Automatic Verification of String Manipulating Programs

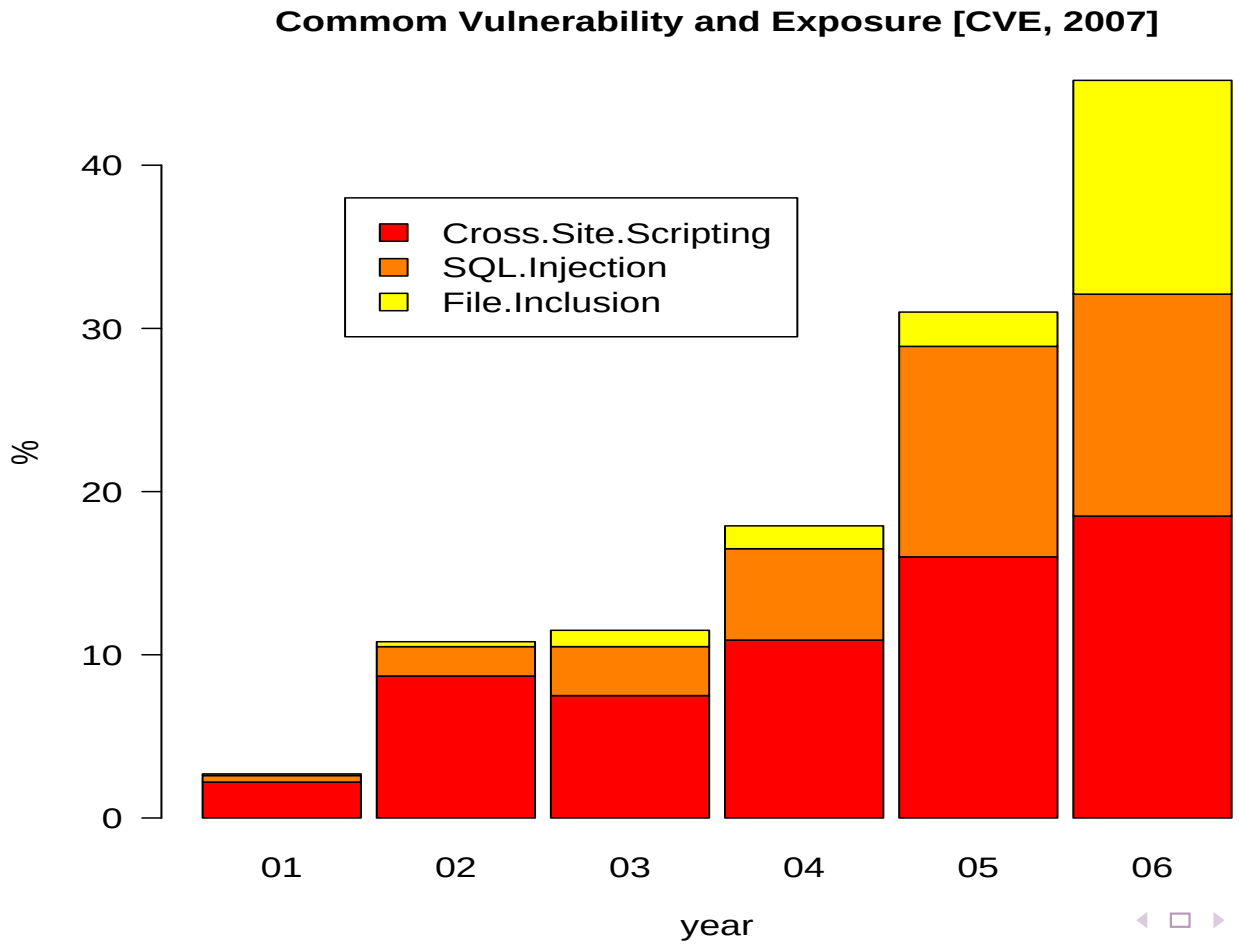
Fang Yu

VLab, Department of Computer Science
University of California, Santa Barbara, USA

May 19, 2010

Web Application Vulnerabilities

String manipulation errors are the most common cause of security vulnerabilities



Web Application Vulnerabilities

- The top three vulnerabilities of the Open Web Application Security Project (OWASP)'s top ten list. [OWASP, 2007]
  1. Cross Site Scripting (XSS)
  2. Injection Flaws (such as SQL Injection)
  3. Malicious File Executions

After three years...

- The top two vulnerabilities of the OWASPs top ten list. [OWASP, 2010]
  1. Injection Flaws (such as SQL Injection)
  2. Cross Site Scripting (XSS)

Cross Site Scripting (XSS) Attack

A PHP Example:

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

- The *echo* statement in line **4** can contain a Cross Site Scripting (XSS) vulnerability

XSS Attack

An attacker may provide an input that contains <script and execute the malicious script.

- 1:<?php
- 2: $www = <script ... >;
- 3: $l_otherinfo = "URL";
- 4: echo "<td>" . $l_otherinfo . ": " .<script ... >. "</td>";
- 5:?>

Is it Vulnerable?

A simple taint analysis, e.g., [Huang et al. WWW04], would report this segment as vulnerable using *taint propagation*.

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

Is it Vulnerable?

Add a sanitization routine at line s.

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

- Taint analysis will assume that $www is untainted after the routine, and conclude that the segment is not vulnerable.

Sanitization Routines are Erroneous

However, ereg_replace("[^A-Za-z0-9 .-@://]","",$www); does not sanitize the input properly.

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

A buggy sanitization routine

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

- A buggy sanitization routine used in MyEasyMarket-4.1 that causes a vulnerable point at line 218 in trans.php [Balzarotti et al., S&P'08]
- Our string analysis identifies that the segment is vulnerable with respect to the attack pattern: Σ^* <scriptΣ^*.

Eliminate Vulnerabilities

Input <!sc+rip!t ...> does not match the attack pattern Σ^* <scriptΣ^*, but still can cause an attack

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

Eliminate Vulnerabilities

- We generate vulnerability signature that characterizes **all** malicious inputs that may generate attacks (with respect to the attack pattern)
- The vulnerability signature for $_GET["www"] is $\Sigma^* < \alpha^* s \alpha^* c \alpha^* r \alpha^* i \alpha^* p \alpha^* t \Sigma^*$, where $\alpha \notin \{$ A-Za-z0-9 .-@:/ $\}$ and Σ is any ASCII character
- Any string accepted by this signature can cause an attack
- Any string that dose not match this signature will not cause an attack. I.e., one can filter out all malicious inputs using our signature

Prove the Absence of Vulnerabilities

Fix the buggy routine by inserting the escape character \.

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

Using our approach, this segment is proven not to be vulnerable against the XSS attack pattern: Σ^* <scriptΣ^*.

Multiple Inputs?

Things can be more complicated while there are multiple inputs.

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

- An attack string can be contributed from one input, another input, or their combination
- We can generate relational vulnerability signatures and automatically synthesize effective patches.

About This Work

We present an automata-based approach for automatic verification of string manipulating programs. Given a program that manipulates strings, we verify assertions about string variables.

- Symbolic String Vulnerability Analysis
- Relational String Analysis
- Composite String Analysis

Summary of Contributions

- An automata-based approach for analyzing string manipulating programs using symbolic string analysis. The approach features language-based replacement, fixpoint acceleration, symbolic automata encoding, and alphabet and relation abstraction techniques [SPIN'08, ASE'09]
- An automata-based string analysis tool: Stranger can automatically detect, eliminate, and prove the absence of XSS, SQLCI, and MFE vulnerabilities (with respect to attack patterns) in PHP web applications [TACAS'10]

Summary of Contributions

- A composite analysis technique that combines string analysis with size analysis showing how the precision of both analyses can be improved by using length automata [TACAS'09]
- A relational string verification technique using multi-track automata and abstraction: This approach does not only enhance the earlier results in string analysis in terms of both precison and performance, but also enables verification of properties that depend on relations among string variables [UCSB-CS-TR]

Automatic Verification of String Manipulating Programs

- Symbolic String Vulnerability Analysis
- Relational String Analysis
- Composite String Analysis

Symbolic String Vulnerability Analysis

Given a program, types of sensitive functions, and an attack pattern, we say

- A program is *vulnerable* if a sensitive function at some program point can take a string that matches the attack pattern as its input
- A program is *not vulnerable* (with respect to the attack pattern) if no such functions exist in the program

Symbolic String Vulnerability Analysis

- Converts programs to dependency graphs focusing on string manipulation operations
- Performs forward symbolic reachability analyses to detect vulnerabilities
- Performs backward symbolic reachability analysis to generate vulnerability signatures

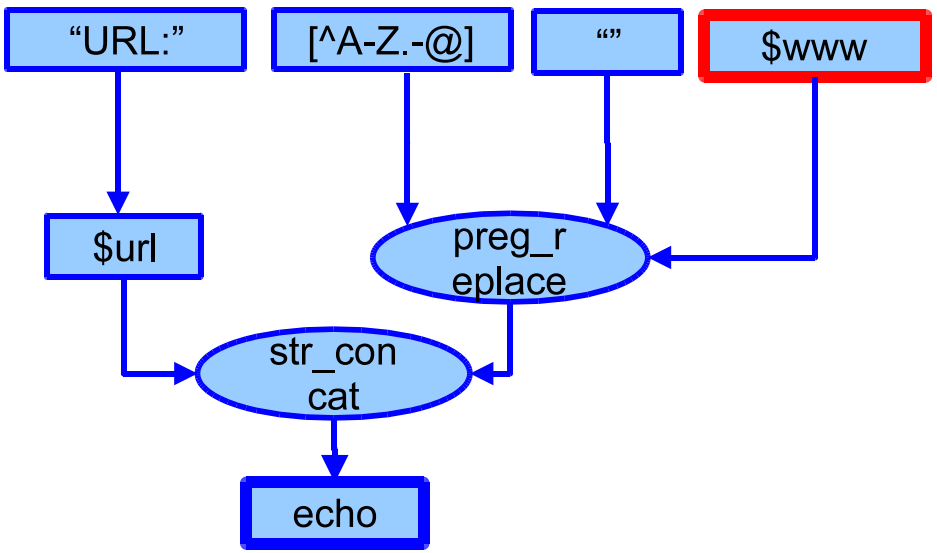
Front End

Consider the following segment.

```
<?php
1: $www = $_GET["www"];
2: $url = "URL:";
3: $www = preg_replace("[^A-Z.-@]","",$www);
4: echo $url. $www;
?>
```

Front End

A dependency graph specifies how the values of input nodes flow to a sink node (i.e., a sensitive function)



NEXT: Compute all possible values of a sink node

Detecting Vulnerabilities

- Associates each node with an automaton that accepts an over approximation of its possible values
- Uses automata-based forward symbolic analysis to identify the possible values of each node
- Uses *post*-image computations of string operations:
  - `postConcat(`M_1`,` M_2`)` returns M, where $M{=}M_1.M_2$
  - `postReplace(`M_1`,` M_2`,` M_3`)` returns M, where $M{=}\textsc{replace}(M_1, M_2, M_3)$

A Language-based Replacement

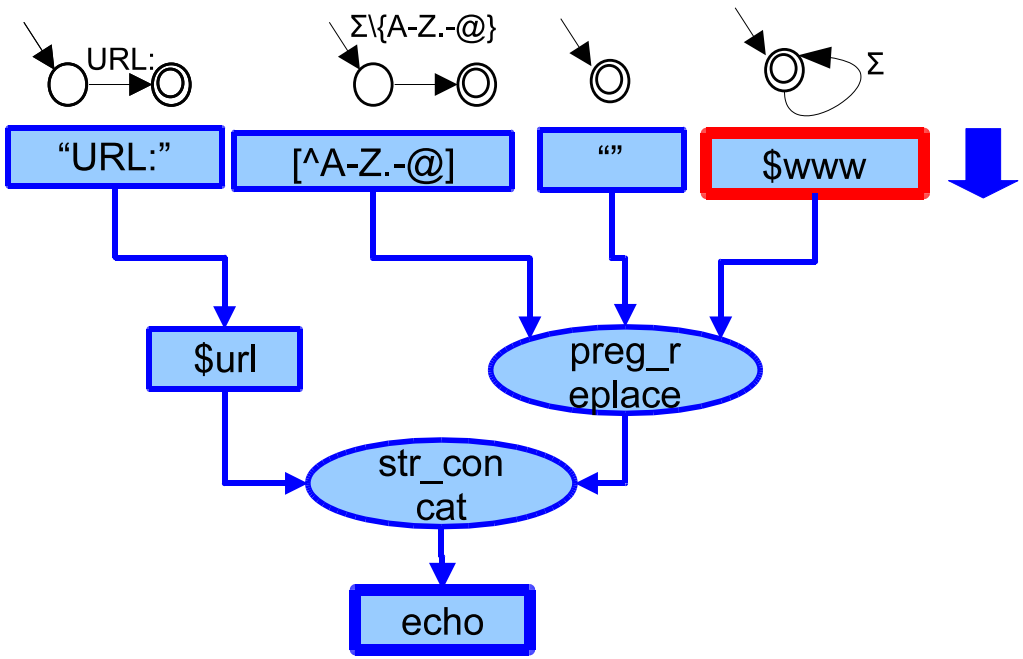
$M = \text{REPLACE}(M_1, M_2, M_3)$

- M_1, M_2, and M_3 are Deterministic Finite Automata (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.

$L(M_1)$	$L(M_2)$	$L(M_3)$	$L(M)$
{baaabaa}	{aa}	{c}	{bacbc, bcabc}
{baaabaa}	a^+	ϵ	{bb}
ba^+b	a^+	{c}	bc^+b

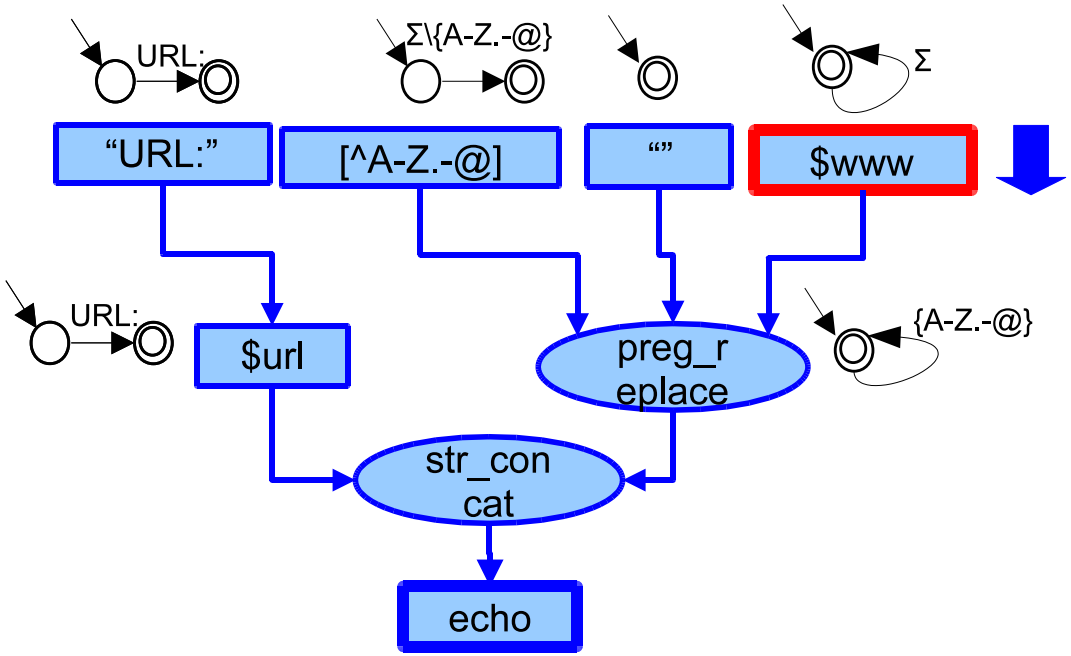
Forward Analysis

- Allows *arbitrary* values, i.e., Σ^*, from user inputs
- Propagates post-images to next nodes iteratively until a fixed point is reached



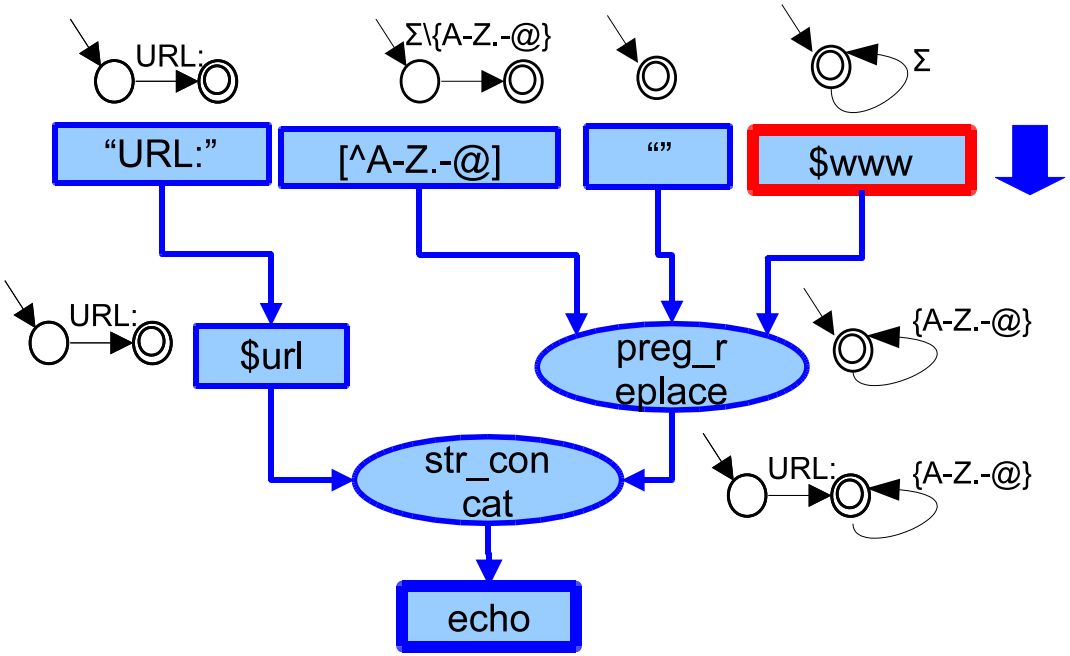
Forward Analysis

- At the first iteration, for the replace node, we call `postReplace`$(\Sigma^*,\ \Sigma \setminus \{A-Z.-@\},\ \texttt{""})$



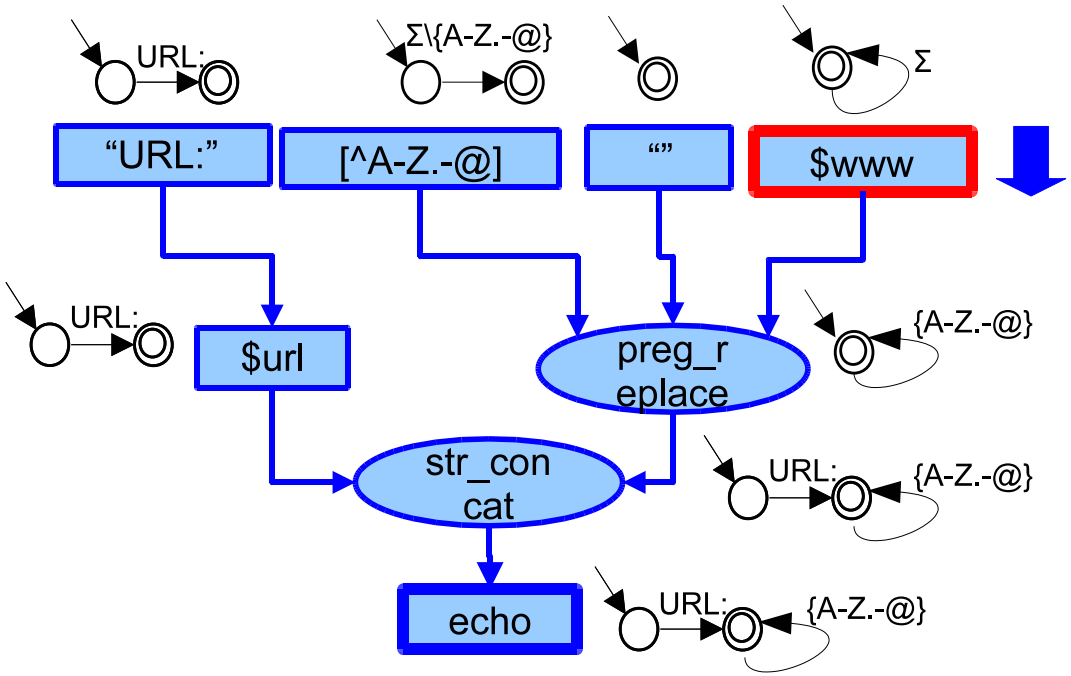
Forward Analysis

- At the second iteration, we call `postConcat("URL:", {A – Z. – @}*)`



Forward Analysis

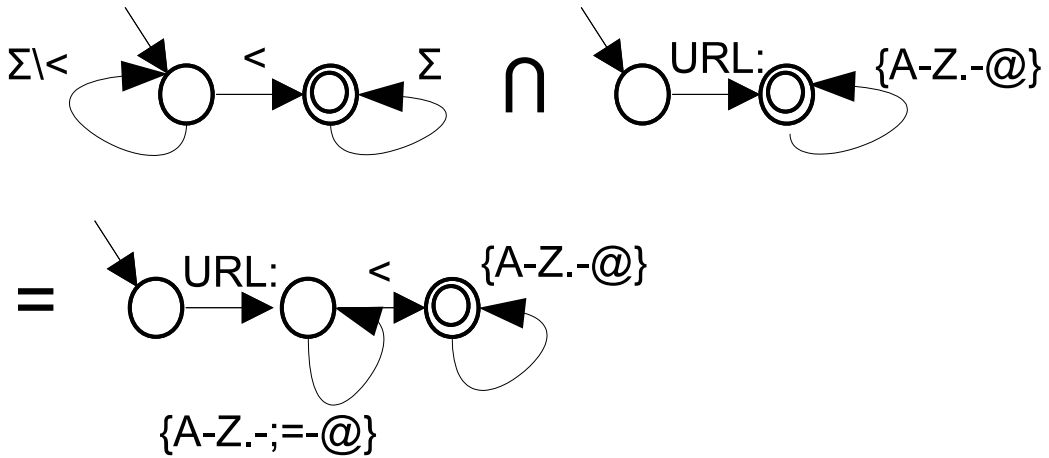
- The third iteration is a simple assignment
- After the third iteration, we reach a fixed point



NEXT: Is it vulnerable?

Detecting Vulnerabilities

- We know all possible values of the sink node (echo)
- Given an attack pattern, e.g., $(\Sigma \setminus <)^* < \Sigma^*$, if the intersection is not an empty set, the program is vulnerable. Otherwise, it is not vulnerable with respect to the attack pattern



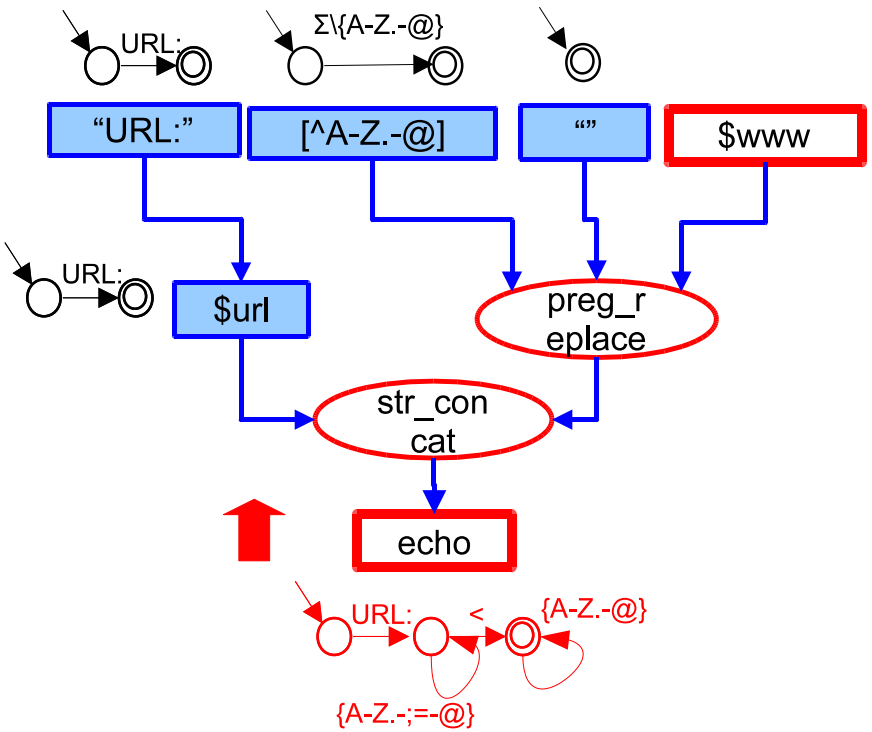
NEXT: What are the malicious inputs?

Generating Vulnerability Signatures

- A vulnerability signature is a characterization that includes all malicious inputs that can be used to generate attack strings
- Uses backward analysis starting from the sink node
- Uses *pre*-image computations on string operations:
  - `preConcatPrefix(`M`,` M_2`)` returns M_1 and `preConcatSuffix(`M`,` M_1`)` returns M_2, where $M = M_1.M_2$.
  - `preReplace(`M`,` M_2`,` M_3`)` retunrs M_1, where M=REPLACE(M_1, M_2, M_3).

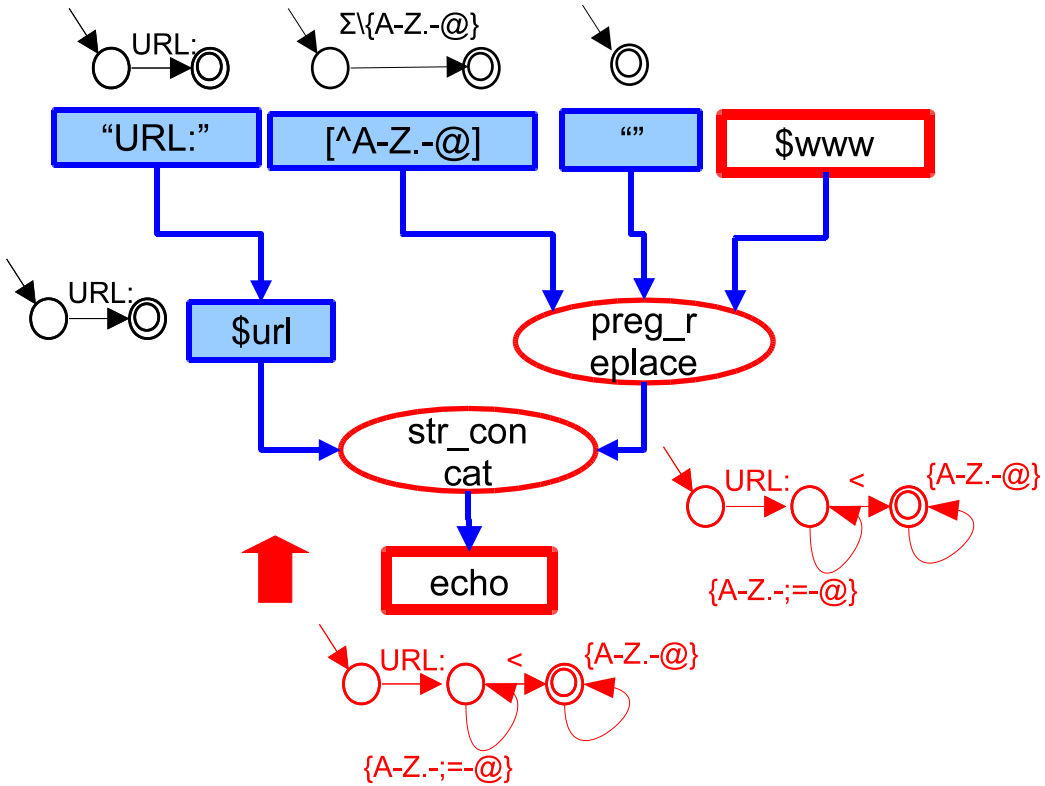
Backward Analysis

- Computes pre-images along with the path from the sink node to the input node
- Uses forward analysis results while computing pre-images



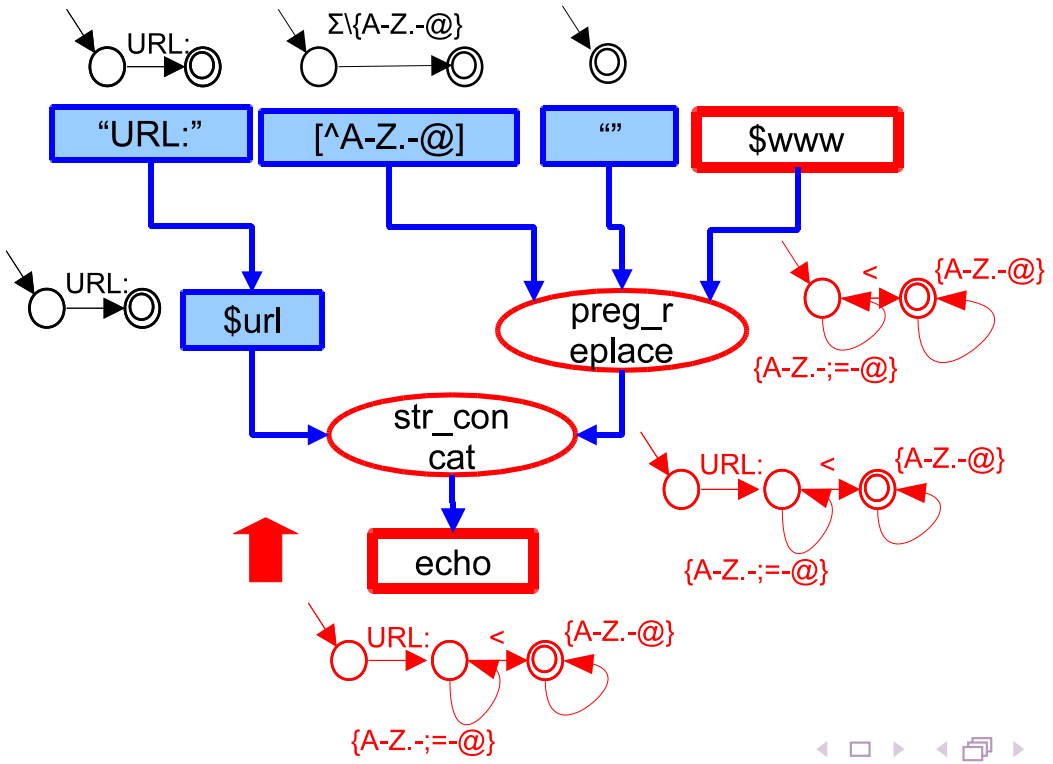
Backward Analysis

- The first iteration is a simple assignment.



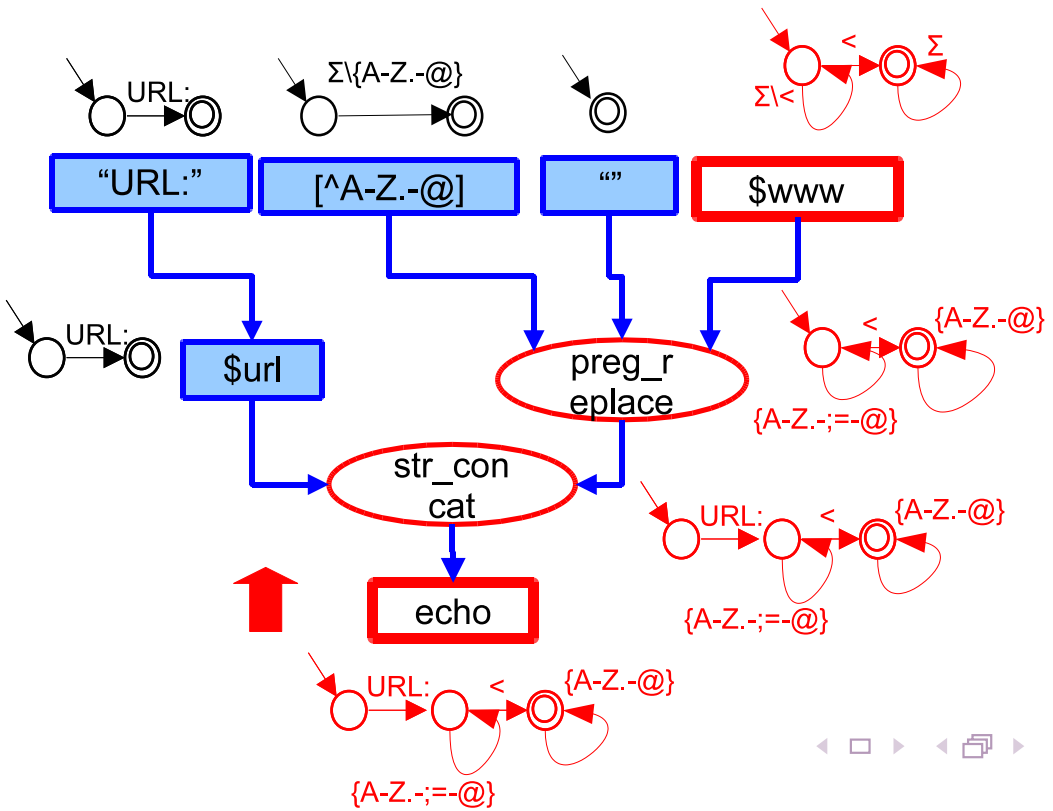
Backward Analysis

- At the second iteration, we call `preConcatSuffix(`$URL : \{A-Z.-;=-@\}^* < \{A-Z.-@\}^*$`, "URL:")`.
- $M = M_1.M_2$



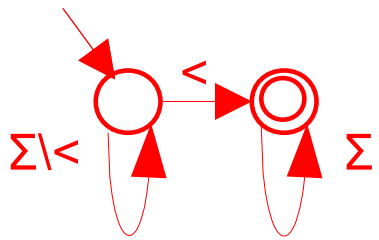
Backward Analysis

- We call $\texttt{preReplace}(\{A-Z.-;=-@\}^* < \{A-Z.-@\}^*, \Sigma \setminus \{A-Z.-@\}, \texttt{""})$ at the third iteration.
- $M = \texttt{replace}(M_1, M_2, M_3)$
- After the third iteration, we reach a fixed point.



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

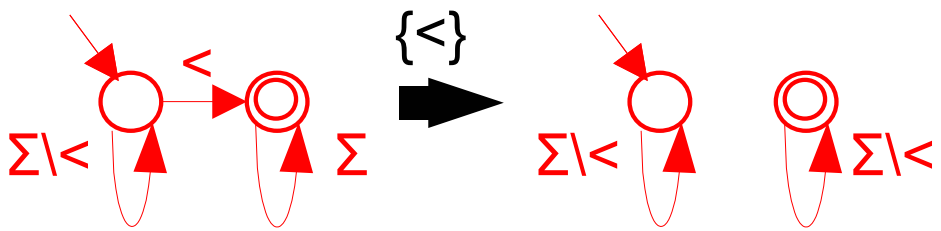


NEXT: How to detect and prevent malicious inputs

Patch Vulnerable Applications

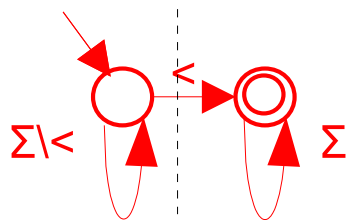
The idea is to modify the input (as little as possible) so that it does not match the vulnerability signature

- Deletes certain characters (an alphabet cut) in the input
- 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



An Alphabet Cut

- Finding a minimal alphabet cut of a DFA is an NP-hard problem (one can reduce the vertex cover problem to this problem)
- We apply a min-cut algorithm to find a cut that separates the initial state and the final states of the DFA
- The set of characters that are associated with the edges of the min cut is an alphabet cut



{<} is an alphabet cut

Patch Vulnerable Applications

Patch: If the input matches the vulnerability signature, delete all characters in the alphabet cut

```
<?php
if (preg_match('/[^ <]*<.*/',$_GET["www"]))
  $_GET["www"] = preg_replace(<,"",$_GET["www"]);
1: $www = $_GET["www"];
2: $url = "URL:";
3: $www = preg_replace("[^A-Z.-@]","",$www);
4: echo $url. $www;
?>
```

Experiments

We evaluated our 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), and
- (3) `proManager-0.72` (a project management system).

We used the following XSS attack pattern $\Sigma^* < \mathit{SCRIPT}\Sigma^*$.

Dependency Graphs

- The dependency graphs of these benchmarks are built for sensitive sinks
- Unrelated parts have been removed using slicing

	#nodes	#edges	#concat	#replace	#constant	#sinks	#inputs
1	21	20	6	1	46	1	1
2	29	29	13	7	108	1	1
3	25	25	6	6	220	1	2
4	23	22	10	9	357	1	1
5	25	25	14	12	357	1	1

Table: Dependency Graphs. #constant: the sum of the length of the constants

Vulnerability Analysis Performance

Forward analysis seems quite efficient.

	time(s)	mem(kb)	res.	#states / #bdds	#inputs
1	0.08	2599	vul	23/219	1
2	0.53	13633	vul	48/495	1
3	0.12	1955	vul	125/1200	2
4	0.12	4022	vul	133/1222	1
5	0.12	3387	vul	125/1200	1

Table: #states /#bdds of the final DFA (after the intersection with the attack pattern)

Signature Generation Performance

Backward analysis takes more time. Benchmark 2 involves a long sequence of replace operations.

	time(s)	mem(kb)	#states /#bdds
1	0.46	2963	9/199
2	41.03	1859767	811/8389
3	2.35	5673	20/302, 20/302
4	2.33	32035	91/1127
5	5.02	14958	20/302

Table: #states /#bdds of the vulnerability signature

Cuts

Sig.	1	2	3	4	5
input	i_1	i_1	i_1, i_2	i_1	i_1
#edges	1	8	4, 4	4	4
alp.-cut	$\{<\}$	$\{S,',"\}$	Σ, Σ	$\{<,',"\}$	$\{<,',"\}$

Table: Cuts. #edges: the number of edges in the min-cut.

- For 3 (two user inputs), the patch will block everything and delete everything
- Our analysis over approximates the relations among input variables (e.g. the concatenation of two inputs contains an attack)
- There may be no way to prevent it by restricting only one input

Automatic Verification of String Manipulating Programs

- Symbolic String Vulnerability Analysis
- Relational String Analysis
- Composite String Analysis

Relational String Analysis

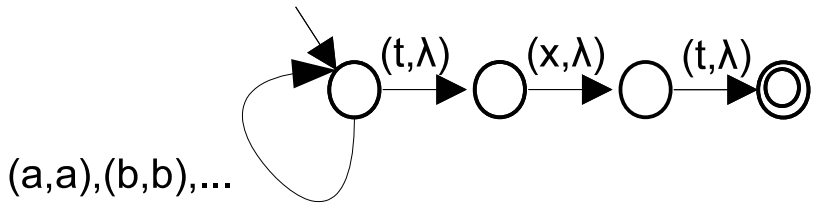
Instead of multiple *single*-track DFAs, we use one *multi*-track DFA, where each track represents the values of one string variable.

Using multi-track DFAs we are able to:

- Identify the relations among string variables
- Generate relational vulnerability signatures for multiple user inputs of a vulnerable application
- Prove properties that depend on relations among string variables, e.g., \$file = \$usr.txt (while the user is Fang, the open file is Fang.txt)
- Summarize procedures
- Improve the precision of the path-sensitive analysis

Multi-track Automata

- Let X (the first track), Y (the second track), be two string variables
- λ is a padding symbol
- A multi-track automaton that encodes X = Y.txt



Relational Vulnerability Signature

- Performs 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
- Intersects the auxiliary track with the attack pattern upon termination

Relational Vulnerability Signature

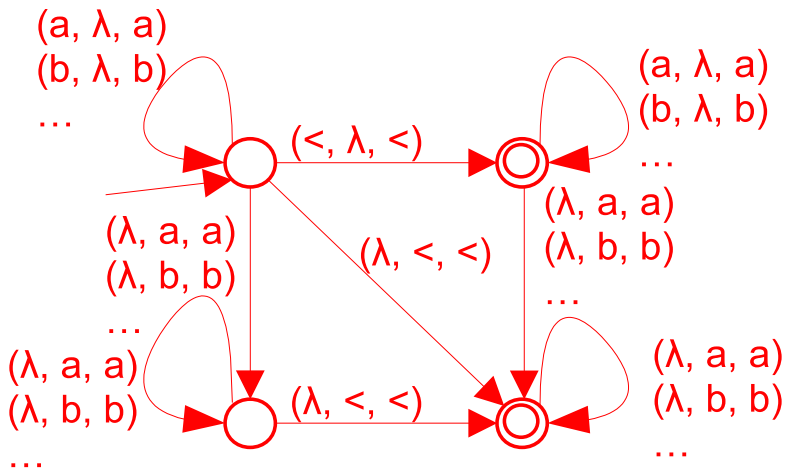
Consider a simple example having multiple user inputs

```
<?php
1: $www = $_GET["www"];
2: $url =$_GET["url"];
3: echo $url. $www;
?>
```

Let the attack pattern be $(\Sigma \setminus <)^* < \Sigma^*$

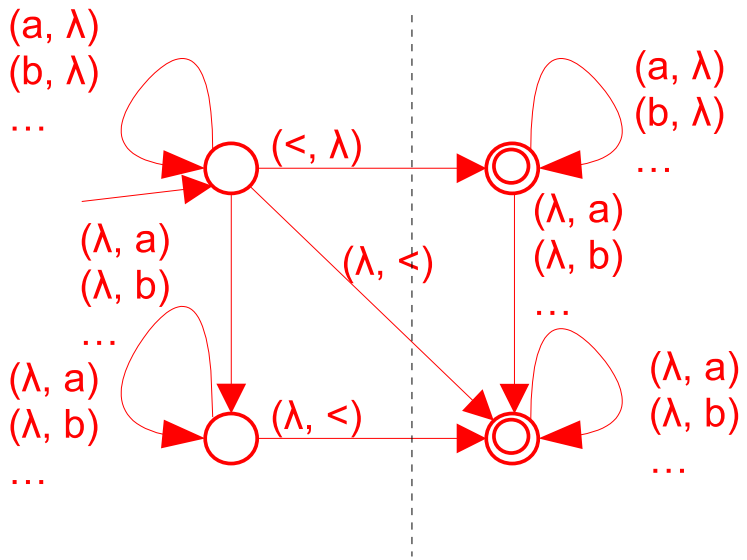
Relational Vulnerability Signature

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



Relational Vulnerability Signature

- Projects away the auxiliary track
- Finds a min-cut
- This min-cut identifies the alphabet cuts:
  - $\{<\}$ for the first track ($url)
  - $\{<\}$ for the second track ($www)



Patch Vulnerable Applications with Multi 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"]);
}
1: $www = $_GET["www"];
2: $url = $_GET["url"];
3: echo $url. $www;
?>
```

Previous Benchmark: Single V.S. Relational Signatures

ben.	type	time(s)	mem(kb)	#states /#bdds
3	Single-track	2.35	5673	20/302, 20/302
	Multi-track	0.66	6428	113/1682

3	Single-track	Multi-track
#edges	4	3
alp.-cut	Σ, Σ	$\{<\}$, $\{S\}$

Other Technical Issues

To conduct relational string analysis, we need a meaningful "intersection" of multi-track automata

- Intersection are closed under aligned multi-track automata
- λs are **right justified** in all tracks, e.g., $ab\lambda\lambda$ instead of $a\lambda b\lambda$
- However, there exist unaligned multi-track automata that are not describable by aligned ones
- We propose an alignment algorithm that constructs aligned automata which under/over approximate unaligned ones

Other Technical Issues

Modeling Word Equations:

- Intractability of $X = cZ$: The number of states of the corresponding aligned multi-track DFA is exponential to the length of c.
- Irregularity of $X = YZ$: $X = YZ$ is not describable by an aligned multi-track automata

We have proven the above results and proposed a conservative analysis.

Automatic Verification of String Manipulating Programs

- Symbolic String Vulnerability Analysis
- Relational String Verification
- Composite String Analysis

Composite Verification

We aim to extend our string analysis techniques to analyze systems that have unbounded string and integer variables.

We propose a composite static analysis approach that combines string analysis and size analysis.

Size Analysis

Integer Analysis: At each program point, statically compute the possible states of the values of all integer variables.

These infinite states are symbolically over-approximated as linear arithmetic constraints that can be represented as an arithmetic automaton

Integer analysis can be used to perform **Size Analysis** by representing lengths of string variables as integer variables.

What is Missing?

Consider the following segment.

- 1:<?php
- 2: $www = $_GET["www"];
- 3: $l_otherinfo = "URL";
- 4: $www = ereg_replace("[^A-Za-z0-9 ./-@://]","",$www);
- 5: if(strlen($www) < $limit)
- 6: echo "<td>" . $l_otherinfo . ": " . $www . "</td>";
- 7:?>

What is Missing?

If we perform size analysis solely, after line 4, we do not know the length of $www.

- 1:<?php
- 2: $www = $_GET["www"];
- 3: $l_otherinfo = "URL";
- 4: $www = ereg_replace("[^A-Za-z0-9 ./-@://]","",$www);
- 5: if(strlen($www) < $limit)
- 6: echo "<td>" . $l_otherinfo . ": " . $www . "</td>";
- 7:?>

What is Missing?

If we perform string analysis solely, at line 5, we cannot check/enforce the branch condition.

- 1:<?php
- 2: $www = $_GET["www"];
- 3: $l_otherinfo = "URL";
- 4: $www = ereg_replace("[^A-Za-z0-9 ./-@://]","",$www);
- 5: if(strlen($www) < $limit)
- 6: echo "<td>" . $l_otherinfo . ": " . $www . "</td>";
- 7:?>

What is Missing?

We need a **composite analysis** that combines string analysis with size analysis.

Challenge: How to transfer information between string automata and arithmetic automata?

Some Facts about String Automata

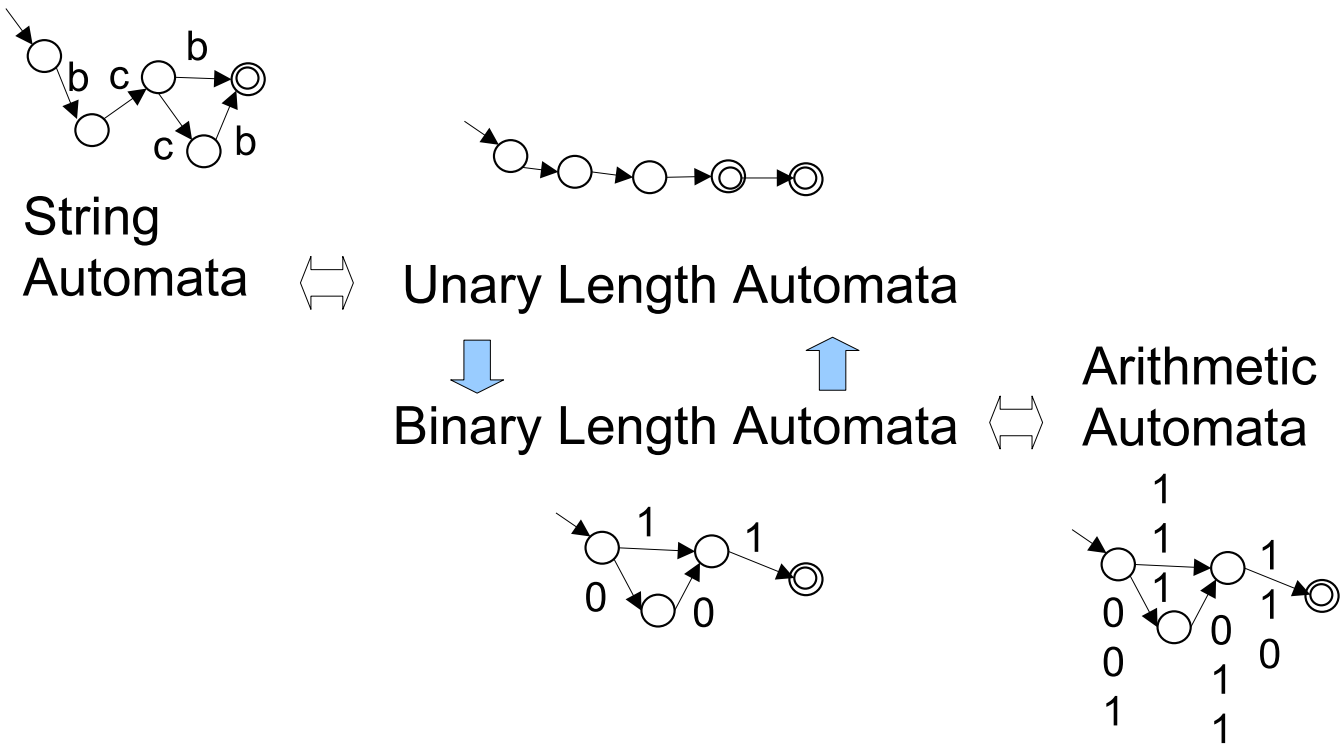
- A string automaton is a single-track DFA that accepts a regular language, whose length forms a semi-linear set, .e.g., $\{4, 6\} \cup \{2 + 3k \mid k \geq 0\}$
- The unary encoding of a semi-linear set is uniquely identified by a unary automaton
- The unary automaton can be constructed by replacing the alphabet of a string automaton with a unary alphabet

Some Facts about Arithmetic Automata

- An arithmetic automaton is a multi-track DFA, where each track represents the value of one variable over a binary alphabet
- If the language of an arithmetic automaton satisfies a Presburger formula, the value of each variable forms a semi-linear set
- The semi-linear set is accepted by the binary automaton that projects away all other tracks from the arithmetic automaton

An Overview

To connect the dots, we propose a novel algorithm to convert unary automata to binary automata and vice versa.
More details can be found in [TACAS'09].



Experiments

In [TACAS09], we manually generate several benchmarks from:

- C string library
- Buffer overflow benchmarks (buggy/fixed) [Ku et al., ASE'07]
- Web vulnerable applications (vulnerable/sanitized) [Balzarotti et al., S&P'08]

These benchmarks are small ($<$100 statements and $<$ 10 variables) but demonstrate typical relations among string and integer variables.

Experimental Results

The results show some promise in terms of both precision and performance

Test case (*bad*/*ok*)	Result	Time (s)	Memory (kb)
int strlen(char *s)	T	0.037	522
char *strrchr(char *s, int c)	T	0.011	360
gxine (CVE-2007-0406)	F/T	0.014/0.018	216/252
samba (CVE-2007-0453)	F/T	0.015/0.021	218/252
MyEasyMarket-4.1 (trans.php:218)	F/T	0.032/0.041	704/712
PBLguestbook-1.32 (pblguestbook.php:1210)	F/T	0.021/0.022	496/662
BloggIT 1.0 (admin.php:27)	F/T	0.719/0.721	5857/7067

Table: T: The property holds (buffer overflow free or not vulnerable with respect to the attack pattern)

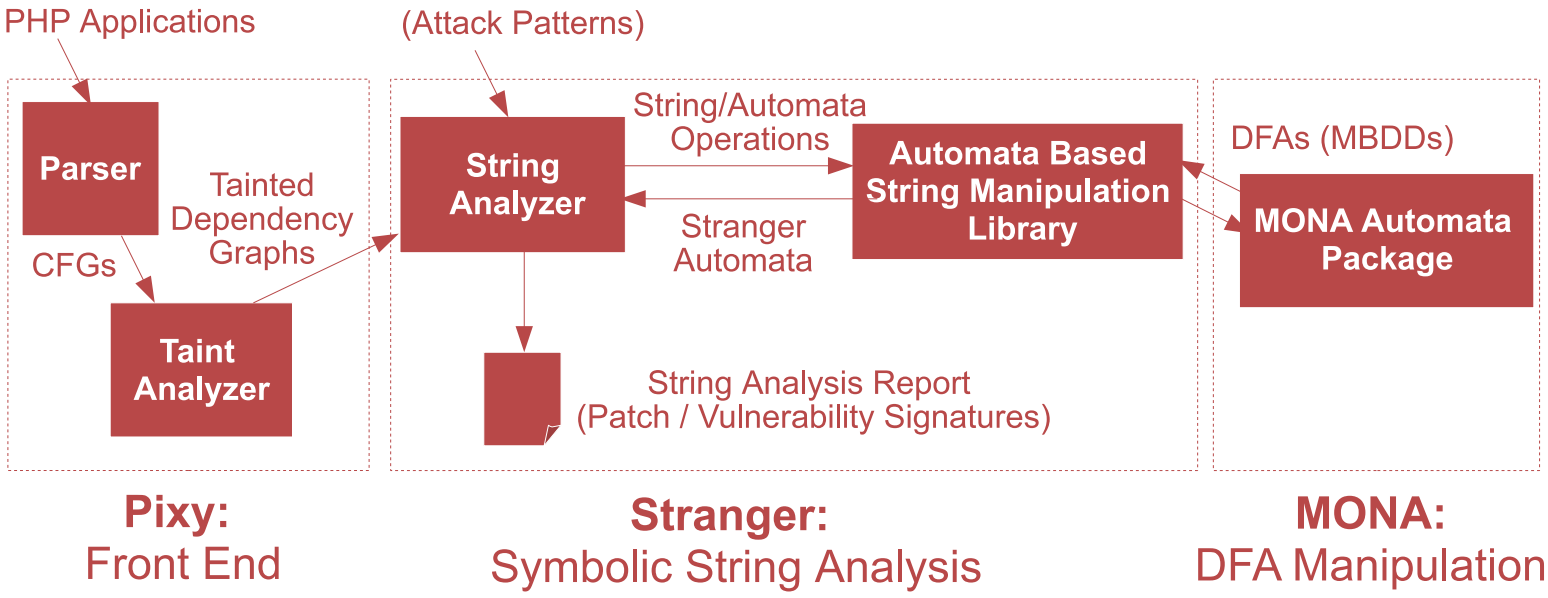
STRANGER Tool

We have developed STRANGER (STRing AutomatoN GEneratoR)

- A public automata-based string analysis tool for PHP
- Takes a PHP application (and attack patterns) as input, and automatically analyzes all its scripts and outputs the possible XSS, SQL Injection, or MFE vulnerabilities in the application

STRANGER Tool

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



The tool, detailed documents, and several benchmarks are available: http://www.cs.ucsb.edu/~vlab/stranger.

STRANGER Tool

A case study on `Schoolmate 1.5.4`

- 63 php files containing 8000+ lines of code
- Intel Core 2 Due 2.5 GHz with 4GB of memory running Linux Ubuntu 8.04
- STRANGER took 22 minutes / 281MB to reveal 153 XSS from 898 sinks
- After manual inspection, we found 105 actual vulnerabilities (false positive rate: 31.3%)
- We inserted patches for all actual vulnerabilities
- Stranger proved that our patches are correct with respect to the attack pattern we are using

STRANGER Tool

Another case study on `SimpGB-1.49.0`, a PHP guestbook web application

- 153 php files containing 44000+ lines of code
- Intel Core 2 Due 2.5 GHz with 4GB of memory running Linux Ubuntu 8.04
- For all executable entries, STRANGER took
  - 231 minutes to reveal 304 XSS from 15115 sinks,
  - 175 minutes to reveal 172 SQLI from 1082 sinks, and
  - 151 minutes to reveal 26 MFE from 236 sinks

Related Work on String Analysis

- String analysis based on context free grammars: [Christensen et al., SAS'03] [Minamide, WWW'05]
- String analysis based on symbolic execution: [Bjorner et al., TACAS'09]
- Bounded string analysis : [Kiezun et al., ISSTA'09]
- Automata based string analysis: [Xiang et al., COMPSAC'07] [Shannon et al., MUTATION'07] [Barlzarotti et al. S&P'08]
- Application of string analysis to web applications: [Wassermann and Su, PLDI'07, ICSE'08] [Halfond and Orso, ASE'05, ICSE'06]

Related Work on Size Analysis and Composite Analysis

- Size analysis : [Dor et al., SIGPLAN Notice'03] [Hughes et al., POPL'96] [Chin et al., ICSE'05] [Yu et al., FSE'07] [Yang et al., CAV'08]
- Composite analysis:
  - Composite Framework: [Bultan et al., TOSEM'00]
  - Symbolic Execution: [Xu et al., ISSTA'08] [Saxena et al., UCB-TR'10]
  - Abstract Interpretation: [Gulwani et al., POPL'08] [Halbwachs et al., PLDI'08]

Related Work on Vulnerability Signature Generation

- Test input/Attack generation: [Wassermann et al., ISSTA'08] [Kiezun et al., ICSE'09]
- Vulnerability signature generation: [Brumley et al., S&P'06] [Brumley et al., CSF'07] [Costa et al., SOSP'07]

My Publications

String Analysis:

- STRANGER: *An Automata-based String Analysis Tool for PHP* [TACAS'10]
- *Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses* [ASE'09]
- *Symbolic String Verification: Combining String Analysis and Size Analysis* [TACAS'09]
- *Symbolic String Verification: An Automata-based Approach* [SPIN'08]

My Publications

Web Service/Application Verification:

- *Modular Verification of Web Services Using Efficient Symbolic Encoding and Summarization* [FSE'08]
- *Verifying Web Applications Using Bounded Model Checking* [DSN'04]
- *Securing Web Application Code by Static Analysis and Runtime Protection* [WWW'04] (best paper nominee)

Size Analysis:

- *Automated Size Analysis for OCL* [FSE'07]

My Other Publications

Membrane Computing: [UC'06][NC'07]

Verification of Real-time Systems:

- SAT-based Techniques: [FORMATS-FTRTFT'04] [ATVA'04] [IJFCS'06]
- BDD-based Techniques: [CIAA'03] [FORTE'03] [RTCSA'03a,RTCSA'03b] [JEC'04] [IEEE TSE'04, TSE'06]

Thank you for your attention.

Questions?