Automatic Verification of String Manipulating Programs

Fang Yu

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

CS267 Guest Leture
November 18, 2009

Overview

We investigate string verification problem and present an automata-based approach for *automatic verification* of string manipulating programs based on symbolic string analysis.

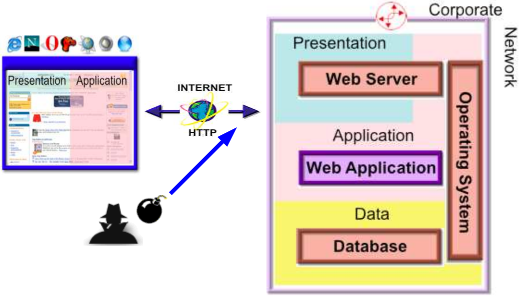
String analysis plays an important role in the security area. For instance, one can detect various web vulnerabilities like SQL Command Injection and Cross Site Scripting (XSS) attacks.

Web Application Vulnerabilities

- The top three vulnerabilities in OWASPs top ten list (2007)
  1. Cross Site Scripting (XSS)
  2. Injection Flaws (such as SQL injection)
  3. Malicious File Execution (MFE)

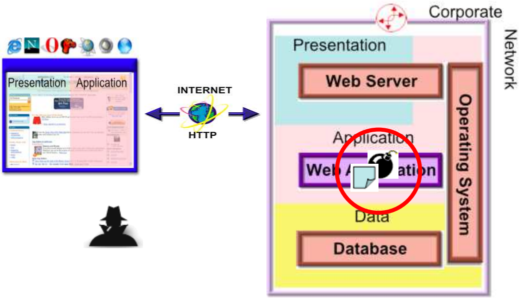
Injection Flaws

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



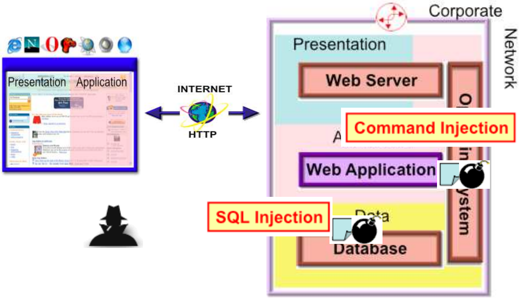
Injection Flaws

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



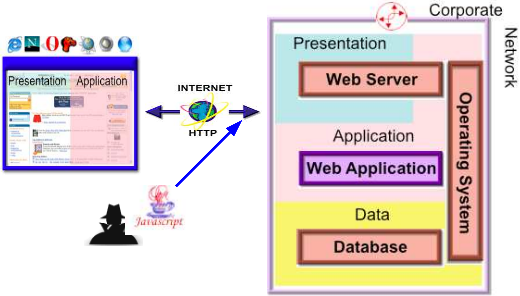
Injection Flaws

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



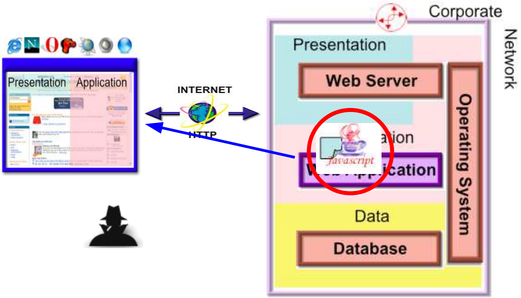
XSS Attacks

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



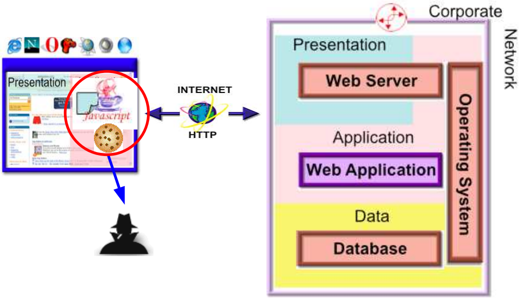
XSS Attacks

- 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.



Is it Vulnerable?

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

Is it Vulnerable?

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], 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 used in MyEasyMarket-4.1 that causes a known vulnerable point at line 218 in trans.php

Sanitization Routines are Erroneous

Our string analysis identifies that the segment is vulnerable. Furthermore,

- We generate vulnerability signature that characterizes *all* malicious inputs that may generate attacks
- 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 .-@:/ $\}$
- Any string accepted by this signature may yield an attack

Sanitization Routines are Erroneous

For example, a malicious input can be <!sc+rip!t ...> which does not match the attack pattern Σ^* <scriptΣ^*.

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

- One can filter out all malicious inputs using our signature

Is it Vulnerable?

Fix the sanitization 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 vulnerable against the XSS attack pattern: $\Sigma^* <\text{script}\Sigma^*$.

Automatic Verification of String Manipulating Programs

We can

1. Detect vulnerabilities in web applications that are due to string manipulation
2. Prove the absence of vulnerabilities in web applications that use proper sanitization
3. Generate a characterization of all malicious inputs that may compromise a vulnerable web application

We achieve this goal by an automata-based symbolic string analysis approach.

Part I: String Verification

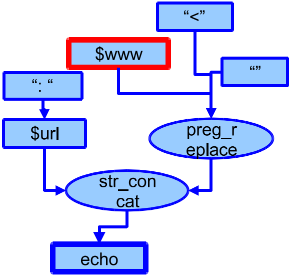
Verification Framework

- Convert PHP programs to dependency graphs with string manipulation operations
- Associate each node with an automaton that accepts an over approximation of its possible values
- Combine forward and backward symbolic reachability analyses

Verification Framework

- A dependency graph specifies how the values of input nodes flow to a sink

```
<?php
  $www = $_GET["www"];
  $url = “:”;
  $www = preg_replace(
               "<",
               "",
               $www);
  echo  $url . $www;
 ?>
```



Verification Framework

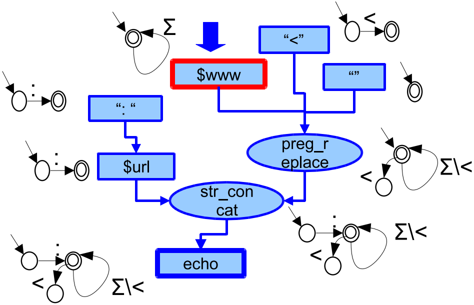
Detecting vulnerabilities

- 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`)` for $M := M_1.M_2$, and
  - `postReplace(`M_1`,` M_2`,` M_3`)` for $M := replace(M_1, M_2, M_3)$

Verification Framework

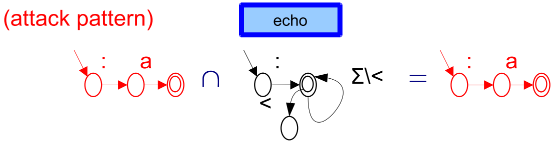
Forward analysis

- Allows *arbitrary* values from user inputs
- Propagates post-images to next nodes



Detecting Vulnerabilities

- Intersects the result of the sink node with the attack pattern
- If the intersection is empty then the program is not vulnerable with respect to the attack pattern. Otherwise, it is vulnerable



Verification Framework

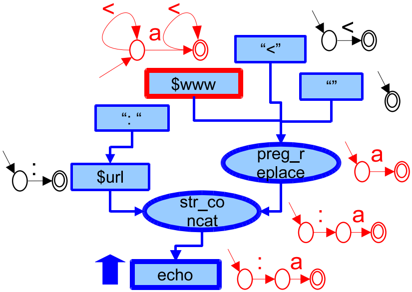
Generating vulnerability signatures

- 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`$(M,\ M_2)$, `preConcatSuffix`$(M,\ M_1)$ for $M := M_1.M_2$ and
  - `preReplace`$(M,\ M_2,\ M_3)$ for $M := replace(M_1, M_2, M_3)$.

Verification Framework

Backward analysis

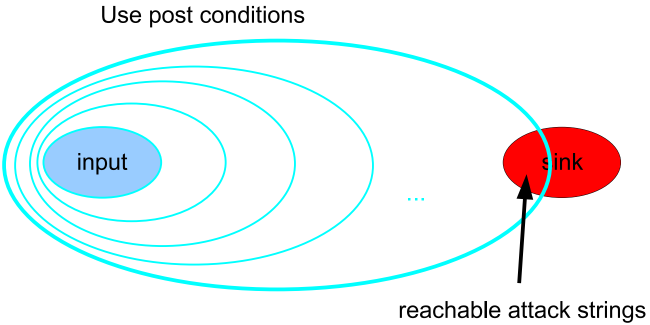
- Computes pre-images along with the path to the user input
- Uses results from forward analysis



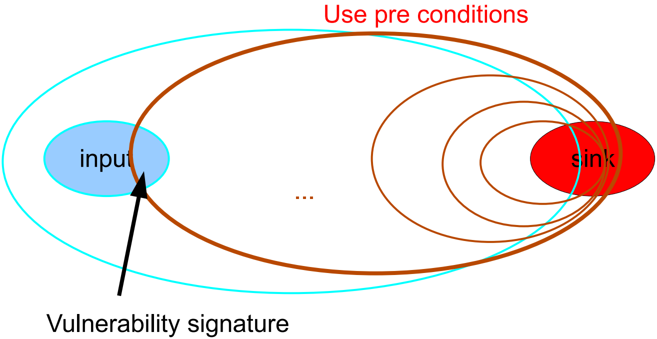
Symbolic Fixpoint Computations

- Iteratively,
    - Computes the next state of current automata against string operations and
    - Updates automata by joining the result to the automata at the next statement
- Terminates the execution upon reaching a fixed point
- We use an automata based widening operation that over-approximates the least fixpoint and accelerates convergence

Forward Fixpoint Computation



Backward Fixpoint Computation



Challenges

- **Precision**: Need to deal with sanitization routines having PHP string functions, e.g., `ereg_replacement`.
- **Complexity**: The problem in general is undecidable. The fixed point may not exist and even if it exists the fixpoint computation may not converge.
- **Performance**: Need to perform automata manipulations efficiently in terms of both time and memory.

Selected Features of Our Approach

We propose:

- A Language-based Replacement: To model *replacement* operations in PHP programs.
- A Pre-condition computation: To perform backward analysis
- An Automata Widening Operator: To accelerate fixed point computation.
- A Symbolic Encoding: Using Multi-terminal Binary Decision Diagrams (MBDDs) from MONA DFA packages.

Technical Details

1. **Replacement**
2. Pre-condition
3. Widening
4. Symbolic Encoding

A Language-based Replacement

M=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.

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

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}	{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.

A Language-based Replacement

Implementation of 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 alphabet: $\Sigma' = \{\alpha' | \alpha \in \Sigma\}$.

An Example

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

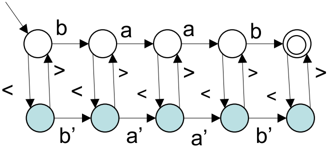
- $L(M_1) = \{baab\}$
- $L(M_2) = a^+ = \{a, aa, aaa, \ldots\}$
- $L(M_3) = \{c\}$

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 $b < a'a' > b$, $b < a' > ab$.

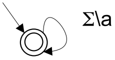


Step 2

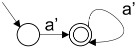
Construct M_2' from M_2:

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

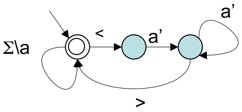
For instance, M_2' accepts $b < a'a' > b$, $b < a' > bc < a' >$.



(a)



(b)



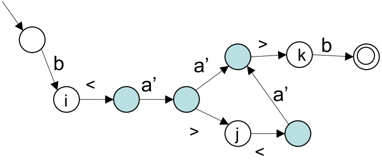
(c)

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 identify three pairs:(i,j), (i,k), (j,k).

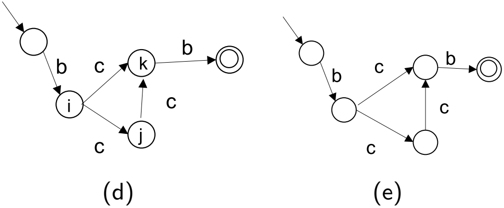


Step 4

Construct M:

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

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



***The details can be found in [SPIN08]*

Technical Details

1. Replacement
2. **Pre-condition**
3. Widening
4. Symbolic Encoding

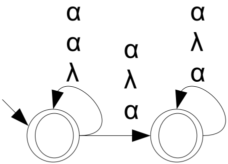
Pre-conditions of String Concatenation

We introduce *concatenation transducer* to specify the relation of $X = YZ$.

- 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 $w[3]$

Concatenation Transducer: $X = YZ$

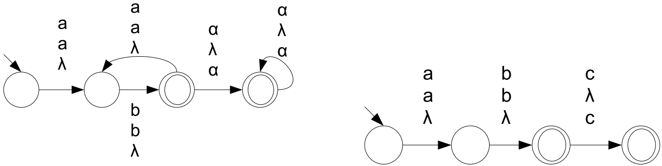
Let α be any character in Σ.



Suffix: Given X and Y to Compute Z

Consider the precondition of an assignment $X := (ab)^+.Z$. 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., $\{\epsilon, c\}$.

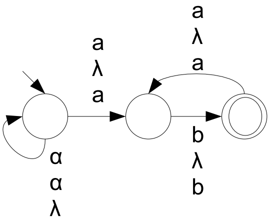


(a) M (b) After the intersection

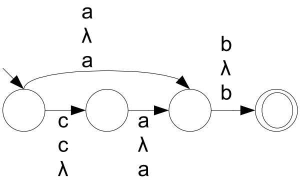
Prefix: Given X and Z to Compute Y

Consider the precondition of an assignment $X := Y.(ab)^+$.
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., $\{\epsilon, c\}$.



(a) M



(b) After the intersection

Technical Details

1. Replacement
2. Pre-condition
3. **Widening**
4. Symbolic Encoding

Widening Automata: $M \nabla M'$

This widening operator was originally proposed by Bartzis and Bultan [CAV04]. Intuitively,

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

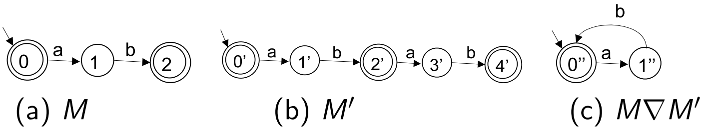
State Equivalence

q, q' are equivalent if one of the following conditions 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

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:

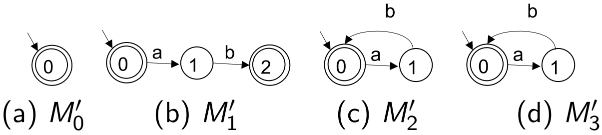
- Start from an empty string and concatenate *ab* in a loop
- The exact computation sequence M_0, M_1, ..., M_i, ... will never converge, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 1 \leq k \leq i\} \cup \{\epsilon\}$.

Accelerate The Fixed Point Computation

Use the widening operator ∇.

- Compute an over-approximation 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*-approximation sequence for the simple example:



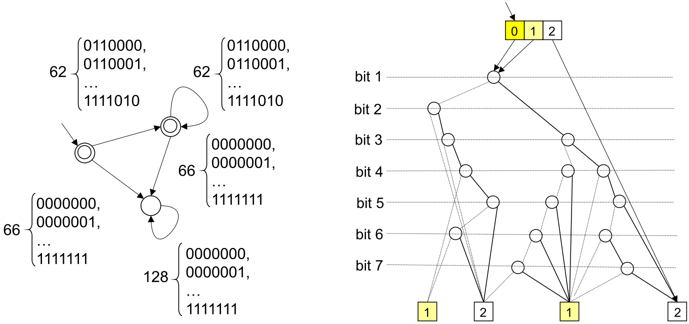
(a) M'_0 (b) M'_1 (c) M'_2 (d) M'_3

Technical Details

1. Replacement
2. Pre-condition
3. Widening
4. **Symbolic Encoding**

Automata Representation

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



(a) Explicit Representation

(b) Symbolic Representation

Implementation

We used the MONA DFA Package. [Klarlund and Møller, 2001]

- Compact Representation:
  - Canonical form and
  - Shared BDD nodes
- Efficient MBDD Manipulations:
  - Union, Intersection, and Emptiness Checking
  - Projection and Minimization
- Cannot Handle Nondeterminism:
  - We used dummy bits to encode nondeterminism

Benchmarks

In [SPIN08], we reported experiments on test cases extracted from real-world, open source applications:

- MyEasyMarket-4.1 (a shopping cart program)
- PBLguestbook-1.32 (a guestbook application)
- Aphpkb-0.71 (a knowledge base management system)
- BloggIT-1.0 (a blog engine)
- proManager-0.72 (a project management system)

Benchmarks

Generate benchmarks.

- Select vulnerable points based on the result of Saner [SPP08]. (with Marco Cova)
- For each selection, we manually generate two test cases:
  - A sliced code segment from the original program (denoted as "o"), in which we only consider statements that influence the selected vulnerable point(s)
  - A modified segment with extra/fixed sanitization routines (denoted as "m")

Benchmarks

Here are some statistics about the benchmarks:

Application File(line)	Benchmark Index	No. of Constr.	No. of Concat.	No. of Repl.
MyEasyMarket-4.1	o1	11	4	1
trans.php(218)	m1	11	4	1
PBLguestbook-1.32	o2	19	15	1
pblguestbook.php(1210)	m2	19	16	1
PBLguestbook-1.32	o3	6	7	0
pblguestbook.php(182)	m3	14	8	4
Aphpkb-0.71	o4	4	3	1
saa.php(87)	m4	8	3	3
BloggIT 1.0	o5	21	12	8
admin.php(23, 25, 27)	m5	23	12	10
proManager-0.72	o6	39	31	9
message.php(91)	m6	45	31	12

Experimental Results

We compare our results against Saner [SPP08].

Idx	Res.	Final DFA state(bdd)	Peak DFA state(bdd)	Time user+sys(sec)	Mem (kb)	Saner n(type)	Saner Time(sec)
o1	y	17(133)	17(148)	0.010+0.002	444	1(xss)	1.173
m1	n	17(132)	17(147)	0.009+0.001	451	0	1.139
o4	y	27(219)	289(2637)	0.045+0.003	2436	1(xss)	1.220
m4	n	18(157)	1324(15435)	0.177+0.009	11388	0	1.622
o6	y	387(3166)	2697(29907)	1.771+0.042	13900	1(xss)	6.980
m6	n	423(3470)	2697(29907)	2.091+0.051	19353	0	7.201

- Res.
    - y: the intersection of attack strings is not empty (vulnerable)
    - n: the intersection of attack strings is empty (secure).

Experimental Results

We compare our results against Saner [SPP08].

Idx	Res.	Final DFA state(bdd)	Peak DFA state(bdd)	Time user+sys(sec)	Mem (kb)	Saner n(type)	Saner Time(sec)
o2	y	42(329)	42(376)	0.019+0.001	490	1(sql)	1.264
m2	n	49(329)	42(376)	0.016+0.002	626	1(sql)	1.665
o3	y	842(6749)	842(7589)	2.57+0.061	13310	1(reg)	4.618
m3	n	774(6192)	740(6674)	1.221+0.007	8184	1(reg)	4.331
o5.1	y	79(633)	79(710)	0.499+0.002	3569	0	0.558
o5.2	y	126(999)	126(1123)				
o5.3	y	138(1095)	138(1231)				
m5.1	n	79(637)	93(1026)	0.391+0.006	5820	0	0.559
m5.2	n	115(919)	127(1140)				
m5.3	n	127(1015)	220(2000)				

- type:(1) xss - cross site scripting vulnerablity, (2) sql - SQL injection vulnerability, (3) reg - regular expression error.

Vulnerability Experiments

We conduct vulnerability analysis on the following vulnerable benchmarks. The results are reported in [ASE09].

- (1) `MyEasyMarket-4.1` (a shopping cart program),
- (2) `PBLguestbook-1.32` (a guestbook application),
- (3) `BloggIT-1.0` (a blog engine), and
- (4) `proManager-0.72` (a project management system).

Basic information

Here are some data of the dependency graphs.

	vul	#nodes	#edges	#sinks	#inputs	#literals
1	1(xss)	21	20	1	1	51
2	1(sql)	41	44	1	2	99
3	1(xss)	32	31	1	1	142
4	3(xss)	119	117	3	3	450

Table: Dependency Graphs

Fwd v.s. Bwd Analyses

Time Performance.

	total time(s)	fwd time(s)	bwd time(s)	mem(kb)
1	0.569	0.093	0.474	2700
2	3.449	0.124	3.317	5728
3	1.087	0.248	0.836	18890
4	16.931	0.462	16.374	116097

Table: Total Performance

Post- v.s. Pre-condition Computations

	CONCAT	REPLACE	PRECONCAT	PREREPLACE
	#operations/time(s)			
1	6/0.015	1/0.004	2/0.411	1/0.004
2	19/0.082	1/0.004	11/3.166	1/0.0
3	22/0.038	4/0.112	2/0.081	4/0.54
4	14/0.014	12/0.058	26/11.892	24/3.458

Table: String Function Performance

Vulnerability Signatures

Here are some data about the generated automata.

	Reachable Attack (Sink)		Vulnerability Signature (Input)	
	#states	#bdd nodes	#states	#bdd nodes
1	24	225	10	222
2	66	593	2	9
			2	9
3	29	267	92	983
4	131	1221	57	634
	136	1234	174	1854
	147	1333	174	1854

Table: Attack and Vulnerability Signatures

What should you know?

A symbolic approach for string verification on PHP programs

- A general forward and backward verification framework
- A language-based replacement
- A weakest pre-condition computation on concatenation
- An automaton-based widening operator

Our string analysis tool can be downloaded from:
http://www.cs.ucsb.edu/∼ vlab/stranger

Part II: Composite Verification

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 a Presburger arithmetic and represented as an arithmetic automaton [Bartzis and Bultan, CAV03].

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 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?

To do so, we introduce **Length Automata**.

Some Facts about String Automata

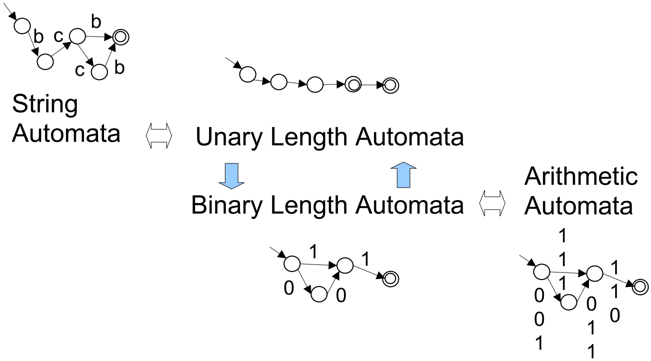
- A string automaton is a single-track DFA that accepts a regular language, whose length forms a semi-linear set, .e.g., $\{4, 6\} \cup \{2 + 3k \mid k \geq 0\}$
- The unary encoding of a semi-linear set is uniquely identified by a unary automaton
- The unary automaton can be constructed by replacing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



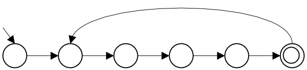
String Automata ⟺ Unary Length Automata

Binary Length Automata ⟺ Arithmetic Automata

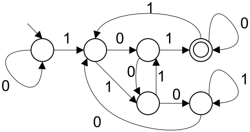
An Example of Length Automata

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**.



Unary

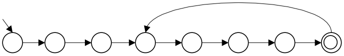


Binary

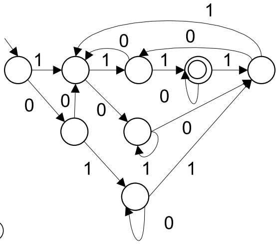
Another Example of Length Automata

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**.



Unary



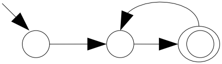
Binary

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

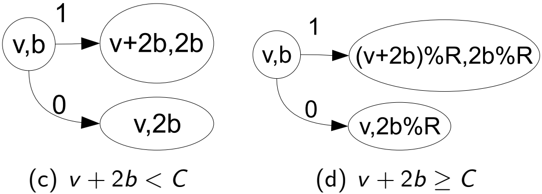
A binary state is a pair (v, b):

- 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 $\exists r.r = (C + v)\%R$
- The number of binary states is $O(C^2 + R^2)$

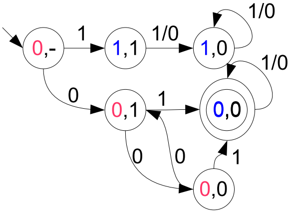


(c) $v + 2b < C$ (d) $v + 2b \geq C$

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.

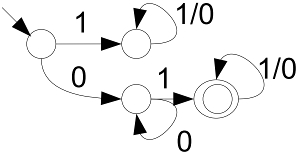


Figure: A binary automaton that accepts {2+2k}

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

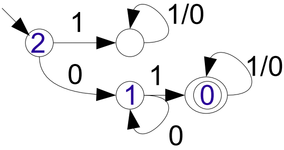
Compute the Minimal/Maximal Values

- 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

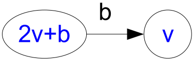
The Unary Automaton Construction

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

Composite Verification

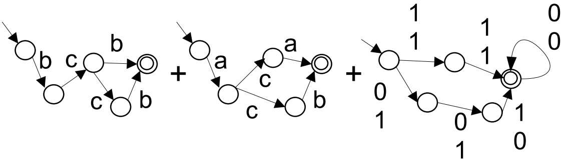
We perform composite verification on a simple imperative language that supports:

- Branch and goto statements (path sensitive)
  - Branch conditions can be membership of regexp on string variables or a Presburger formula on integers and the length of string variables.
- String operations including concatenation, prefix, suffix, and language-based replacement.
- Linear arithmetic computations on integers

A Composite State

A composite state consists of the states of :

- Multiple single-track string automata (Each string automaton accepts the values of a string variable)
- A multi-track arithmetic automaton (Each track accepts the length of a string variable or the value of an integer variable)



Forward Fixpoint Computation

- We iteratively compute and add the post images of the composite states for each program label until reaching a fixpoint
- The post image is defined on the composite state
  - String → (Unary → Binary) → Arithmetic
  - Arithmetic → (Binary → Unary) → String
- We also incorporate the widening operator on automata to accelerate the fixpoint computation

Implementation

We implemented a prototype tool on top of

- Symbolic String Analysis [Yu et al. SPIN08]
- Arithmetic Analysis [Bartzis et al. CAV03]
- Automata Widening [Bartzis et al. CAV04]

***Both string and arithmetic automata are symbolically encoded by using the MONA DFA Package.*

Experiments

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

- C string library
- Buffer overflow benchmarks [Ku et al., ASE07]
- Web vulnerable applications [Balzarotti et al., SSP08]

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 SQL attack free)

Related Publications

- STRANGER*: An Automata-based String Analysis Tool for PHP*
  Fang Yu, Muath Alkhalaf, Tevfik Bultan.
  Under submission.
- *Verification of String Manipulating Programs Using Multi-track Automata*
  Fang Yu, Tevfik Bultan, Oscar H. Ibarra.
  Under submission.
- *Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses*
  Fang Yu, Muath Alkhalaf, and Tevfik Bultan.
  Short paper. Accepted for Publication in the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009).
- *Symbolic String Verification: Combining String Analysis and Size Analysis*
  Fang Yu, Tevfik Bultan, and Oscar H. Ibarra.
  In Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), LNCS 5505, pages 322-336, York, UK, Mar. 2009.
- *Symbolic String Verification: An Automata-based Approach*
  Fang Yu, Tevfik Bultan, Marco Cova, Oscar H. Ibarra.
  In Proceedings of the 15th International SPIN Workshop on Model Checking of Software (SPIN 2008), LNCS 5156, pages 306-324, Los Angeles, CA, August 2008.

Related Work on Static String Analysis

Grammar-based String Analysis:

- Java String Analyzer [Chris and Moller, SAS03]
- Valid Web Pages [Minamide, WWW05]
- Injection Vulnerability [Wassermann and Su, PLDI07]

Our papers [SPIN08, TACAS09] are also cited by:
A decision procedure for subset constraints over regular languages.
[P. Hooimeijer and W. Weimer, PLDI09]

Thank you for your attention.

Questions?