String Analysis: Techniques and Applications

Lu Zhang

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
  - String Constraint Solver
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
  - String Constraint Solver
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...

Basic Concepts

- String Variables

✓ In strongly typed languages (e.g., Java), String Variables are variables in the program with a string type.

```
str  in  String str;
```

✓ In weakly typed languages (e.g., PHP), String Variables are variables that may be assigned a string value.

```
$str in $str = “abc”;
```

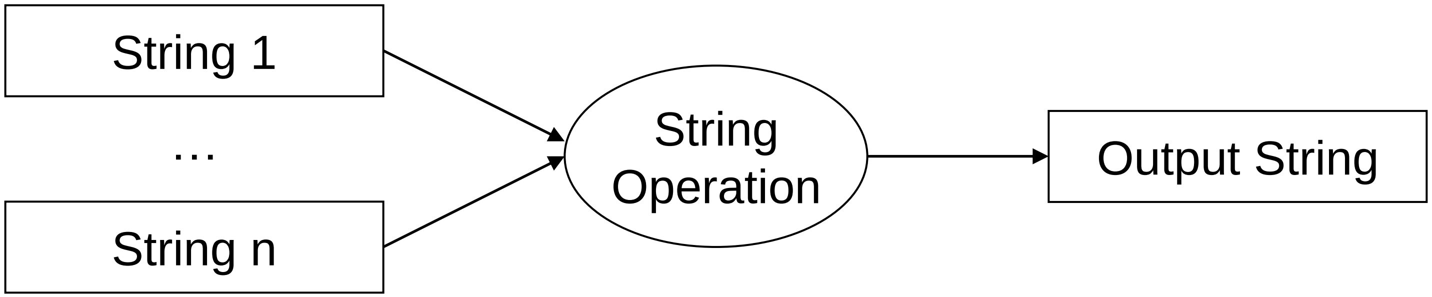
Basic Concepts

- String Constants

  A sequence of characters within a pair of double quotation

- String operations

✓ String operations are library functions that takes several string variables as inputs and output a string variable (i.e., String.length() is usually not considered a string operation)



Basic Concepts

- Common string operations
  - Concatenation

    x = a + b;
  - Replace

    x = a.replace (“a”, “b”);
  - Substring

    x = a.substring(3,5);
  - Tokenize

    x = a.nextToken();
  - …

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...

Basic String Analysis

- Purpose

  Approximately estimate the possible values of a certain string variable in a program

- Hot Spot

  A hot spot is a certain occurrence O of a certain string variable v in the source code, the possible values of the string variable v at the occurrence O require to be estimated.

Basic String Analysis

- String variable with finite possible values

```
01 String str = “abc”
02 if(x>5){
03     str = str + “cd”
04 }
05 System.out.println(str)   <- Hot Spot
```

Possible value of variable str at 05: “abc”, “abccd”

Basic String Analysis

- String variable with infinite possible values

```
01 String str = “|”
02 while(x<readNumber()){
03     str = str + “a”+”|”;
04      x++;
05 }
06 System.out.println(str)   <- Hot Spot
```

Possible value of variable str at 06: “|”, “|a|”, “|a|a|”…

Techniques

- How to deal with infinite possible values?
  - Using formal languages to represent the set of possible values
  - Two options
    - ✓ Automaton (Regular Grammar) Based String Analysis
    - ✓ CFG Based String Analysis

Automaton Based String Analysis

- Use an automaton M to represent the possible values of a hot spot
- The set of strings that the automaton M accepts is a super set of the possible values of a hot spot
- Proposed by Christensen et al. from University of Aarhus, Denmark in 2003

Automaton Based String Analysis

- Steps
  - Extract String Flow Graph from the source code of the need-to-analyze program
  - Transform the String Flow Graph to a Context Free Grammar G with string operations
  - Calculate the automaton approximation Linear Grammar of G
  - Use automaton transformations to represent string operations, and construct automaton M for the linear grammar

Running Example

```
public class Tricky{
    static String bar (int k, String op) {
        if (k==0) return "";
        return op+bar(k-1,op)+“]”;
    }
    static String foo (int n) {
        String b = “”;
        for (int i=0; i<n; i++) b = b + “(”;
        String s = bar(n-1,"*");
        return b + s.replace(']',')');
    }
    public static void main (String args[]) {
        String hot = foo(Integer.parseInt(args[0]));
    }
}
```

Hot Spot

Output:

n n-1 n-1

((…(**…*))…)

Extracting String Flow Graph

- Transform the source code to SSA form

  Static Single Assignment form of a program make sure that each variable is assigned once in the code

  Example:

```
x="ab";
x="cd" + x;
```

→

```
x="ab";
x1="cd" + x;
```

```
b = ""
for(i=1:n)
b = b + "(";
```

→

```
b="";
b1=φ(b, b2);
b2= b1 + "(";
```

Extracting String Flow Graph

- Extracting String Flow Graph *graph* from SSA Form F

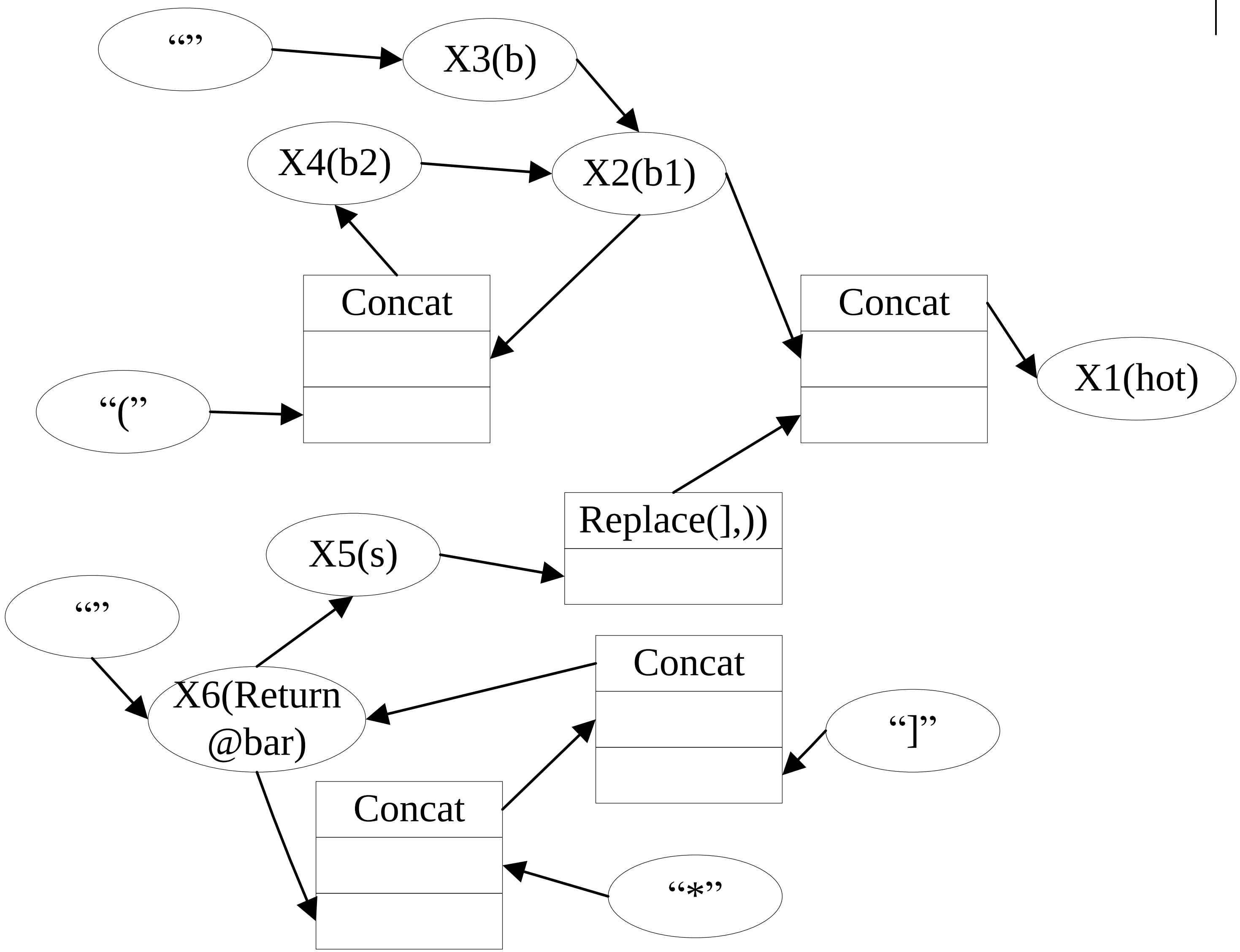
Rules:

A string variable in $F \rightarrow$ A node in *graph*

A string assignment in $F \rightarrow$ An edge in *graph*

A string operation in $F \rightarrow$ An operation in *graph*

String Flow Graph of the Running Example



Transform String Flow Graph to Context Free Grammar with operations

- Rules:

A node in *graph* → A Non-Terminal in Grammar *G*

An edge in *graph* → A production in Grammar *G*

A concat operation in *graph* → A concatenation at the right hand side of a production

Other operations in *graph* → An operation at the right hand side of a production

The node for hot spot in *graph* → The start Non-Terminal of Grammar *G*

Context Free Grammar with operations of the running example

- Non-Terminal set: {X1, X2, X3, X4, X5, X6}
- Terminal set: {*, (,],)}
- Start Non-Terminal: X1
- Productions:

X1→X2X5.replace(],))   X2 →X3 | X4

X3 →   X4 →X2(X5 →X6

X6 → | *X6]

Normalize the grammar

X1→X2X5.replace(],))

X2 →X3 | X4

X3 →

X4 →X2(

X5 →X6

X6 → | *X6]

X1→X2X6

X2 → X11|X2X7

X7 → (

X6 → X5.replace(],))

X5 → X11|X8X10

X8 →X9X5

X9 →*

X10 →]

X11 →

Automaton approximation of the grammar

- Analyze cycles in productions

X1→X2X6

X2 → X11|X2X7

X7 → (

X6 → X5.replace(],))

X5 → X11|X8X10

X8 →X9X5

X9 →*

X10 →]

X11 →

⇨ Right generating, can be exactly represented by an automaton

⇨ Both left and right generating
Called non-regular component
Cannot be exactly represented by an automaton

Removing non-regular components

- Mohri - Nederhof Algorithm

Rules: for each non-terminal A in non-regular component M

Do:

A -> X	=>	A ->X A'
A->B	=>	A->B, B'->A'
A->X Y	=>	A->R A', R->X Y
A->X B	=>	A->X B, B'->A'
A->B X	=>	A->B, B'->X A'
A->B C	=>	A->B, B'->C, C'->A'
A->reg	=>	A -> R A', R->reg

B and C represents non-terminals in M
X and Y represents non-terminals out of M
R is a newly added non-terminal

Regular approximation of the running example

- Non-regular component:{X5, X8}

X1→X2X6   X2 → X11|X2X7   X7 → (

X6 → X5.replace(],))

X5 → X11X5'

X5 → X8

X8'→ X10X5

X8 →X9X5

X5'→X8'

X9 →*

X10 →]

X11 →

Left generating Now!

Dealing with string operations

- Build an automaton transformation for each string operation
- For example: replace(],)) can be represented by replace all the transition labels ‘]’ in the input automaton to ‘)’
- Transformations can be automatically built according to the parameters of the operation

Construct the automaton

- Building the automaton using the Topological sorting algorithm
  - First of all, build automatons for the non-terminals that deduce only terminals. If a non-terminal has an automaton built, we call it a free non-terminal
  - Then, build automatons for the non-terminals that deduce only free non-terminals, and repeat this step
  - If a non-terminal is involved in a left-generating or right-generating component, use the classical algorithm to convert the whole component to an automaton
  - If a non-terminal is an input of a string operation, use the transformation of the operation to calculate the output

Problems

- String operations in a cycle
  - How to deal with the case below?

X5 → X5. replace(],))

- Current technique cannot handle it, use the closure of the character set of X5 as the approximation

X5 → {*,)}*

CFG Based String Analysis

- Context Free Grammar is more expressive than Automatons
- So it is more precise to use CFG to estimate the possible values of a hot spot
- Proposed by Minamide from University of Tsukuba, Japan, 2005

Similarity & Difference

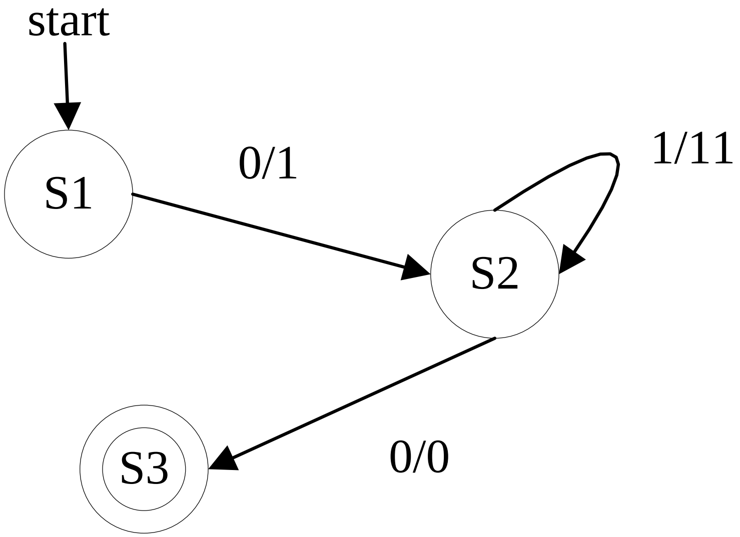
- Similarity
  - Transform the source code to SSA form
  - Extract String Flow Graph from the SSA form
  - Transform the String Flow Graph to a CFG with operations
- Difference
  - Do not calculate the regular approximation
  - Use FST (Finite State Transducer) instead of automaton transformations to represent string operations

CFG Based String Analysis

- Steps
  - Generate the CFG with operations
  - Resolve the string operations in the CFG using the CFG-FST intersection algorithm

Finite State Transitor

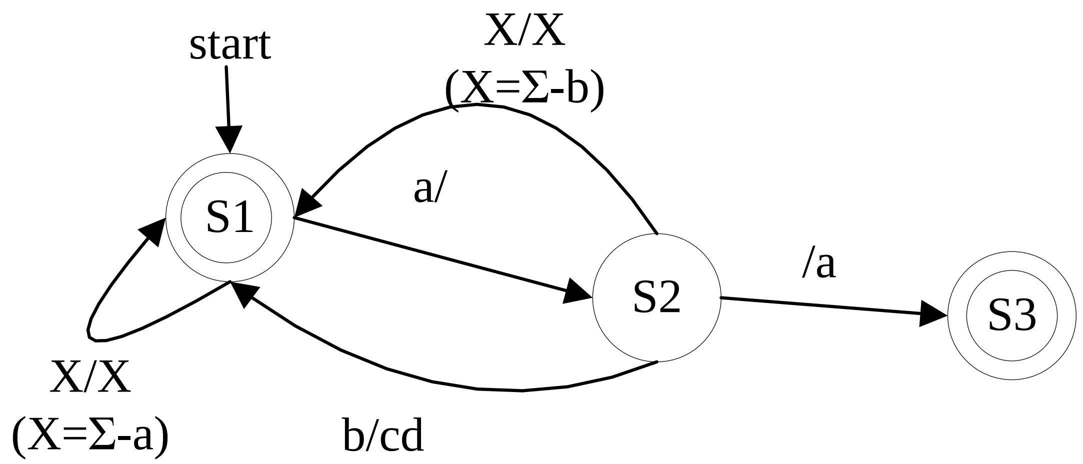
- Finite State Transducer (FST) is a Finite State Automaton with output
- For each Transition, an FST not only accept a character, but also output one or more characters
- An example:



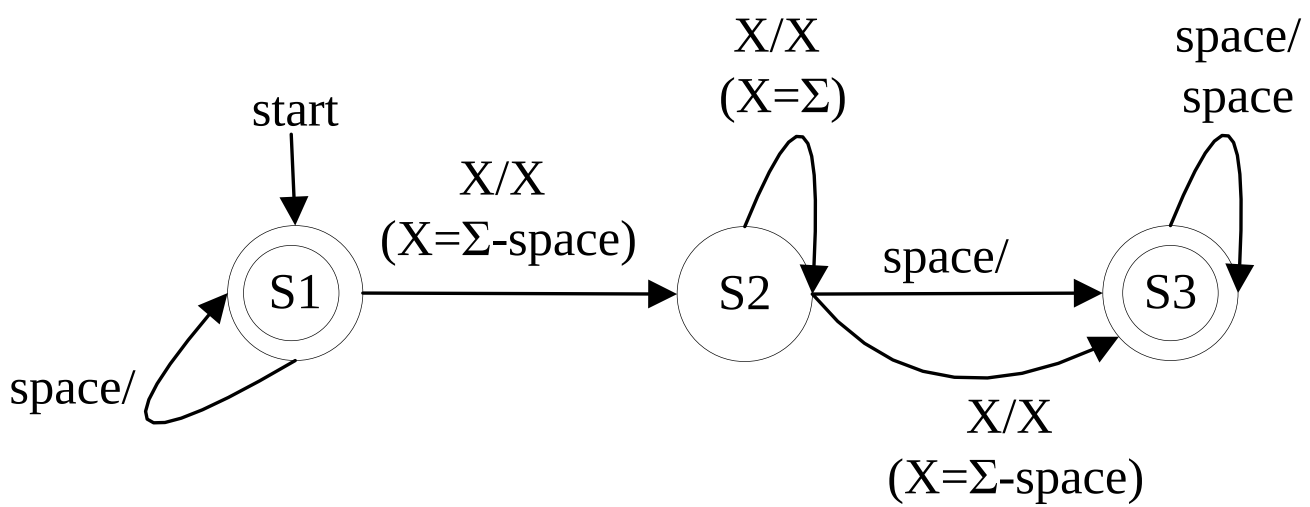
FST for string operations

- Use FSTs to simulate string operations
  - Replace

Replace("ab", "cd")



- Trim



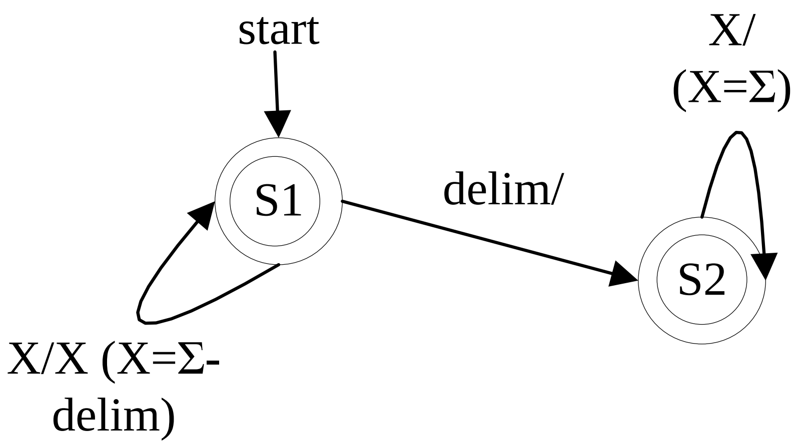
FST for string operations

- Tokenize (explode)

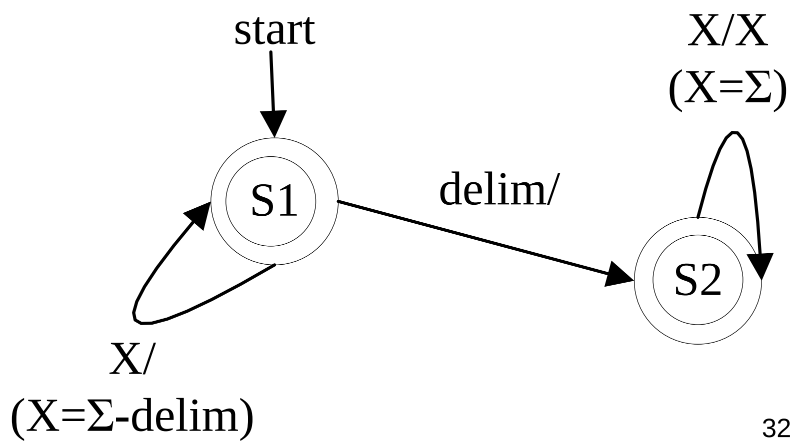
Transform one string operation to two operations

String str = tokens.nextToken()	⇨	String str1 = str.getToken() String str2 = str.removeToken()

FST for getToken



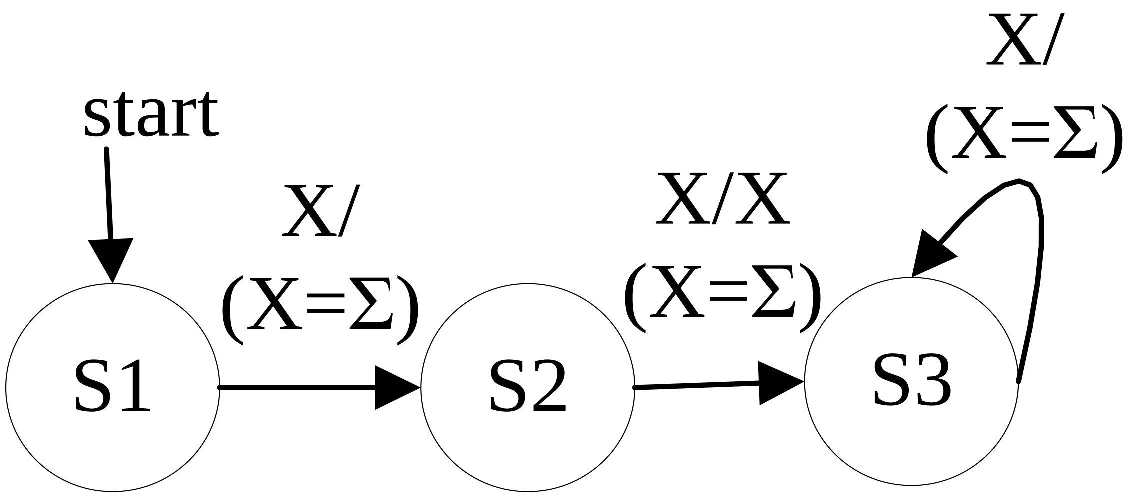
FST for removeToken



FST for string operations

- Substring

  substring(1,2)



CFG-FST intersection

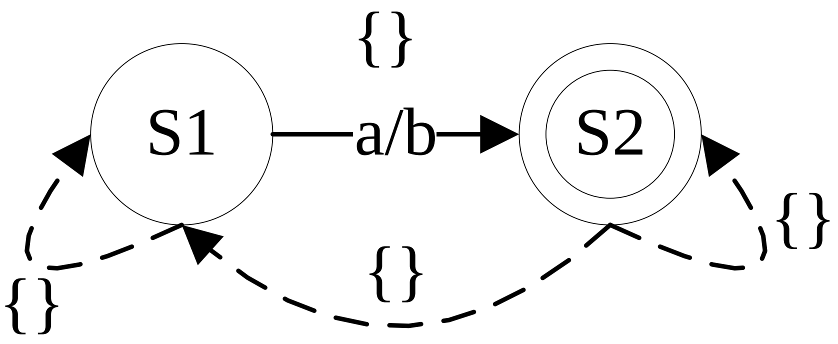
- Given a CFG G, and a FST T, try to calculate a CFG G' , satisfying that:

$$x \in G \Leftrightarrow T(x) \in G'$$

, in which x is any string, and T (x) is the output of T with x as input

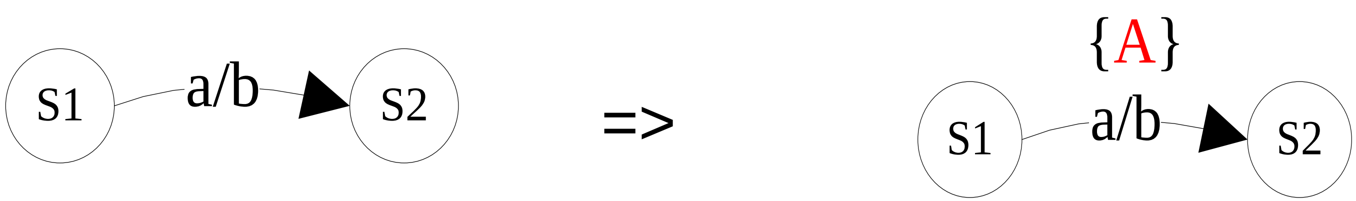
CFG-FST Intersection Algorithm

- Transform the CFL to Chomsky Normal Form (the right hand sides of all productions contain only two non-terminals) e.g., S->ABC => S->DC, D->AB
- For each pair of states in the FST, add an empty generating non-terminal set



CFG-FST Intersection Algorithm

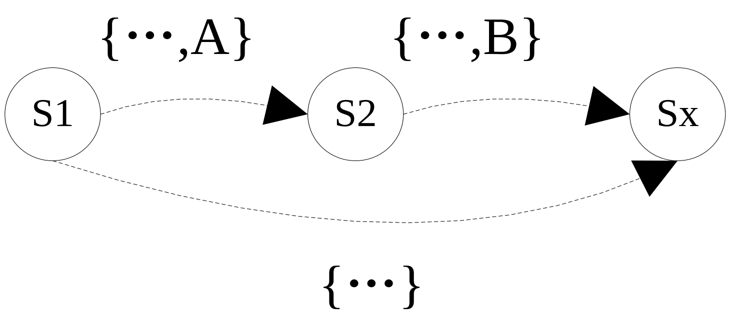
- Initialize the generating non-terminal set of all pairs of states.
- Rule: If transition (s_1,s_2) in FST accept character *t* and A -> *t* in CFG, add A to the generating non-terminal set of (s_1,s_2)



A->a

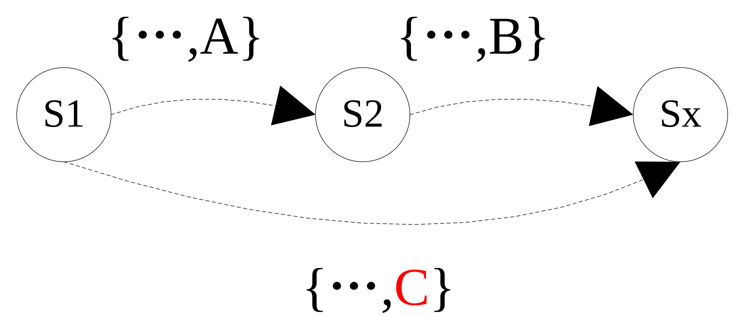
Solution of CFL-Reachability Problem, cont.

- For each non-terminal A on each pair of states $<s_1,s_2>$, if $B \in$ generating-set(s_2,s_x) $\wedge$ C -> AB $\in$ Productions, add C to generating-set(s_1,s_x)



C->AB

=>

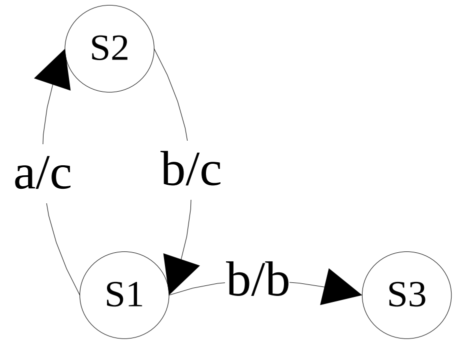


Solution of CFL-Reachability Problem, cont.

- For each non-terminal A on each pair of states $<s_1,s_2>$, if $B \in$ generating-set$(s_x,s_1) \wedge$ $C \to BA \in$ Productions, add C to generating-set(s_x,s_2)
- Iteratively execute last two steps until no more non-terminals are added to the generating sets
- Each time add a non-terminal to a generating set, output the production used
- The output productions are the intersection of FST and CFG

An Example

The FST:



The CFG Grammar:

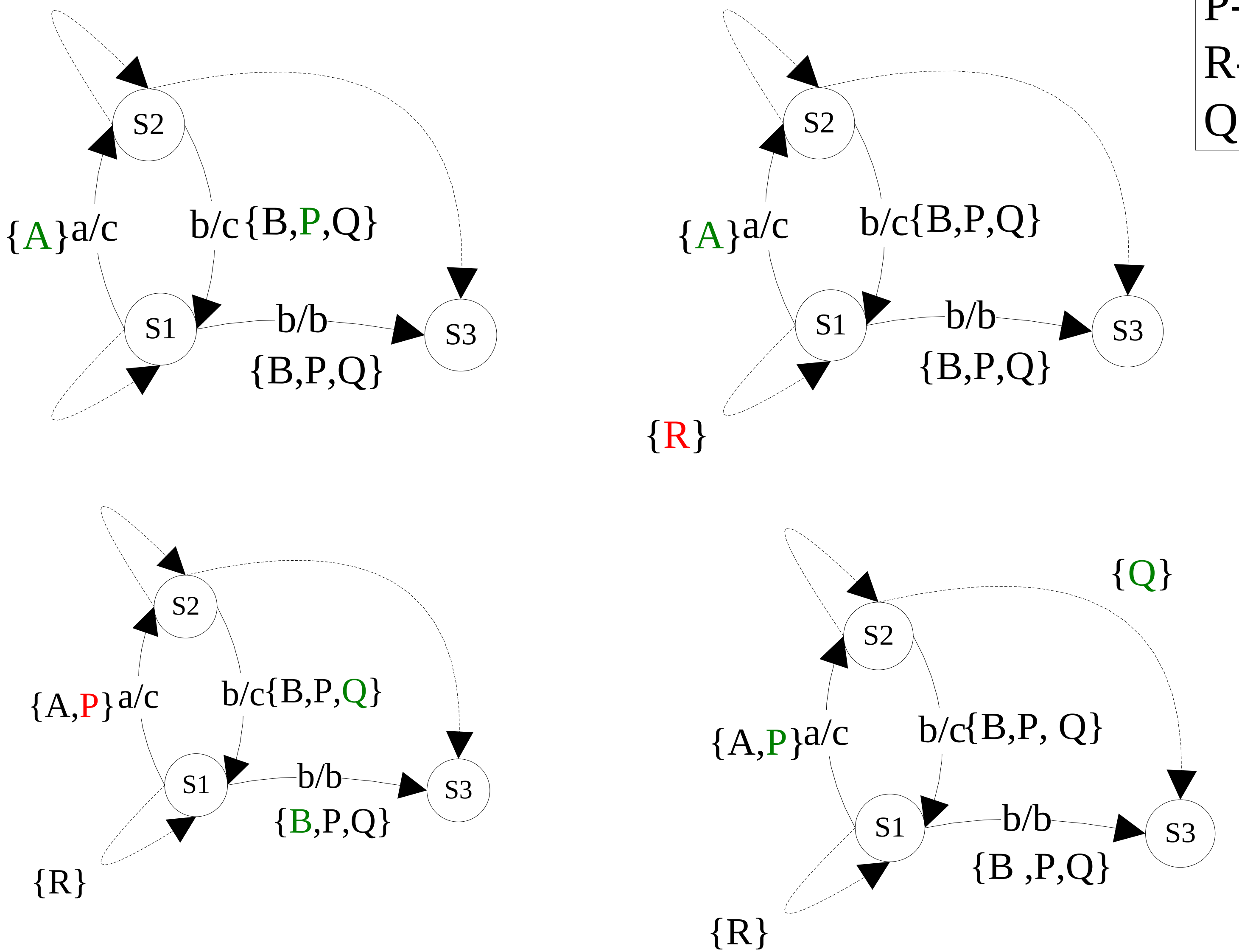
S->PQ
P->aPa | b
Q->Qb | b

The Normalized Grammar:

S->PQ
A->a
B->b
P->RA | b
R->AP
Q->QB | b

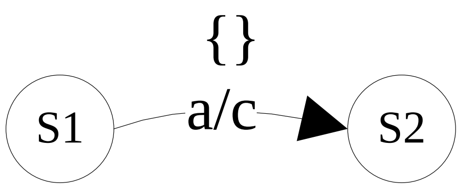
An Example, cont.

S->PQ
A->a
B->b
P->RA | b
R->AP
Q->QB | b

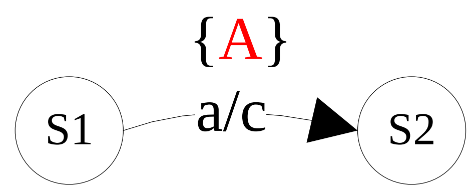


Output productions used

- When initialize the generating sets, output the production with output terminal instead of the accepted terminal



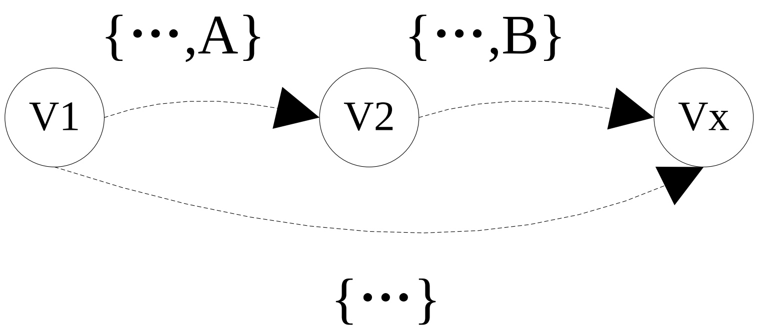
A ->a



A_{12} ->c

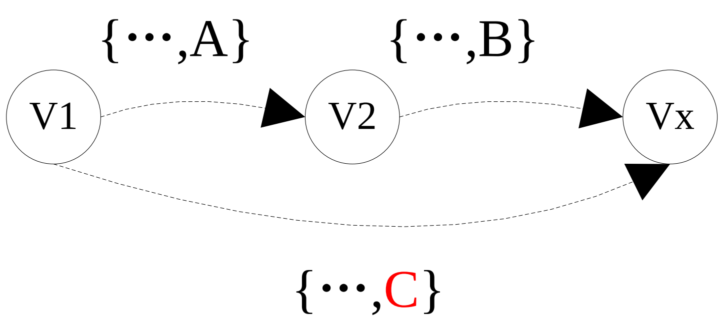
Output productions used

- For the non-terminals added later, use the rule below:



C->AB

=>



C_{1x} ->A_{12} B_{2x}

Resolve string operations in a CFG with operations

- Resolve the string operations using the topological sorting algorithm
  - If the input non-terminals of a string operation *Op* deduce pure CFG, resolve *Op*
  - Repeat the above step until there are no string operations in the CFG
  - Example:

X1->X2X3
X2->X4.replace(*,)) … op1
X4->X5X6
X6->X7.replace([,]) … op2
X7->[X7]+

input of op1:
X4->X5X6
X6->X7.replace([,])
X7->[X7]+

input of op2:
X7->[X7]+

Resolve op2 first,
Then op1

Problems

- String operations in a deduction cycle
  - How to deal with the case below?

X7 → X7. replace([,])

- Current technique cannot handle it, use the closure of the character set of X7 as the approximation

X7 → {], +}*

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...

String Taint Analysis

- Purpose

  The basic string analysis estimates the possible values of a hot spot, but it can not determine the data source of the hot spot

  String taint analysis tries to determine the data source of a given hot spot

  The original purpose of string taint analysis is to determine whether the value of a hot spot comes from user input

Running Example

```
public class Tricky{
    static String bar (int k, String op) {
        if (k==0) return "";
        return op+bar(k-1,op)+”]”;
    }
    static String foo (int n) {
        String b = “”;
        for (int i=0; i<n; i++) b = b + “(”;
        String s = bar(n-1,readChar());
        return b + s.replace(']',')');
    }
    public static void main (String args[]) {
        String hot = foo(Integer.parseInt(args[0]));
    }
}
```

Hot Spot

Output:

n n-1 n-1

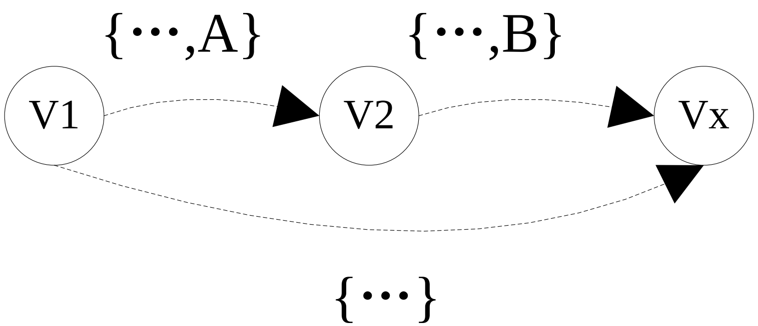
((…(**…*))…)

* is the input char

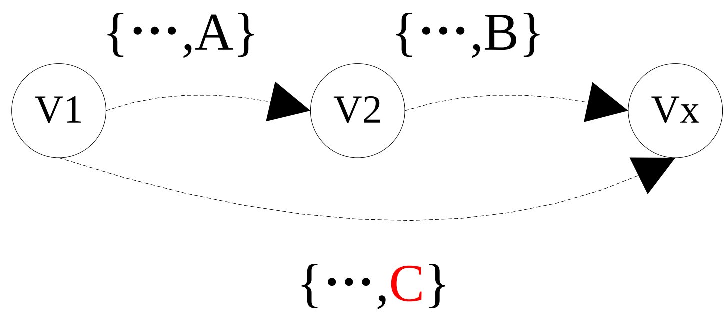
Basic Steps

- Extract a CFG with operations from the source code
- Add a Boolean taint for each non-terminal and terminal in the CFG
- For each terminal corresponding to a user input function (e.g., readInput()), set the its taint to true
- For each production, propagate the taint value from the right hand side to the non-terminal at the left hand side

Propagating Taints Through FST

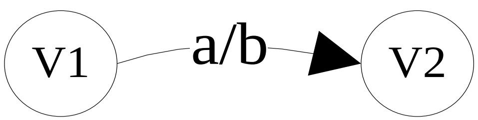


=>

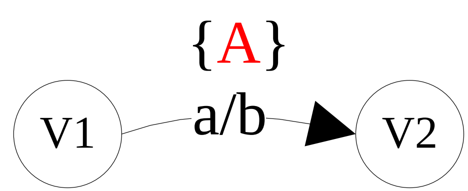


$C(t) \rightarrow AB$

$C_{1x}(t) \rightarrow A_{12} B_{2x}$



=>



$A(t) \rightarrow a$

$A_{12}(t) \rightarrow b$

Generalized String Taint Analysis

- Traditional string taint analysis handles only Boolean values, so it can only differentiate two data sources of a hot spot
- Generalized String Taint Analysis
  - Use a set instead of a Boolean value to represent a taint
  - Allow more complex operations among taints of the non-terminals/terminals of a production
  - Example: $A(t1) \rightarrow B(t2)C(t3) \Rightarrow t1 = t2 \cup t3$

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...

String Order Analysis

- Limitations of basic string analysis and string taint analysis

  With basic string analysis and string taint analysis, we are able to know the possible values and data sources of a hot spot, but we do not know the order of the data sources appearing in the value of a hot spot

- String order analysis tries to answer questions like "Is constant string a always after constant string b when they co-appear in hot spot t?"

String Order Analysis

- Example

```
$a = ‘abc’;
$t = ‘f<br name=’;
echo $a.$t.’de’.’>’;
```

- We want to decide whether “abc” is inside a HTML tag (i.e., whether “abc” is after “<” and before “>”)

Flag Propagation Algorithm: Basic Idea

- Given a CFG, identify which terminals / terminal parts are inside tags (i.e., between ‘<’ and ‘>’)

- Basic Solution:

1. Initialize known places (i.e., the terminals containing ‘<’ or ‘>’),

   e.g., T➔(O)’f**<**br name=‘(I)   O: outside, I: inside

2. Iterate propagating position information (I/O flags) to other places in the CFG (via a list of rules)
3. End iterations if none of the flags in the CFG changes

abcf<br name=de>

Flag Propagation Algorithm

- Add a left flag and a right flag to each variable in the CFG. A flag may be of one of the four values:
  - ✓ O: Indicate that the place where the flag stays is **outside a tag**
  - ✓ I: Indicate that the place where the flag stays is **inside a tag**
  - ✓ U: Indicate that the place where the flag stays is **unknown**
  - ✓ C: Indicate that the place where the flag stays may be **both inside/outside a tag** (e.g. $c=‘abc’; echo $c.’<tag name=‘.$c’>’;)
- Initialize the flags of terminals
  - ✓ Terminals with ‘>’ or ‘<’: Initialize with **“I”** or **“O”** accordingly
  - ✓ Others: initialize with **“U”**

Flag Propagation Algorithm

- Propagate flags in the CFG using the flag operation and four propagating rules iteratively

- The Flag Operation (+)

When two flags meet, we use the flag operation to calculate the propagation result of the two flags

U+U = U O+U = O I+U = I

O+O = O I+I = I I+O = C

C+* = C

Flag Propagation Algorithm

- Four Propagation Rules

✓ Neighboring Rule (for neighboring variables)
S->A(R)(L)B
e.g.: S->A(O)(U)B => S->A(O)(O)B
S->A(U)(O)B => S->A(O)(O)B

✓ Transitive Rule (for terminals without ‘<’ and ‘>’)
S->(L)’abc’(R)
e.g.: S->(O)’abc’(U) => S->(O)’abc’(O)

✓ Left Deducing Rule
(L)S->(L)AB
e.g.: (U)S->(O)AB => (O)S->(O)AB

✓ Right Deducing Rule
S(R)->AB(R)

Example CFG

- A➔’abc’
- T➔’f<br name=‘
- D➔’de’
- E➔’>’
- S➔ATDE

abcf<br name=de>

Initialization

- (U)A(U)→(U)'abc' (U)
- (U)T(U)→(**O**)'f<br name='(**I**)
- (U)D(U)→(U)'de' (U)
- (U)E(U)→(**I**)'>' (**O**)
- (U)S(U)→(U)A(U) (U)T(U) (U)D(U) (U)E(U)

U: unknown

abcf<br name=de>

Propagation

- (U)A(U O)→(U O)'abc' (U O)
- (U O)T(U I)→(O)'f<br name='(I)
- (U I)D(U I)→(U)'de' (U)
- (U I)E(U O)→(I)'>' (O)
- (U)S(U O)→(U)A(U O) (U O)T(U I) (U I)D(U I) (U I)E(U O)

Left Deducing Rule

(L)S→(L)AB

Right Deducing Rule

S(R)→AB(R)

Transitive Rule (for terminals without '<' and '>')

S→ (L)'abc'(R)

Neighboring Rule

S→A(R)(L)B

abcf<br name=de>

Final CFG with Differentiated Terminals

- (O)A(O)→(O)'abc' (O)
- (O)T(I)→(O)'f<br name=' (I)
- (I)D(I)→(I)'de' (I)
- (I)E(O)→(I)'>' (O)
- (O)S(O)→(O)A(O) (O)T(I) (I)D(I) (I)E(O)

abcf<br name=de>

Conflict Cases

- Code

$a = ‘abc’
echo $a.’<’.$a.’>’

abc<abc>

- CFG

()A(O)→‘abc’
(O)B(I)→(O)’<’(I)
(I)C(O)→(I)’>’(O)
S(O)→A(?)(O)B(I)
A(?)(I)C(O)

- Final Result

A→(**C**)‘abc’(**C**)
B→(O)’<’(I)
C→(I)’>’ (O)
S→(**C**)A(**C**) (**C**)B(**C**)
(**C**)A(**C**) (**C**)C(**C**)

Complication: **abc** is used **both inside and outside** tags

Solution: C Flag for Conflict: O + I = C; C + O|I|U|C = C except for the flags of initialized known places

Example Code

PHP Code:

```
$s = "";
for($i=0;$i<$n;$i++){
$a = "Name:";
$b = "StudentName“.$i.”\””;
$b = “ value=";
$c = $attr."\"default";
$p = $a."<input name=\"“
    .$b.$c;
$p = $p."\">";
$s = $s."\n".$p;
$i++;}
echo $table;
```

HTML Texts:

```
Name:<input
name="StudentName$i"
value="default">
…
```

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...

Why database applications?

- Database software projects depends on SQL queries to manipulate the database
- SQL queries are usually dynamically generated to make the program more flexible
- Dynamically generated SQL queries, an example:

```
Connection con = DriverManager.getConnection ("students.db");
String q = "SELECT * FROM address";
if (id!=0) q = q + " WHERE studentid=" + id;
ResultSet rs = con.createStatement().executeQuery(q);
```

Recent important applications

- Verify the correctness of dynamically generated SQL queries
- Detect SQL injection vulnerability
- Determine the impact of database schema changes

Verify the correctness of dynamically generated SQL queries

- Proposed by Christensen et al. in 2003
- Purpose:

  Verify whether all the possible values of the dynamically generated SQL queries are legal according to the SQL syntax

An example

- Legal dynamically generated SQL queries

  int id = readInt();
  String query = "SELECT * FROM address";
  if (id!=0) query = query + " WHERE studentid=" + id;

- Possibly illegal dynamically generated SQL queries

  int id = readInt();
  String query = "SELECT * FROM address";
  if (id!=0) query = query + " WHERE studentid=" + id;
  else query = query + "WHERE studentid=" + id;

missing space!!

Approach

- Identify all the query execution statements in the source code and mark the variables representing a query as hot spots
- Use basic string analysis to estimate the possible values of each hot spot t, represented as an automaton $M(t)$
- Approximate the SQL syntax as a finite state automaton MS with 631 states, and calculate its complement MS'
- For each t, check whether $M(t) \cap MS' = \Phi$

Evaluation

Evaluation on 9 programs

Example	Lines	Exps	Hotspots	Total	Memory	Errors	False Errors
Decades	26	63	1	1.344	27	0	0
SelectFromPer	51	50	1	1.480	27	0	0
LoadDriver	78	154	1	0.981	28	0	0
DB2Appl	105	59	2	0.784	27	0	0
AxionExample	162	37	7	1.008	29	0	0
Sample	178	157	4	1.261	28	0	0
GuestBookServlet	344	320	4	3.167	33	1	0
DBTest	384	412	5	2.387	31	1	0
CoercionTest	591	1,133	4	5.664	42	0	0

in Mbs

time in seconds

Limitations

- Sound but incomplete (may have false positives)
- Can find only syntax errors, cannot find run-time errors (e.g., type inconsistencies)

Detect SQL injection vulnerability

- Proposed by Gary Wassermann and Zhendong Su, 2007
- Purpose

  Check whether a dynamically generated SQL query may involve in a SQL injection vulnerability

An Example of SQL injection

- Consider the query below:

  query = "SELECT * FROM accounts WHERE name='"+readName()+"' AND password='"+readPassword();

- If input ‘ OR 'a'='a‘, we get:

  SELECT * FROM accounts WHERE

  name='badguy' AND password=‘ ' OR 'a'='a'

Approach

- Build regular policy for each value field in the SQL statement
- For each query and its corresponding CFL, compute the intersection of the CFL and the regular policy
- If the intersection is not empty and contains substrings from un-trusted source (user input), a SQL injection is found

Evaluation

Evaluation on 5 real world projects

Name (version)	Files	Lines	Time (h:m:s)		Errors		
			String Analysis	SQLCIV Check	direct		indirect
					Real	False	
e107 (0.7.5)	741	132,850	3:39:26.23	35:36.12	1	0	4
EVE Activity Tracker (1.0)	8	905	0.40	0.06	4	0	1
Tiger PHP News System (1.0 beta 39)	16	7,961	3:14:06.95	5.39	0	3	2
Utopia News Pro (1.3.0)	25	5,611	25:00.08	2:08.69	14	2	12
Warp Content MS (1.2.1)	42	23,003	21.10	0.08	0	0	0
Totals					19	5	17

Indirect errors: a user-input string goes to the dangerous part of a SQL query through the database

Example:

String insert = "insert into table values ("+readString()+"," readInt()+")";
executeQuery (insert);
ResultSet rs = executeQuery ("select * from table");
String query = "select * from table where id="+rs.getString(0);

Limitations

- Sound but incomplete, may has false positives
- Can not provide test cases for the developer to understand the vulnerability

Determine the impact of database schema changes

- Proposed by Andy Maule et al., in 2008
- Purpose:

  Determine which statements in the source code may require fix after a change on the database schema (e.g., a change on the name of a table/column, adding/removing table/columns)

Impact of schema change: An example

schema

Experiments			
ExperimentId	Date	Name	Description
VARCHAR(30)	DATE	VARCHAR(30)	TEXT
req.	req.	not req.	not req.

Remove this column

```
queryResult = QueryRunner.Run(
"SELECT Experiments.Name,Experiments.ExperimentId"+
" FROM Experiments"+
" WHERE Experiments.Date={@ExpDate}", dbParams);
```

Approach

- Mark all the SQL queries that goes to a SQL query execution statement as hot spots
- For each hot spot, estimate its possible values using basic string analysis
- For the name of each table column in the schema, build an automaton like "$\Sigma^{*}\text{name}\Sigma^{*}$", which represents all strings containing the name
- Intersect the automaton $M(t)$ of each hot spot t and of each table column $M(c)$
- $M(t) \cap M(c) \neq \Phi \Rightarrow$ *a change on c affects t*

Evaluation

Do evaluation on the irPublish Content Management System, which consists of 127KLOC C# code

The database include 101 tables and 615 columns

Schema Changes:

ChangeSc1	Added a column to a table
ChangeSc2	Added 3 columns to a table
ChangeSc3	Altered data type of a column
ChangeSc4	Added a new constraint to column
ChangeSp1	Added 3 new parameters to a stored proc.
ChangeSp2	Added new return columns to a stored proc.
ChangeSp3	Added new return columns to a stored proc.
ChangeSp4	Added a new parameter in a stored proc.

Predicted Changes vs. Real Changes:

Change	Predicted	True positives	False positives
ChangeSc1	5 warns	2	3
ChangeSc2	4 warns	0	4
ChangeSc3	4 warns	0	4
ChangeSc4	4 warns	0	4
ChangeSp1	1 err	1	0
ChangeSp2	1 warns	1	0
ChangeSp3	1 warn	0	1
ChangeSp4	none	0	0

Limitations

- Sound and incomplete, with low precision because whenever the changed column is involved in a statement, it raise a warning

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...

Why web applications?

- Web-based software projects use html text to present web pages
- Html texts are usually dynamically generated to make the program more flexible
- Dynamically generated html texts, an example in PHP:

```
$x = _Post[Color]
$content = _Post[content]
if ($errMsg == "")
echo ("<p><h2><font color='".$x."'>".$content.
"</font></h2><p>\n");
```

Recent Important Applications

- Verify the correctness of dynamically generated web pages
- Detect cross-site-scripting vulnerabilities

Verify the correctness of dynamically generated web pages

- Proposed by Minamide in 2005
- Purpose:

Verify whether all the possible values of the dynamically generated web page comply with the html syntax

An example

- Legal dynamically generated SQL queries

```
echo "<html>";
echo "<h1>";
if($head!="")
echo $head;
echo "</h1></html>“;
```

- Possibly illegal dynamically generated SQL queries

```
echo "<html>";
echo "<h1>";
if($head!="")
echo $head.</h1>;
echo "</html>“;
```

If $head == “”, the <h1> tag will be unclosed due to the missing </h1>

Approach

- Add a statement to concatenate all the outputs of a web page generating unit (e.g., a .php file), and set the concatenation result as the hot spot
- Use basic string analysis to estimate the possible values of the hot spot, represented as a CFG G
- Approximate the HTML syntax as a finite state automaton M by limit the recursive depth of the tags, and calculate its complement M'
- Check whether $G \cap M' = \Phi$

Evaluation

Evaluation on 6 programs

Program	#lines	#non-terminals	#productions	Time (sec)
webchess	2224	300	450	0.36
schoolmate	8085	7985	9505	39.92
faqforge	843	180	443	0.16
phpwims	726	82	226	0.13
timeclock	462	656	1233	0.15

Validation Results

Time to generate CFG

Program	Depth	Bugs	Time (sec)
webchess	9	1	123.33
schoolmate	17	14	7580.69
faqforge	10	30	45.64
phpwims	9	7	63.93
timeclock	14	11	145.61

Max Recursive Depth

Limitations

- Sound but incomplete (may have false positives)
- Can find only syntax errors, cannot find run-time errors (e.g., script refer to illegal variables)

Detect cross-site-scripting vulnerabilities

- Proposed by Gary Wassermann and Zhendong Su, 2008
- Purpose

Check whether a dynamically generated web page may involve in a cross-site-scripting vulnerability

Example

An cross-site-scripting vulnerability:

In form.php: <form action='view.php'><input id=1 name='content'></form>

In view.php:

echo "<div></td>Content: " . _POST('content')

if we input "<script>badcode</script>" to the 'content' item of form.php, bad code goes to view.php

Approach

- Build regular policy for all the HTML texts that will invoke a script interpreter
- For the CFL of the HTML text, compute the intersection of the CFL and the regular policy
- If the intersection is not empty and contains substrings from un-trusted source (user input), a XSS vulnerability is found

Evaluation

Subject	Files	Lines Per File			Total
		mean	std dev	max	lines
Claroline	1144	148	248	5,207	169,232
FishCart	218	230	196	1,182	50,047
GecBBLite	11	29	30	95	323
PhPetition	17	159	75	281	2,701
PhPoll	40	144	112	512	5,757
Warp	44	554	520	2,276	24,365
Yapig	50	170	191	946	8,500

The information of subjects

Caused by user input

Subject	Direct				Indirect
	GPC		Uninit		
	t	f	t	f	
Claroline 1.5.3	32	43	38	25	42
FishCart 3.1	2	2	30	12	2
GecBBLite 0.1	1	1	0	0	7
PhPetition 0.3.1b	0	0	7	8	7
PhPoll 0.96 beta	5	6	0	0	0
Warp CMS 1.2.1	1	1	22	19	18
Yapig 0.95b	15	13	9	1	14

Result of the detection

Caused by un-initialized variables, which can be set by a user when export global is true in PHP

Limitations

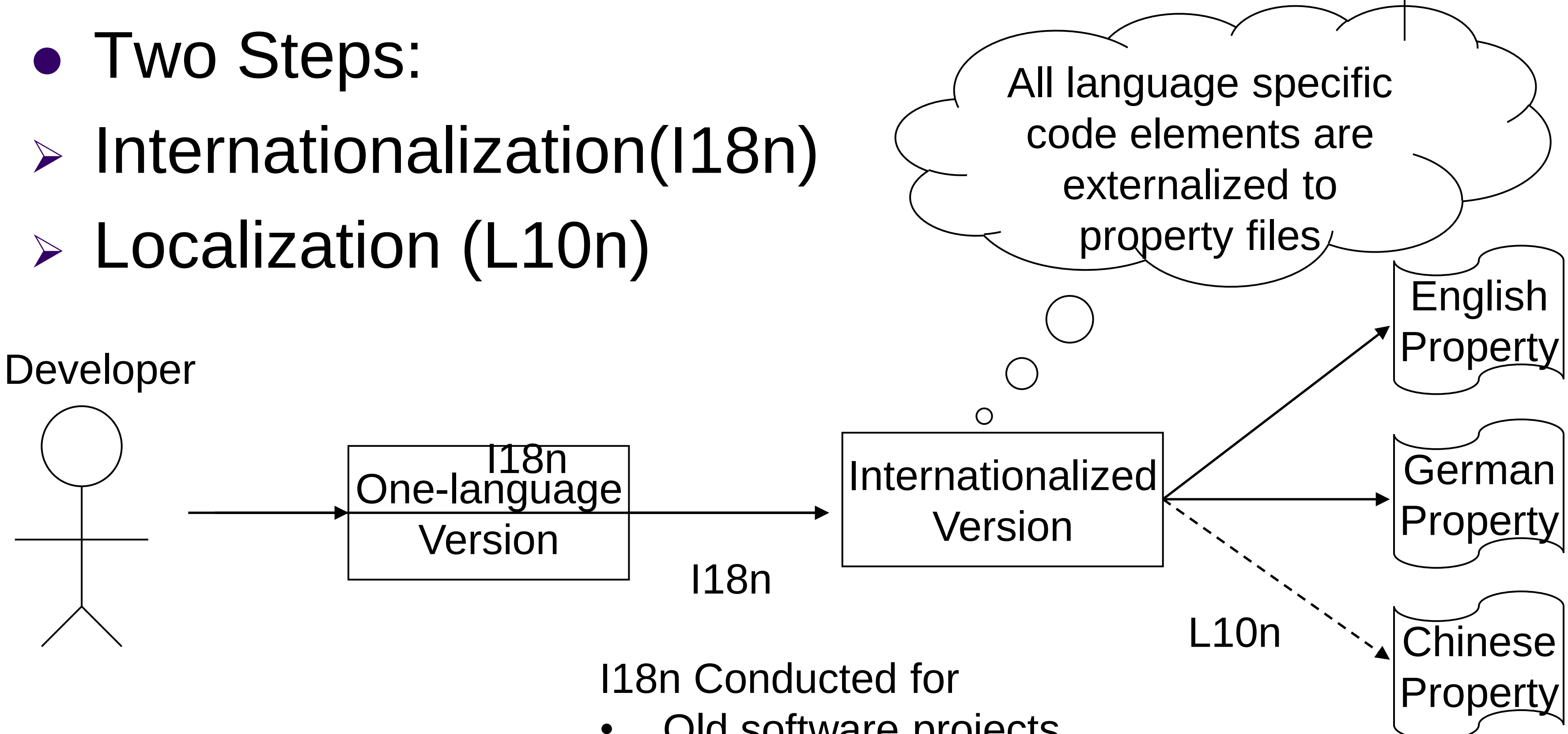
- Can not handle DOM-based cross-site-scripting vulnerabilities which read malicious code from the DOM
- Can not follow complex data flow such as web page visits and dynamic code

Outline

- Basic Concepts
- Techniques
  - Basic String Analysis
  - String Taint Analysis
  - String Order Analysis
- Applications
  - Database Applications
  - Web Applications
  - Software Internationalization
  - ...
- ...

Globalization Process

- Two Steps:
  - Internationalization(I18n)
  - Localization (L10n)



I18n Conducted for
- Old software projects
- New project with no global plan at first
- Using old components

Example of I18n and L10n

- Original Code Elements

```
JButton gManual = new JButton("Manual");
JButton gAbout = new JButton("About");
JButton gQuit = new JButton("Quit");
```

- Externalized Code Elements

```
JButton gManual = new JButton(resb.getString("swing.menu.manual"));
JButton gAbout = new JButton(resb.getString("swing.menu.about"));
JButton gQuit = new JButton(resb.getString("swing.menu.quit"));
```

- Property files

Risk.txt

```
swing.menu.options=Options
swing.menu.manual=Manual
swing.menu.help=Help
swing.menu.about=About
swing.menu.quit=Quit
```

Risk.txt | Risk_de.properties

```
swing.menu.options     =Einstellung
swing.menu.manual      =Handbuch
swing.menu.help        =Hilfe
swing.menu.about       =Über
swing.menu.quit        =Beenden
```

Language Specific Code Elements

- Constant Strings
- Date/Number Formats
- Currency/Measures
- Writing Direction
- Color/Culture related elements
- …

Constant Strings are of the largest number, and some of them are very hard to be located.

Motivation of our work

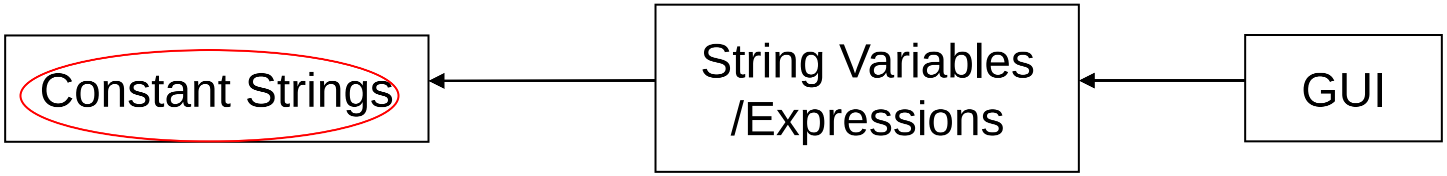
- There are a lot of constant strings
- We should not translate all of them

Application/ Version	#LOC	#Constant Strings	#Need-to-Translate Strings (Not externalized in the subsequent version)
Rtext0.8.6.9 (Core Package)	17k	1252	408(121)
Risk1.0.7.5	19k	1510	509(55)
ArtOfIllusion1.1	71k	2889	1221(816)
Megamek0.29.72	110k	10464	1734(678)

- It is sometimes hard to decide which string is need-to-translate

Basic Idea

We assume that all need-to-translate strings are those strings that are sent to the GUI



Output API Methods

- Output API Methods are methods that pass at least one of its parameters to the GUI
- Example

  java.awt.Graphics2D.drawString(java.lang.String, int, int)
  drawString 1 false 0

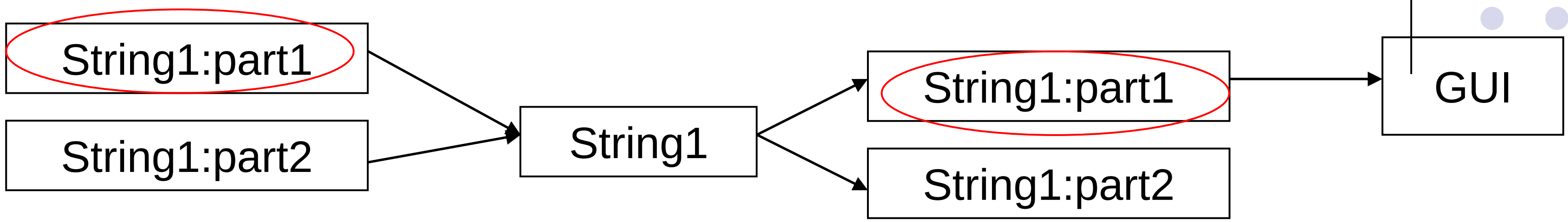
- Initial Output Strings are the arguments sent to Output API Methods

  g.drawString (**weaponMessage**, 30,20)

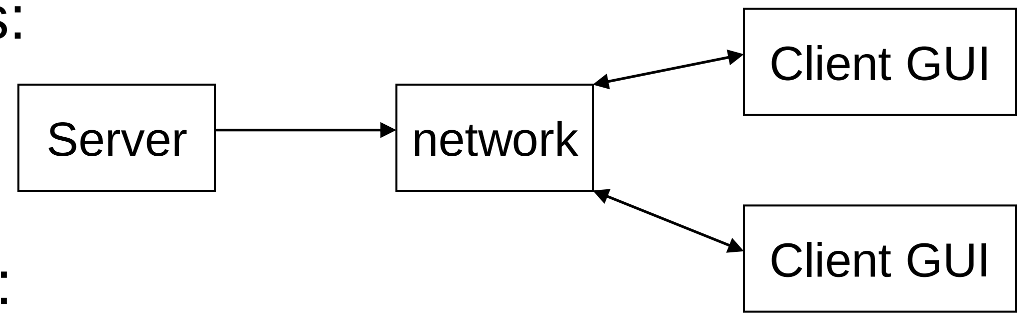
- We locate the string using Eclipse API Search Engine

Challenges

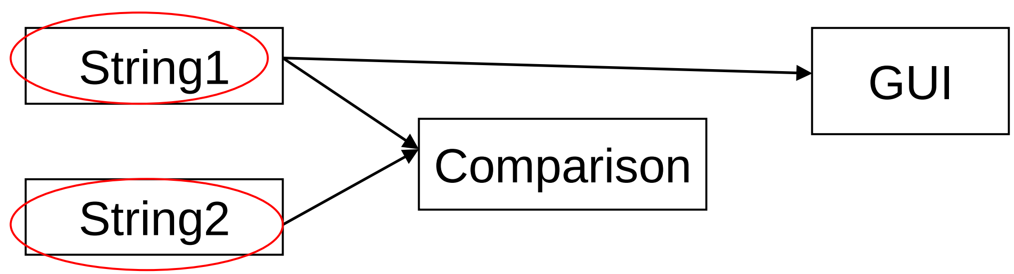
- ✓ String operations (concatenate, tokenize, substring, etc..)



- ✓ String transmissions:



- ✓ String Comparisons:



- ✓ Trivial Strings: “123”, “ ”, “Risk”, …

Experimental subjects

- RText : Simple Editor
- Risk : Board Game
- ArtOfIllusion : Graph Drawing Project
- Megamek : Big Real Time Strategy Game

Application/Version	Starting Month	#Developers	#LOC	#Files	#Constant Strings
RText 0.8.6.9	11/2003	16	17k	55	1252
Risk 1.0.7.5	05/2004	4	19k	38	1510
AOI 1.1	11/2000	2	71k	258	2889
Megamek 0.29.72	02/2002	33	110k	338	10464

Bugs found

- We found 17 not-externalized need-to-translate strings in the latest version of Megamek and reported them as report 2085049. The developers confirmed and externalized them.

Web Applications: Problems

- Web applications will not only output user-visible strings but also tags.

Code

```
$name = 'Xiaoyin Wang';
$position = 'Ph.D. Candidate';
$part = 'Software Engineering Institute';
$part_ref = 'http://www.sei.pku.edu.cn/';
$univ = 'Peking University';
$univ_ref = 'http://www.pku.edu.cn/';
echo '<DIV><FONT size=6><B>'.$name.'</B></FONT>'
echo '<P>'.$position.'<BR><A href=\"'.$part_ref.'\">'
   .$part.'</A><BR><A href=\"'.$univ_ref.'\">'.$univ.'</A>'
```

HTML

```
<DIV><FONT size=6><B>Xiaoyin Wang</B></FONT>
<P>Ph.D. Candidate<BR><A href="http://www.sei.pku.edu.cn/">Software
Engineering Institute</A><BR><A href="http://www.pku.edu.cn/">Peking
University</A>
```

Screen

Xiaoyin Wang

Ph.D. Candidate
Software Engineering Institute
Peking University

User-Visible Constant Strings in Web Applications

- Constant Strings outside Tags

```
echo "and pressed 'refresh' on your browser.
    In this case, your responses have<br/>\n";
echo "already been saved."
echo "</font></center><br /><br />";
```

(from question.php, Lime Survey 0.97)

- Constant Strings in value attribute of input tags

```
if (substr(strtolower($reply_subj), 0, 3) != "re:")
    $reply_subj = "Re: ".$reply_subj;
echo "        <INPUT TYPE=TEXT NAME=passed_subject
    SIZE=60 VALUE=\"$reply_subj\">";
```

(from compose.php, SquirrelMail 0.2.1)

Not-visible Constant Strings in Web Applications

- Constant String inside Tags

```
if ( $t == $timetohighlight) { $c = "red";} else{
    $c = "white";
}
echo "<td bgcolor=$c>";
```

(from day.php3, MRBS version 0.6)

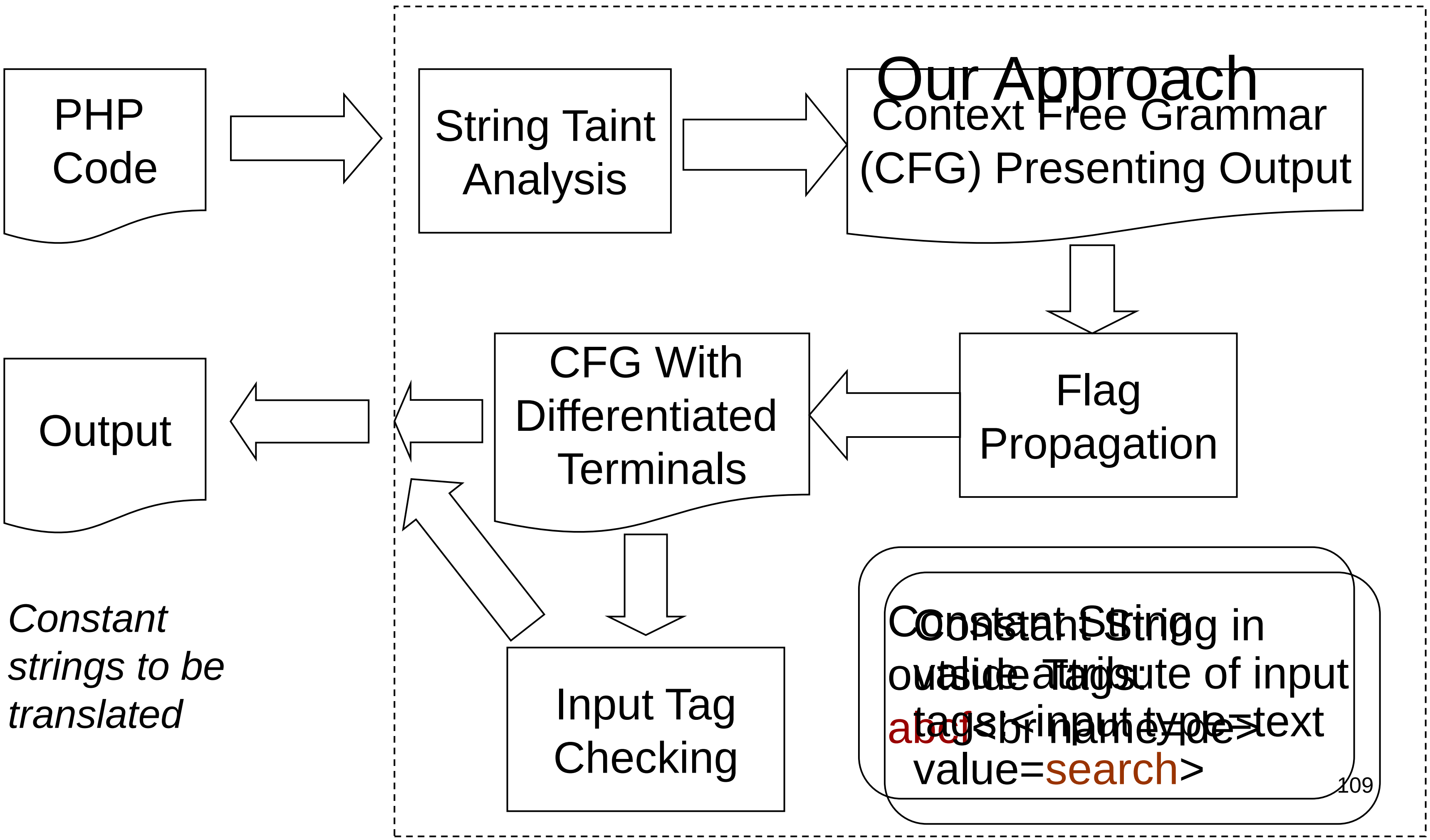
Challenges

- Differentiate constant strings inside and outside tags
- Identify constant strings that are parts of certain attribute of certain tags, such as "value" attribute of <input> tags.

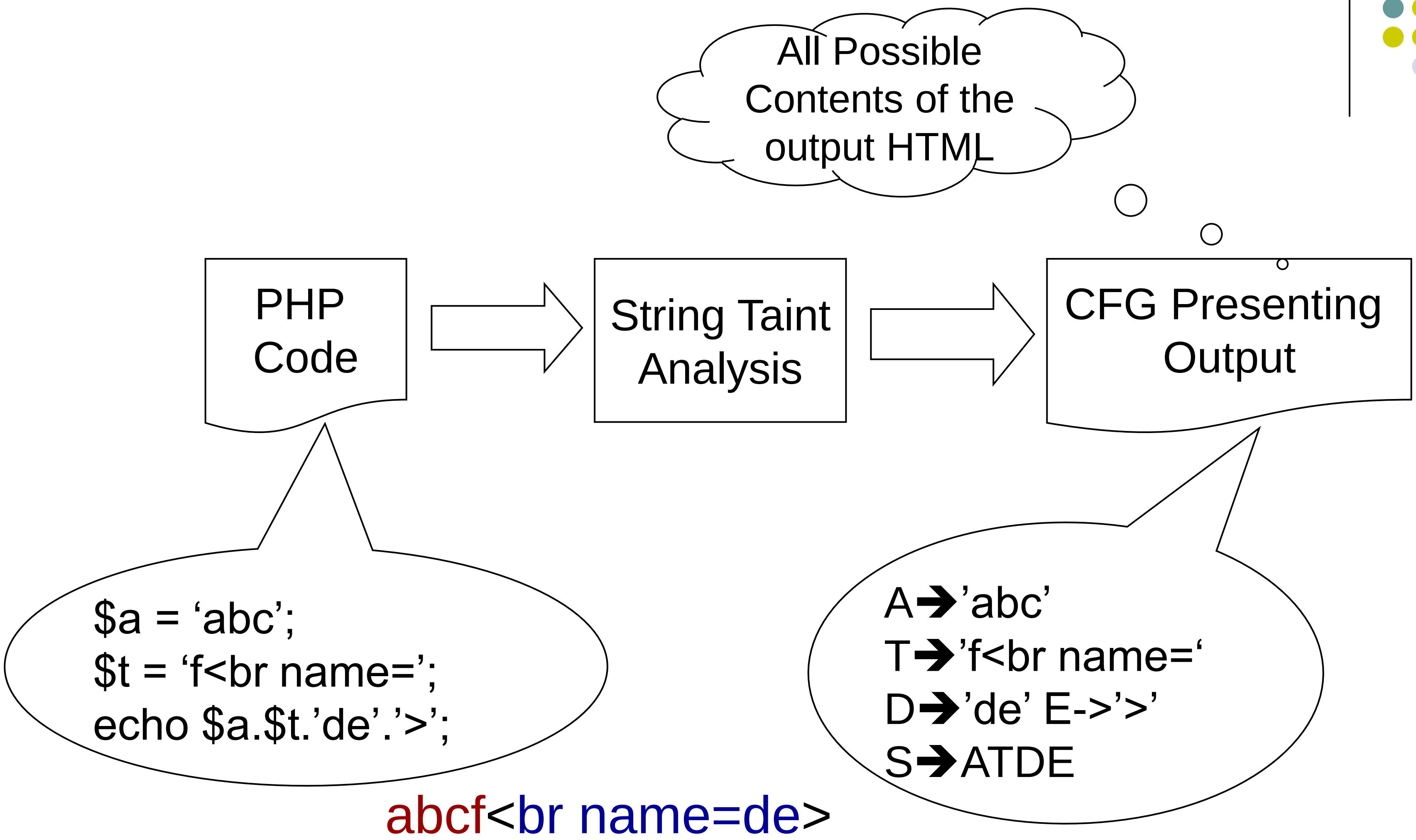
✓ Easy for static html texts, but difficult dynamic html texts

  ✓ the generated html texts by code can be **various** and **infinite**

Approach Overview

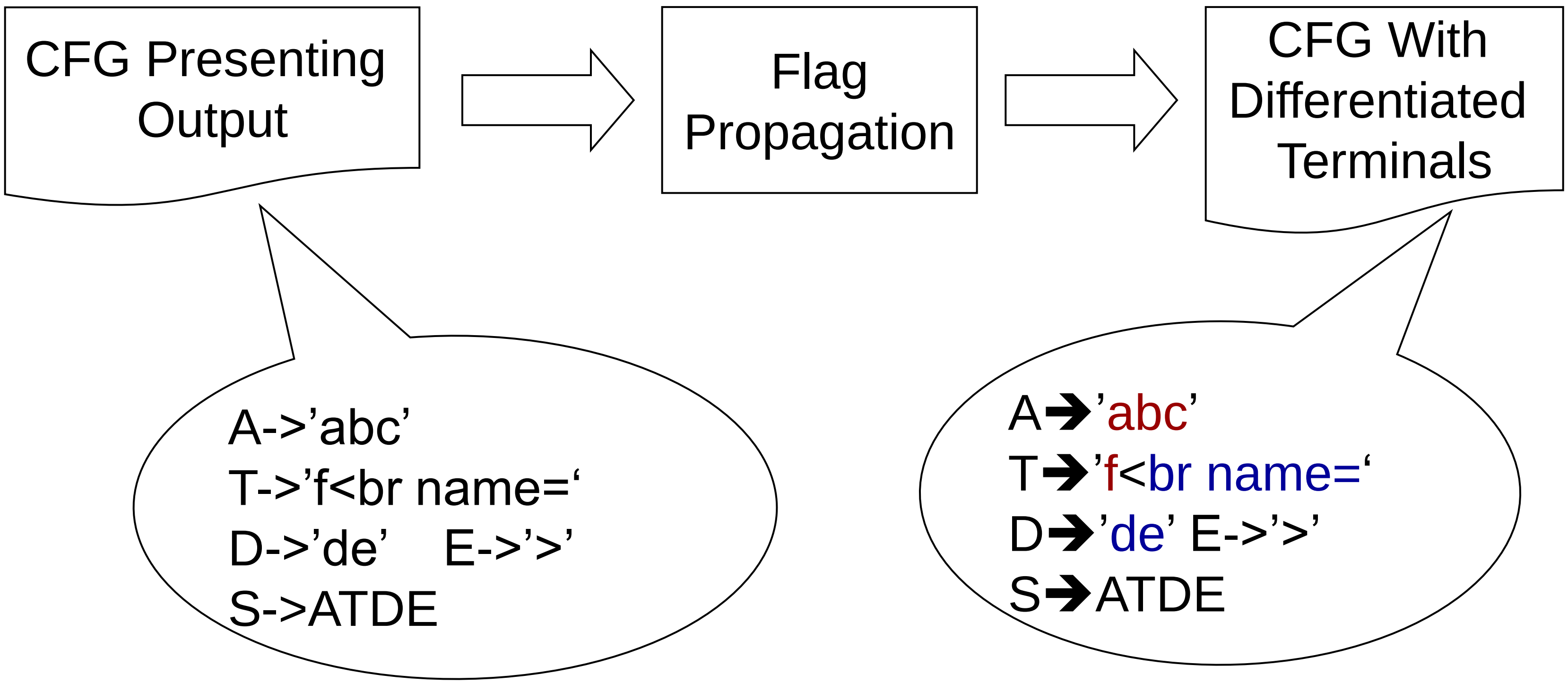


Step 1 - String Taint Analysis



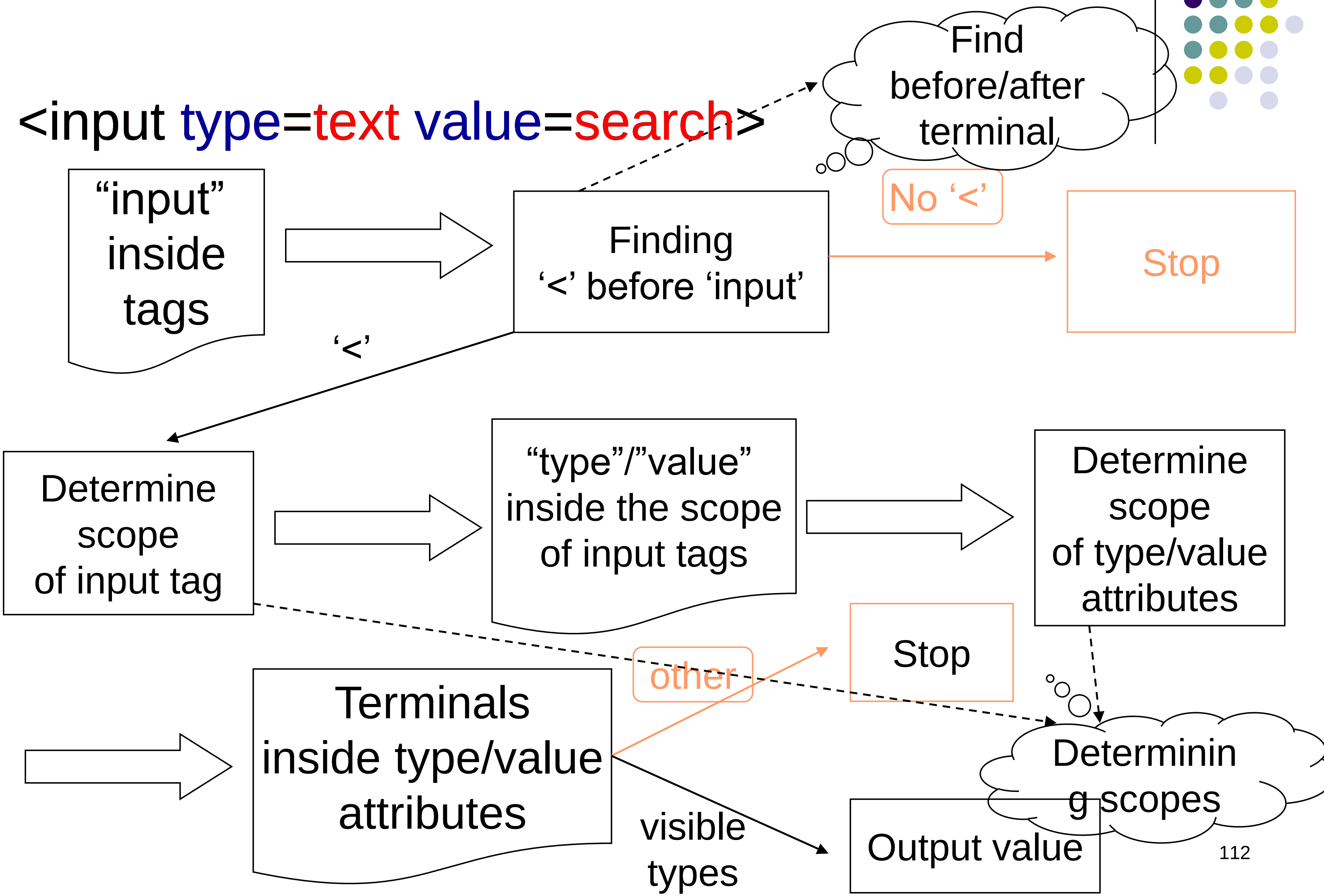
Static detection of SQL injection vulnerabilities, Wassermann and Su PLDI'07

Step 2 – Tag Range Analysis



abcf<br name=de>

Step 3- Input Tag Checking



Evaluation Subjects

- Three PHP projects
  - Lime Survey
  - Squirrel
  - Mrbs

Only a small percentage of constant strings are need-to-translate

PJ/Ver	#LOC	#Constant Strings	#Need-to-Translate
Lime Survey 0.97	11.3K	6493	290
Squirrel0.2.1	4.0K	2457	184
MRBS 0.6	1.4K	704	57

432 externalized by developers at v+1 version
62 externalized by developers at later versions
37 manually verified/confirmed by us

Evaluation Result

BS: string taint analysis

BS+O: string taint analysis + **flag propagation**

ALL: string taint analysis + flag propagation + **input tag checking**

Subject (Approach)	Need-to-Translate	Located	FN	FP
Lime (ALL)	**290**	**219**	**89(31%)**	**18(6%)**
Lime (BS+O)	290	198	110(38%)	18(6%)
Lime (BS)	290	599	89(31%)	398(137%)
Squirrel (ALL)	**184**	**192**	**0(0%)**	**8(4%)**
Squirrel (BS+O)	184	180	12(7%)	8(4%)
Squirrel (BS)	184	718	0(0%)	534(290%)
Mrbs (ALL)	**57**	**42**	**17(30%)**	**2(4%)**
Mrbs (BS+O)	57	42	17(30%)	2(4%)
Mrbs (BS)	57	140	17(30%)	100(175%)

Most need-to-translate constant strings are outside tags [illegible]

[illegible] false negatives

Found Constant Strings Externalized in Later Versions

- Our approach found 62 constant strings (5: Lime Survey, 44: Squirrel Mail, 13: MRBS)
  - not externalized at the internationalization
  - but externalized later
- Example (smtp.php of Squirrel Mail, externalized 3 years later)

```
switch ($err_num) {
    ...
    case 502:$message = "Command not implemented";
          $status = 0;
          break;
    ...
}
```

Thank you!