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

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

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 $\Sigma^*$

- 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

Attack Pattern = Σ*<Σ*

L(Σ*<Σ*) ≠ ∅
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 FORWARD ANALYSIS ($L$, $F$, $V$)

1: $I := \{l \mid \forall l'. (l', l) \not\in F\}$;
2: for $l \in L \setminus I$, $v \in V$ do
3: \quad $\tilde{A}[l, v] = A(\emptyset)$;
4: end for
5: for $l \in I$, $v \in V$ do
6: \quad $\tilde{A}[l, v] = A_{init}(v)$;
7: end for
8: queue $WQ := NULL$;
9: $WQ.enqueue(l_1)$;
10: while $WQ \neq NULL$ do
11: \quad $l := WQ.dequeue()$;
12: \quad for $(l, l') \in F$ do
13: \quad \quad if POST($\tilde{A}[l], (l, l')) \not\subseteq \mathcal{L}(\tilde{A}[l'])$ then
14: \quad \quad \quad $\tilde{A}(l') = \tilde{A}(l') \triangledown (\tilde{A}(l') \sqcup \text{POST}(\tilde{A}(l), l))$;
15: \quad \quad end if
16: \quad end for
17: 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
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

```php
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 \( A_{i+1} = A_i \cup \text{post}(A_i) \),
  – compute \( A'_1, A'_2, \ldots \) where \( A'_{i+1} = A'_i \nabla (A'_i \cup \text{post}(A'_i)) \)

• By definition \( A \cup B \subseteq A \nabla B \)

• The goal is to find a widening operator \( \nabla \) 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:
  - \(\text{preConcatPrefix}(M, M2)\)
    returns \(M_1\) and where \(M = M_1.M_2\)
  - \(\text{preConcatSuffix}(M, M1)\)
    returns \(M_2\), where \(M = M_1.M_2\)
  - \(\text{preReplace}(M, M2, M3)\)
    returns \(M_1\), where \(M=\text{replace}(M1, M2, M3)\)
Backward Analysis

Vulnerability Signature = $^{[\wedge<]}*^{<\Sigma}$
Vulnerability Signature Automaton

\[
\Sigma
\]

\[
[<]<^*
\]

Non-ASCII

0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 1 0 1 1 1 1 1 1
0 1 1 1 1 1 x x 0 1 1 1 0 1
x 0 1 1 1 1 x x x 0 1 1 0 1
x x 0 1 1 1 x x x x 0 1 1 1
x x x 0 1 1 x x x x x x 0 1
x x x x 0 1 x x x x x x 0 1
x, x, x, x, 1, x, x, x, x, x, x, x
x, x, x, x

[^<]*<[^<]*
**Backward Symbolic Reachability**

Algorithm 6 \textsc{BackwardAnalysis}(L, F, V)

1. \( T := \{l \mid \forall l'. (l, l') \notin F\}; \)
2. \textbf{for} \( l \in L \setminus T, v \in V \) \textbf{do}
3. \( \tilde{A}[l, v] = A(\emptyset); \)
4. \textbf{end for}
5. \textbf{for} \( l \in T, v \in V \) \textbf{do}
6. \( \tilde{A}[l, v] = A_{init}(v); \)
7. \textbf{end for}
8. \textbf{queue} \( WQ := NULL; \)
9. \( WQ.enqueue(l_t); \)
10. \textbf{while} \( WQ \neq NULL \) \textbf{do}
11. \( l := WQ.dequeue(); \)
12. \textbf{for} \( (l, l') \in F \) \textbf{do}
13. \textbf{if} \( \text{PRE}(\tilde{A}[l], (l, l')) \nsubseteq \mathcal{L}(\tilde{A}[l']) \) \textbf{then}
14. \( \tilde{A}(l') = \tilde{A}(l') \nabla (\tilde{A}(l') \cup \text{PRE}(\tilde{A}(l), l)); \)
15. \( WQ.enqueue(l'); \)
16. \textbf{end if}
17. \textbf{end for}
18. \textbf{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