Introduction to Stranger

Fang Yu

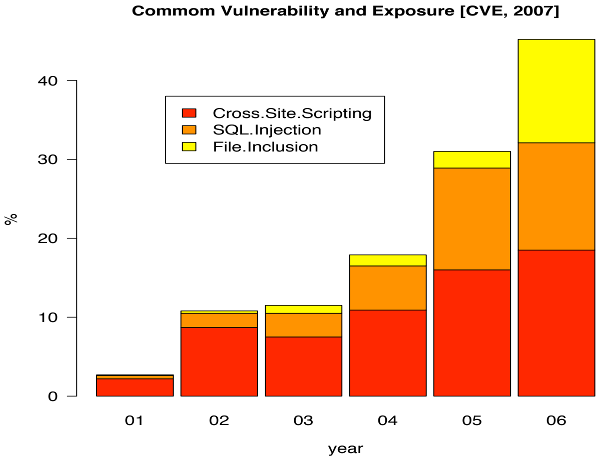
Department of Management Information Systems
National Chengchi University, Taipei, Taiwan

joint work with
Muath Alkhalaf and Tevfik Bultan
University of California, Santa Barbara

March 22, 2011

Web Application Vulnerabilities

Have contributed the majority of common vulnerabilities.



Web Application Vulnerabilities

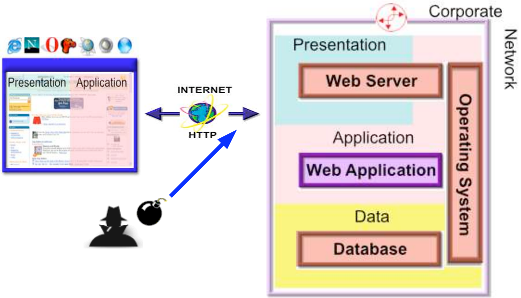
- The top three vulnerabilities of the Open Web Application Security Project (OWASP)'s top ten list. [OWASP, 2007]
  1. Cross Site Scripting (XSS)
  2. Injection Flaws (such as SQL Injection)
  3. Malicious File Executions

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)

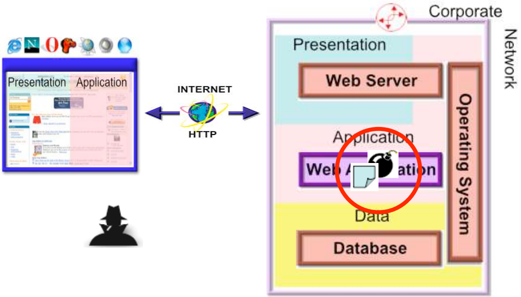
Injection Flaws

- The attacker formulates a malicious command, and sends it as input to the Web application
    - Login / search / registration / etc



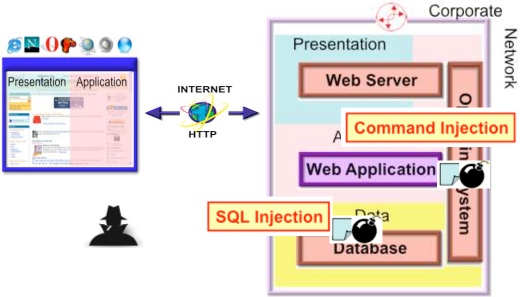
Injection Flaws

- The Web application uses the input to construct commands without prior sanitization



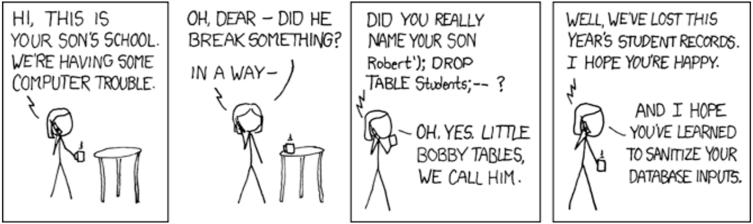
Injection Flaws

- Command delivered to OS: Command injection
- Command delivered to database: SQL injection
- Since arbitrary command is executed, this attack may cause great damage



SQL Injection

Exploits of a Mom.



Source: XKCD.com

SQL Injection

Access students’ data by $name (from a user input).

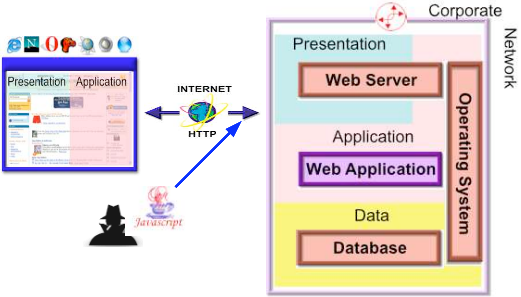
- 1:<?php
- 2: $name =$_GET[”name”];
- 3: $user_data = $db->query(”SELECT * FROM students WHERE (name = ’$name’) ”);
- 4:?>

SQL Injection

- 1:<?php
- 2: $name = $_GET["name"];
- 3: $user_data = $db->query("SELECT * FROM students WHERE (name = 'Robert '); DROP TABLE students; - -') ");
- 4:?>

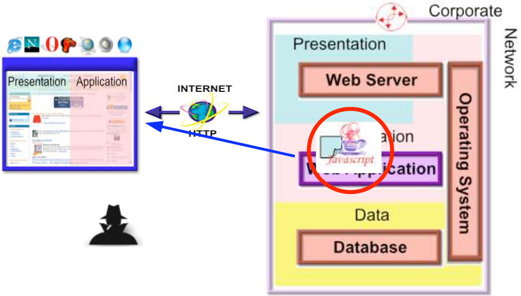
XSS Attacks

- Malicious content injected into a web application can also attack clients
- In XSS, an attacker first inject a malicious script into the Web applications database
    - Through a functionality (e.g., message posting)



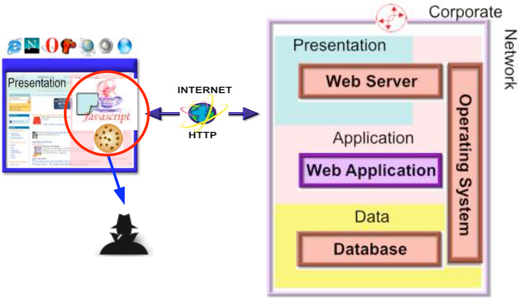
XSS Attacks

- Upon a certain request by a victim, the script is used to construct output
  - E.g., the victim reads the posted message



XSS Attacks

- The script is delivered on behalf of the Web application to the client
  - It has the right to access client's **cookies** and deliver them to attackers.



XSS Attack

Another 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:?>

String Related Vulnerabilities

- All of the listed web application vulnerabilities are caused by: a sensitive function takes an attack string (specified by an attack pattern) as its input
- It occurs due to improper string manipulations on user provided inputs
- We develop a formal and fully automatic approach that can:
  - detect string related vulnerabilities
  - prove the absence of string related vulnerabilities
  - generate sanitization statements for patching vulnerable web applications

Is it Vulnerable?

A simple taint analysis, e.g., [Huang et al. WWW04], can report this segment 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: $\Sigma^* <\mathrm{script}\Sigma^*$.

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^*.

About This Work

We achieve this by automata-based string analysis techniques.

Our approach consists of three phases:

- Vulnerability Analysis
- Vulnerability Signature Generation
- Sanitization Generation

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

Vulnerability Analysis

- Converts programs to dependency graphs focusing on string manipulation operations
- Performs forward symbolic reachability analyses to detect vulnerabilities

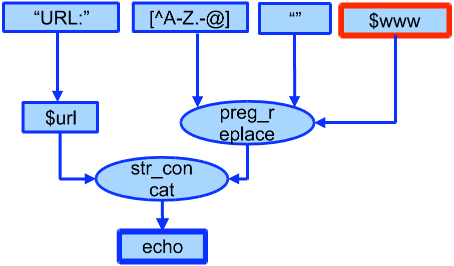
Front End

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

$M = \textsc{replace}(M_1, M_2, M_3)$

- 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.

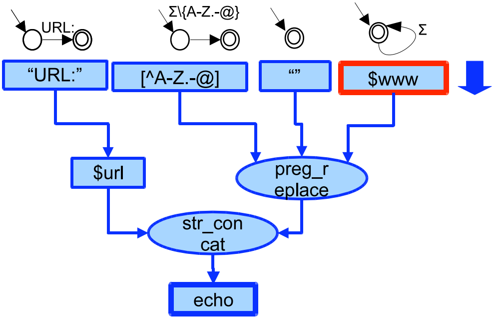
$M = \textsc{replace}(M_1, M_2, M_3)$

Some examples:

$L(M_1)$	$L(M_2)$	$L(M_3)$	$L(M)$
{baaabaa}	{aa}	{c}	{bacbc, bcabc}
{baaabaa}	a^+	ϵ	{bb}
{baaabaa}	a^+b	{c}	{baacaa, bacaa, bcaa}
{baaabaa}	a^+	{c}	{bcccbcc, bcccbc, bccbcc, bccbc, bcbcc, bcbc}
ba^+b	a^+	{c}	bc^+b

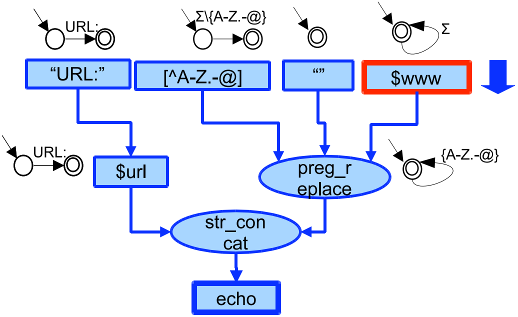
Forward Analysis

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



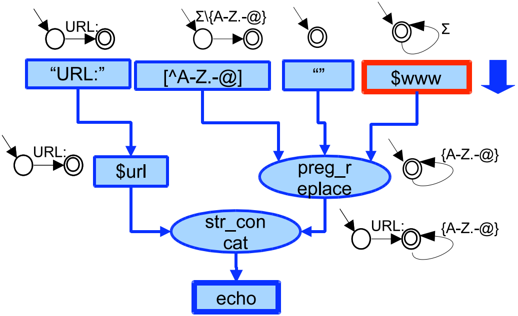
Forward Analysis

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



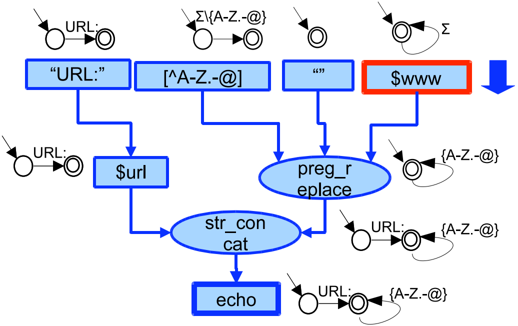
Forward Analysis

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



Forward Analysis

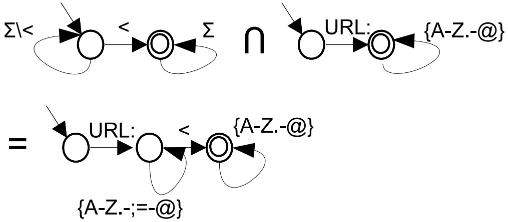
- The third iteration is a simple assignment
- After the third iteration, we reach a fixed point



NEXT: Is it vulnerable?

Detecting Vulnerabilities

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



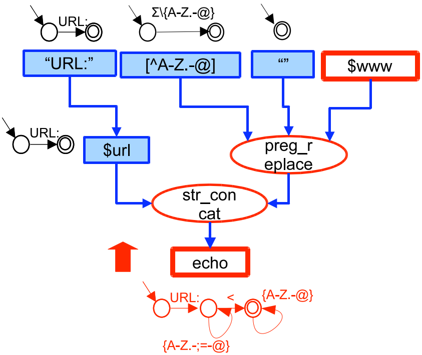
NEXT: What are the malicious inputs?

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`)` returns M_1, where $M = \textsc{replace}(M_1, M_2, M_3)$.

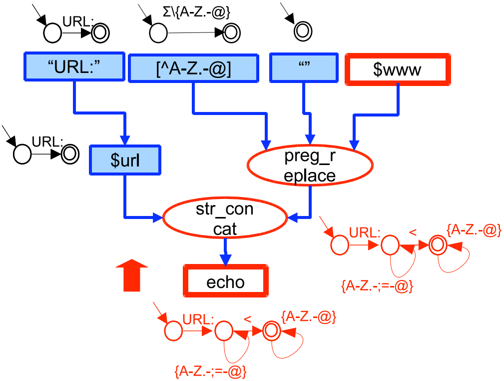
Backward Analysis

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



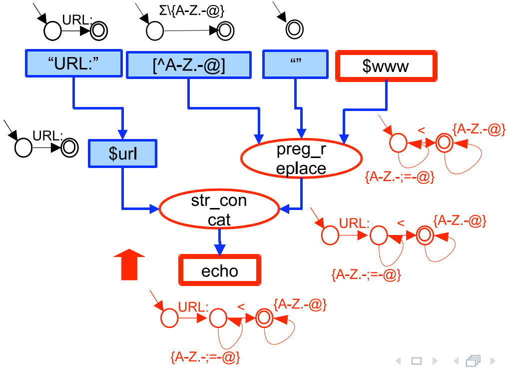
Backward Analysis

- The first iteration is a simple assignment.



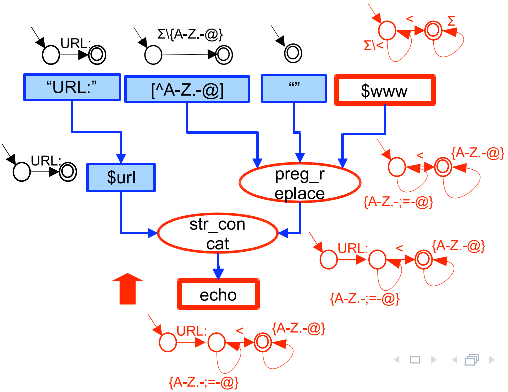
Backward Analysis

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



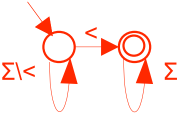
Backward Analysis

- 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

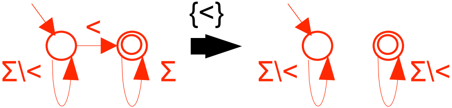
Patch Vulnerable Applications

- Match-and-block: A patch that checks if the input string matches the vulnerability signature and halts the execution if it does
- Match-and-sanitize: A patch that checks if the input string matches the vulnerability signature and modifies the input if it does

Sanitize

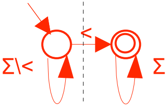
The idea is to modify the input by deleting certain characters (as little as possible) so that it does not match the vulnerability signature

- 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



Find An Alphabet Cut

- Finding a minimum 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
- We give higher weight to edges that are associated with alpha-numeric characters
- 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

A match-and-sanitize 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 $\Sigma^* < SCRIPT\Sigma^*$.

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	$\{<\}$	$\{<,',"\}$	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

Multiple Inputs?

Things can be more complicated while there are multiple inputs.

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

- An attack string can be contributed from one input, another input, or their combination
- Using *single-track* DFAs, the 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

Relational String Analysis

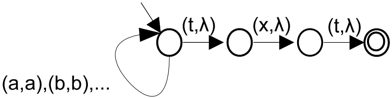
Instead of multiple *single*-track DFAs, we use one *multi*-track DFA, where each track represents the values of one string variable.

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

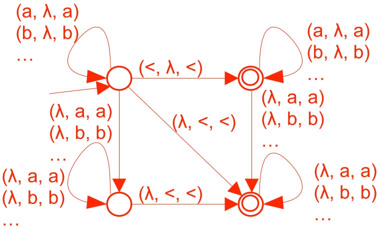
Consider a simple example having multiple user inputs

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

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

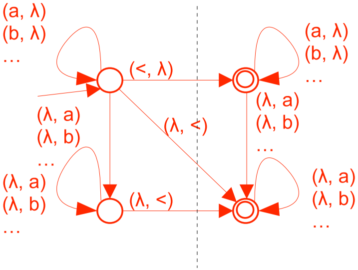
Relational Vulnerability Signature

- 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 for the echo statement
- 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	$\{<\}$, $\{<\}$

Patching Web Applications

We applied our analysis to three open source PHP web applications:

- Webchess 0.9.0 (a server for playing chess over the internet)
- EVE 1.0 (a tracker for players activity for an online game)
- Faqforge 1.3.2 (a document management tool)

	Application	# of PHP files	total loc	# of sinks XSS	 SQLI
1	Webchess 0.9.0	23	3375	421	140
2	EVE 1.0	8	906	114	17
3	Faqforge 1.3.2	10	534	375	133

Attack Patterns

- For XSS attacks: Σ^* $<$scriptΣ^* (indicating an embedded script)
- For SQLI attacks: Σ^* or 1 $=$1 Σ^* (indicating a *true* condition in a query)

Vulnerability Analysis Evaluation

	# of Vul. (single, 2, 3, 4)	Time (seconds) total	 fwd	 bwd	 relational	Mem (Kb) average
XSS Vulnerability Analysis						
1	(24, 3, 0, 0)	46.08	1.73	0.92	6.30	16850
2	(0, 0, 8, 0)	288.50	6.80	–	127.80	125382
3	(20, 0, 0, 0)	7.87	0.22	0.22	–	9948
SQLI Vulnerability Analysis						
1	(43, 3, 1, 2)	110.7	4.87	12.04	38.03	136790
2	(8, 3, 0, 0)	23.9	1.5	8.47	5.2	17280
3	(0, 0, 0, 0)	6.7	–	–	–	< 1

- (single, 2, 3, 4) indicates the number of detected vulnerabilities that have single input, two inputs, three inputs and four inputs

Cut Evaluation

Find the alphabet cut of each vulnerability signature

	XSS		SQLI	
	time (sec)	alp.-cut	time	alp.-cut
1	0.06	$\{<\}$	0.07	$\{=\}$
2	0.3	$\{<\}$	0.1	$\{=\}$
3	0.05	$\{<\}$	none	

- time is the average time per automaton to find its alphabet-cut
- alp.-cut is the deleted character set for each input

Patch Evaluation

- Sanitize the three applications above by placing the automatically generated sanitization statements at the beginning of each vulnerable script.
- Run our forward vulnerability analysis which reported *zero* vulnerabilities with regard to the attack pattern mentioned above demonstrating
- Our analysis is *sound* and guarantees that after the sanitization statements are inserted, sensitive functions will not receive any input that matches the attack pattern

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]

Publications related to this work

String Analysis:

- *Patching Vulnerabilities with Sanitization Synthesis.*
  Fang Yu, Muath Alkahalf, Tevfik Bultan. Accepted by [ICSE'11]
- *Relational String Analysis Using Multi-track Automata.*
  Fang Yu, Tevfik Bultan, Oscar H. Ibarra. [CIAA'10]
- STRANGER*: An Automata-based String Analysis Tool for PHP.*
  Fang Yu, Muath Alkahalf, Tevfik Bultan. [TACAS'10]
- *Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses.*
  Fang Yu, Muath Alkahalf, Tevfik Bultan. [ASE'09]
- *Symbolic String Verification: Combining String Analysis and Size Analysis*
  Fang Yu, Tevfik Bultan, Oscar H. Ibarra. [TACAS'09]
- *Symbolic String Verification: An Automata-based Approach*

  Fang Yu, Tevfik Bultan, Marco Cova, Oscar H. Ibarra. [SPIN'08]

Summary of Contributions

- An automata-based approach for analyzing string manipulating programs using symbolic string analysis. The approach combines forward and backward symbolic reachability analyses, and features language-based replacement, fixpoint acceleration, and symbolic automata encoding [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: We catch relations among string variables using multi-track automata, i.e., each track represents the values of one variable. This approach enables verification of properties that depend on relations among string variables [CIAA10]
- An automatic approach for vulnerability signature generation and patch synthesis: We apply multi-track automata to generate relational vulnerability signatures with which we are able to synthesize effective patches for vulnerable Web applications. [ICSE11]

Thank you for your attention.

Questions?