Symbolic String Verification: An Automata-based Approach

Fang Yu Tevfik Bultan Marco Cova Oscar H. Ibarra

Dept. of Computer Science
University of California Santa Barbara, USA
{yuf, bultan, marco, ibarra}@cs.ucsb.edu

August 11, 2008

1. Motivation
   - Goal
   - Is it vulnerable?
2. Symbolic String Verification
   - Verification Framework
   - A Language-based Replacement
   - Widening Automata
   - Symbolic Encoding
3. Experiments
   - Benchmarks
   - Results
4. Conclusion

Motivation

We aim to develop an *efficient* but rather *precise* string verification tool based on static string analysis.

Static String Analysis: At each program point, statically compute all possible values that string variables can take.

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.

Is it vulnerable?

A program is vulnerable if a sensitive function can take an attack string (specified by an attack pattern) as its input.

A PHP Example: (A XSS attack pattern for echo: $\Sigma^* < script\Sigma^*$)

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

A simple taint analysis [Huang et al. WWW04] can report this segment vulnerable.

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:?>

This segment is identified to be vulnerable by dynamic testing (Balzarotti et al.)[SSP08]. (A vulnerable point at line 218 in trans.php, distributed with MyEasyMarket-4.1.)

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:?>

By our approach, this segment is proven not vulnerable against the XSS attack pattern: $\Sigma^* < script\Sigma^*$.

Verification Framework

- Associate each string variable at each program point with an automaton that accepts an over approximation of its possible values.
- Use these automata to perform a forward symbolic reachability analysis.
- Iteratively
  - Compute the next state of current automata against string operations and
  - Update automata by joining the result to the automata at the next statement
- Terminate the execution upon reaching a fixed point.

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.

Features of Our Approach

We propose:

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

A Language-based Replacement

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

- M_1, M_2, and M_3 are Deterministic Finite Automata (DFAs).
  - M_1 accepts the set of original strings,
  - M_2 accepts the set of match strings, and
  - M_3 accepts the set of replacement strings
- Let $s \in L(M1)$, $x \in L(M2)$, 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.

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

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

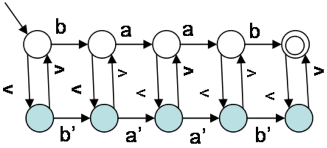
- $L(M_1) = \{baab\}$
- $L(M_2) = a^+ = \{a, aa, aaa, \ldots\}$
- $L(M_3) = \{c\}$

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

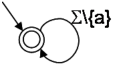


Step 2

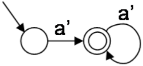
Construct M_2' from M_2:

- (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.

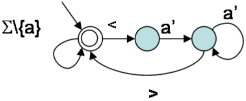
For instance, M_2' accepts $b < a'a' > b$, $b < a' > bc < a' >$.



(a)



(b)



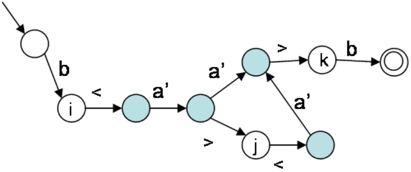
(c)

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).

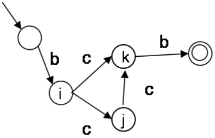


Step 4

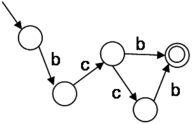
Construct M:

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

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



(d)



(e)

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
- $L(M \nabla M') \supseteq L(M) \cup L(M')$

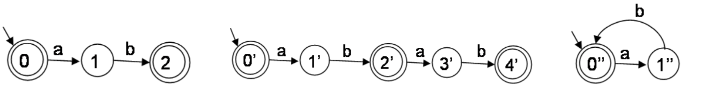
State Equivalence

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:

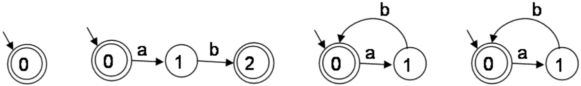
- Start from an empty string and concatenate *ab* in a loop
- The exact computation sequence M_0, M_1, ..., M_i, ... will never converge, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 1 \leq k \leq i\} \cup \{\epsilon\}$.

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 $i > 0$, $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}))$.

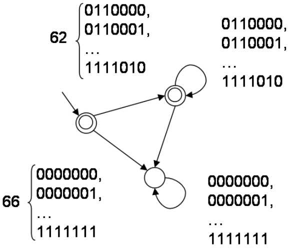
An over-approximation sequence for the simple example:



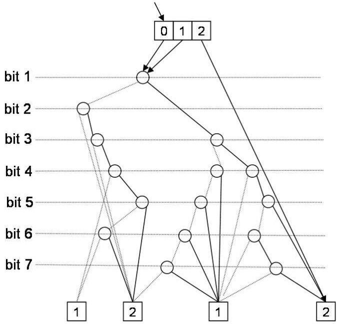
(a) M'_0 (b) M'_1 (c) M'_2 (d) M'_3

Automata Representation

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



(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

We experimented 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].
- For each selection, we manually generate two test cases:
  - A sliced code segment from the original program, in which we only consider statements that influence the selected vulnerable point(s)
  - A modified segment with extra/fixed sanitization routines

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.

Conclusion

A symbolic approach for string verification on PHP programs

- A general verification framework
- A language-based replacement
- An automaton-based widening operator
- Experimental results are promising

Benchmarks can be downloaded from:
http://www.cs.ucsb.edu/∼ yuf/spin.benchmarks.tar.gz

Questions?

Related Works

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

Future Works

- Compact Automata Representation and Manipulation
- Composite Analysis on Strings and Integers