Symbolic String Verification: Combining String Analysis and Size Analysis

Fang Yu Tevfik Bultan Oscar H. Ibarra

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

TACAS 2009, York, UK

Motivation

We aim to develop a verification tool for analyzing infinite state 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 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?

A motivating example from trans.php, distributed with MyEasyMarket-4.1.

- 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

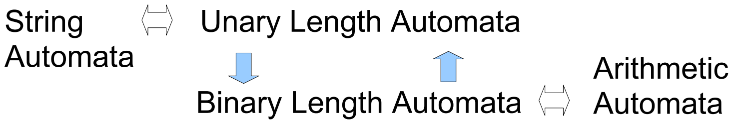
- A string automaton is a single-track DFA that accepts a regular language, whose length forms a semi-linear set, .e.g., $\{4, 6\} \cup \{2 + 3k \mid k \geq 0\}$.
- The unary encoding of a semi-linear set is uniquely identified by a unary automaton
- The unary automaton can be constructed by replacing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

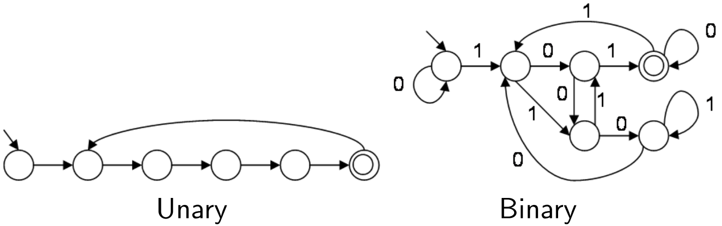
To connect the dots, we need to convert unary automata to binary automata and vice versa.



An Example of Length Automata

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

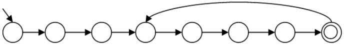
- 5: in unary 11111, in binary 101, from lsb **101**.
- 1000: in binary 1111101000, from lsb **0001011111**.

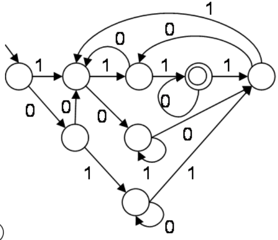


Another Example of Length Automata

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



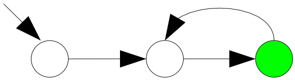
Binary

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

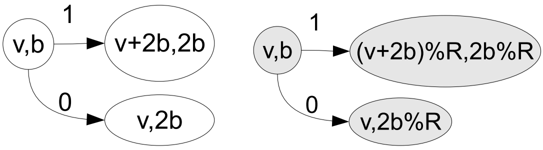
A binary state is a pair (v, b):

- 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 (case (b))
- (v, b) is an accepting state if $\exists r.r = (C + v)\%R$

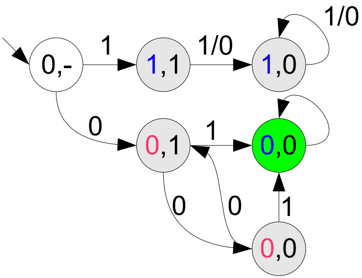


(a) $v + 2b < C$ (b) $v + 2b \geq C$

The Binary Automaton Construction

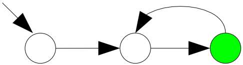
Consider the previous example, where $C = 1$, $R = 2$, $r = 1$.

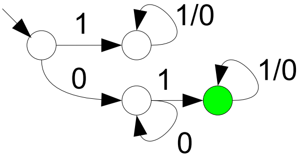
- $0 = (C + r)\%R = (1 + 1)\%2$
- The number of binary states is $O(N^2)$. N is the size of the unary automaton



The Binary Automaton Construction

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


Unary



Binary

From Binary to Unary

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

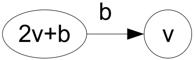
An Over Approximation:

- 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

- Observations:
  - The minimal value forms the shortest accepted path
  - The m
    aximal value forms the longest loop-free accepted path (If there exists any accepted path containing a cycle, the maximal value is $\inf$)

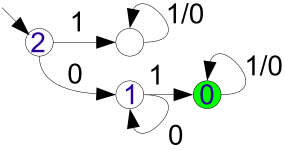
- Perform BFS from the accepting states up to the length of the shortest/longest path. (Both are bounded by the number of states)
  - Initially, both values of the accepting states are set to 0
  - Update the minimal/maximal values for each state accordingly



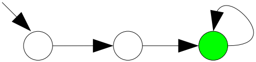
The Unary Automaton Construction

Consider our previous example,

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



The Minimal Value



The Unary Automaton

Some Remarks: From Binary to Unary

- In general, we cannot convert binary to unary automata precisely. (e.g., $\{2^k \mid k \geq 0\}$)
- A unary automaton can only specify a semi-linear set
- Leroux [LICS04] presented an algorithm to identify the presburger formula from an arithmetic automaton, which can be used to improve the precision of our approach

A Simple Imperative Language

We support:

- branch and goto statements
  - 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

Composite State

At each program point, we compute the reachable composite states that consist 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

The computation is based on a standard work queue algorithm.

- We iteratively compute and add the post images 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 incorporate a 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. [Klarlund and Møller, 2001]

- Compact representation and efficient MBDD manipulations

Benchmarks

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 string manipulations.

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: buffer overflow free or SQL attack free

Related Work

- String Analysis:
  - Java String Analyzer (Finite Automata) [Christensen et al., SAS03]
  - PHP String Analyzer (Context Free Grammar) [Minamide, WWW05]

Related Work

- String Analysis:
  - Java String Analyzer (Finite Automata) [Christensen et al., SAS03]
  - PHP String Analyzer (Context Free Grammar) [Minamide, WWW05]
- Integer Analysis:
  - Automaton Construction [Wolper et al., TACAS00]

Related Work

- String Analysis:
  - Java String Analyzer (Finite Automata) [Christensen et al., SAS03]
  - PHP String Analyzer (Context Free Grammar) [Minamide, WWW05]
- Integer Analysis:
  - Automaton Construction [Wolper et al., TACAS00]
- Size Analysis:
  - Buffer Overflow Detection [Dor et al., 2003] [Ganapathy et al., CCS03] [Wagner et al., NDSS00]

Related Work

- String Analysis:
  - Java String Analyzer (Finite Automata) [Christensen et al., SAS03]
  - PHP String Analyzer (Context Free Grammar) [Minamide, WWW05]
- Integer Analysis:
  - Automaton Construction [Wolper et al., TACAS00]
- Size Analysis:
  - Buffer Overflow Detection [Dor et al., 2003] [Ganapathy et al., CCS03] [Wagner et al., NDSS00]
- Composite Analysis:
  - Test Input Generation (Splat) [Xu et al., ISSTA08]

Conclusion

- We presented an automata-based approach for symbolic verification of infinite state systems with unbounded string and integer variables
- We presented a composite verification framework that combines string analysis and size analysis
- We improved the precision of both string and size analysis by connecting the information between them

Thank you for your attention.

Questions?

More Information:
http://www.cs.ucsb.edu/∼bultan/vlab
http://www.cs.ucsb.edu/∼yuf