo $var
7:?>
```

Our forward analysis with widening would compute that the value of the variable $var in line 6 is (head)(tail)*

A widening operator

- Idea:
  - Instead of computing a sequence of automata

  $A_1, A_2, \ldots$ where $A_{i+1}=A_i \cup post(A_i)$,
  - compute

  $A'_1, A'_2, \ldots$ where $A'_{i+1}=A'_i \nabla (A'_i \cup post(A'_i))$

- By definition $A \cup B \subseteq A \nabla B$

- The goal is to find a widening operator ∇ such that:

1. The sequence $A'_1, A'_2, \ldots$ **converges**
2. It converges **fast**
3. The computed fixpoint is as close as possible to the **exact** set of reachable states

Backward Analysis

- A ***vulnerability signature*** is a characterization of all malicious inputs that can be used to generate attack strings
- Identify vulnerability signatures using an automata-based backward symbolic analysis starting from the sink node

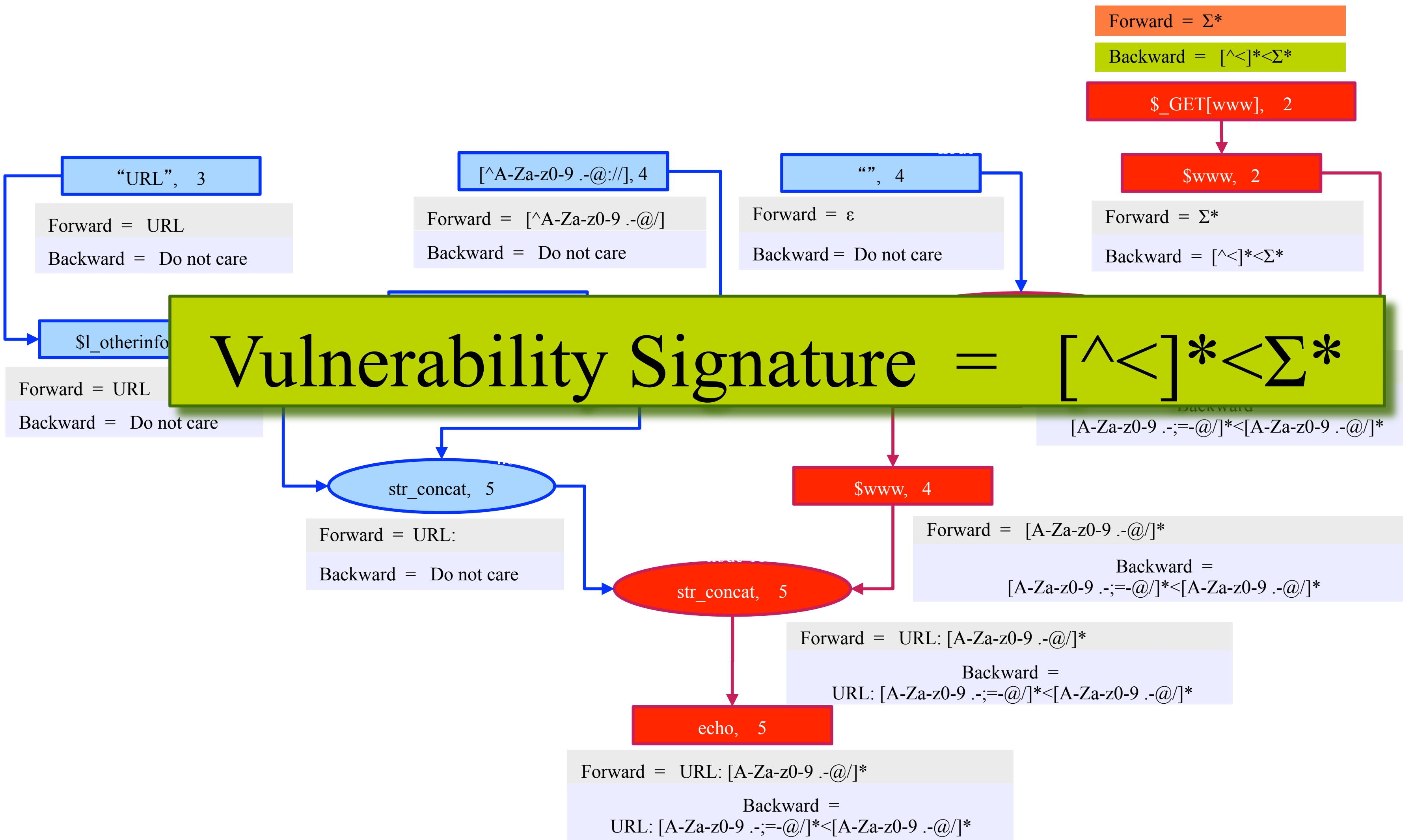
- Need to implement **Pre-image computations** on string operations:
  - **preConcatPrefix**(M, M2)

    returns M1 and where M = M1.M2
  - **preConcatSuffix**(M, M1)

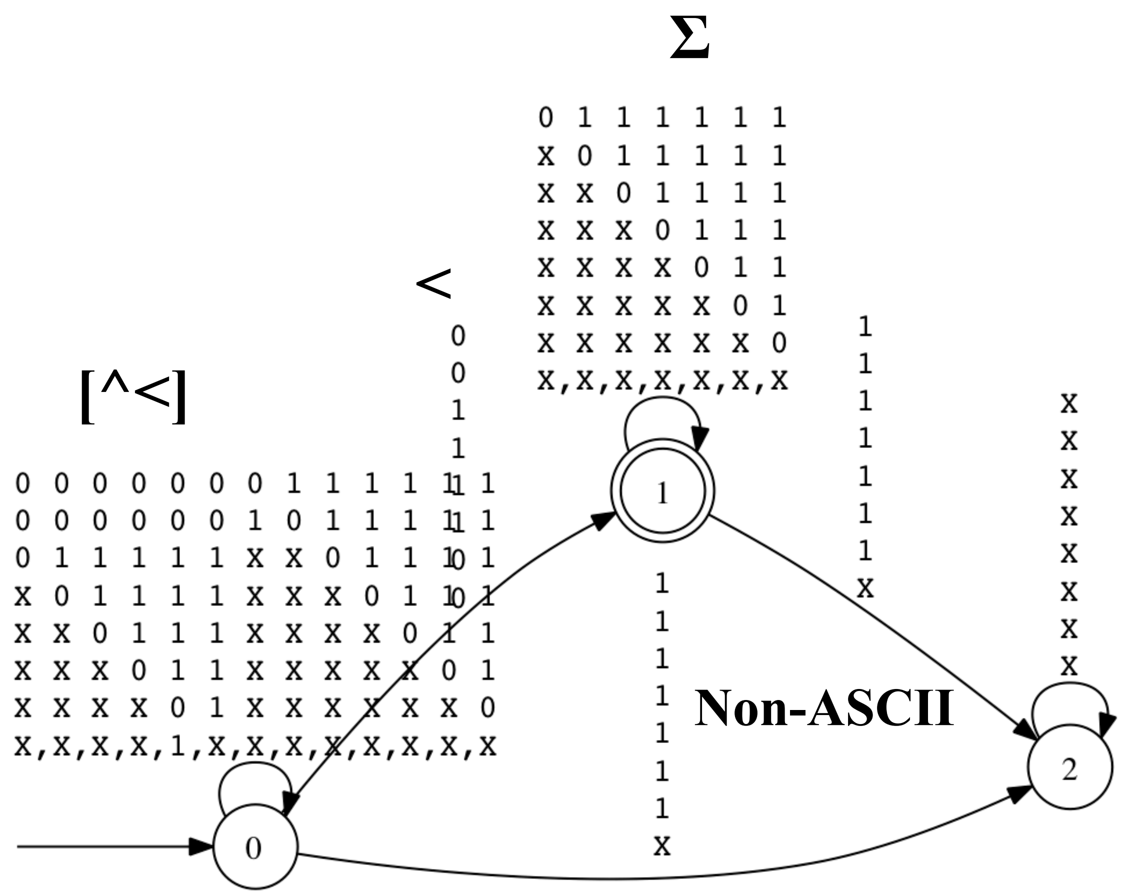
    returns M2, where M = M1.M2
  - **preReplace**(M, M2, M3)

    returns M1, where M=replace(M1, M2, M3)

Backward Analysis



Vulnerability Signature Automaton



[^<]*<Σ*

Backward Symbolic Reachability

Algorithm 6 BACKWARDANALYSIS(L, F, V)

$T := \{l \mid \forall l'.(l,l') \notin F\}$;
for $l \in L \setminus T, v \in V$ **do**
    $\vec{A}[l,v] = A(\emptyset)$;
end for
for $l \in T, v \in V$ **do**
    $\vec{A}[l,v] = A_{init}(v)$;
end for
queue $WQ := NULL$;
WQ.enqueue(l_t);
while do$WQ \neq NULL$
    $l := WQ$.dequeue();
    for $(l,l') \in F$ **do**
        if $\textsc{Pre}(\vec{A}[l],(l,l')) \not\sqsubseteq \mathcal{L}(\vec{A}[l'])$ **then**
            $\vec{A}(l') = \vec{A}(l')\nabla(\vec{A}(l') \sqcup \textsc{Pre}(\vec{A}(l),l))$;
            WQ.enqueue(l');
        end if
    end for
end while

Recap

Given an automata-based string analyzer:

•**Vulnerability Analysis:** We can do a forward analysis to detect all the strings that reach the sink and that match the attack pattern

- We can compute an automaton that accepts all such strings
- If there is any such string the application might be vulnerable to the type of attack specified by the attack pattern

•**Vulnerability Signature:** We can do a backward analysis to compute the vulnerability signature

- Vulnerability signature is the set of all input strings that can generate a string value at the sink that matches the attack pattern
- We can compute an automaton that accepts all such strings