String Analysis for Software Verification and Security

Fang Yu

Software Security Lab.
Department of Management Information Systems
College of Commerce, National Chengchi University
http://soslab.nccu.edu.tw

FLOLAC 2019, August 27, 2019

About Me

Yu, Fang

- 2014-present: Associate Professor, Department of Management Information Systems, National Chengchi University
- 2010-2014: Assistant Professor, Department of Management Information Systems, National Chengchi University
- 2005-2010: Ph.D. and M.S., Department of Computer Science, University of California at Santa Barbara
- 2001-2005: Institute of Information Science, Academia Sinica
- 1994-2000: M.B.A. and B.B.A., Department of Information Management, National Taiwan University

Book Reference

- *String Analysis for Software Verification and Security*
  Tevfik Bultan, Fang Yu, Muath Alkhalaf, Abdulbaki Aydin. [Springer. 2018]
- https://www.springer.com/gp/book/9783319686684

More Recent Work

- *Parameterized Model Counting for String and Numeric Constraints*
  Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan and Fang Yu. [ACM ESEC/FSE '18]
- *A Symbolic Model Checking Approach to the Analysis of String and Length Constraints*
  Hung-En Wang, Shih-Yu Chen, Fang Yu, Jie-Hong R. Jiang. [ACM ASE'18]
- *Static API Call Vulnerability Detection in iOS Applications*
  Chun-Han Lin, Fang Yu, Jie-Hong Jiang, and Tevfik Bultan. [ACM/IEEE ICSE'18]
- *Optimal Sanitization Synthesis for Web Application Vulnerability Repair*
  Fang Yu, ChinYuan Shueh, ChunHan Lin, YuFang Chen, BowYaw Wang, Tevfik Bultan. [ACM ISSTA'16]
- *String Analysis via Automata Manipulation with Logic Circuit Representation*
  HungEn Wang, ThungLin Tsai, ChunHan Lin, Fang Yu, JieHong R Jiang.
  [CAV'16]

Automatic Verification of String Manipulating Programs

Web Applications = String Manipulating Programs

Web Applications

Web applications are used extensively in many areas

- Commerce: online banking, online shopping, etc.
- Entertainment: online game, music and videos, etc.
- Interaction: social networks

Web Applications

We may rely on web applications more in the future

- Health Records: Google Health, Microsoft HealthVault
- Controlling and monitoring national infrastructures: Google Powermeter

Google health

Microsoft HealthVault Microsoft

Web Applications

Web software is also rapidly replacing desktop applications.

One Major Road Block

Web applications are not trustworthy!

Web applications are notorious for security vulnerabilities

- Their global accessibility makes them a target for many malicious users

Web applications are becoming increasingly dominant and their use in safety critical areas is increasing

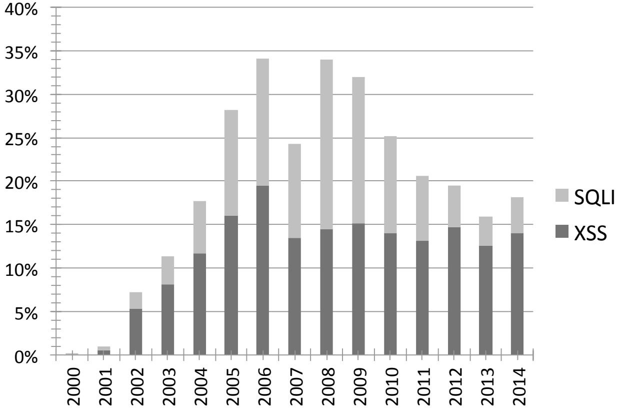
- Their trustworthiness is becoming a critical issue

Web Application Vulnerabilities

- The top two vulnerabilities of the Open Web Application Security Project (OWASP)'s top ten list in 2007, 2010, 2013, and 2017
  1. Cross Site Scripting (XSS)
  2. Injection Flaws (such as SQL Injection)

Web Application Vulnerabilities

Percentage of the Cross-site Scripting (XSS) and SQL Injection (SQLI) vulnerabilities among all the computer security vulnerabilities reported in the CVE repository.



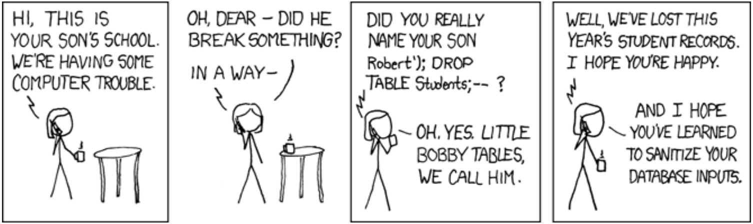
Why are web applications error prone?

Extensive string manipulation:

- Web applications use extensive string manipulation
  - To construct html pages, to construct database queries in SQL, to construct system commands
- 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

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:?>

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.

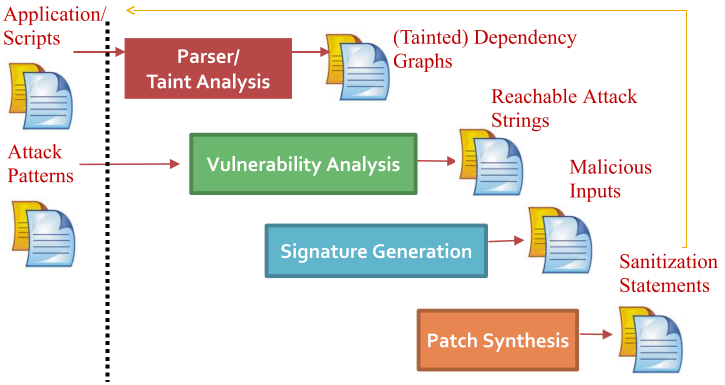
String Analysis

- 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
- 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 that characterizes all malicious inputs

Automata-based String Analysis

- Finite State Automata can be used to characterize sets of string values
- We use 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
- Attack patterns are specified as regular expressions

String Analysis Stages



A Language-based Replacement

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

- M_1, M_2, and M_3 are 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 $s \in L(M1)$, $x \in L(M2)$, and $c \in L(M3)$:
  - Replaces all parts of any s that match any x with any c.
  - Outputs a DFA that accepts the result to M.

M=REPLACE(M_1, M_2, M_3)

Some examples:

$L(M_1)$	$L(M_2)$	$L(M_3)$	$L(M)$
{ baaabaa}	{aa}	{c}	
{baaabaa}	a^+	ϵ	
{baaabaa}	a^+b	{c}	
{baaabaa}	a^+	{c}	
ba^+b	a^+	{c}	

M=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	
{baaabaa}	a^+b	{c}	
{baaabaa}	a^+	{c}	
ba^+b	a^+	{c}	

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

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}	
{baaabaa}	a^+	{c}	
ba^+b	a^+	{c}	

M=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}	
ba^+b	a^+	{c}	

M=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}	

M=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

M=REPLACE(M_1, M_2, M_3)

- An over approximation with respect to the leftmost/longest(first) constraints
- Many string functions in PHP can be converted to this form:
  - htmlspecialchars, tolower, toupper, str_replace, trim, and
  - preg_replace and ereg_replace that have regular expressions as their arguments.

Formal Definition

A DFA M is a replaced-DFA of a DFA tuple (M_1, M_2, M_3), if and only if $L(M) = \{w \mid k > 0, w_1 x_1 w_2 \ldots w_k x_k w_{k+1} \in L(M_1),$ $w = w_1 c_1 w_2 \ldots w_k c_k w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), c_i \in L(M_3), \forall 1 \leq i \leq k+1, w_i \notin \{w_1' x' w_2' \mid x' \in L(M_2), w_1', w_2' \in \Sigma^*\}\}$.

A Language-based Replacement

Implementation of $\text{REPLACE}(M_1, M_2, M_3)$:

- Mark matching sub-strings
  - Insert marks to M_1
  - Insert marks to M_2
- Replace matching sub-strings
  - Identify marked paths
  - Insert replacement automata

In the following, we use two marks: $<$ and $>$ (not in Σ), and a duplicate set of alphabet: $\Sigma' = \{\alpha' | \alpha \in \Sigma\}$. We use an example to illustrate our approach.

An Example

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

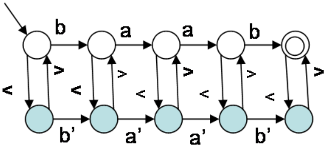
- $L(M_1) = \{baab\}$
- $L(M_2) = a^+ = \{a, aa, aaa, \ldots\}$
- $L(M_3) = \{c\}$

Step 1

Construct M_1' from M_1:

- Duplicate M_1 using Σ'
- Connect the original and duplicated states with $<$ and $>$

For instance, M_1' accepts $b < a'a' > b$, $b < a' > ab$.

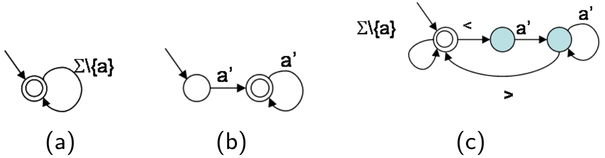


Step 2

Construct M'_2 from M_2:

- Construct $M_{\bar{2}}$ that accepts strings that do not contain any substring in $L(M_2)$. (a)
- Duplicate M_2 using Σ'. (b)
- Connect (a) and (b) with marks. (c)

For instance, M'_2 accepts $b < a'a' > b$, $b < a' > bc < a' >$.

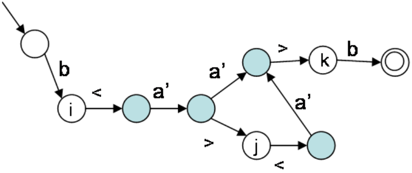


Step 3

Intersect M'_1 and M'_2.

- The matched substrings are marked in Σ'.
- Identify (s, s'), so that $s \rightarrow^{<} \ldots \rightarrow^{>} s'$.

In the example, we idenitfy three pairs:(i,j), (i,k), (j,k).

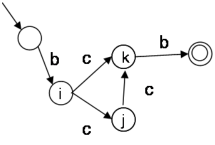


Step 4

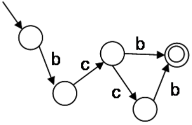
Construct M:

- Insert M_3 for each identified pair. (d)
- Determinize and minimize the result. (e)

$L(M) = \{bcb, bccb\}$.



(d)



(e)

Quiz 1

Compute M=REPLACE(M_1, M_2, M_3), where $L(M_1) = \{baabc\}$, $L(M_2) = a^+b$, $L(M_3) = \{c\}$.

Concatenation

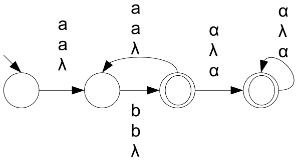
We introduce concatenation transducers to specify the relation $X = YZ$.

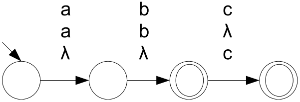
- A concatenation transducer is a 3-track DFA M over the alphabet $\Sigma \times (\Sigma \cup \{\lambda\}) \times (\Sigma \cup \{\lambda\})$, where $\lambda \notin \Sigma$ is a special symbol for padding.
- $\forall w \in L(M)$, $w[1] = w'[2].w'[3]$
  - $w[i]$ $(1 \leq i \leq 3)$ to denote the i^{th} track of $w \in \Sigma^3$
  - $w'[2] \in \Sigma^*$ is the λ-free prefix of $w[2]$ and
  - $w'[3] \in \Sigma^*$ is the λ-free suffix of $w[3]$

Suffix

Consider $X = (ab)^+.Z$
Assume $L(M_X) = \{ab, abc\}$. What are the values of Z?

- We first build the transducer M for $X = (ab)^+ Z$
- We intersect M with M_X on the first track
- The result is the third track of the intersection, i.e., $\{\epsilon, c\}$.

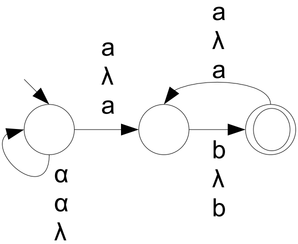


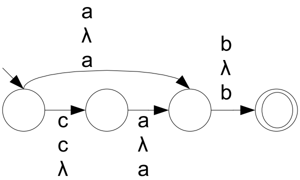


Prefix

Consider $X = Y.(ab)^+$.
Assume $L(M_X) = \{ab, cab\}$. What are the values of Y?

- We first build the transducer M for $X = Y.(ab)^+$
- We intersect M with M_X on the first track
- The result is the second track of the intersection, i.e., $\{\epsilon, c\}$.





Quiz 2

What is the concatenation transducer for the general case X=YZ, i.e., X, Y, Z $\in \Sigma^*$?

Quiz 2.1

Consider $X = Y.(abc)^*$.
Assume $L(M_X) = (cab)^+c$. What are the values of Y?

Widening Automata: $M \nabla M'$

Compute an automaton so that $L(M \nabla M') \supseteq L(M) \cup L(M')$. We can use widening to accelerate the fixpoint computation.

Widening Automata: $M \nabla M'$

Here we introduce one widening operator originally proposed by Bartzis and Bultan [CAV04]. Intuitively,

- Identify equivalence classes, and
- Merge states in an equivalence class
- $L(M \nabla M') \supseteq L(M) \cup L(M')$

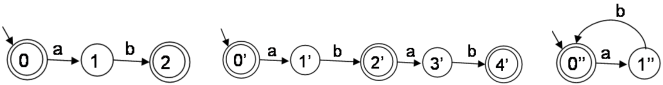
State Equivalence

q, q' are equivalent if one of the following condition holds:

- $\forall w \in \Sigma^*$, w is accepted by M from q then w is accepted by M' from q', and vice versa.
- $\exists w \in \Sigma^*$, M reaches state q and M' reaches state q' after consuming w from its initial state respectively.
- $\exists q''$, q and q'' are equivalent, and q' and q'' are equivalent.

An Example for $M \nabla M'$

- $L(M) = \{\epsilon, ab\}$ and $L(M') = \{\epsilon, ab, abab\}$.
- The set of equivalence classes: $C = \{q''_0, q''_1\}$, where $q''_0 = \{q_0, q'_0, q_2, q'_2, q'_4\}$ and $q''_1 = \{q_1, q'_1, q'_3\}$.



(a) M (b) M' (c) $M \nabla M'$

Figure: Widening automata

Quiz 3

Compute $M \nabla M'$, where $L(M) = \{a, ab, ac\}$ and $L(M') = \{a, ab, ac, abc, acc\}$.

A Fixed Point Computation

Recall that we want to compute the least fixpoint that corresponds to the reachable values of string expressions.

- The fixpoint computation will compute a sequence M_0, M_1, ..., M_i, ..., where $M_0 = I$ and $M_i = M_{i-1} \cup post(M_{i-1})$

A Fixed Point Computation

Consider a simple example:

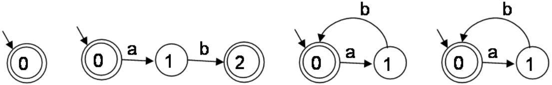
- Start from an empty string and concatenate *ab* at each iteration
- The exact computation sequence M_0, M_1, ..., M_i, ... will never converge, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 1 \leq k \leq i\} \cup \{\epsilon\}$.

Accelerate The Fixed Point Computation

Use the widening operator ∇.

- Compute an over-approximate sequence instead: M'_0, M'_1, ..., M'_i, ...
- $M'_0 = M_0$, and for $i > 0$, $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}))$.

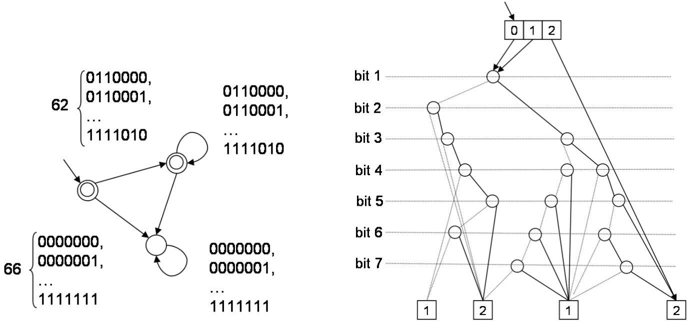
An over-approximate sequence for the simple example:



(a) M'_0 (b) M'_1 (c) M'_2 (d) M'_3

Automata Representation

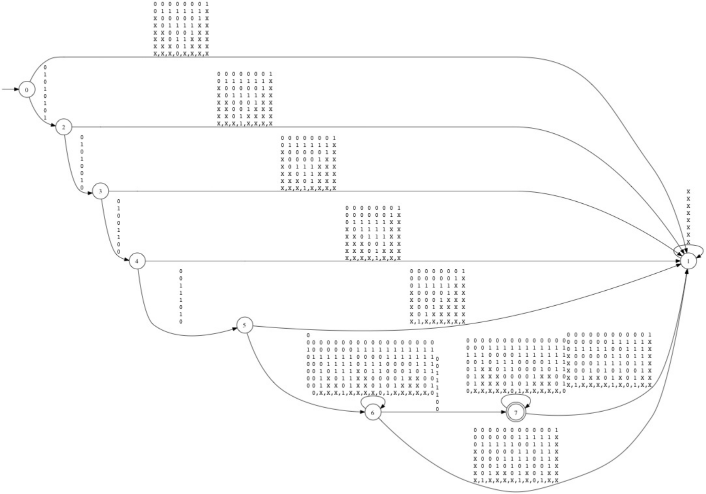
A DFA Accepting [A-Za-z0-9]* (ASC II).



(a) Explicit Representation

(b) Symbolic Representation

Another Automata Example



Automatic Verification of String Manipulating Programs

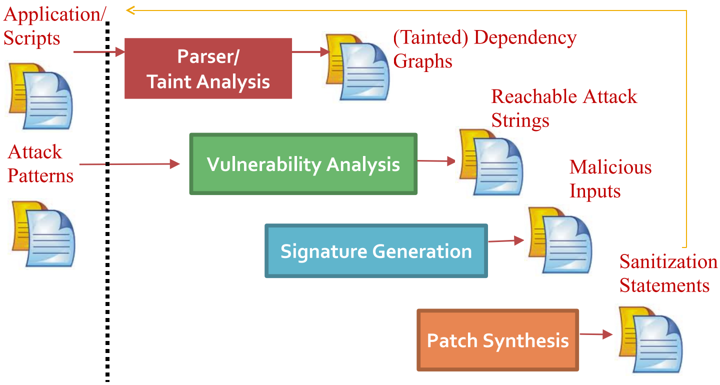
- Symbolic String Vulnerability Analysis
- Relational String Analysis
- Composite String Analysis

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

String Analysis Stages



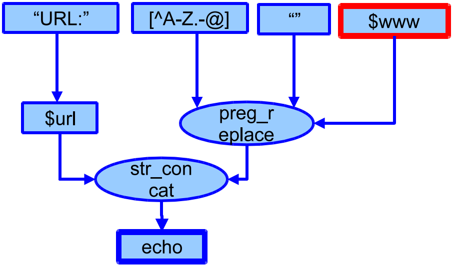
Front End

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)



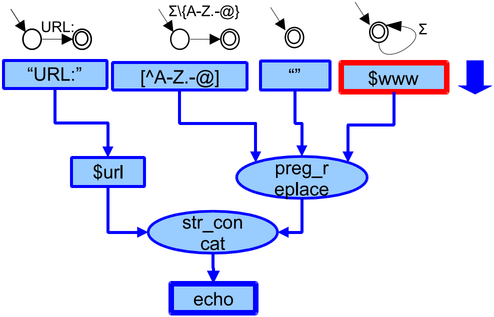
NEXT: Compute all possible values of a sink node

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

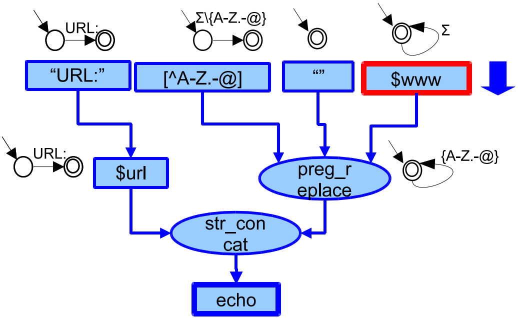
Forward Analysis

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



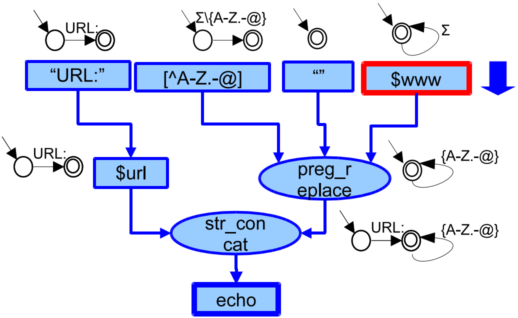
Forward Analysis

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



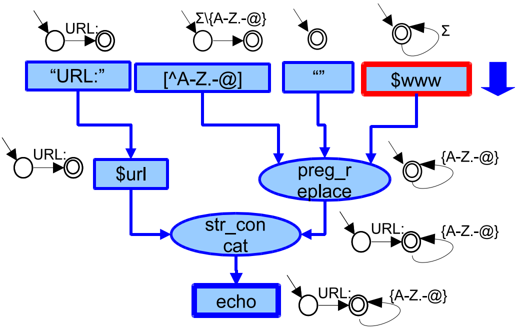
Forward Analysis

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



Forward Analysis

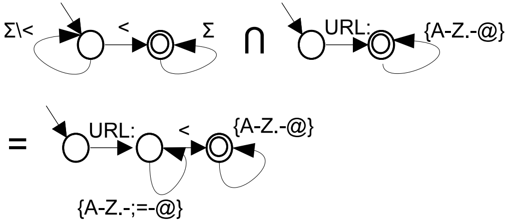
- The third iteration is a simple assignment
- After the third iteration, we reach a fixed point



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



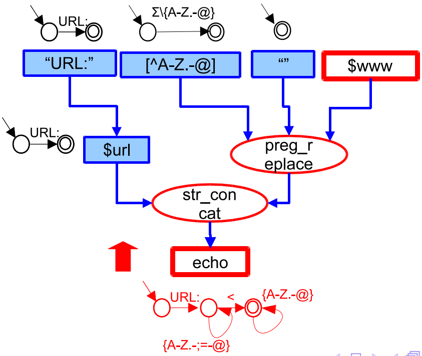
NEXT: What are the malicious inputs?

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`)` retunrs M_1, where $M = \text{REPLACE}(M_1, M_2, M_3)$.

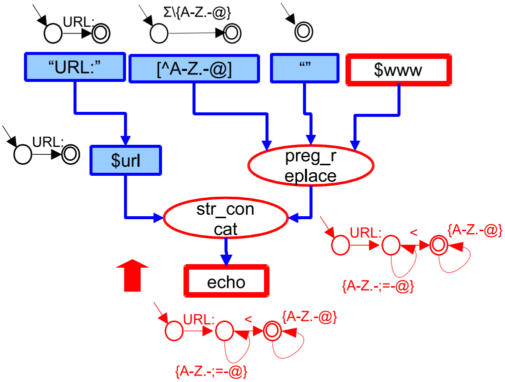
Backward Analysis

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



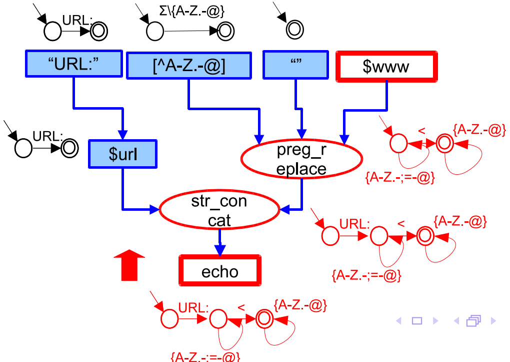
Backward Analysis

- The first iteration is a simple assignment.



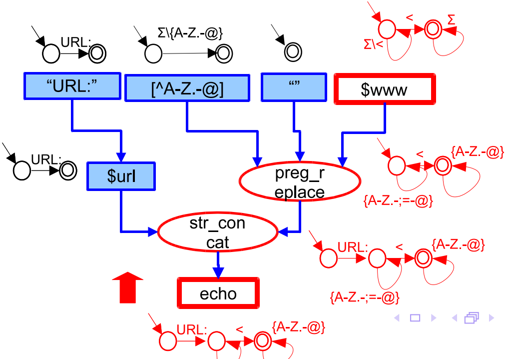
Backward Analysis

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



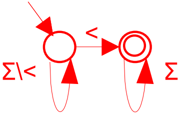
Backward Analysis

- We call `preReplace`($\{A-Z.-;=-@\}^* < \{A-Z.-@\}^*$, $\Sigma \setminus \{A-Z.-@\}$, `""`) at the third iteration.
- M = `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

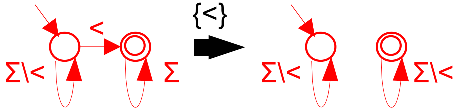
Patch Vulnerable Applications

- 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

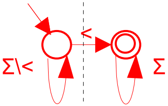
The idea is to modify the input by deleting certain characters (as little as possible) so that it does not match the vulnerability signature

- 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.

```
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
- 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

Automatic Verification of String Manipulating Programs

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

Relational String Analysis

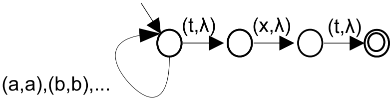
Instead of multiple *single*-track DFAs, we use one *multi*-track DFA, where each track represents the values of one string variable.

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
- Each constant node is a single track automaton (the auxiliary track) accepting the constant string
- Each user input node is a two track automaton (an input track + the auxiliary track) accepting strings that two tracks have the same value

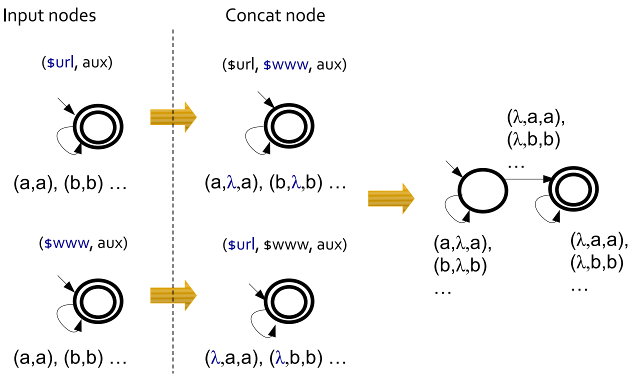
Relational Vulnerability Signature

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 $(\Sigma \setminus <)^* < \Sigma^*$

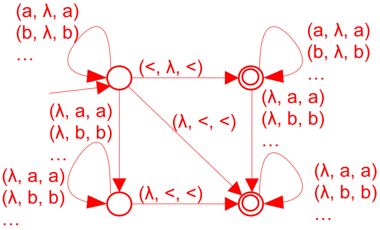
Signature Generation



Relational Vulnerability Signature

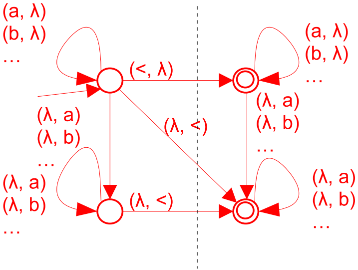
Upon termination, intersects the auxiliary track with the attack pattern

- 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.

Experiments on Relational String Analysis

Basic benchmarks:

- Implicit equality properties
- Branch and loop structures

MFE benchmarks:

- Each benchmark represents a MFE vulnerability
  - M1: PBLguestbook-1.32, pblguestbook.php(536)
  - M2, M3: MyEasyMarket-4.1, prod.php (94, 189)
  - M4, M5: php-fusion-6.01, db_backup.php (111), forums_prune.php (28).
- We check whether the retrieved files and the external inputs are consistent with what the developers intend.

Experimental Results

Use single-track automata.

	Single-track			
Ben	Result	DFAs/ Composed DFA state(bdd)	Time user+sys(sec)	Mem (kb)
B1	false	15(107), 15(107) /33(477)	0.027 + 0.006	410
B2	false	6(40), 6(40) / 9(120)	0.022+0.008	484
M1	false	2(8), 28(208) / 56(801)	0.027+0.003	621
M2	false	2(20), 11(89) / 22(495)	0.013+0.004	555
M3	false	2(20), 2(20) / 5(113)	0.008+0.002	417
M4	false	24(181), 2(8), 25(188) / 1201(25949)	0.226+0.025	9495
M5	false	2(8), 14(101), 15(108) / 211(3195)	0.049+0.008	1676

Table: false: The property can be violated (false alarms), DFAs: the final DFAs

Experimental Results

Use multi-track automata.

	Multi-track			
Ben	Result	DFA state(bdd)	Time user+sys(sec)	Mem (kb)
B1	true	14(193)	0.070 + 0.009	918
B2	true	5(60)	0.025+0.006	293
M1	true	50(3551)	0.059+0.002	1294
M2	true	21(604)	0.040+0.004	996
M3	true	3(276)	0.018+0.001	465
M4	true	181(9893)	0.784+0.07	19322
M5	true	62(2423)	0.097+0.005	1756

Table: true: The property holds, DFA: the final DFA

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.

String Analysis

Static String Analysis: At each program point, statically compute the possible values of **each string variable**.

The values of each string variable are over approximated as a regular language accepted by a **string automaton** [Yu et al. SPIN08].

String analysis can be used to detect **web vulnerabilities** like SQL Command Injection [Wassermann et al, PLDI07] and Cross Site Scripting (XSS) attacks [Wassermann et al., ICSE08].

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

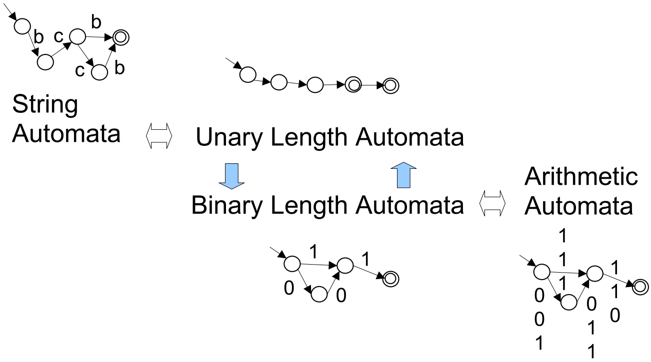
- A string automaton is a single-track DFA that accepts a regular language, whose length forms a semi-linear set, .e.g., $\{4, 6\} \cup \{2 + 3k \mid k \geq 0\}$
- The unary encoding of a semi-linear set is uniquely identified by a unary automaton
- The unary automaton can be constructed by replacing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

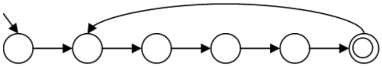
To connect the dots, we propose a novel algorithm to convert unary automata to binary automata and vice versa.



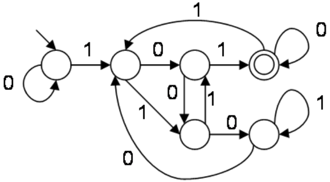
An Example of Length Automata

Consider a string automaton that accepts $(great)^{+}$.
The length set is $\{5 + 5k | k \geq 0\}$.

- 5: in unary 11111, in binary 101, from lsb **101**.
- 1000: in binary 1111101000, from lsb **0001011111**.



(c) Unary

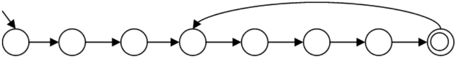


(d) Binary

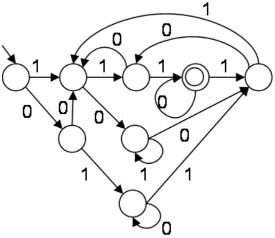
Another Example of Length Automata

Consider a string automaton that accepts $(great)^{+}cs$.
The length set is $\{7 + 5k | k \geq 0\}$.

- 7: in unary 1111111, in binary 1100, from lsb **0011**.
- 107: in binary 1101011, from lsb **1101011**.
- 1077: in binary 10000110101, from lsb **10101100001**.



(e) Unary



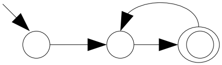
(f) Binary

From Unary to Binary

Given a unary automaton, construct the binary automaton that accepts the same set of values in binary encodings (starting from the least significant bit)

- Identify the semi-linear sets
- Add binary states incrementally
- Construct the binary automaton according to those binary states

Identify the semi-linear set



- A unary automaton M is in the form of a lasso
- Let C be the length of the tail, R be the length of the cycle
- $\{C + r + Rk \mid k \geq 0\} \subseteq L(M)$ if there exists an accepting state in the cycle and r is its length in the cycle
- For the above example
  - $C = 1$, $R = 2$, $r = 1$
  - $\{1 + 1 + 2k \mid k \geq 0\}$

Binary states

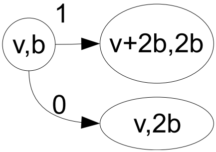
A binary state is a pair (v, b):

- v is the integer value of all the bits that have been read so far
- b is the integer value of the last bit that has been read
- Initially, v is 0 and b is undefined.

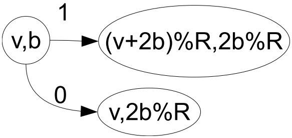
The Binary Automaton Construction

We construct the binary automaton by adding binary states accordingly

- Once $v + 2b \geq C$, v and b are the remainder of the values divided by R
- (v, b) is an *accepting* state if v is a remainder and $\exists r . v = (C + r)\%R$
- The number of binary states is $O(C^2 + R^2)$



(g) $v + 2b < C$

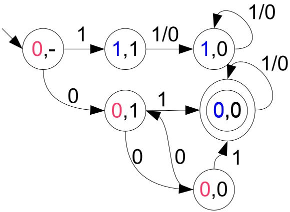


(h) $v + 2b \geq C$

The Binary Automaton Construction

Consider the previous example, where $C = 1$, $R = 2$, $r = 1$.

- (0, 0) is an accepting state, since $\exists r.r = 1, (C + v)\%R = (1 + 0)\%2 = 1$



The Binary Automaton Construction

After the construction, we apply *minimization* and get the final result.

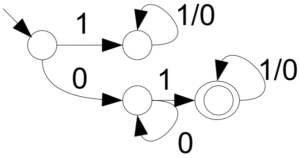


Figure: A binary automaton that accepts {2+2k}

Quiz 4

Consider a string automaton that accepts $auto(good)^+$.

- Compute the semi-linear set.
- Construct the binary automata that accepts the semi-linear set

From Binary to Unary

Given a binary automaton, construct the unary automaton that accepts the same set of values in unary encodings

- There exists a binary automaton, e.g., $\{2^k \mid k \geq 0\}$, that cannot be converted to a unary automaton precisely.
- We adopt an *over*- approximation:
  - Compute the minimal and maximal accepted values of the binary automaton
  - Construct the unary automaton that accepts the values in between

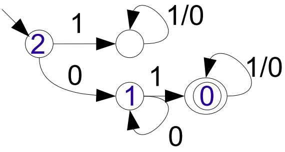
Compute the Minimal/Maximal Values

- The minimal value forms the shortest accepted path
- The maximal value forms the longest loop-free accepted path (If there exists any accepted path containing a cycle, the maximal value is inf)
- Perform BFS from the accepting states (depth is bounded by the number of states)
    - Initially, both values of the accepting states are set to 0
    - Update the minimal/maximal values for each state accordingly

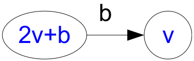
The Unary Automaton Construction

Consider our previous example,

- min = 2, max = inf
- An *over* approximation: $\{2 + 2k \mid k \geq 0\} \subseteq \{2 + k \mid k \geq 0\}$



Computing the minimal value



The value of the previous state

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)

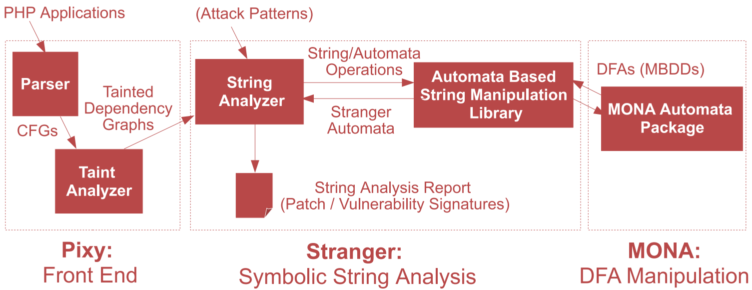
STRANGER Tool

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][Veneas et al., POPL'15][Wang et al. CAV'16]
- String constraint solving: [CVC4] [Z3, Z3-Str, Z3-Str2,2016] [SSS, S3P] [Norn] [Slog, Slender (Wang, Jiang and Yu. Coming soon)]
- 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][Yu et al. ISSTA'16]

Thank you for your attention.

Questions?