PLDI 2016 Tutorial
Automata-Based String Analysis

Tevfik Bultan, Abdulbaki Aydin, Lucas Bang
Verification Laboratory (VLab)
University of California, Santa Barbara, USA
bultan@cs.ucsb.edu, baki@cs.ucsb.edu, bang@cs.ucsb.edu

String Analysis @ UCSB VLab

- *Symbolic String Verification: An Automata-based Approach* [Yu et al., SPIN'08]
- *Symbolic String Verification: Combining String Analysis and Size Analysis* [Yu et al., TACAS'09]
- *Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses* [Yu et al., ASE'09]
- *Stranger: An Automata-based String Analysis Tool for PHP* [Yu et al., TACAS'10]
- *Relational String Verification Using Multi-Track Automata* [Yu et al., CIAA'10, IJFCS'11]
- *String Abstractions for String Verification* [Yu et al., SPIN'11]
- *Patching Vulnerabilities with Sanitization Synthesis* [Yu et al., ICSE'11]
- *Verifying Client-Side Input Validation Functions Using String Analysis* [Alkhalaf et al., ICSE'12]
- *ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies* [Alkhalaf et al., ISSTA'12]
- *Automata-Based Symbolic String Analysis for Vulnerability Detection* [Yu et al., FMSD'14]
- *Semantic Differential Repair for Input Validation and Sanitization* [Alkhalaf et al. ISSTA'14]
- *Automated Test Generation from Vulnerability Signatures* [Aydin et al., ICST'14]
- *Automata-based model counting for string constraints* [Aydin et al., CAV'15]

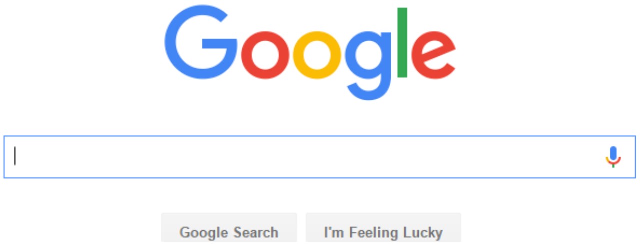
OUTLINE

- **Motivation**
- Symbolic string analysis
- Automated repair
- String constraint solving
- Model counting

Modern Software Applications

Common Usages of Strings

- **Input validation and sanitization**

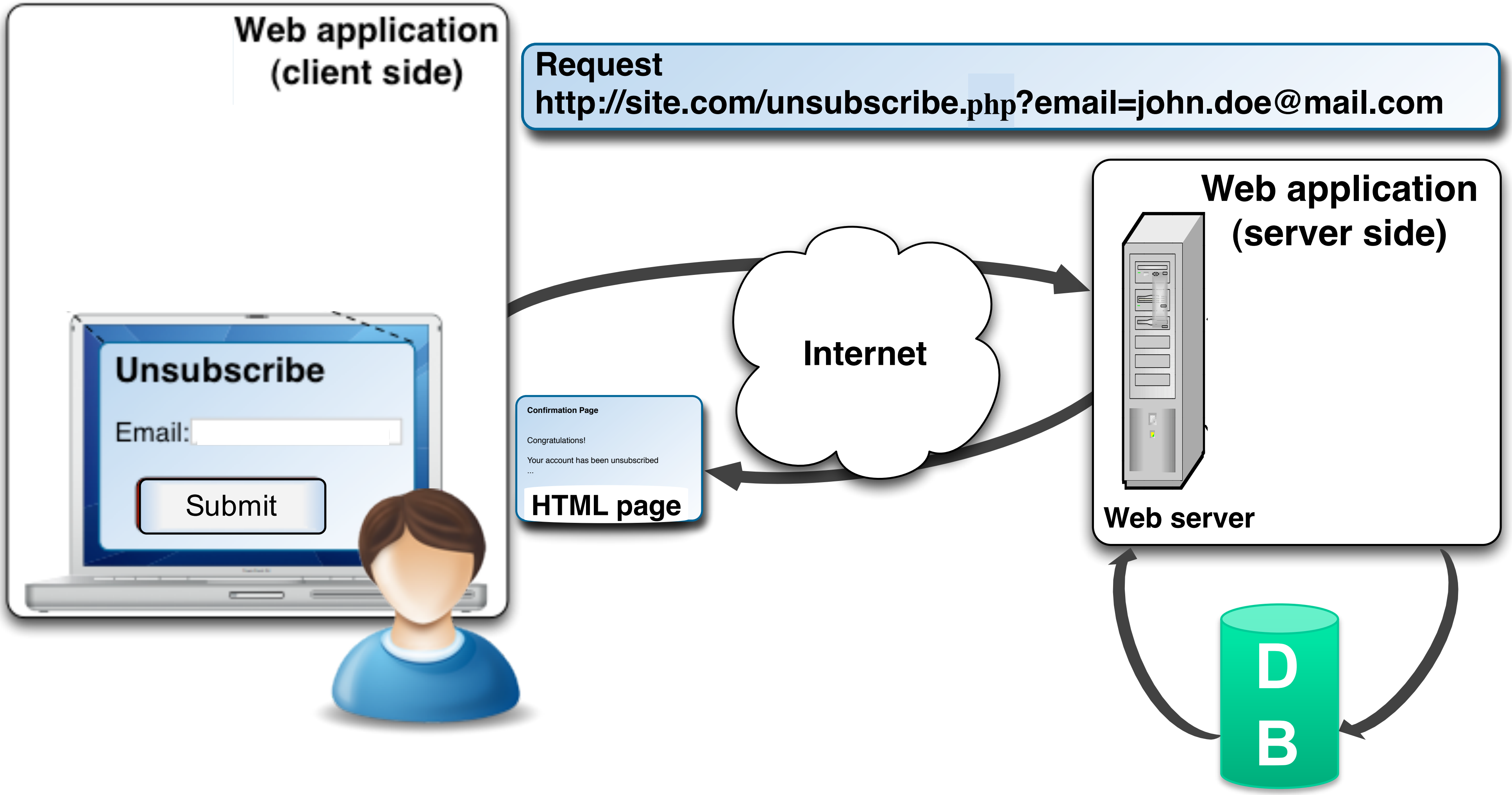


- Database query generation

- Formatted data generation

- Dynamic code generation

Anatomy of a Web Application



Web application (client side)

Unsubscribe

Email:

Submit

Request
http://site.com/unsubscribe.php?email=john.doe@mail.com

Internet

Confirmation Page

Congratulations!

Your account has been unsubscribed

...

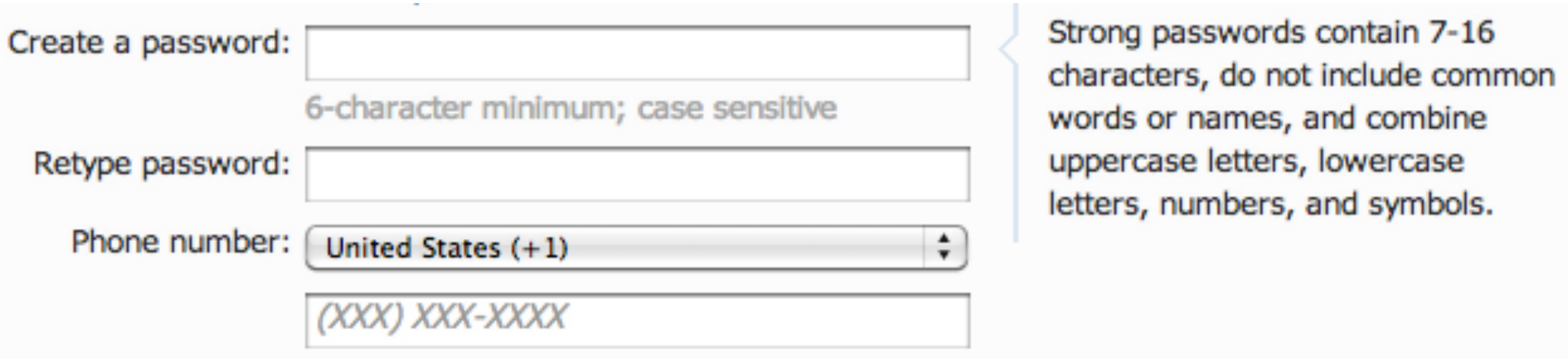
HTML page

Web application (server side)

Web server

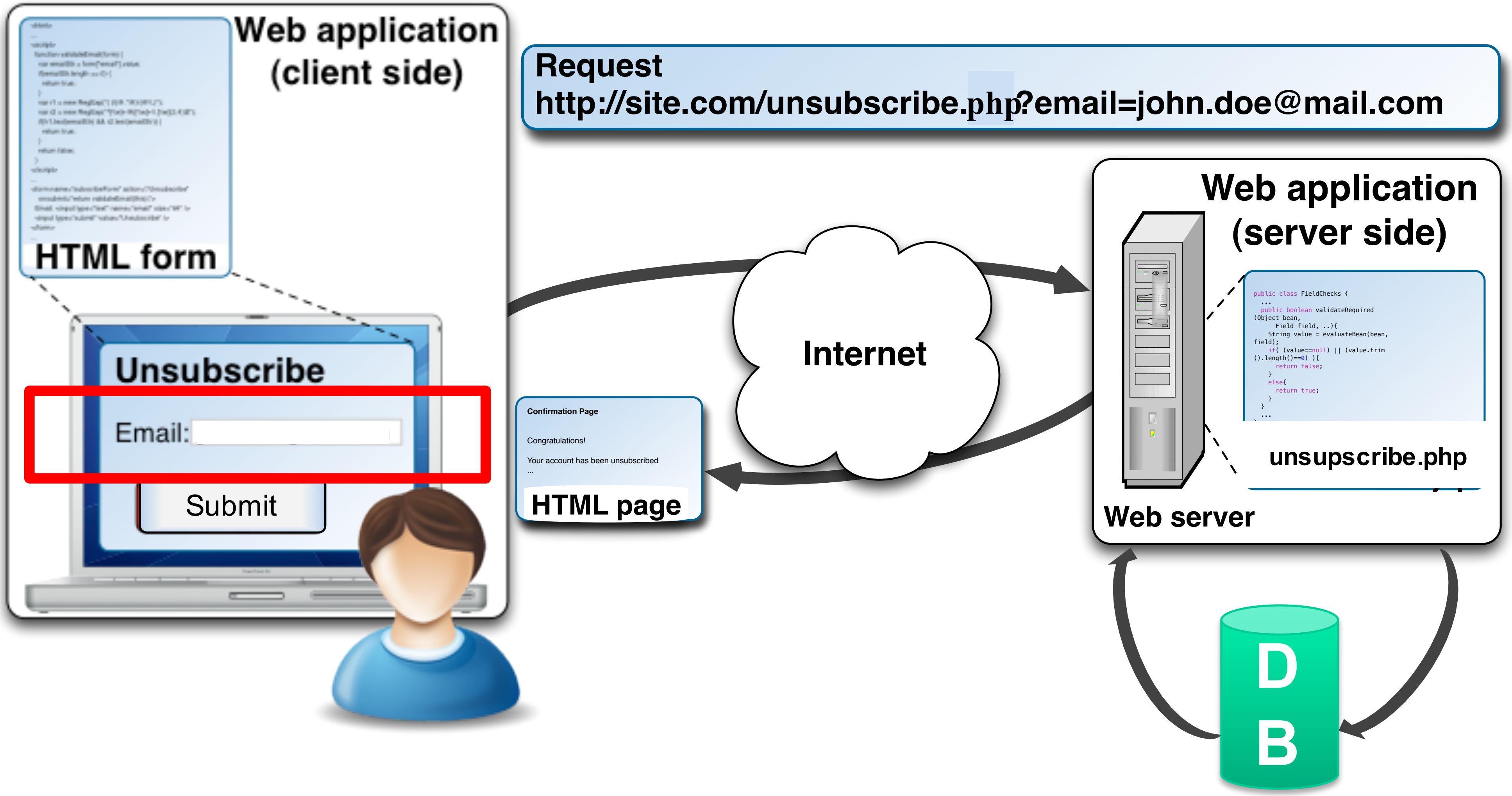
DB

Web Application Inputs are Strings



Strong passwords contain 7-16 characters, do not include common words or names, and combine uppercase letters, lowercase letters, numbers, and symbols.

Web Application Inputs are Strings



Request
http://site.com/unsubscribe.php?email=john.doe@mail.com

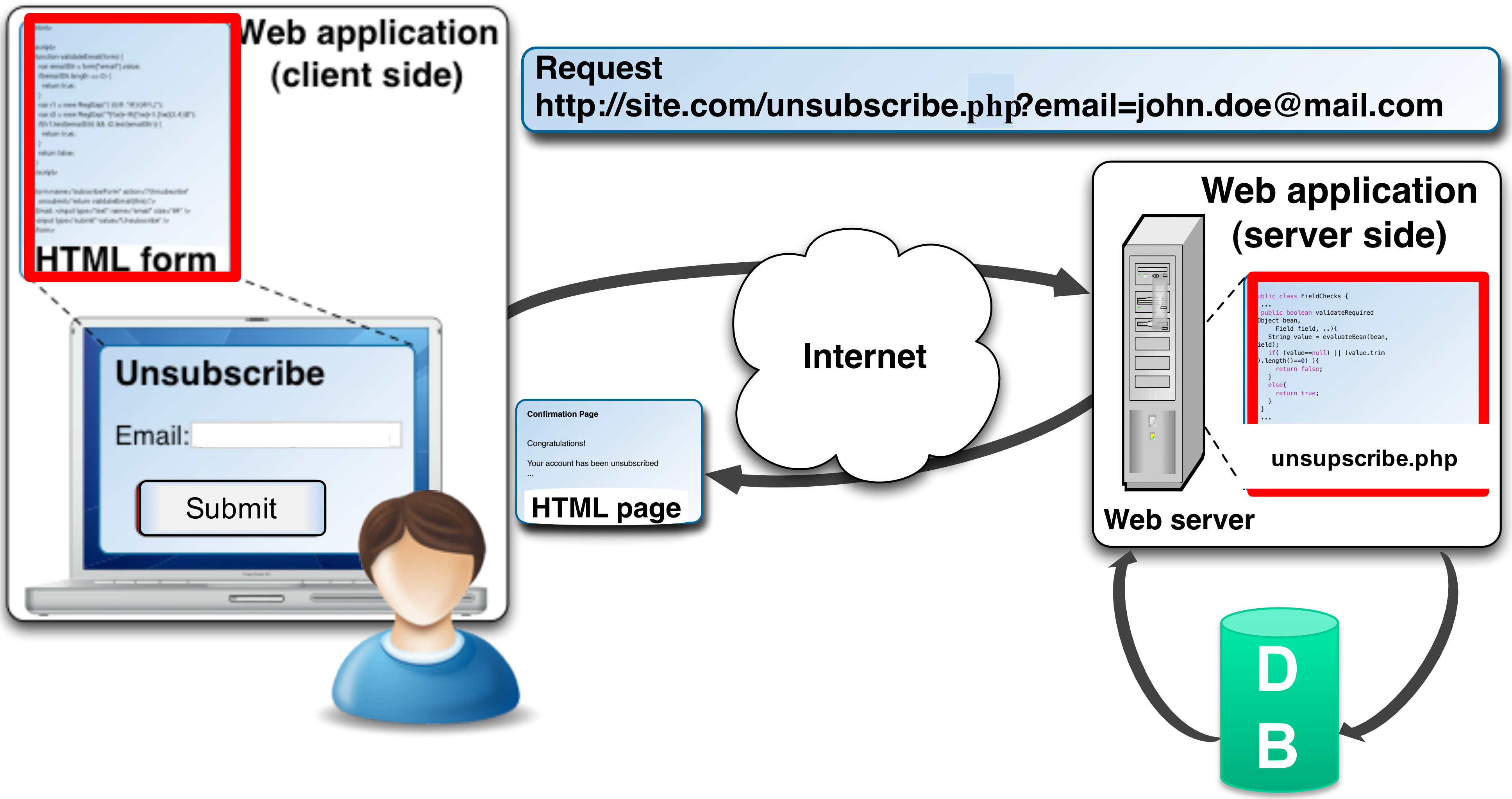
Confirmation Page

Congratulations!

Your account has been unsubscribed
...

```
public class FieldChecks {
  ...
  public boolean validateRequired
(Object bean,
      Field field, ..){
    String value = evaluateBean(bean,
field);
    if( (value==null) || (value.trim
().length()==0) ){
      return false;
    }
    else{
      return true;
    }
  }
  ...
```

Input Needs to be Validated and/or Sanitized



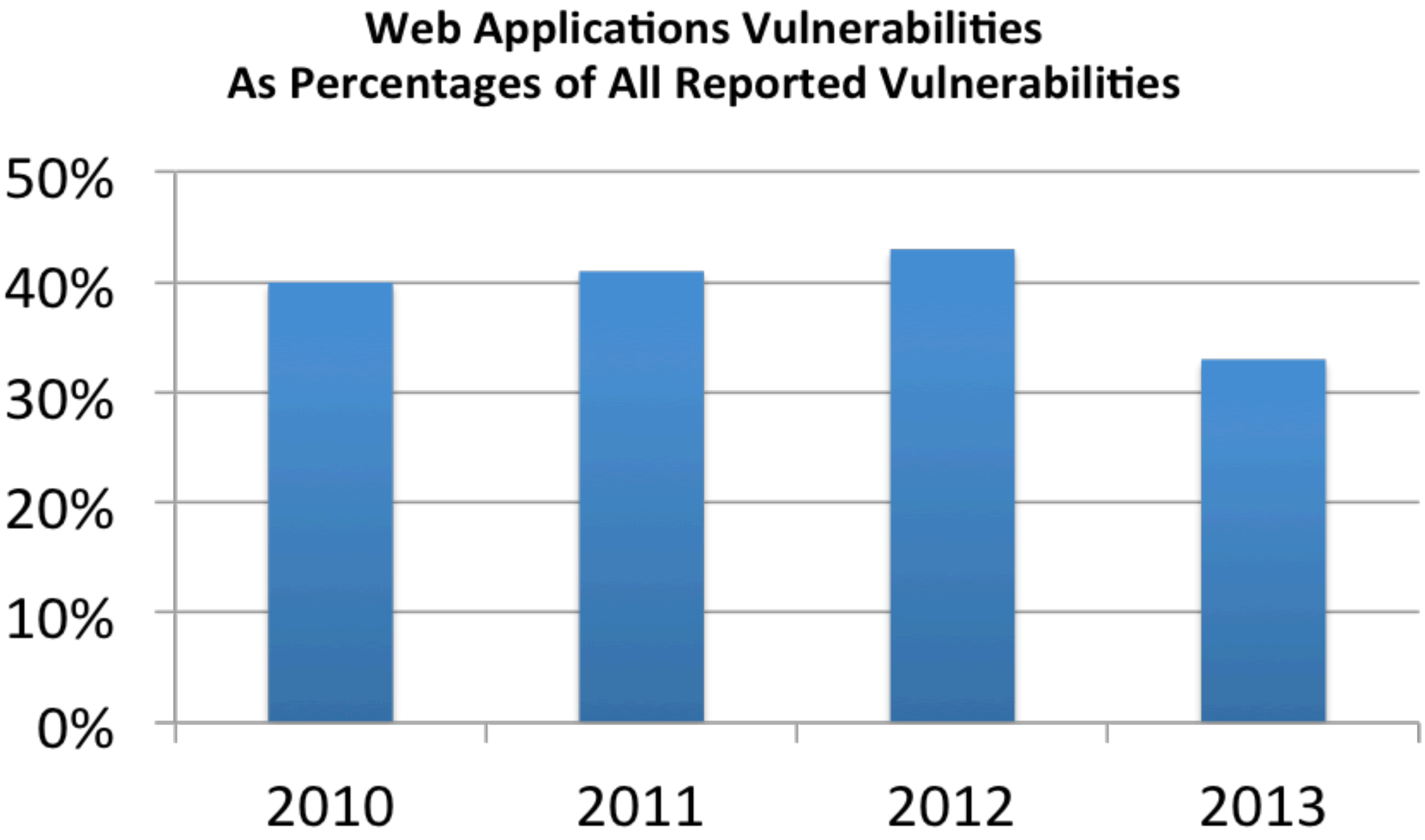
```
ublic class FieldChecks {
 ...
 public boolean validateRequired
Object bean,
     Field field, ..){
   String value = evaluateBean(bean,
ield);
   if( (value==null) || (value.trim
).length()==0) ){
     return false;
   }
   else{
     return true;
   }
 }
 ...
```

Vulnerabilities in Web Applications

- There are many well-known security vulnerabilities that exist in many web applications. Here are some examples:
  - **SQL injection:** where a malicious user executes SQL commands on the back-end database by providing specially formatted input
  - **Cross site scripting (XSS):** causes the attacker to execute a malicious script at a user's browser
  - **Malicious file execution:** where a malicious user causes the server to execute malicious code

- These vulnerabilities are typically due to
  - errors in user input validation and sanitization or
  - lack of user input validation and sanitization

Web Applications are Full of Bugs



Source: IBM X-Force report

Top Web Application Vulnerabilities

2007

1. Injection Flaws
2. XSS
3. Malicious File Execution

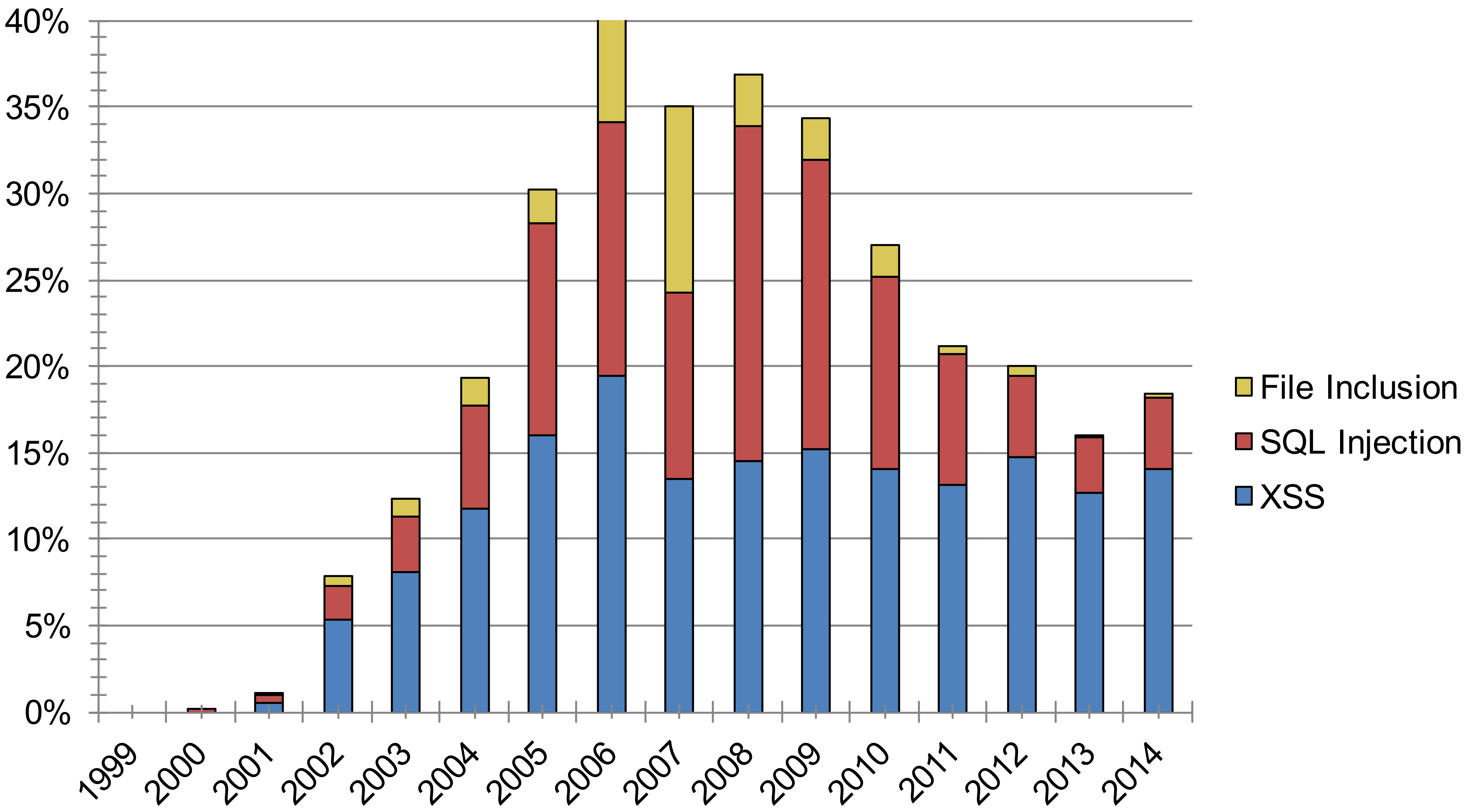
2010

1. Injection Flaws
2. XSS
3. Broken Auth. Session Management

2013

1. Injection Flaws
2. Broken Auth. Session Management
3. XSS

As Percentage of All Vulnerabilities



- SQL Injection, XSS, File Inclusion as percentage of all computer security vulnerabilities (extracted from the CVE repository)

Why Is Input Validation Error-prone?

- ***Extensive string manipulation***:
  - Web applications use extensive string manipulation
    - To construct html pages, to construct database queries in SQL, etc.
  - The user input comes in string form and must be validated and sanitized before it can be used
    - This requires the use of complex string manipulation functions such as string-replace
  - String manipulation is error prone

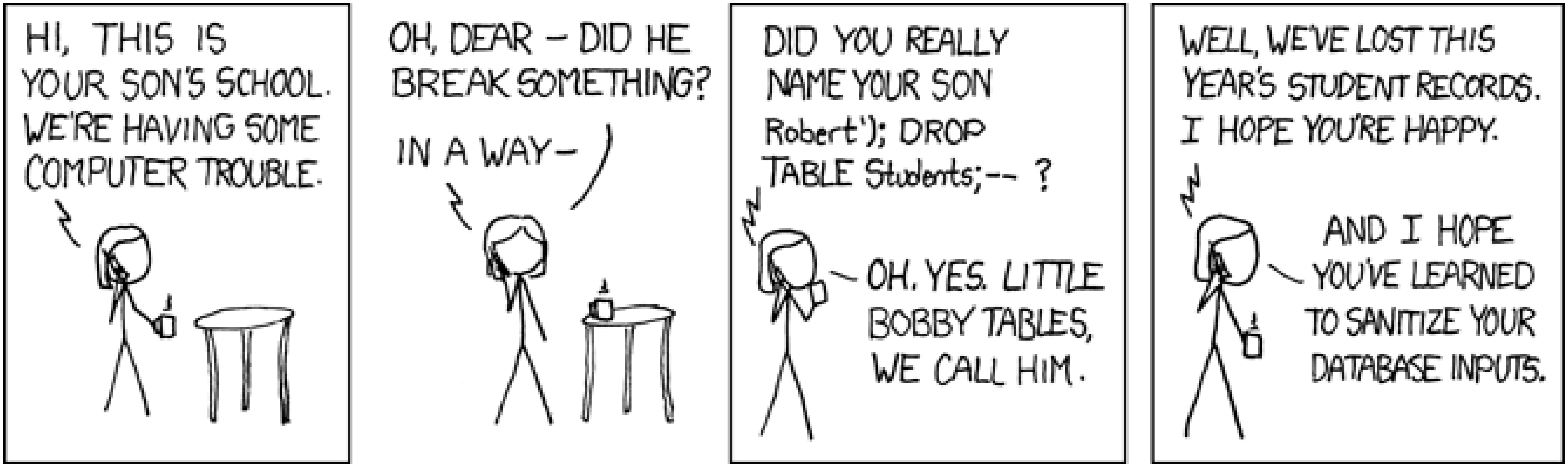
String Related Vulnerabilities

- String related web application vulnerabilities occur when:
  - a ***sensitive function*** is passed a ***malicious string input from the user***
  - This input contains an ***attack***
  - It is not ***properly sanitized*** before it reaches the sensitive function

- Using **string analysis** we can discover these vulnerabilities automatically

Computer Trouble at School

Exploits of a Mom.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Source: XKCD.com

SQL Injection

- A PHP example
- Access students’ data by $name (from a user input).

```
<?php
 $name = $GET[”name”];
 $user data = $db->query(“SELECT * FROM students
                              WHERE name = ‘$name’”);
?>
```

SQL Injection

- A PHP Example:
- Access students’ data by $name (from a user input).

```
<?php
 $name = $GET[”name”];
 $user data = $db->query(“SELECT * FROM students
WHERE name = ‘Robert ’); DROP TABLE students; - -”);
?>
```

What is a String?

- Given alphabet Σ, a string is a finite sequence of alphabet symbols

  $<c_1, c_2, \ldots, c_n>$ for all i, c_i is a character from Σ

- Σ = English = {a,…,z, A,…Z}

Σ = {a}

Σ = {a, b},

Σ = ASCII = {NULL, …, !, “, …, 0, …, 9, …, a, …, z, …}

Σ = Unicode

Σ = ASCII

“Foo”

“Ldkh#$klj54”

“123”

Σ = English

“Hello”

“Welcome”

“good”

Σ = {a}

“a”

“aa”

“aaa”

“aaaa”

“aaaaa”

Σ = {a,b}

“a”

“aba”

“bbb”

“ababaa”

“aaa”

String Manipulation Operations

- Concatenation
  - "1" + "2" → "12"
  - "Foo" + "bAaR" → "FoobAaR"

- Replacement
  - replace(s, "a", "A") bAaR → bAAR
  - replace (s, "2","") 234 → 34
  - toUpperCase(s) abC → ABC

String Filtering Operations

- Branch conditions

  length(s) < 4 ?

  ✅ “Foo”

  ❌ “bAaR”

  match(s, /^[0-9]+$/) ?

  ✅ “234”

  ❌ “a3v%6”

  substring(s, 2, 4) == “aR” ?

  ✅ ”bAaR”

  ❌ “Foo”

A Simple Example

- Another PHP Example:

<script ...

```
1:<?php
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
4: echo ”<td>” . $l_otherinfo . ”: ” . $www . ”</td>”;
5:?>
```

- The **`echo`** statement in line 4 is a sensitive function
- It contains a Cross Site Scripting (**XSS**) vulnerability

Is It Vulnerable?

- A simple ***taint analysis*** can report this segment vulnerable using taint propagation

tainted

```
<?php
 $www = $_GET[”www”];
 $l_otherinfo = ”URL”;
 echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
?>
```

- **echo** is tainted → script is **vulnerable**

How to Fix it?

- To fix the vulnerability we added a sanitization routine at line ***s***
- Taint analysis will assume that $www is ***untainted*** and report that the segment is ***NOT*** vulnerable

```
1:<?php              tainted
2:  $www = $_GET[”www”];
3:  $l_otherinfo = ”URL”;
   untainted
s:  $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4:  echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

Is It Really Sanitized?

<script …>

```
1:<?php
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
```

<script …>

```
s: $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4: echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

Sanitization Routines can be Erroneous

- The sanitization statement is not correct!

```
ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
```

  - Removes all characters that are not in { A-Za-z0-9 .-@:/ }
  - `.-@` denotes **all characters between “`.`” and “`@`”** (including **“<”** and **“>”**)
  - “`.-@`” should be “`.\-@`”

- This example is from a buggy sanitization routine used in MyEasyMarket-4.1 (line 218 in file trans.php)

String Analysis

- String analysis determines all possible values that a string expression can take during any program execution

- Using string analysis we can identify all possible input values of the sensitive functions
  - Then we can check if inputs of sensitive functions can contain attack strings

- How can we characterize attack strings?
  - Use regular expressions to specify the attack patterns

  - An attack pattern for XSS: **`Σ*<scriptΣ*`**

Vulnerabilities Can Be Tricky

- Input <!sc+rip!t ...> does not match the attack pattern
  - but it matches the vulnerability signature and it can cause an attack

```
1:<?php            <!sc+rip!t ...>
2:  $www = $_GET[”www”];
3:  $l_otherinfo = ”URL”;
    <script ...>
s:  $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4:  echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

String Analysis

- If string analysis determines that the intersection of the attack pattern and possible inputs of the sensitive function is empty
  - then we can conclude that the program is secure

- If the intersection is not empty, then we can again use string analysis to generate a ***vulnerability signature***
  - characterizes all malicious inputs

- Given $\Sigma^*\texttt{<script}\Sigma^*$ as an attack pattern:
  - The vulnerability signature for $_GET[”www”] is

    $$\Sigma^*\texttt{<}\alpha^*\texttt{s}\alpha^*\texttt{c}\alpha^*\texttt{r}\alpha^*\texttt{i}\alpha^*\texttt{p}\alpha^*\texttt{t}\Sigma^*$$

    where $\alpha \notin$ { A-Za-z0-9 .-@:/ }

OUTLINE

- Motivation
- **Symbolic string analysis**
- Automated repair
- String constraint solving
- Model counting

Automata-based String Analysis

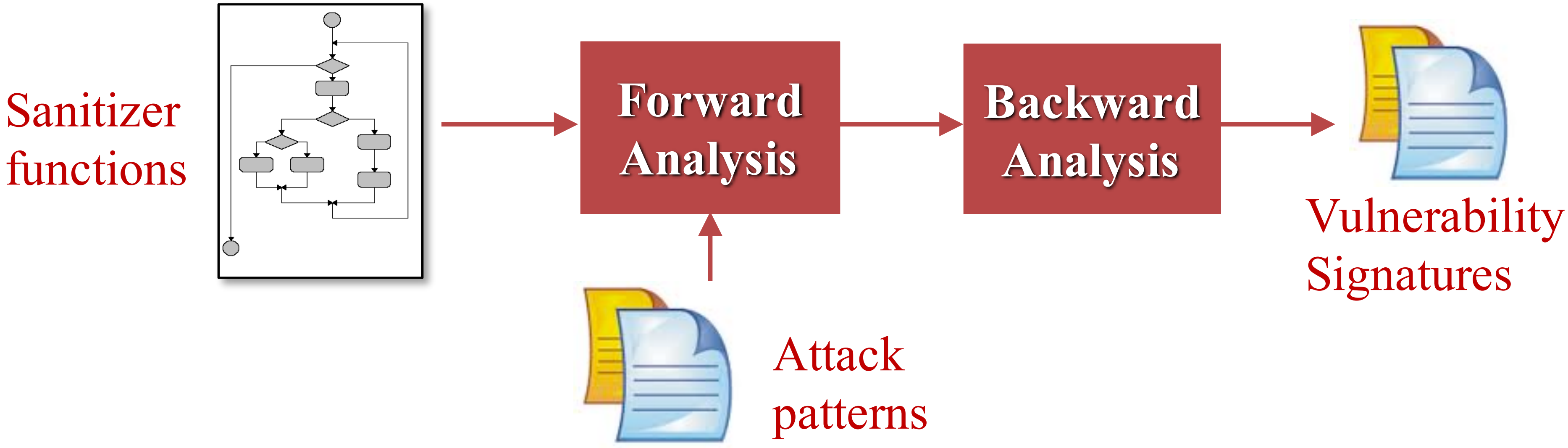
- Finite State Automata can be used to characterize sets of string values

- 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



Sanitizer functions

Forward Analysis

Backward Analysis

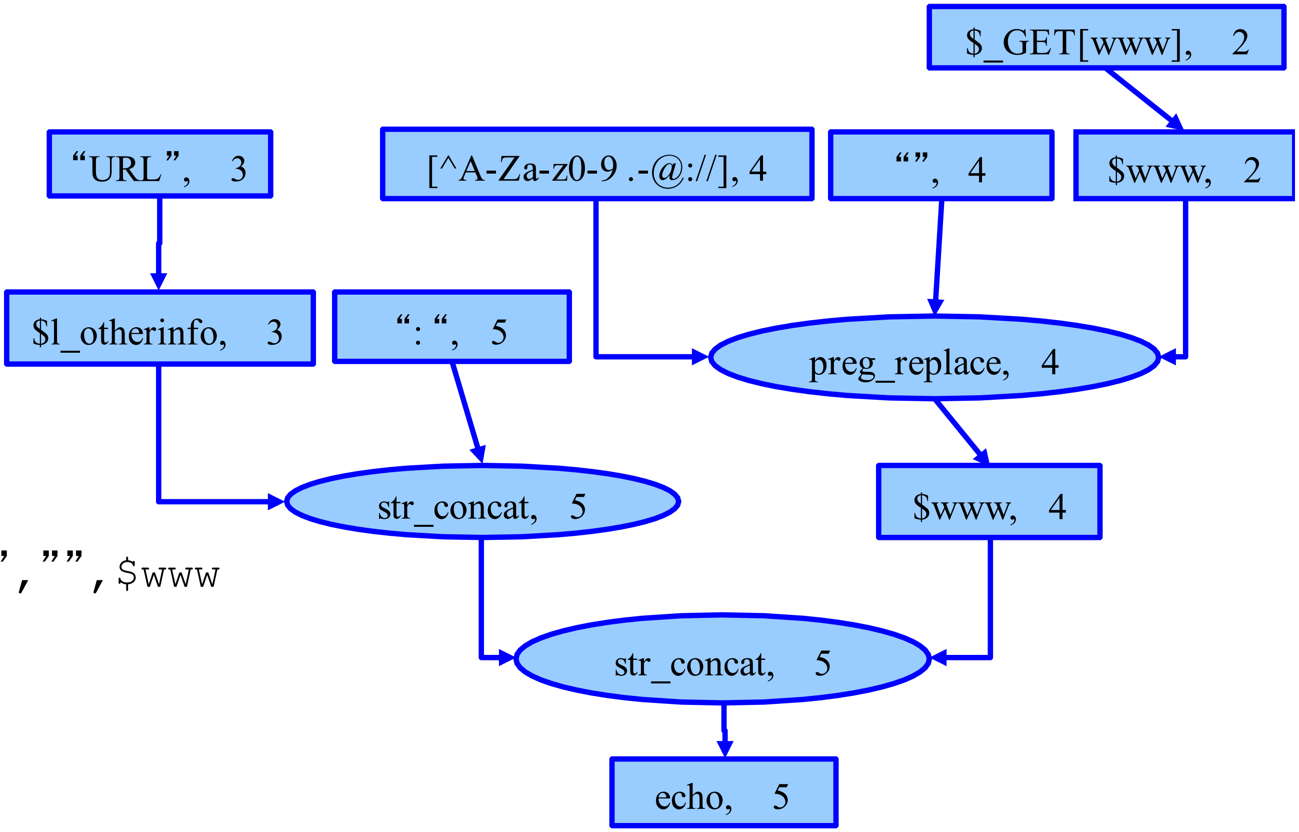
Vulnerability Signatures

Attack patterns

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

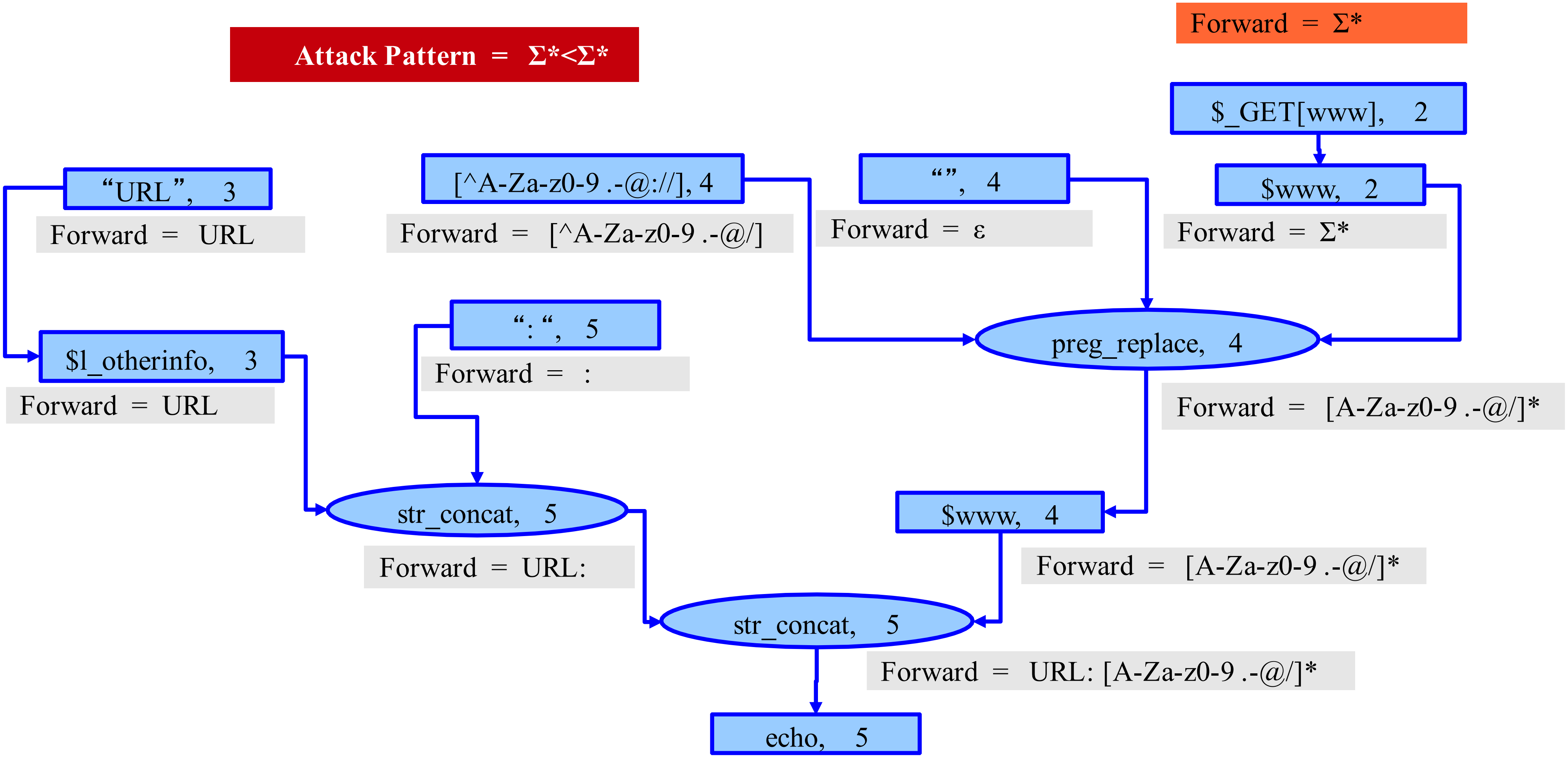
- Need to implement **post-image computations** for string operations:
  - **postConcat**(M1, M2)

    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



Forward = Σ*

Attack Pattern = Σ*<Σ*

$_GET[www], 2

“URL”, 3

Forward = URL

[^A-Za-z0-9 .-@://], 4

Forward = [^A-Za-z0-9 .-@/]

“”, 4

Forward = ε

$www, 2

Forward = Σ*

“: “, 5

Forward = :

$l_otherinfo, 3

Forward = URL

preg_replace, 4

Forward = [A-Za-z0-9 .-@/]*

str_concat, 5

Forward = URL:

$www, 4

Forward = [A-Za-z0-9 .-@/]*

str_concat, 5

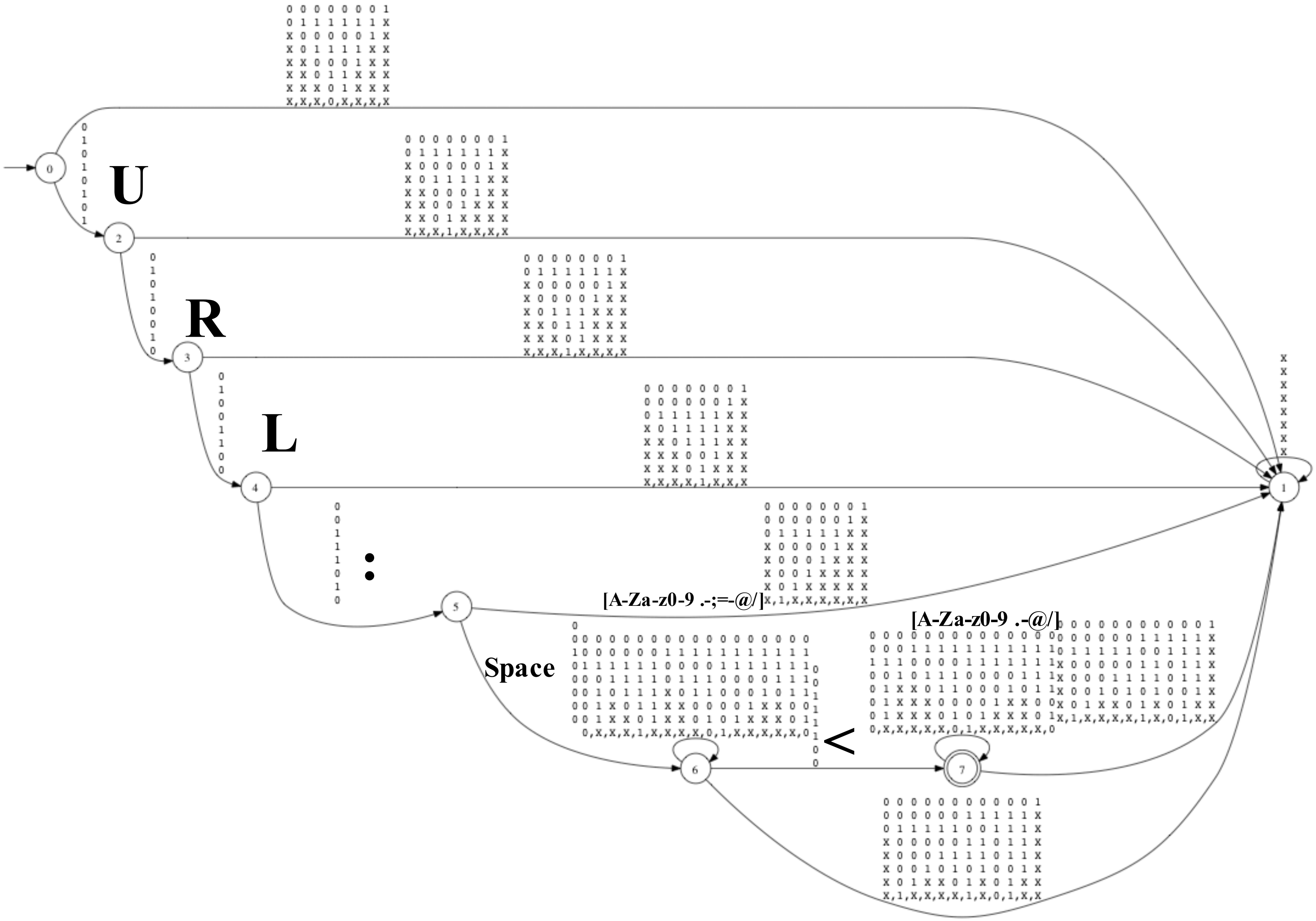
Forward = URL: [A-Za-z0-9 .-@/]*

echo, 5

L(Σ*<Σ*) ∩ L(URL: [A-Za-z0-9 .-@/]*) =

L(URL: [A-Za-z0-9 .-;=-@/]*<[A-Za-z0-9 .-@/]*) ≠ Ø

Result Automaton



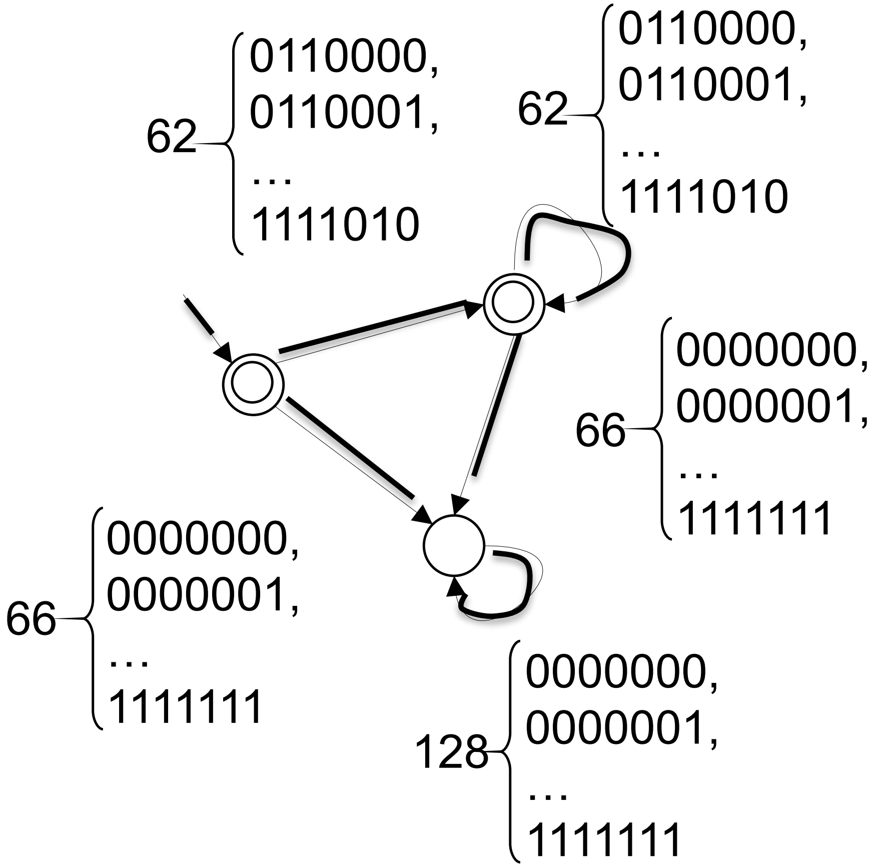
URL: [A-Za-z0-9 .-;=-@/]*<[A-Za-z0-9 .-@/]*

Symbolic Automata Representation

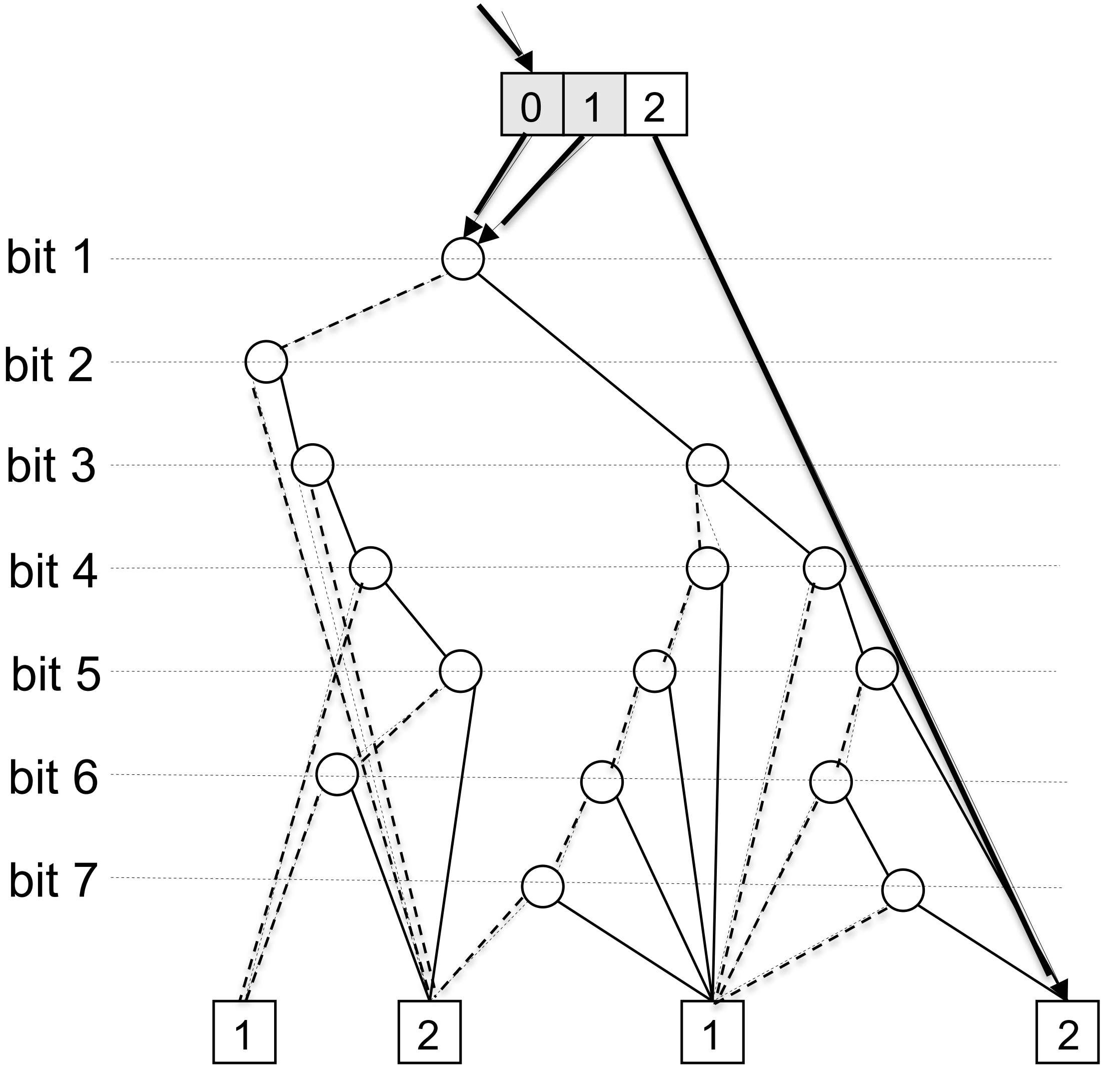
- MONA DFA Package for automata manipulation
  - [Klarlund and Møller, 2001]
- Compact Representation:
  - Canonical form and
  - Shared BDD nodes
- Efficient MBDD Manipulations:
  - Union, Intersection, and Emptiness Checking
  - Projection and Minimization
- Cannot Handle Nondeterminism:
  - Use dummy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

Symbolic Automata Representation

Explicit DFA representation



Symbolic DFA representation



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

Widening Automata

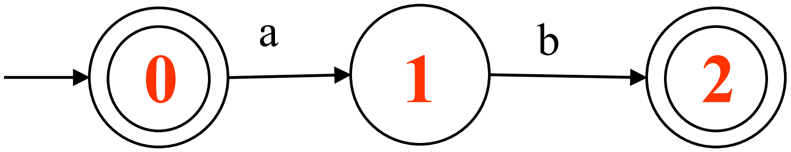
- Given automata A and A' we want to compute $A \nabla A'$

- Basic idea: Merge states in the automaton to create an automaton that accepts a larger language
  - Merge states that are similar (equivalent) in some way

- We say that states k and k' are **equivalent** ($k \equiv k'$) if either
  - k and k' can be reached from initial state with the same string (unless k or k' is a sink state)
  - or, the languages accepted from k and k' are equal
  - or, for some state k'', $k \equiv k''$ and $k' \equiv k''$

- The states of $A \nabla A'$ are the equivalence classes of $\equiv$

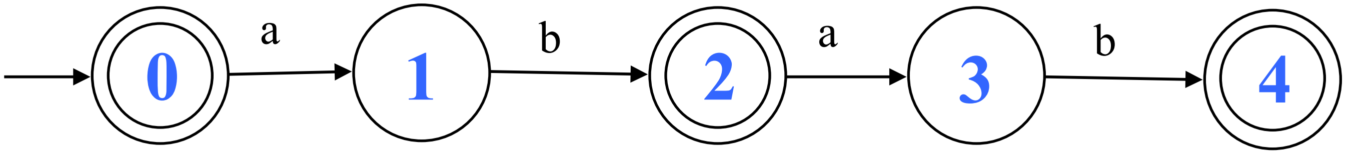
Example



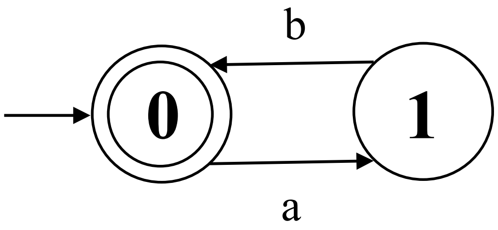
∇

Equivalence classes:

{0, 0, 2, 2, 4} {1,1,3}



$=$



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

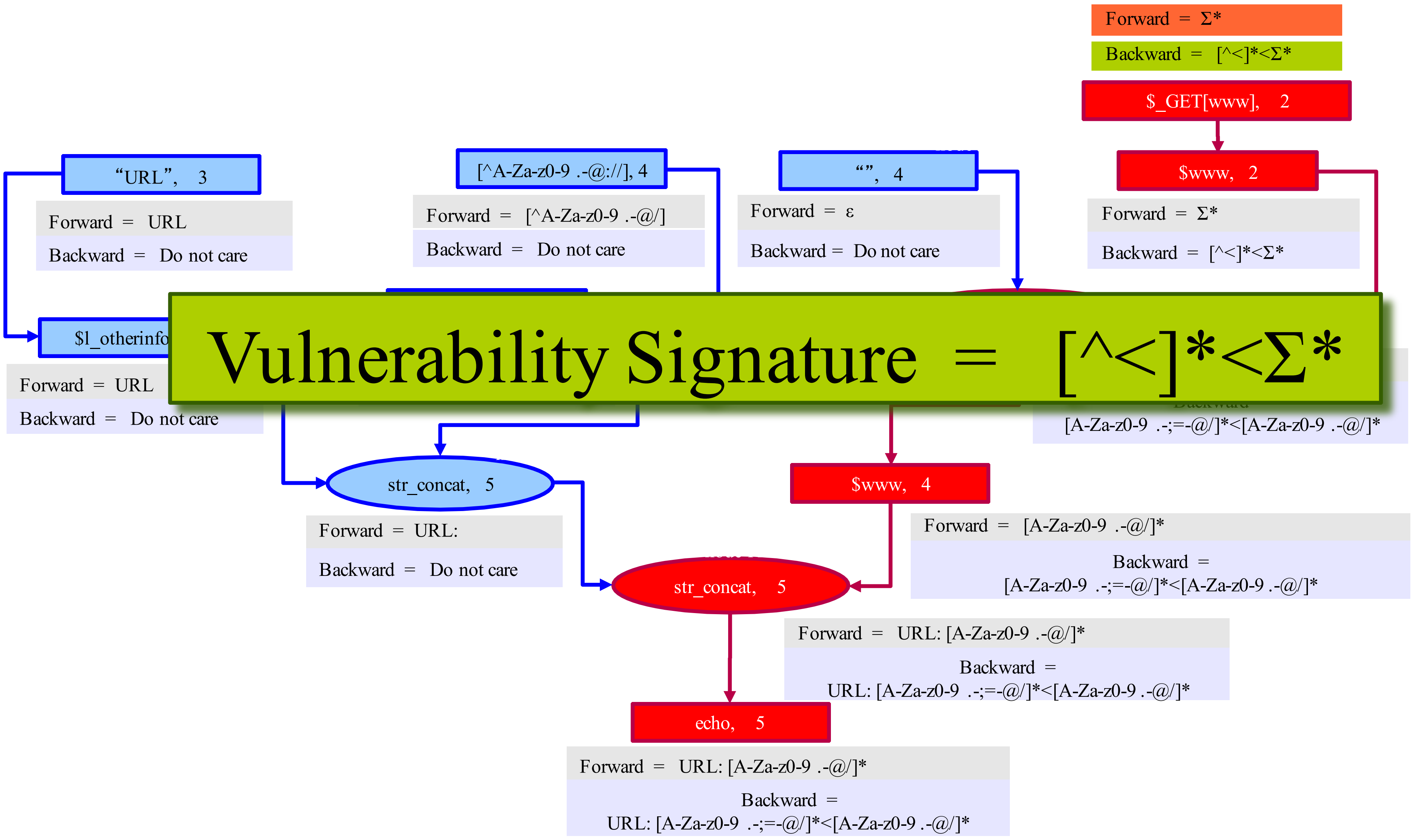
- Need to implement **Pre-image computations** on string operations:
  - **preConcatPrefix**(M, M2)

    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



Forward = Σ*

Backward = [^<]*<Σ*

$_GET[www], 2

$www, 2

Forward = Σ*

Backward = [^<]*<Σ*

“URL”, 3

Forward = URL

Backward = Do not care

[^A-Za-z0-9 .-@://], 4

Forward = [^A-Za-z0-9 .-@/]

Backward = Do not care

“”, 4

Forward = ε

Backward = Do not care

$l_otherinfo

Forward = URL

Backward = Do not care

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

str_concat, 5

Forward = URL:

Backward = Do not care

$www, 4

Forward = [A-Za-z0-9 .-@/]*

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

str_concat, 5

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

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

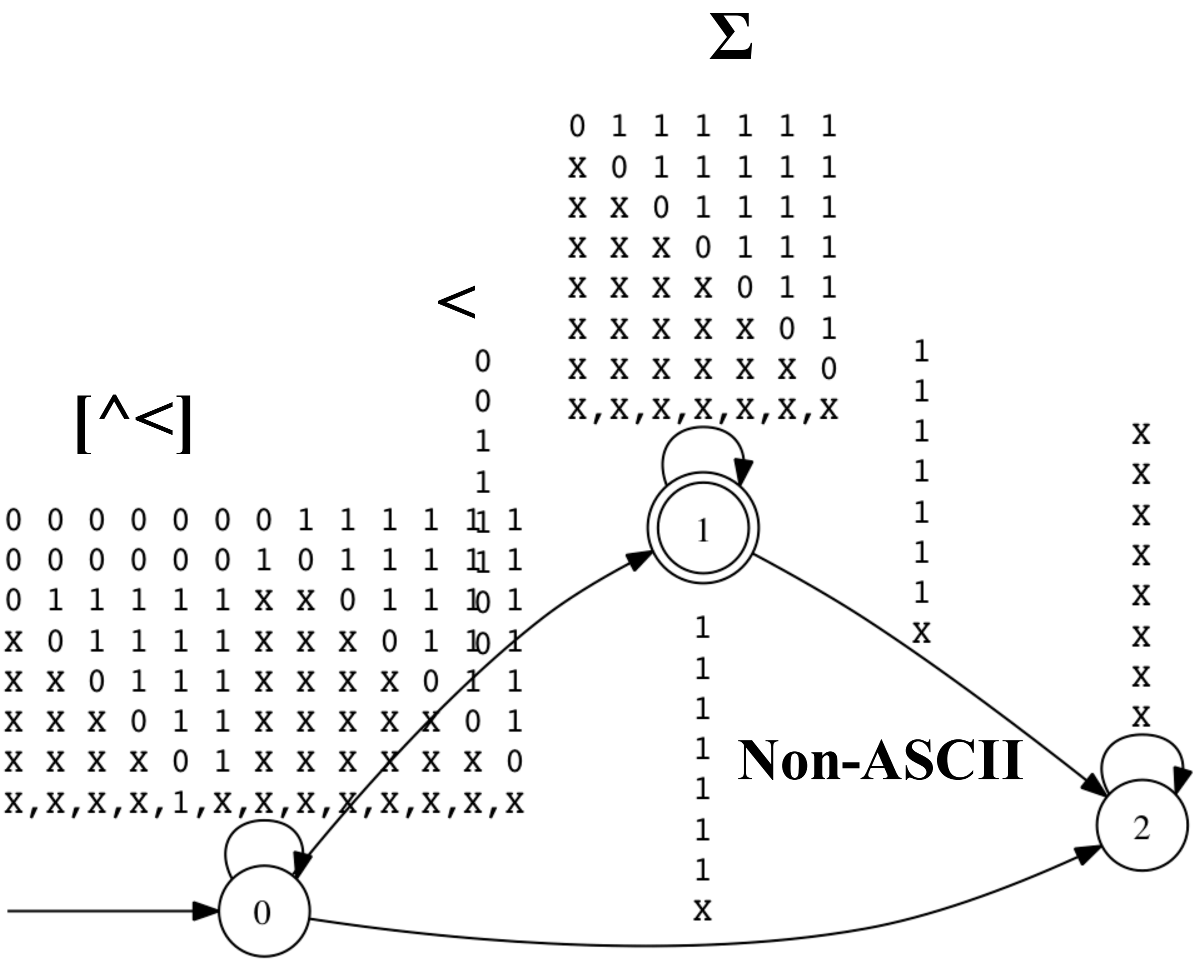
echo, 5

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

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

Vulnerability Signature = [^<]*<Σ*

Vulnerability Signature Automaton



[^<]*<Σ*

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

- **What else can we do?**
  - Can we automatically repair a vulnerability if we detect one?

OUTLINE

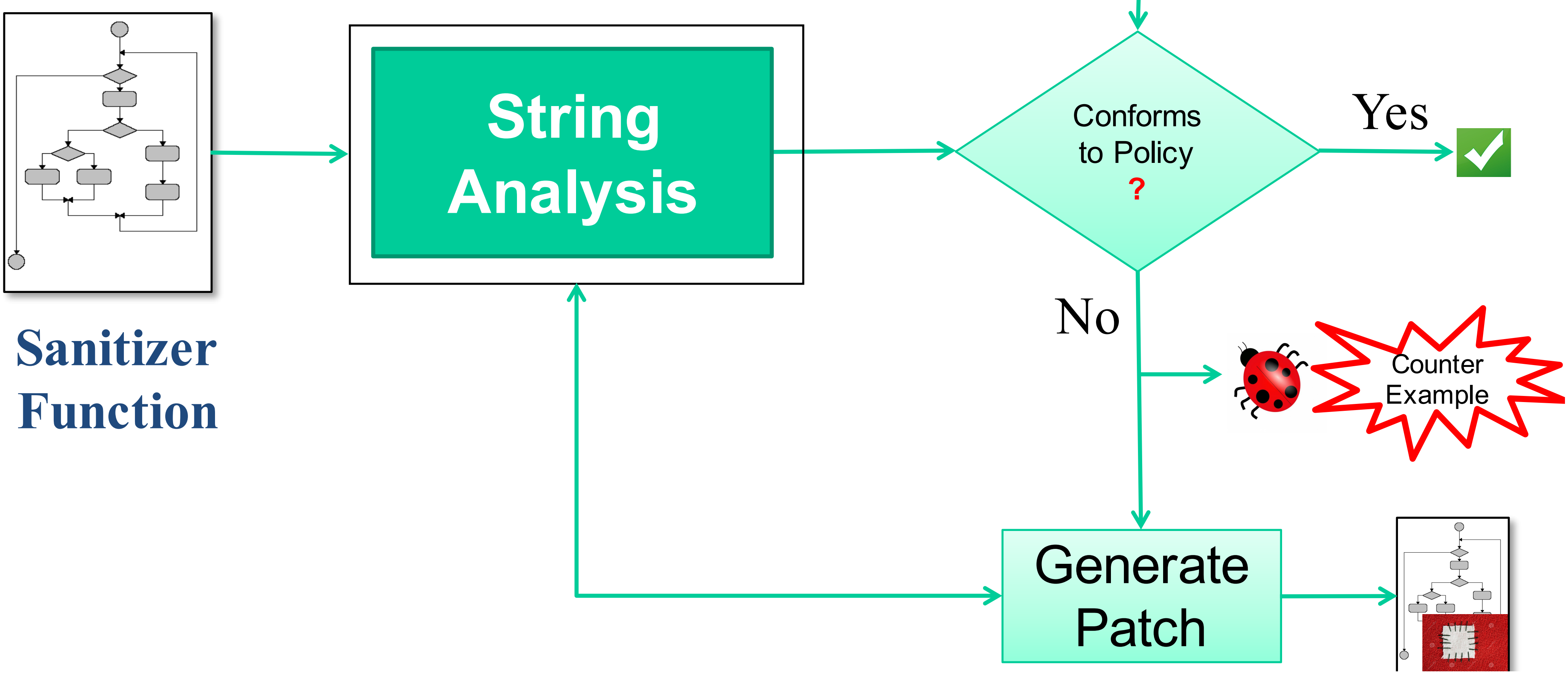
- Motivation
- Symbolic string analysis
- **Automated repair**
- String constraint solving
- Model counting

Vulnerability Detection and Repair

- Symbolic string analysis for detection of vulnerabilities
- Use regular expressions for specification of input validation policy
- Generate a patch when a vulnerability is detected

Policy

(regular expression)



Sanitizer Function

String Analysis

Conforms to Policy ?

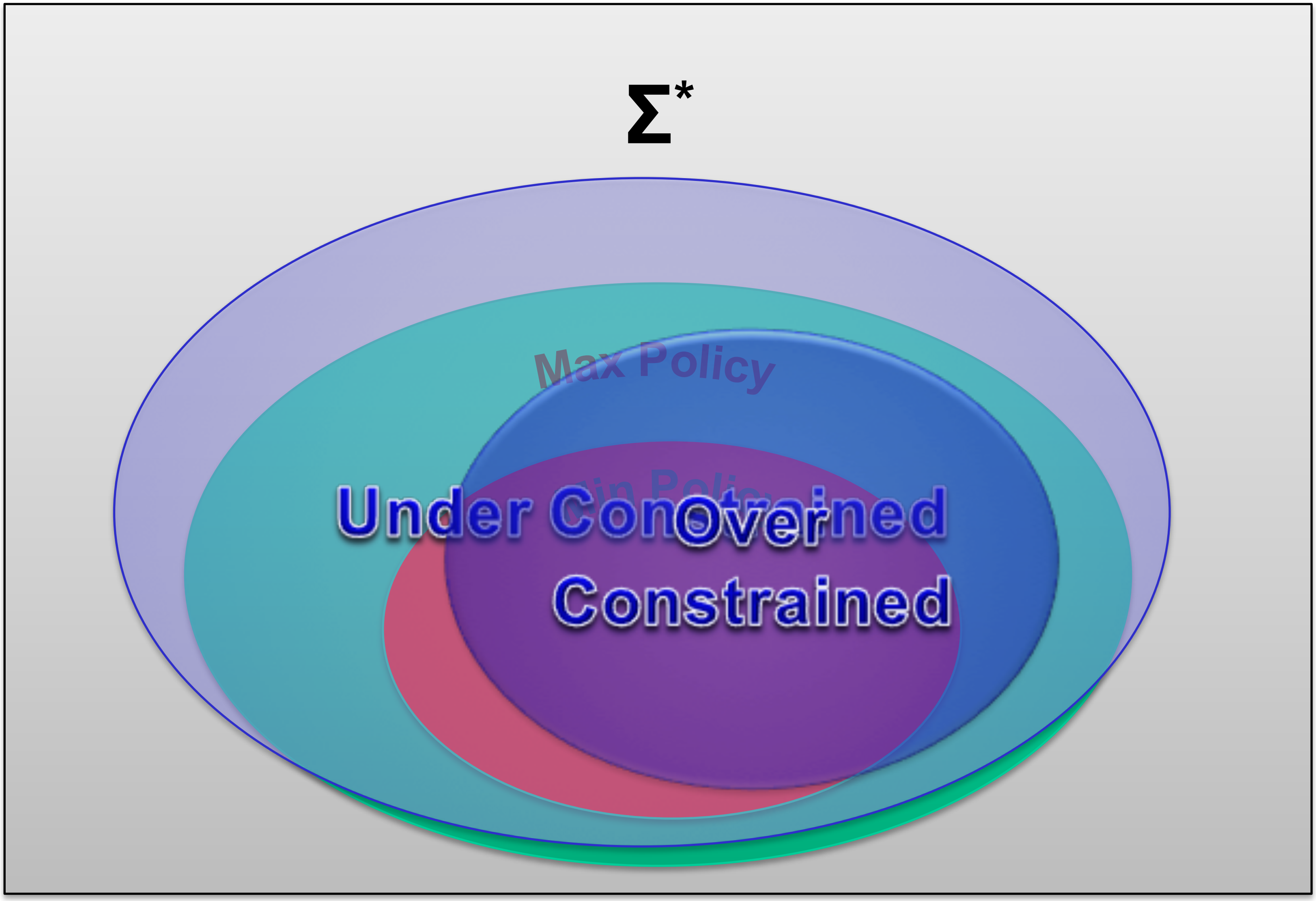
Yes

No

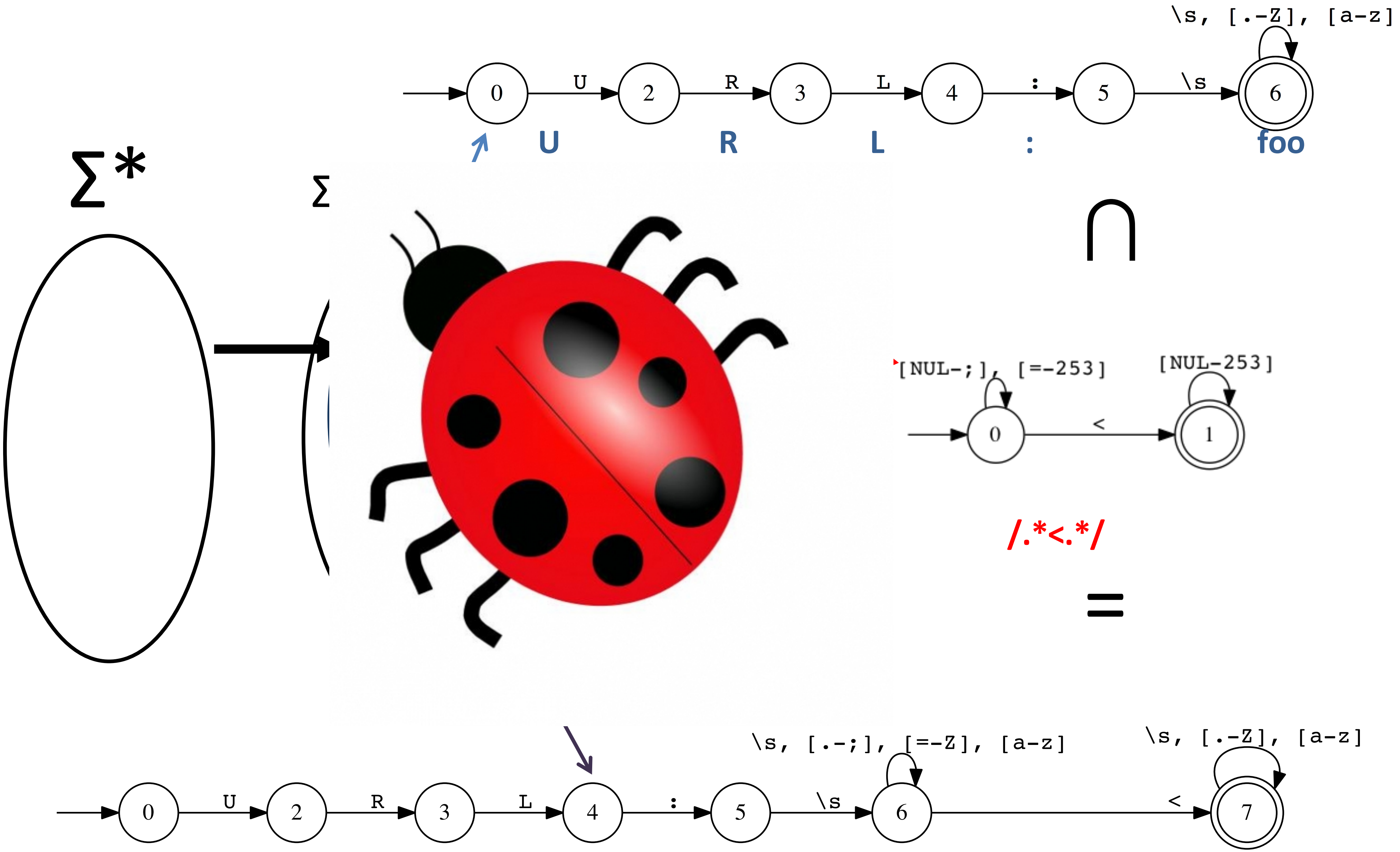
Counter Example

Generate Patch

Min – Max Policies



Vulnerability Detection



Σ*

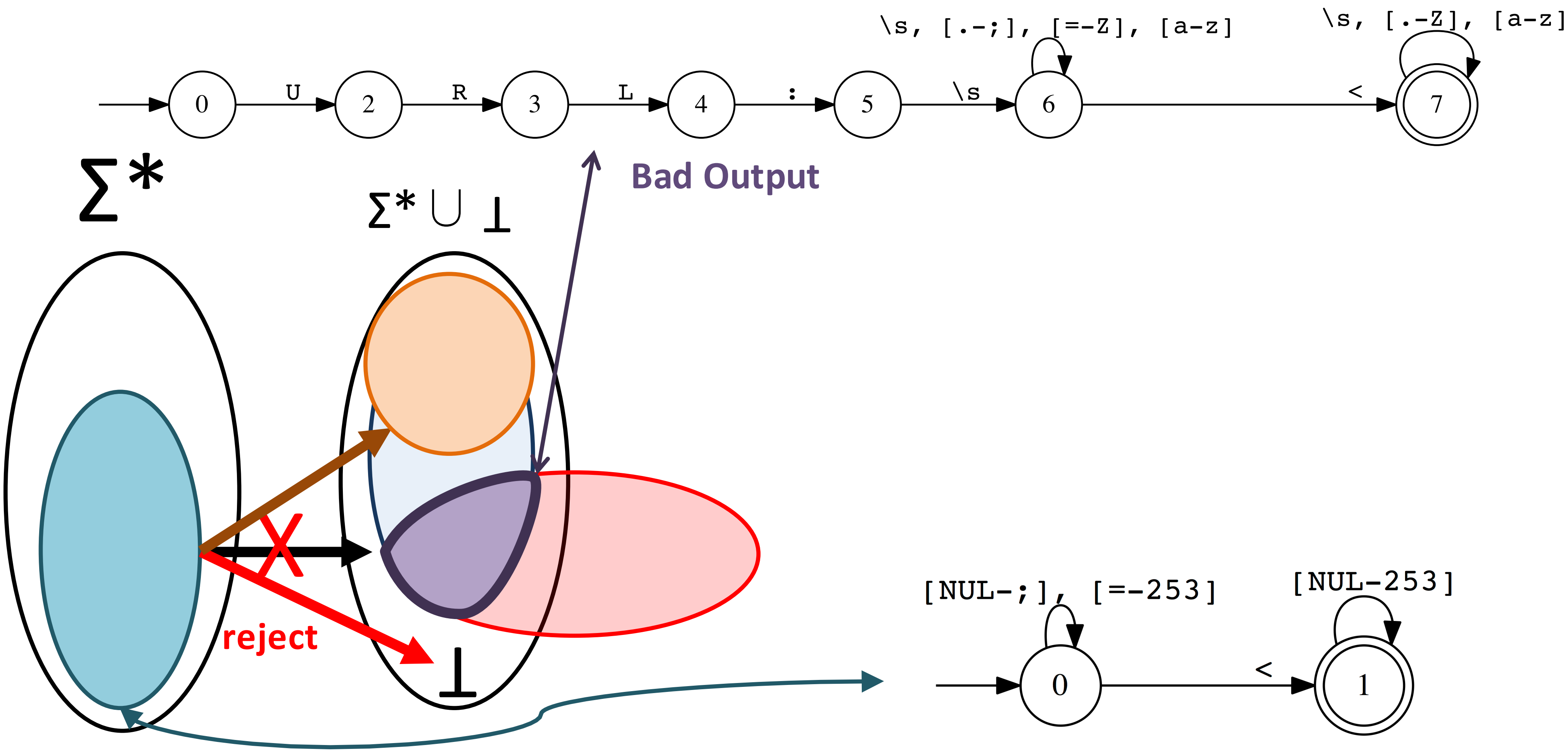
U R L : foo

∩

/.*<.*/

=

Vulnerability Signature Generation and Vulnerability Repair



Pre-Image (Bad Input)

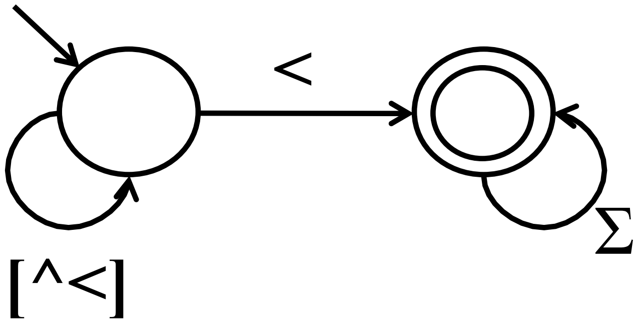
Vulnerability Signature

All inputs that can exploit the vulnerability
(or an over approximation of that set)

Vulnerability Signatures

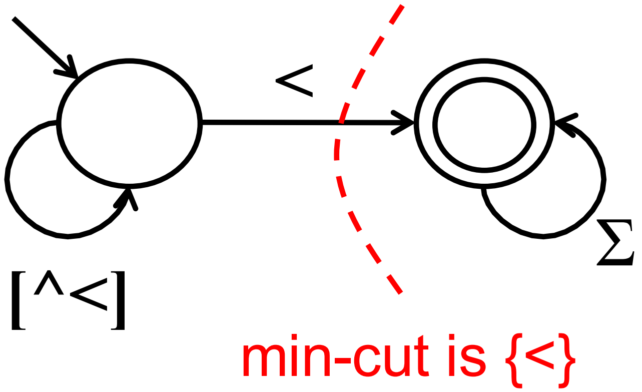
- The vulnerability signature is the result of the input node, which includes all possible malicious inputs
- An input that does not match this signature cannot exploit the vulnerability
- After generating the vulnerability signature
  - Can we generate a patch based on the vulnerability signature?

Example vulnerability signature automaton:



Patches from Vulnerability Signatures

- Main idea:
  - Given a vulnerability signature automaton, find a cut that separates initial and accepting states
  - Remove the characters in the cut from the user input to sanitize

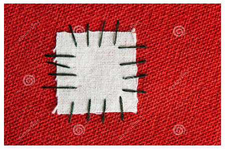


Patches from Vulnerability Signatures

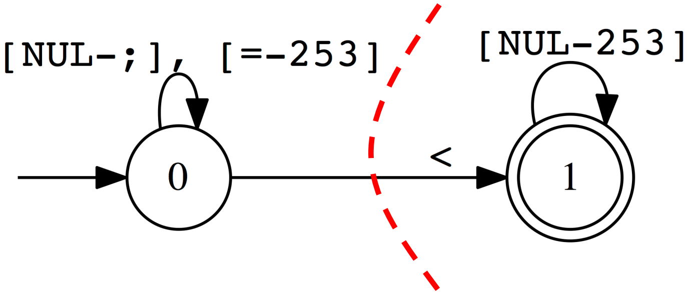
- Ideally, we want to modify the input (as little as possible) so that it does not match the vulnerability signature
- Given a DFA, an ***alphabet cut*** is
  - a set of characters that after ”removing” the edges that are associated with the characters in the set, the modified DFA does not accept any non-empty string
- Finding a minimal alphabet cut of a DFA is an NP-hard problem (one can reduce the vertex cover problem to this problem)
  - We use a min-cut algorithm instead
  - The set of characters that are associated with the edges of the min cut is an alphabet cut
    - but not necessarily the minimum alphabet cut

Generated Patch

```
1: <?php
P:  if(preg match(’/[^ <]*<.*/’,$_GET[″www″]))
      $_GET[″www″] = preg replace(’<’,″″,$_GET[″www″]);
2:  $www = $_GET[”www”];
3:  $l_otherinfo = ”URL”;
4:  $www = preg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
5:  echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
6: ?>
```



Input	Original Output	New Output
Foobar	URL: Foobar	URL: Foobar
Foo<bar	URL: Foo<bar	URL: Foobar
a<b<c<d	URL: a<b<c<d	URL: abcd



min-cut is {<}

Experiments

- We evaluated this approach on five vulnerabilities from three open source web applications:

  (1) MyEasyMarket-4.1: A shopping cart program

  (2) BloggIT-1.0: A blog engine

  (3) proManager-0.72: A project management system

- We used the following XSS attack pattern:

  `Σ*<scriptΣ*`

Forward Analysis Results

- The dependency graphs of these benchmarks are simplified based on the sinks
  - Unrelated parts are removed using slicing

Input				Results		
#nodes	#edges	#sinks	#inputs	Time(s)	Mem (kb)	#states/# bdds
21	20	1	1	0.08	2599	23/219
29	29	1	1	0.53	13633	48/495
25	25	1	2	0.12	1955	125/1200
23	22	1	1	0.12	4022	133/1222
25	25	1	1	0.12	3387	125/1200

Backward Analysis Results

- We use the backward analysis to generate the vulnerability signatures
  - Backward analysis starts from the vulnerable sinks identified during forward analysis

Input				Results		
#nodes	#edges	#sinks	#inputs	Time(s)	Mem (kb)	#states/#bdds
21	20	1	1	0.46	2963	9/199
29	29	1	1	41.03	1859767	811/8389
25	25	1	2	2.35	5673	20/302, 20/302
23	22	1	1	2.33	32035	91/1127
25	25	1	1	5.02	14958	20/302

Alphabet Cuts

- We generate cuts from the vulnerability signatures using a min-cut algorithm

Input				Results
#nodes	#edges	#sinks	#inputs	Alphabet Cut
21	20	1	1	{<}
29	29	1	1	{S,’ ,”}
25	25	1	2	Σ , Σ
23	22	1	1	{<,’ ,”}
25	25	1	1	{<,’ ,”}

Vulnerability signature depends on two inputs

- **Problem:** When there are two user inputs the patch will block everything and delete everything
  - Overlooks the relations among input variables (e.g., the concatenation of two inputs contains < SCRIPT)

Relational Vulnerability Signature

- Perform forward analysis using multi-track automata to generate relational vulnerability signatures

- Each track represents one user input
  - An auxiliary track represents the values of the current node
  - We intersect the auxiliary track with the attack pattern upon termination

Relational Vulnerability Signature

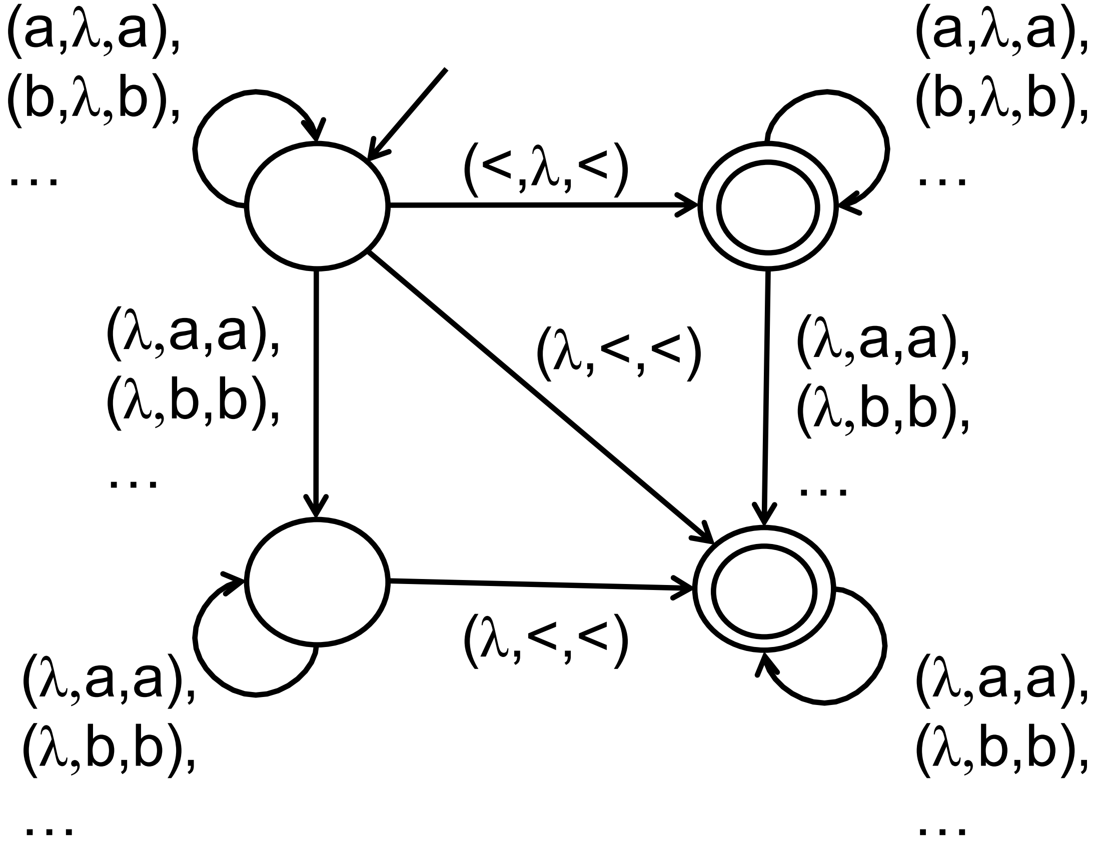
- Consider a simple example having multiple user inputs

```php
<?php
$www = $_GET[”www”];
$url = $_GET[”url”];
echo $url. $www;
?>
```

- Let the attack pattern be $\Sigma * < \Sigma *$

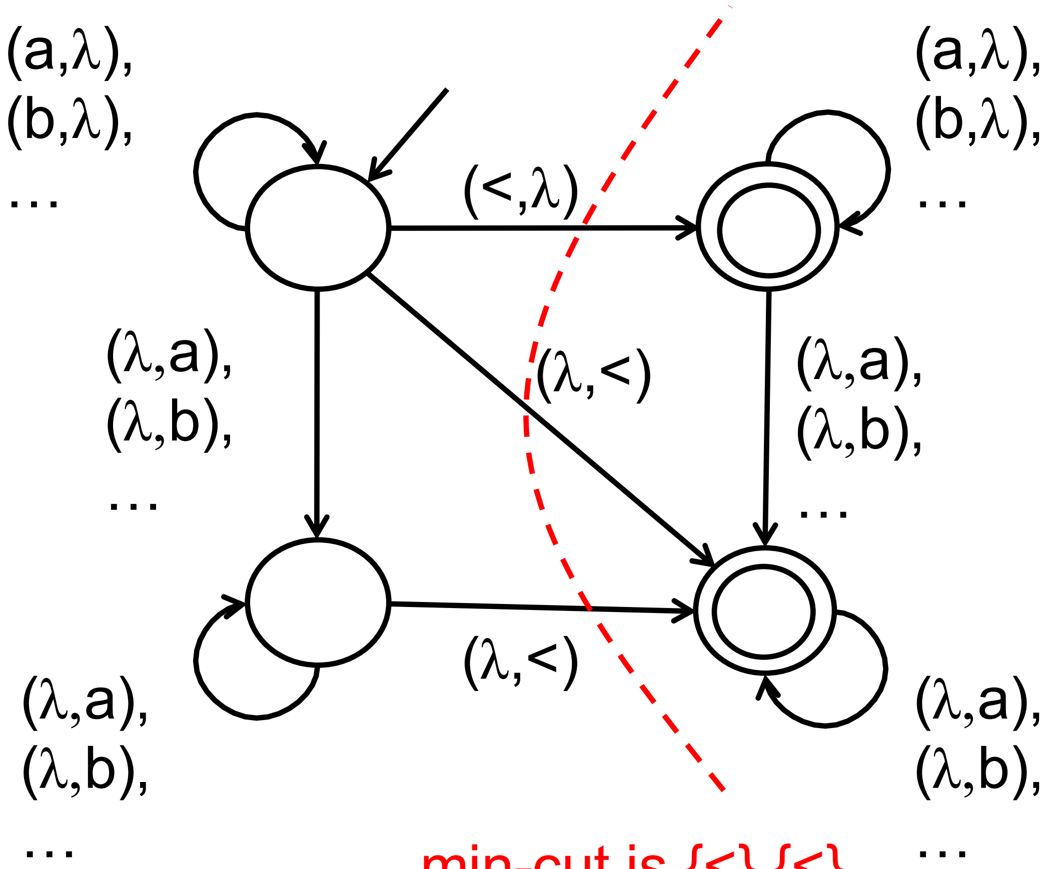
Relational Vulnerability Signature

- A multi-track automaton: ($url, $www, aux)
- Identifies the fact that the concatenation of two inputs contains <



Relational Vulnerability Signature

- Project away the auxiliary variable
- Find the min-cut
- This min-cut identifies the alphabet cuts {<} for the first track ($url) and {<} for the second track ($www)



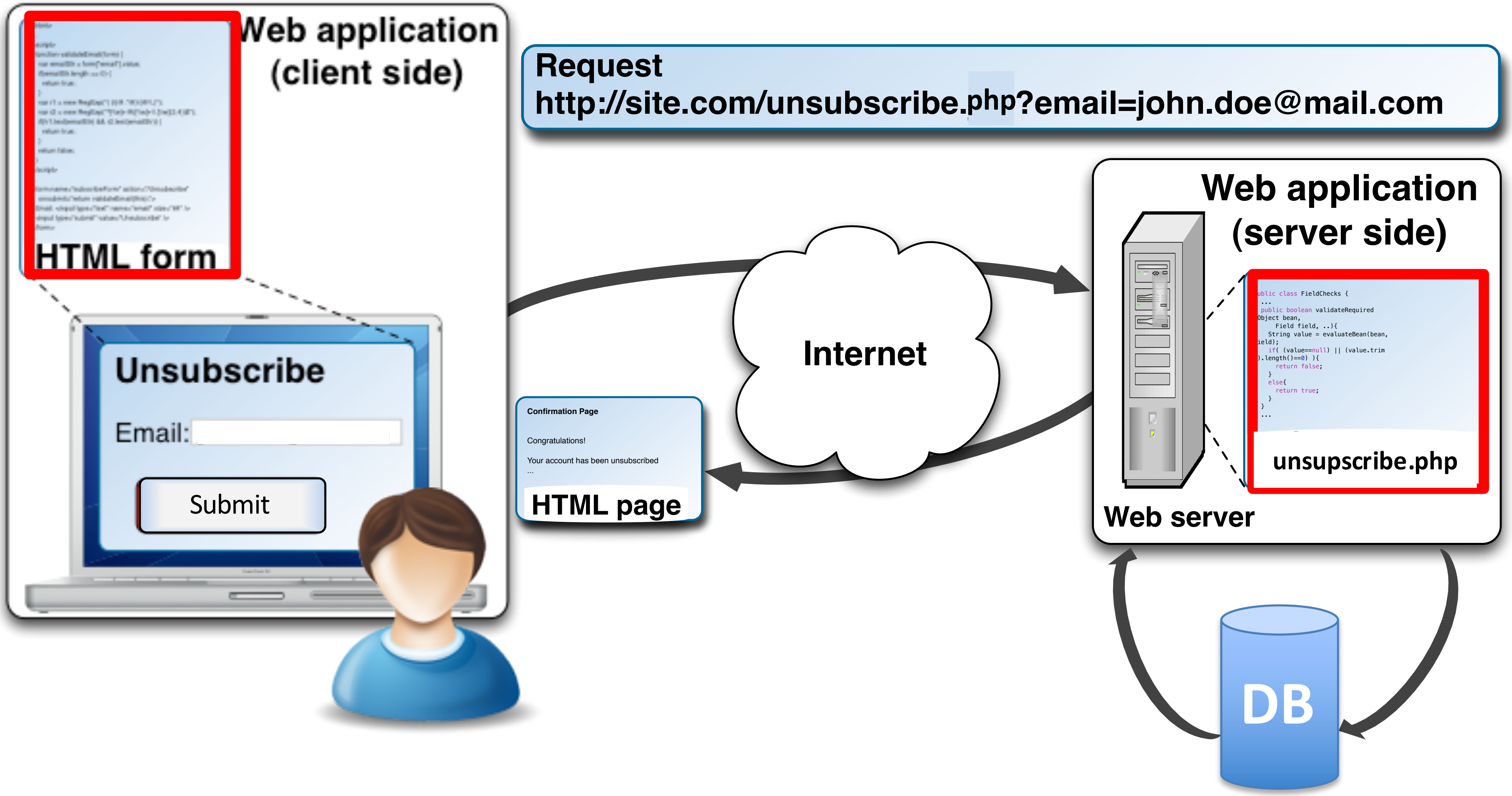
min-cut is {<},{<}

Patch for Multiple Inputs

- Patch: If the inputs match the signature, delete its alphabet cut

```
<?php
  if (preg match(’/[^ <]*<.*/’, $ GET[”url”].$ GET[”www”]))
  {
    $ GET[”url”] = preg replace(<,””,$ GET[”url”]);
    $ GET[”www”] = preg replace(<,””,$ GET[”www”]);
  }
     $www = $ GET[”www”];
     $url = $ GET[”url”];
     echo $url. $www;
?>
```

Differential String Analysis: Verification and Repair without a Policy Specification



Request
http://site.com/unsubscribe.php?email=john.doe@mail.com

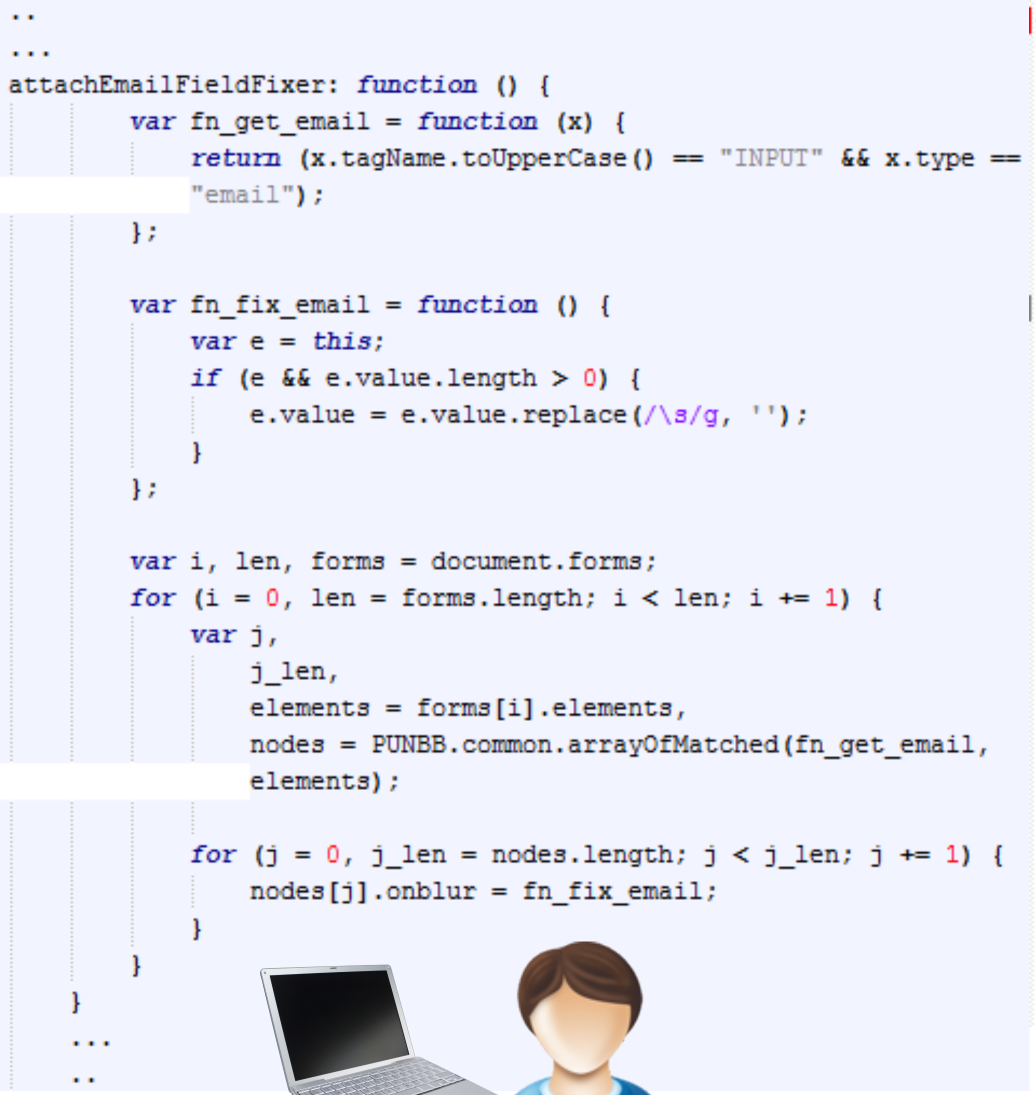
Confirmation Page

Congratulations!

Your account has been unsubscribed
...

```
ublic class FieldChecks {
 ...
 public boolean validateRequired
Object bean,
     Field field, ..){
   String value = evaluateBean(bean,
ield);
   if( (value==null) || (value.trim
).length()==0) ){
     return false;
   }
   else{
     return true;
   }
 }
 ...
```

Differential Analysis: Verification without Specification



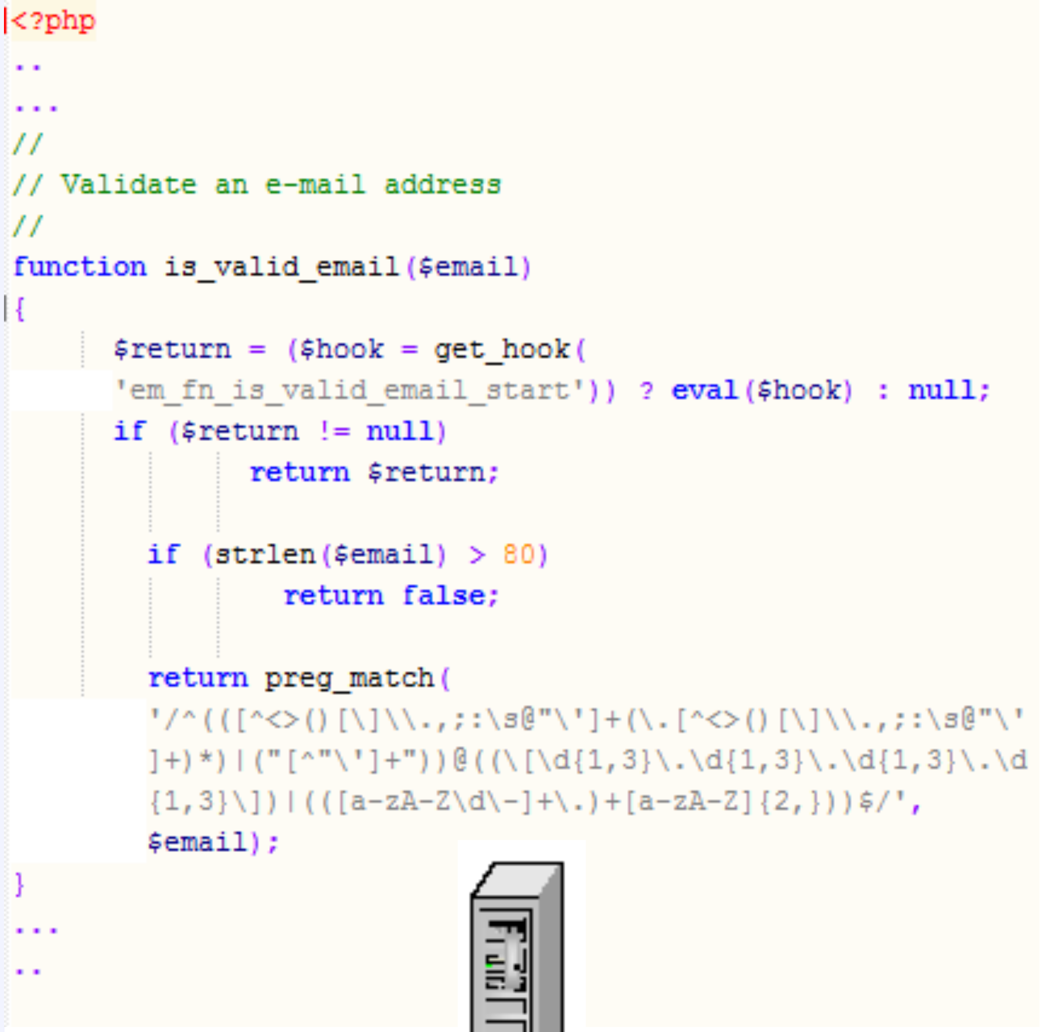
```
..
...
attachEmailFieldFixer: function () {
        var fn_get_email = function (x) {
            return (x.tagName.toUpperCase() == "INPUT" && x.type ==
            "email");
        };

        var fn_fix_email = function () {
            var e = this;
            if (e && e.value.length > 0) {
                e.value = e.value.replace(/\s/g, '');
            }
        };

        var i, len, forms = document.forms;
        for (i = 0, len = forms.length; i < len; i += 1) {
            var j,
                j_len,
                elements = forms[i].elements,
                nodes = PUNBB.common.arrayOfMatched(fn_get_email,
                elements);

            for (j = 0, j_len = nodes.length; j < j_len; j += 1) {
                nodes[j].onblur = fn_fix_email;
            }
        }
    }
    ...
    ..
```

Client-side



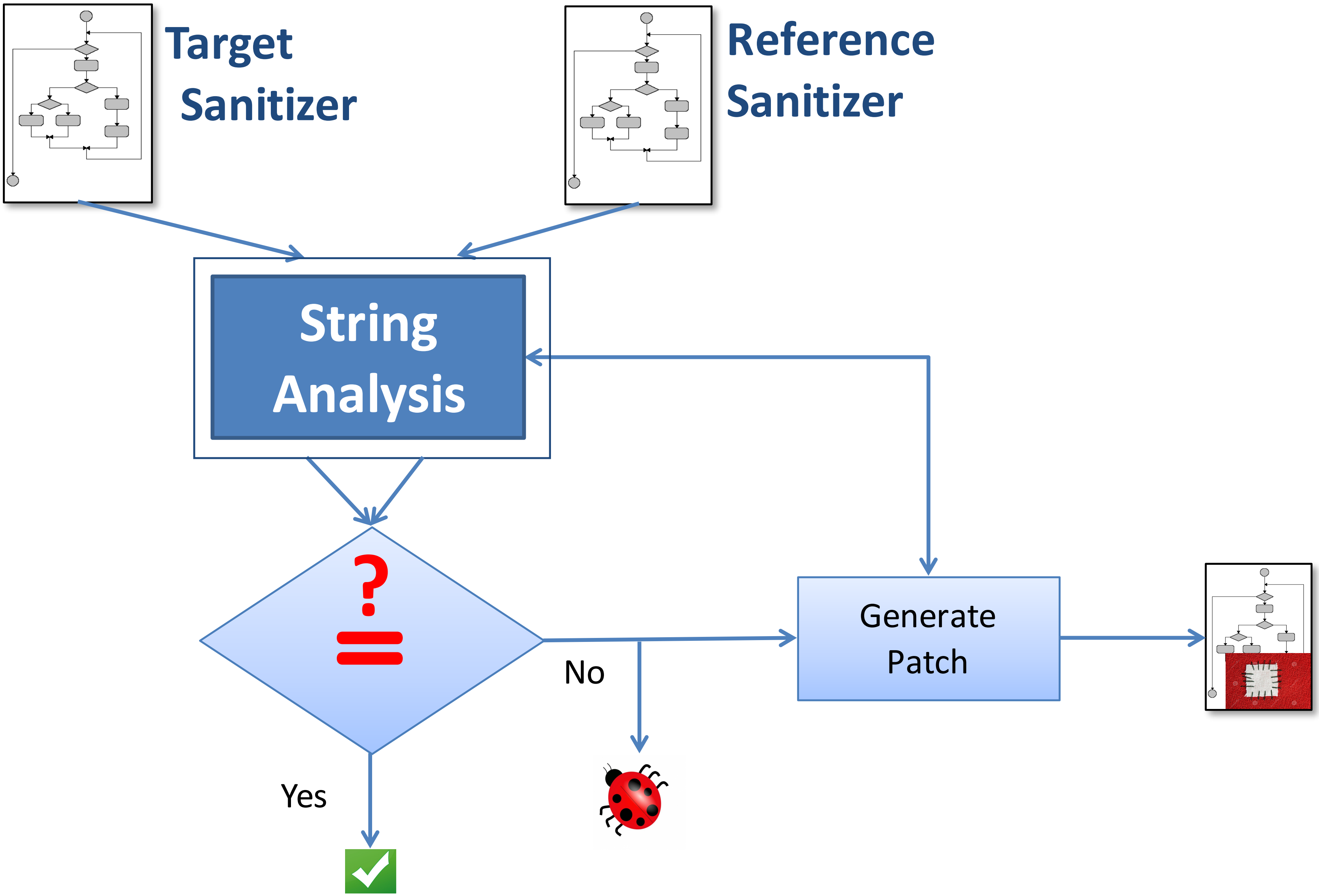
```
<?php
..
...
//
// Validate an e-mail address
//
function is_valid_email($email)
{
        $return = ($hook = get_hook(
        'em_fn_is_valid_email_start')) ? eval($hook) : null;
        if ($return != null)
                return $return;

            if (strlen($email) > 80)
                return false;

            return preg_match(
            '/^(([^<>()[\]\\.,;:\s@"\']+(\.[^<>()[\]\\.,;:\s@"\'
            ]+)*)|("[^"\']+"))@((\[\d{1,3}\.\d{1,3}\.\d{1,3}\.\d
            {1,3}\])|(([a-zA-Z\d\-]+\.)+[a-zA-Z]{2,}))$/',
            $email);
}
...
..
```

Server-side

Differential Analysis and Repair



Target Sanitizer

Reference Sanitizer

String Analysis

$\overset{?}{=}$

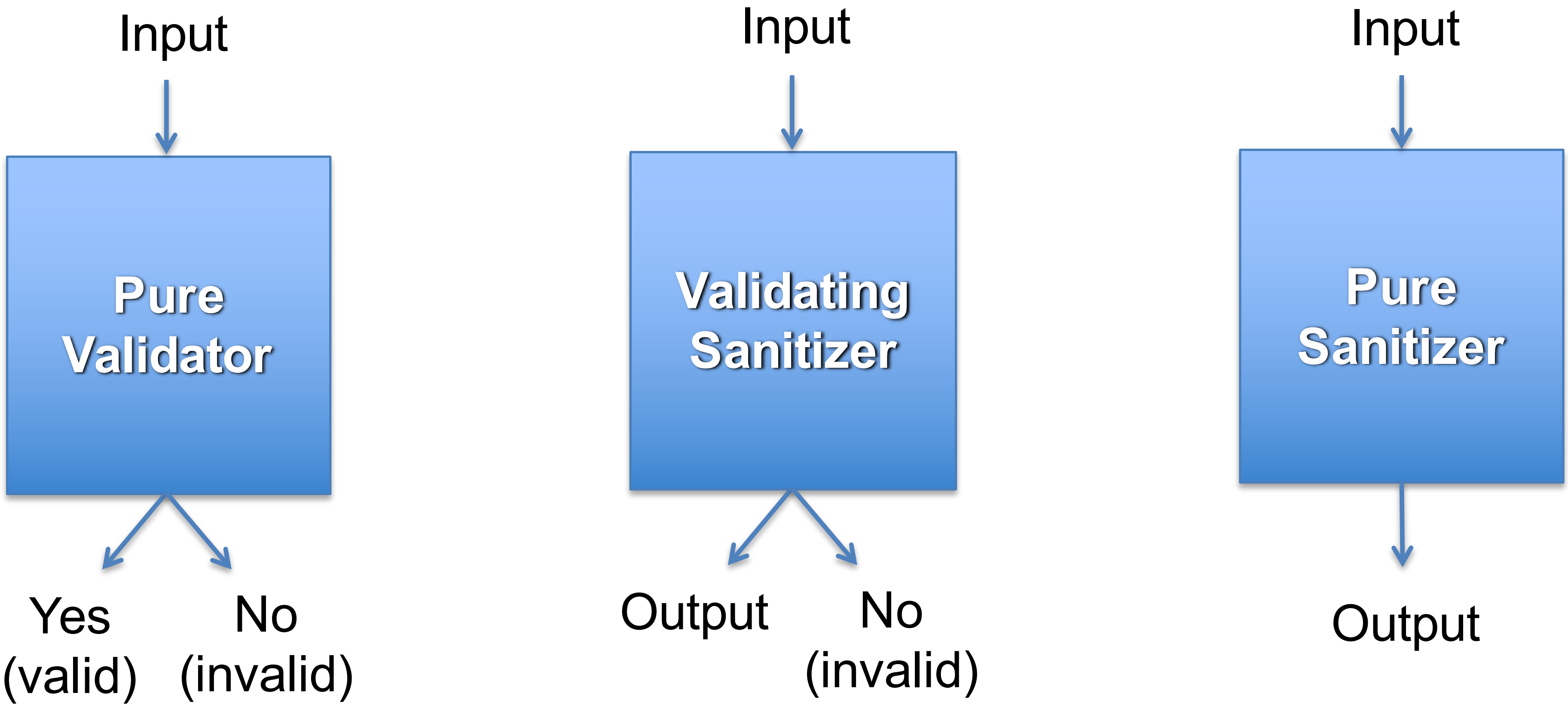
No

Yes

Generate Patch

Categorizing Validation and Sanitization

- There are three types of input validation and sanitization functions



Input

Pure Validator

Yes (valid)

No (invalid)

Input

Validating Sanitizer

Output

No (invalid)

Input

Pure Sanitizer

Output

Most General

A Javascript/Java Input Validation Function

“ “

```
function validateEmail(form) {
  var emailStr = form["email"].value;
  if(emailStr.length == 0) {
     return true;
  }
  var r1
  var r2

  i
        2.
  }
  return
}
```

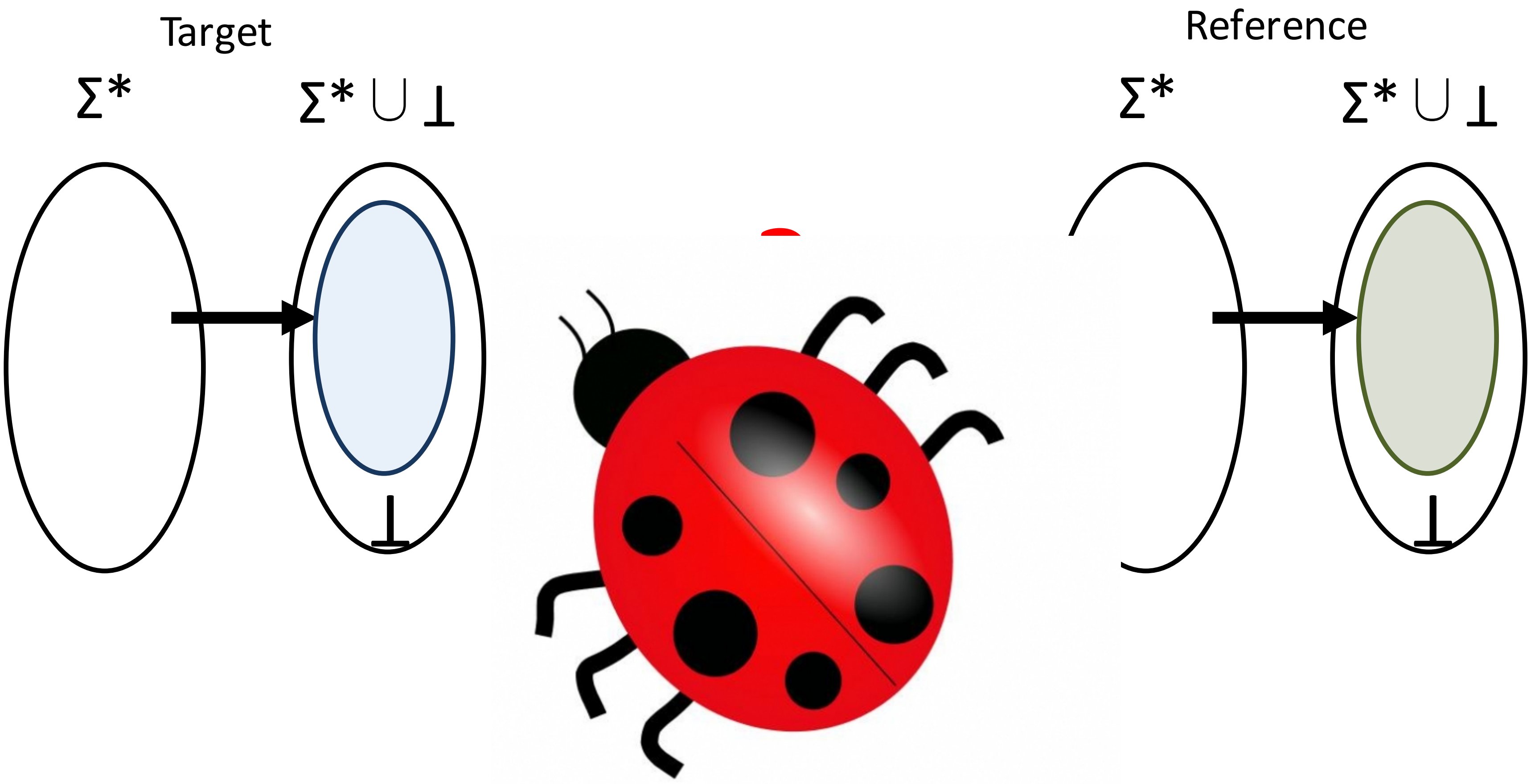
✗

“ “

```
public boolean validateEmail(Object bean, Field f, ..) {
  String val = ValidatorUtils.getValueAsString(bean, f);
  Perl5Util u = new Perl5Util();
  if (!(val == null || val.trim().length == 0)) {
    if ((!u.match("/( )|(@.*@)|(@\\.)/", val)) &&
          u.match("/^[\\w]+@([\\w]+\\.[\\w]{2,4})$/”,
                  val)){
      return true;
    } else {
      return false;
    }
  }
  return true;
}
```

✓

1st Step: Find Inconsistency



Output difference:
Strings returned by target
but not by reference

Differential Analysis Evaluation

- Analyzed a number of Java EE web applications
  - Only looking for differences (inconsistencies)

Name	URL
JGOSSIP	http://sourceforge.net/projects/jgossipforum/
VEHICLE	http://code.google.com/p/vehiclemanage/
MEODIST	http://code.google.com/p/meodist/
MYALUMNI	http://code.google.com/p/myalumni/
CONSUMER	http://code.google.com/p/consumerbasedenforcement
TUDU	http://www.julien-dubois.com/tudu-lists
JCRBIB	http://code.google.com/p/jcrbib/

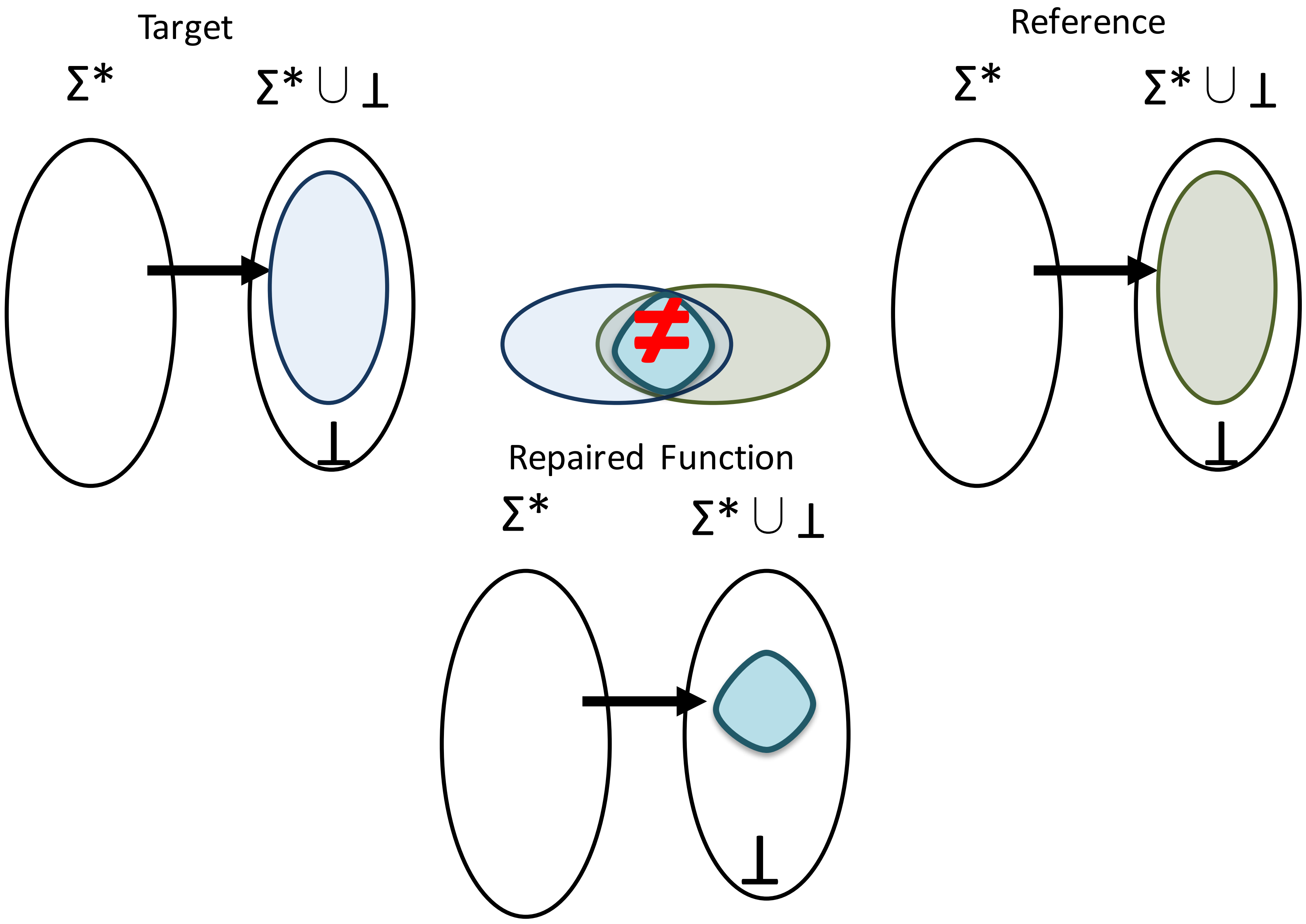
Analysis Phase Time Performance & Inconsistencies That We Found

Subject	Time (s)	$A_{C\text{-}S}$	$A_{S\text{-}C}$
JGossip	3.2	9	2
Vehicle	1.5	0	0
MeoDist	1.7	0	0
MyAlumni	2.9	141	0
Consumer	1.0	7	0
Tudu	0.6	11	0
JcrBib	1.2	45	0

Analysis Phase Memory Usage

Subject	Client-Side DFA							Server-Side DFA						
	Avr size (mb)	Min		Max		Avr		Avr size (mb)	Min		Max		Avr	
		S	B	S	B	S	B		S	B	S	B	S	B
JGOSSIP	6.0	4	10	35	706	6	39	6.1	4	24	35	706	6	41
VEHICLE	4.8	4	24	7	41	5	26	4.8	4	24	7	41	5	26
MEODIST	5.7	5	25	5	25	5	25	5.7	5	25	5	25	5	25
MYALUMNI	3.2	4	10	4	10	4	10	3.2	3	24	5	25	5	25
CONSUMER	5.3	4	10	17	132	5	25	5.3	4	24	17	132	7	41
TUDU	6.1	4	10	4	10	4	10	6.1	3	24	23	264	8	68
JCRBIB	5.4	4	10	4	10	4	10	5.4	5	25	5	25	5	25

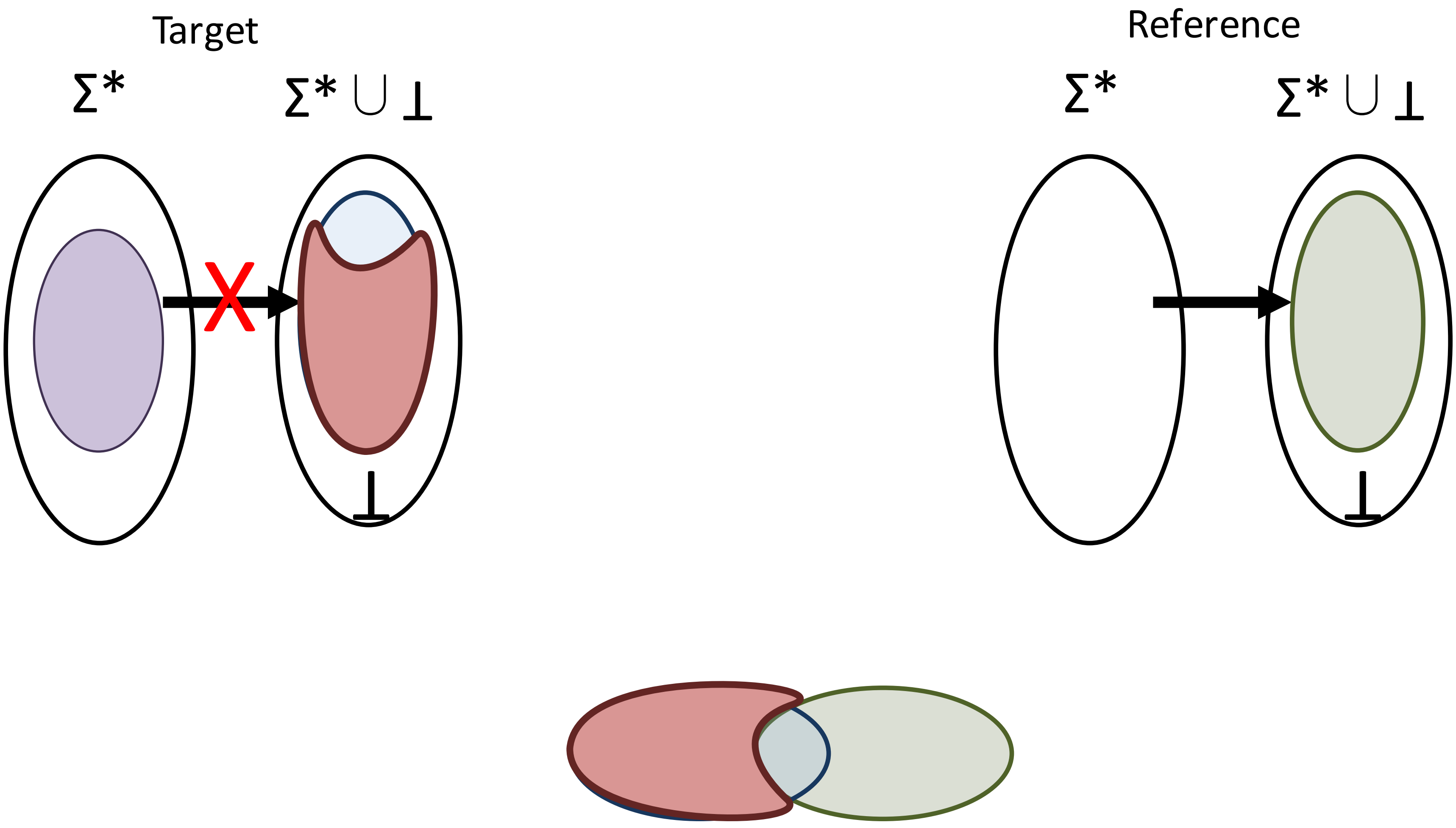
2nd Step: Differential Repair



Composing Sanitizers?

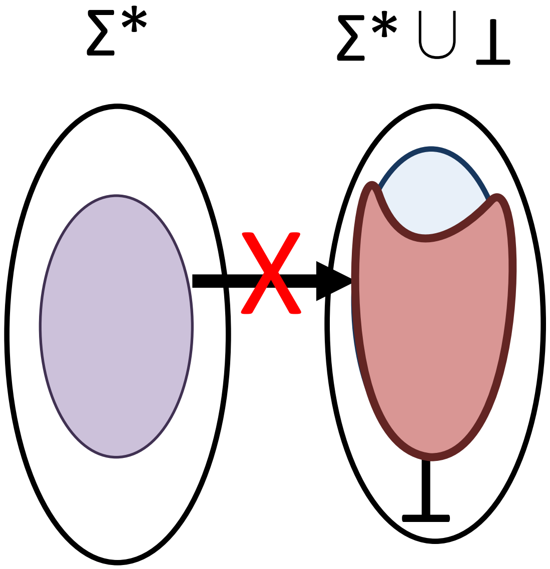
- Can we run the two sanitizers one after the other?
- Does not work due to lack of **Idempotency**
  - Both sanitizers escape ’ with \
  - Input ab’c
  - 1st sanitizer → ab\’c
  - 2nd sanitizer → ab\\’c
    - Security problem (double escaping)
- We need to find the difference

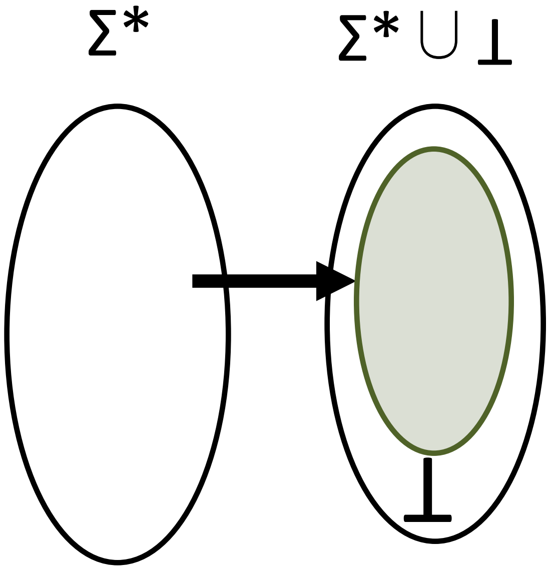
How to repair?



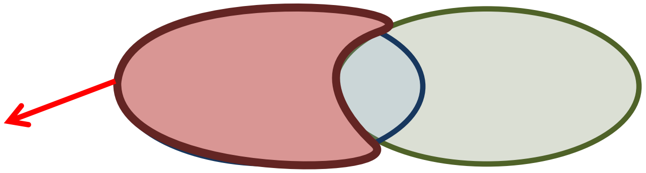
```
function target($x){
        $x = preg_replace(“’”, “\’”,
$x); return $x;
}
```

```
function reference($x){
        $x = preg_replace(“<“, “”,
$x);
        if (strlen($x) < 4)
                return $x;
        else
                die(“error”);
}
```





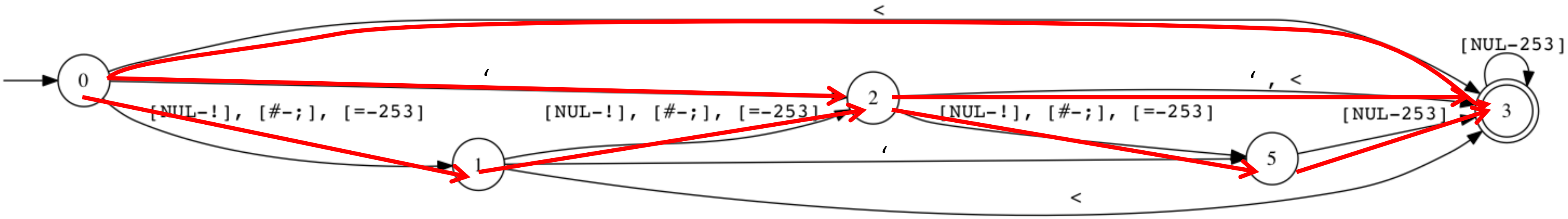
Output difference:
Strings returned by target
but not by reference



```
function target($x){
        $x = preg_replace(“’”, “\’”,
$x); return $x;
}
```

```
function reference($x){
        $x = preg_replace(“<“, “”,
$x);
        if (strlen($x) < 4)
              return $x;
        else
              die(“error”);
}
```

Set of input strings that resulted in the difference



Input	Target	Reference	Diff Type
“<“	“<“	“”	Sanitization
“’’”	“\’\’”	“’’”	Sanitization + Length
“abcd”	“abcd”	⊥	Validation

```
function target($x){
     $x = str_replace(“'”, “\'”, $x);
     return $x;
}
```

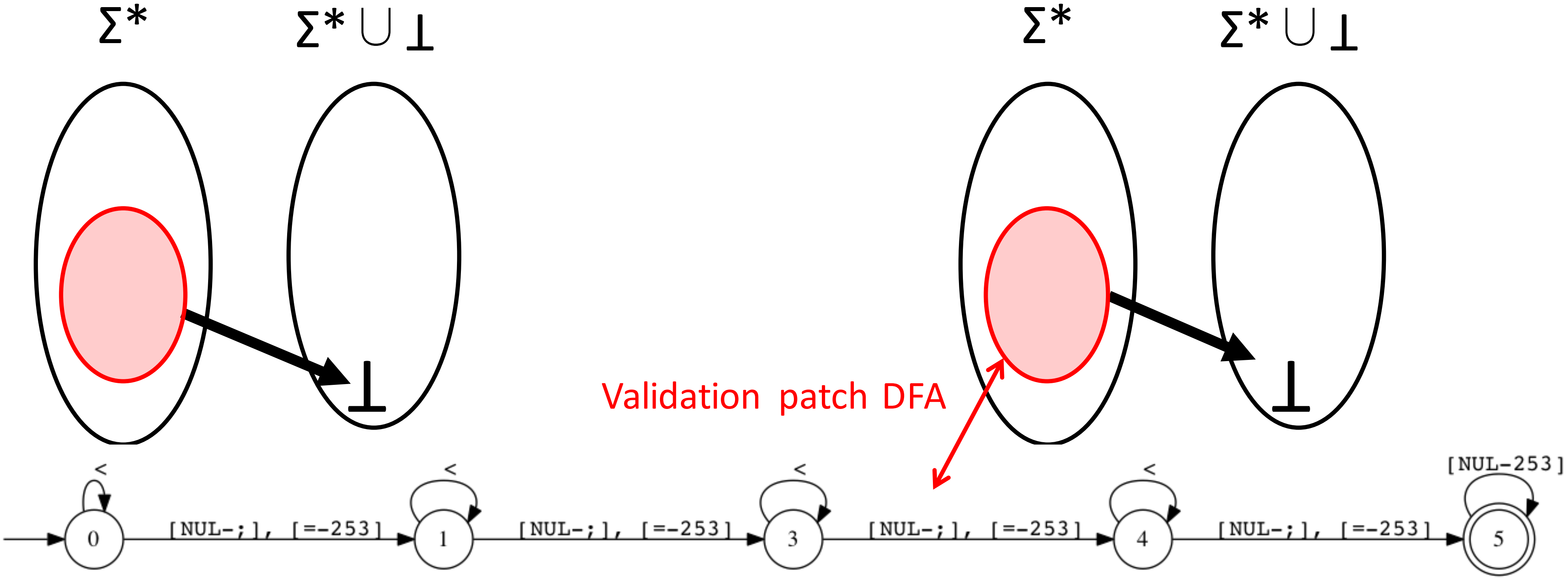
```
function reference($x){
     $x = str_replace(“<“, “”,
$x);
     if (strlen($x) < 4)
          return $x;
     else
          die(“error”);
}
```

- Mincut results in deleting everything
  - “foo” → “”
- Why?
  - You can not remove a validation difference using a sanitization patch

(1) Validation Patch

```
function valid_patch($x){
     if (stranger_match1($x))
          die(“error”);
}
function target($x){
     $x = str_replace(“’”, “\’”,
$x); return $x;
}
```

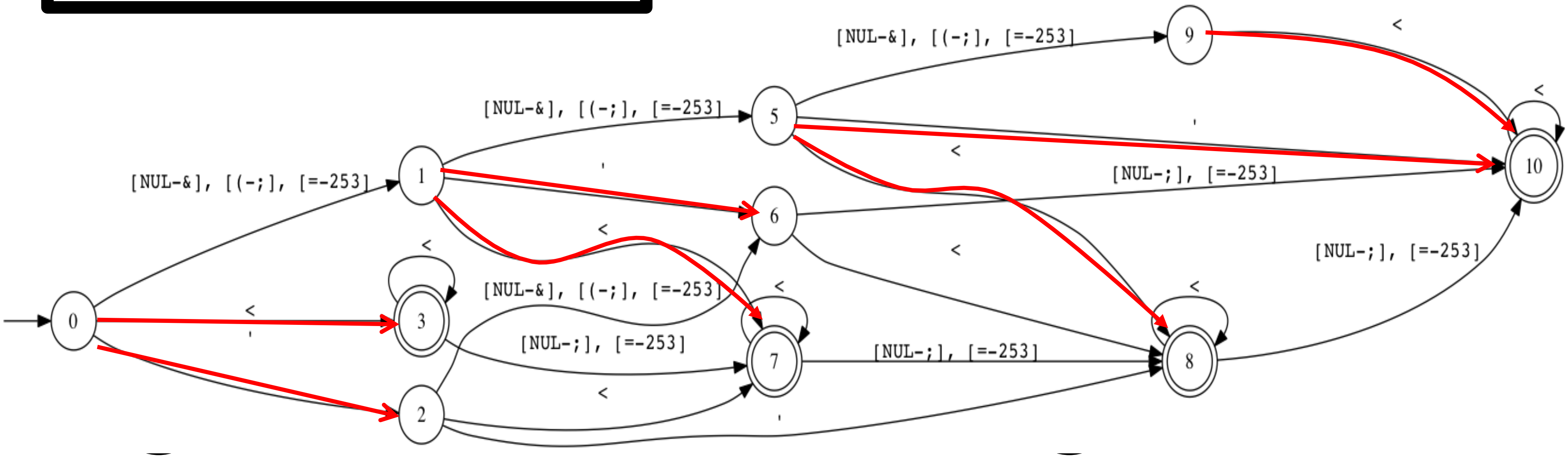
```
function reference($x){
     $x = str_replace(“<“, “”,
$x);
     if (strlen($x) < 4)
          return $x;
     else
          die(“error”);
}
```



```
function valid_patch($x){
     if (stranger_match1($x))
          die(“error”);
}
```

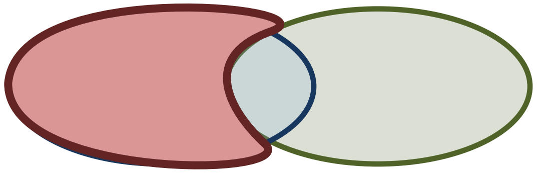
```
function target($x){
     $x = str_replace(“’”, “\’”,
$x); return $x;
}
```

```
function reference($x){
     $x = str replace(“<“, “”,
$x);
     if (strlen($x) < 4)
          return $x;
     else
          die(“error”);
}
```



MinCut = { ‘, <}

“fo ’ ” → “fo\ ’ ”



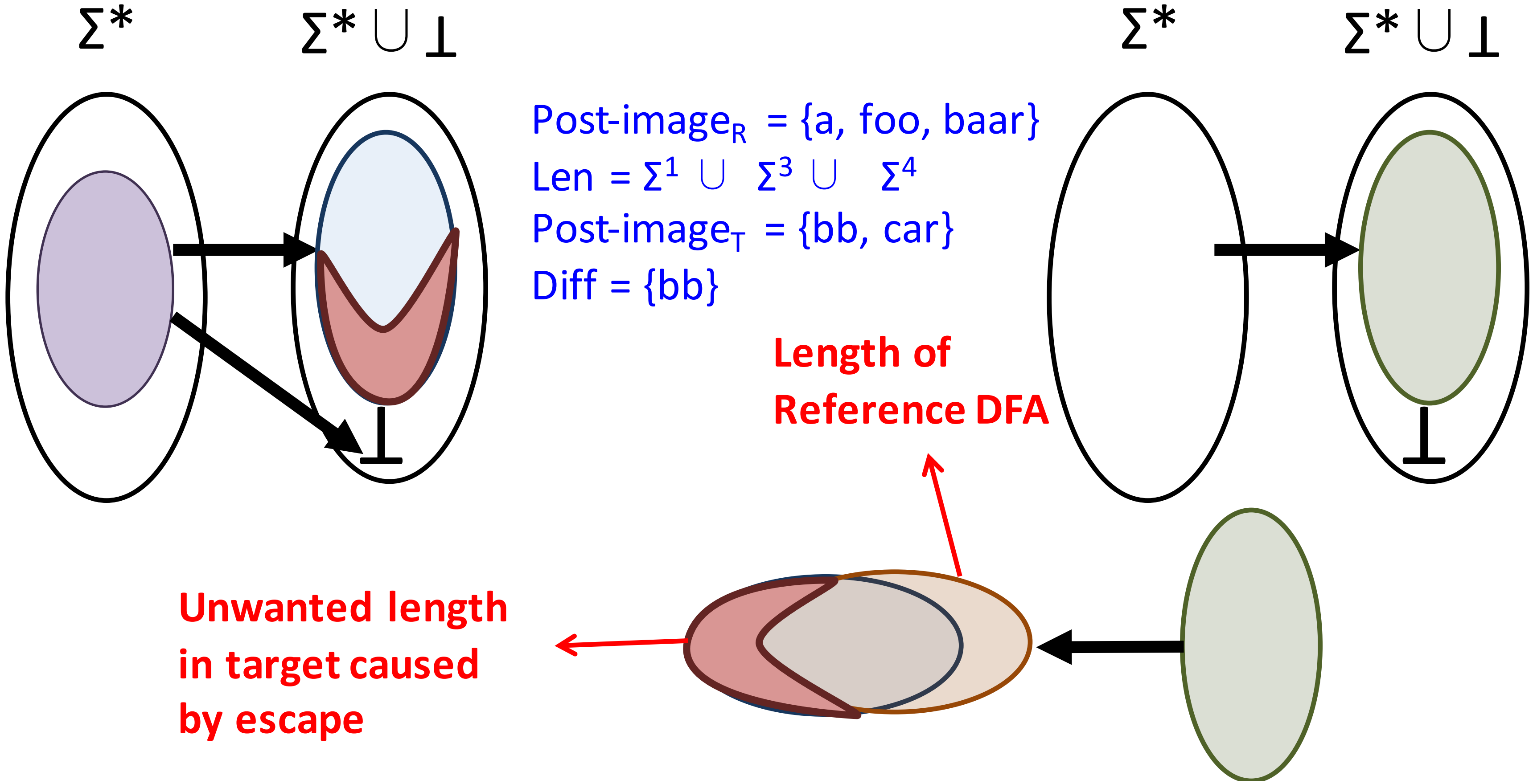
(2) Length Patch

```
function valid_patch($x){
     if (stranger_match1($x))
          die(“error”);
}
```

```
function valid_patch($x){
     if (stranger_match1($x))
          die(“error”);
}
```

```
function target($x){
     $x = str_replace(“’”, “\’”, $x);
     return $x;
}
```

```
function reference($x){
     $x = str_replace(“<“, “”, $x);
     if (strlen($x) < 4)
          return $x;
     else
          die(“error”);
}
```



Post-image_R = {a, foo, baar}

Len = $\Sigma^1 \cup \Sigma^3 \cup \Sigma^4$

Post-image_T = {bb, car}

Diff = {bb}

Length of Reference DFA

Unwanted length in target caused by escape

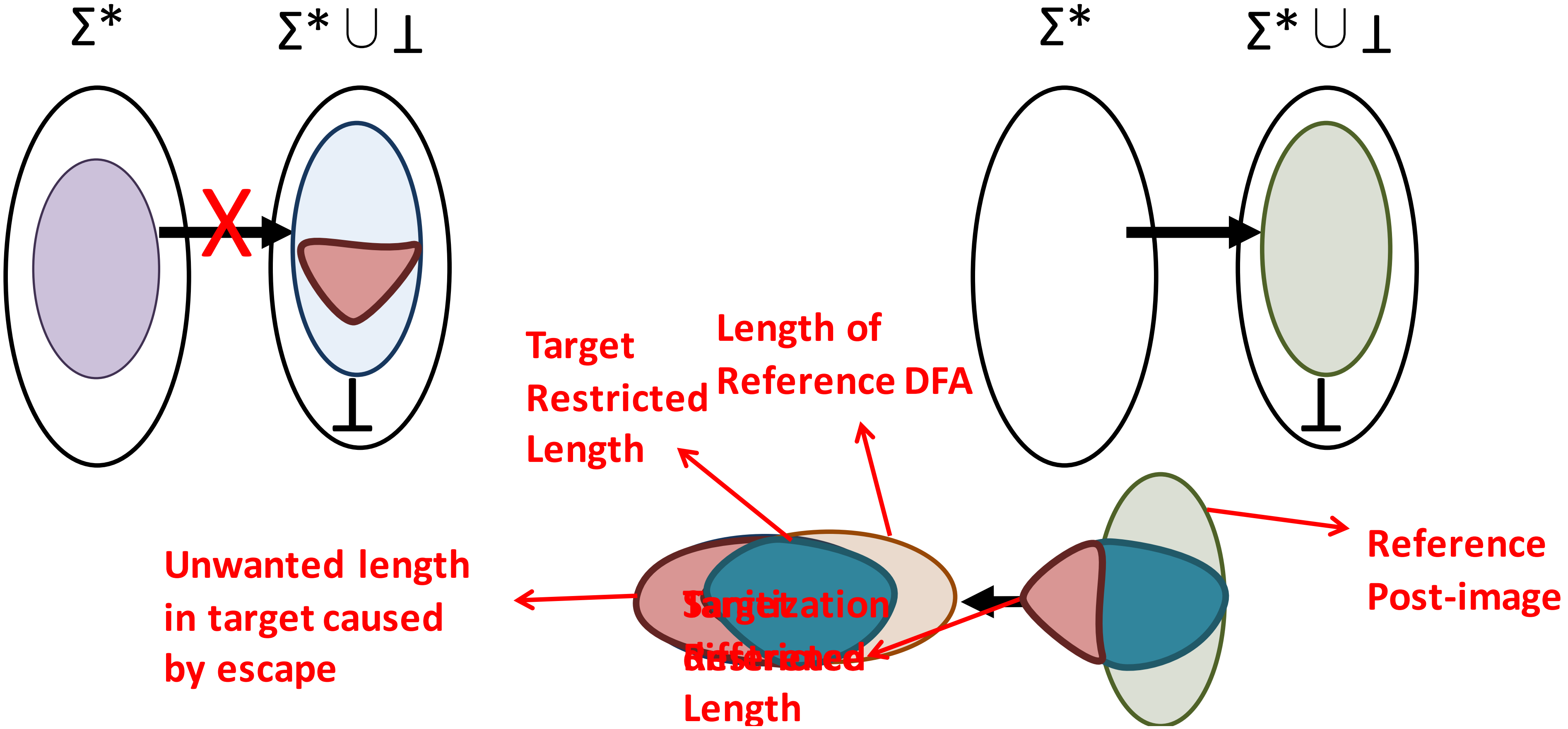
(3) Sanitization Patch

```
function valid_patch($x){
     if (stranger_match1($x))
          die(“error”);
}
```

```
function length_patch($x){
     if (stranger_match2($x))
          die(“error”);
}
```

```
function target($x){
     $x = str_replace(“’”, “\’”,
$x); return $x;
}
```

```
function reference($x){
     $x = str_replace(“<“, “”,
$x);
     if (strlen($x) < 4)
          return $x;
     else
          die(“error”);
}
```



Target Restricted Length

Length of Reference DFA

Unwanted length in target caused by escape

Target Restricted

Sanitization difference

Length

Reference Post-image

(3) Sanitization Patch

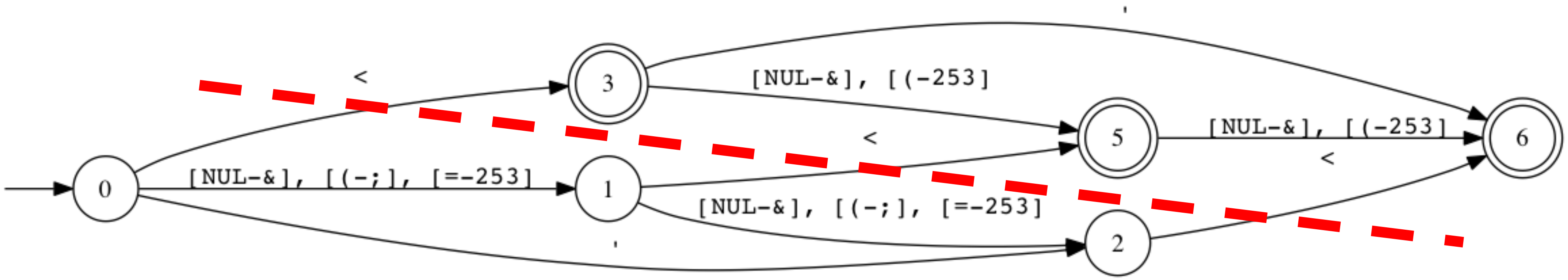
```
function valid_patch($x){
     if (stranger_match1($x))
          die(“error”);
}
```

```
function length_patch($x){
     if (stranger_match2($x))
          die(“error”);
}
```

```
function sanit_patch($x){
     $x = str_replace(“<“, “”,
$x); return $x;
}
```

```
function target($x){
     $x = str_replace(“’”, “\’”,
$x); return $x;
}
```

```
function reference($x){
     $x = str_replace(“<“,  “”,
$x);
     if (strlen($x)  < 4)
          return $x;
     else
          die(“error”);
}
```



MinCut = {<}

MinCut Heuristics

- We use two heuristics for mincut
- Trim:
  - Only if mincut contain space character
  - Test if reference Post-Image is does not have space at the beginning and end
  - Assume it is `trim`()
- Escape:
  - Test if reference Post-Image escapes the mincut characters

Differential Repair Evaluation

- We ran the differential patching algorithm on 5 PHP web applications

Name	Description
PHPNews v1.3.0	News publishing software
UseBB v1.0.16	forum software
Snipe Gallery v3.1.5	Image management system
MyBloggie v2.1.6	Weblog system
Schoolmate v1.5.4	School administration software

Number of Patches Generated

Mapping	# Pairs	# Valid.	# Length.	# Sanit.
Client-Server	122	61	11	0
Server-Client	122	53	2	30
Server-Server	206	49	0	33
Client-Client	19	34	0	5

Sanitization Patch Results

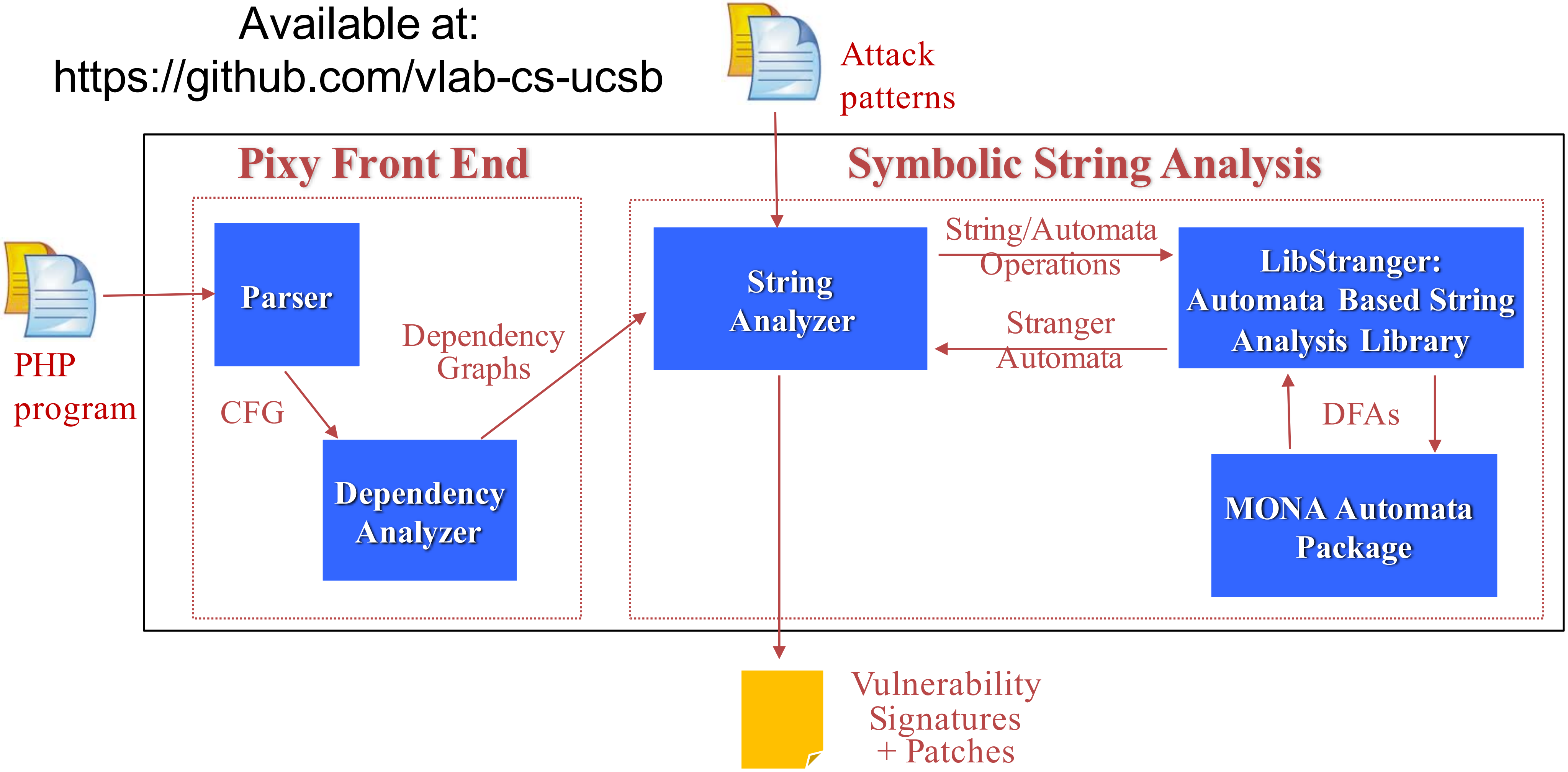
Mapping	mincut Avr. size	mincut Max size	#trim	#escape	#delete
Server-Client	4	10	15	10	20
Server-Server	3	5	23	0	20
Client-Client	7	15	3	0	2

Time and Memory Performance of Differential Repair Algorithm

Repair phase	DFA size (#bddnodes)		peak DFA size (#bddnodes)		time (seconds)	
	avg	max	avg	max	avg	max
Valid.	997	32,650	484	33,041	0.14	4.37
Length	129,606	347,619	245,367	4,911,410	9.39	168.00
Sanit.	2,602	11,951	4,822	588,127	0.17	14.00

Stranger & LibStranger: String Analysis Toolset

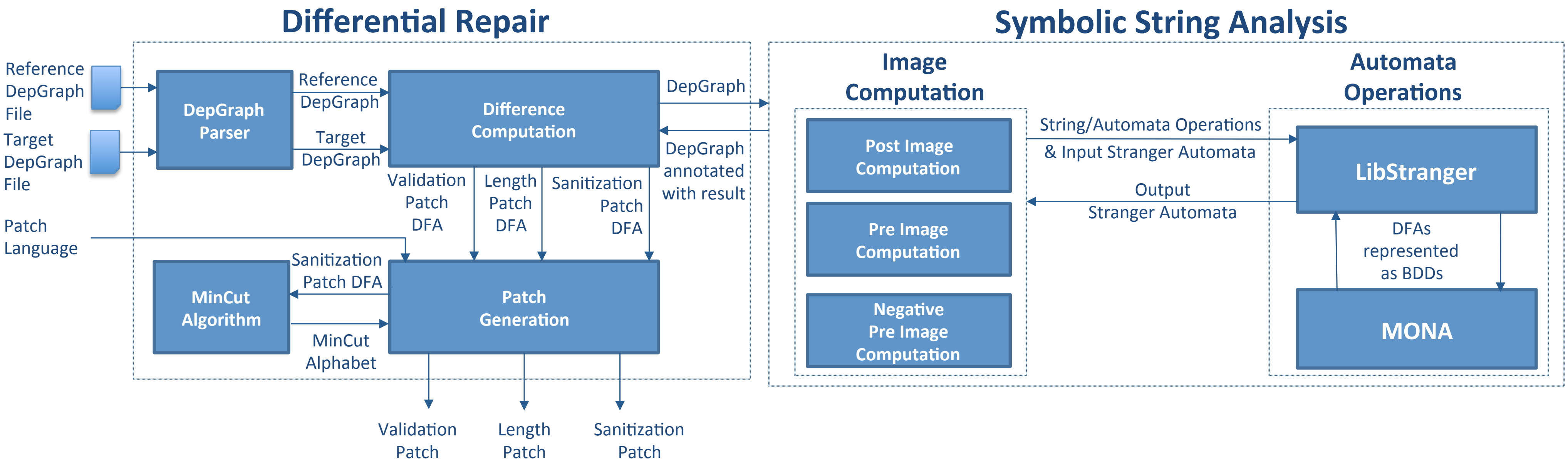
Available at:
https://github.com/vlab-cs-ucsb



- Uses Pixy [Jovanovic et al., 2006] as a PHP front end
- Uses MONA [Klarlund and Møller, 2001] automata package for automata manipulation

SemRep: A Differential Repair Tool

- Available at: https://github.com/vlab-cs-ucsb



- A paper [Kausler, Sherman, ASE’14] that compares sound string constraint solvers: (JSA, LibStranger, Z3-Str, ECLIPSE-Str), reports that **LibStranger is the best**!

STRING ANALYSIS BIBLIOGRAPHY

Automata based String Analysis

- *A static analysis framework for detecting SQL injection vulnerabilities* [Fu et al., COMPSAC’ 07]
- *Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications* [Balzarotti et al., S&P 2008]
- *Symbolic String Verification: An Automata-based Approach* [Yu et al., SPIN’08]
- *Symbolic String Verification: Combining String Analysis and Size Analysis* [Yu et al., TACAS’09]
- *Rex: Symbolic Regular Expression Explorer* [Veanes et al., ICST’10]
- *Stranger: An Automata-based String Analysis Tool for PHP* [Yu et al., TACAS’10]
- *Relational String Verification Using Multi-Track Automata* [Yu et al., CIAA’10, IJFCS’11]
- *Path- and index-sensitive string analysis based on monadic second-order logic* [Tateishi et al., ISSTA’11]

Automata based String Analysis

- *An Evaluation of Automata Algorithms for String Analysis* [Hooimeijer et al., VMCAI'11]
- *Fast and Precise Sanitizer Analysis with BEK* [Hooimeijer et al., Usenix'11]
- *Symbolic finite state transducers: algorithms and applications* [Veanes et al., POPL'12]
- *Static Analysis of String Encoders and Decoders* [D'antoni et al. VMCAI'13]
- *Applications of Symbolic Finite Automata*. [Veanes, CIAA'13]
- *Automata-Based Symbolic String Analysis for Vulnerability Detection* [Yu et al., FMSD'14]

Grammar based String Analysis

- *Precise Analysis of String Expressions* [Christensen et al., SAS' 03]
- *Java String Analyzer (JSA)* [Moller et al.]
- *Static approximation of dynamically generated Web pages* [Minamide, WWW' 05]
- *PHP String Analyzer* [Minamide]
- *Grammar-based analysis string expressions* [Thiemann, TLDI'05]

Symbolic Execution for Strings

- *Abstracting symbolic execution with string analysis* [Shannon et al., MUTATION' 07]
- *Path Feasibility Analysis for String-Manipulating Programs* [Bjorner et al., TACAS' 09]
- *A Symbolic Execution Framework for JavaScript* [Saxena et al., S&P 2010]
- *Symbolic execution of programs with strings* [Redelinghuys et al., ITC'12]

String Abstractions & Widening

- *A Practical String Analyzer by the Widening Approach* [Choi et al. APLAS’06]
- *String Abstractions for String Verification* [Yu et al., SPIN’11]
- *A Suite of Abstract Domains for Static Analysis of String Values* [Constantini et al., SP&E’13, Software Practice & Experience’15]

String size analysis

- *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*. [Wagner et al., NDSS 2000]
- *Buffer overrun detection using linear programming and static analysis.* [Ganapathy et al. ACM CCS 2003]
- *CSSV: towards a realistic tool for statically detecting all buffer overflows in C*. [Dor et al. PLDI 2003]

String Constraint Solvers

- *Reasoning about Strings in Databases* [Grahne at al., JCSS’99]
- *Constraint Reasoning over Strings* [Golden et al., CP’03]
- *A decision procedure for subset constraints over regular languages* [Hooimeijer et al., PLDI’09]
- *Strsolve: solving string constraints lazily* [Hooimeijer et al., ASE’10, ASE’12]
- *An SMT-LIB Format for Sequences and Regular Expressions* [Bjorner et al., SMT’12]
- *Z3-Str: A Z3-Based String Solver for Web Application Analysis* [Zheng et al., ESEC/FSE’13]
- *Word Equations with Length Constraints: What's Decidable?* [Ganesh et al., HVC’12]
- *(Un)Decidability Results for Word Equations with Length and Regular Expression Constraints* [Ganesh et al., ADDCT’13]

String Constraint Solvers

- *A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions* [Liang et al., CAV’14]
- *String Constraints for Verification* [Abdulla et al., CAV’14]
- *S3: A Symbolic String Solver for Vulnerability Detection in Web Applications* [Trinh et al., CCS’14]
- *Evaluation of String Constraint Solvers in the Context of Symbolic Execution* [Kausler et al., ASE’14]
- *Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints* [Zheng et al., CAV’15]
- *A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings* [Liang et al., FroCos’15]
- *Norn: An SMT Solver for String Constraints* [Abdulla et al., CAV’15]

Bounded String Constraint Solvers

•*HAMPI: a solver for string constraints* [Kiezun et al., ISSTA' 09]

•*HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection* [Ganesh et al., CAV'11]

•*HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars* [Kiezun et al., TOSEM'12]

•*Kaluza* [Saxena et al.]

•*PASS: String Solving with Parameterized Array and Interval Automaton* [Li & Ghosh, HVC'14]

Model Counting for String Constraints

- *A model counter for constraints over unbounded strings* [Luu et al., PLDI’14]
- *Automata-based model counting for string constraints* [Aydin et al., CAV'15]

String Analysis for Vulnerability Detection

- *AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks* [Halfond et al., ASE’05]
- *Preventing SQL injection attacks using AMNESIA*. [Halfond et al., ICSE’06]
- *Sound and precise analysis of web applications for injection vulnerabilities* [Wassermann et al., PLDI’07]
- *Static detection of cross-site scripting vulnerabilities* [Su et al., ICSE’08]
- *Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses* [Yu et al., ASE’09]
- *Verifying Client-Side Input Validation Functions Using String Analysis* [Alkhalaf et al., ICSE’12]

String Analysis for Test Generation

- *Dynamic test input generation for database applications* [Emmi et al., ISSTA’07]
- *Dynamic test input generation for web applications.* [Wassermann et al., ISSTA’08]
- *JST: an automatic test generation tool for industrial Java applications with strings* [Ghosh et al., ICSE’13]
- *Automated Test Generation from Vulnerability Signatures* [Aydin et al., ICST’14]

String Analysis for Analyzing Dynamically Generated Code

- *Improving Test Case Generation for Web Applications Using Automated Interface Discovery* [Halfond et al. FSE’07]
- *Automated Identification of Parameter Mismatches in Web Applications* [Halfond et al. FSE’08]
- *Building Call Graphs for Embedded Client-Side Code in Dynamic Web Applications* [Nguyen et al. FSE'15]
- *Varis: IDE Support for Embedded Client Code in PHP Web Applications* [Nguyen et al. ICSE'15]

String Analysis for Specifications

- Lightweight String Reasoning for OCL [Buttner et al., ECMFA’12]
- Lightweight String Reasoning in Model Finding [Buttner et al., SSM’13]

String Analysis for Program Repair

- *Patching Vulnerabilities with Sanitization Synthesis* [Yu et al., ICSE’11]
- *Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving* [Samimi et al., 2012]
- *Patcher: An Online Service for Detecting, Viewing and Patching Web Application Vulnerabilities* [Yu et al., HICSS’14]

Differential String Analysis

- *Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications* [Bisht et al., CCS'10]
- *Waptec: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction.* [Bisht et al., CCS'11]
- *ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies* [Alkhalaf et al., ISSTA'12]
- *Semantic Differential Repair for Input Validation and Sanitization* [Alkhalaf et al. ISSTA'14]

Coming Soon:

- A book on String Analysis!