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.
    
    ```java
    str in String str;
    ```
  
  - In weakly typed languages (e.g., PHP), String Variables are variables that may be assigned a string value.
    
    ```php
    $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
  - ...

7
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

```java
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”, “abcccd”
Basic String Analysis

- String variable with infinite possible values

```java
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
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]));
    }
}
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 \textit{graph} from SSA Form \( F \)

Rules:
- A string variable in \( F \) \( \rightarrow \) A node in \( \textit{graph} \)
- A string assignment in \( F \) \( \rightarrow \) An edge in \( \textit{graph} \)
- A string operation in \( F \) \( \rightarrow \) An operation in \( \textit{graph} \)
String Flow Graph of the Running Example

```
""
X3(b)

X4(b2)  X2(b1)

Concat

"(

X5(s)

Replace([,),])

Concat

X6(Return @bar)

Concat

"

"*

Concat

"

"]

""

X1(hot)

Concat

""

Concat

X1(hot)
```

---

**Graph Details:**
- **"":** Represents concatenation.
- **X1(hot):** Concatenation of "hot".
- **X2(b1):** Concatenation of "b1".
- **X3(b):** Concatenation of "b".
- **X4(b2):** Concatenation of "b2".
- **X5(s):** Replacement of "[,") with "s".
- **X6(Returen @bar):** Concatenation of "Return @bar".
- **"[":** Represents concatenation.
- **"*":** Concatenation.

**Flow Overview:**
1. "" to X3(b) via Concat.
2. X4(b2) to X2(b1) via Concat.
3. "(" to Concat.
4. X5(s) to Replace([,)) via Concat.
5. X6(Returen @bar) to Concat.
6. "*" to "[" via Concat.
7. "[" to X1(hot) via Concat.
Transform String Flow Graph to Context Free Grammar with operations

- Rules:

  A node in $graph \rightarrow$ A Non-Terminal in Grammar $G$
  An edge in $graph \rightarrow$ A production in Grammar $G$
  A concat operation in $graph \rightarrow$ A concatenation at the right hand side of a production
  Other operations in $graph \rightarrow$ An operation at the right hand side of a production
  The node for hot spot in $graph \rightarrow$ 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:
  \[
  \begin{align*}
  X1 & \rightarrow X2X5.\text{replace(][,))} \\
  X2 & \rightarrow X3 \mid X4 \\
  X3 & \rightarrow X4 \rightarrow X2( \\
  X4 & \rightarrow X2( \\
  X5 & \rightarrow X6 \\
  X6 & \rightarrow | *X6]
  \end{align*}
  \]
Normalize the grammar

\[
\begin{align*}
X_1 &\rightarrow X_2 X_5.\text{replace(],))} & X_1 &\rightarrow X_2 X_6 \\
X_2 &\rightarrow X_3 \mid X_4 & X_2 &\rightarrow X_11 \mid X_2 X_7 \\
X_3 &\rightarrow & X_7 &\rightarrow ( \\
X_4 &\rightarrow X_2( & X_6 &\rightarrow X_5.\text{replace(],))} \\
X_5 &\rightarrow X_6 & X_5 &\rightarrow X_11 \mid X_8 X_10 \\
X_6 &\rightarrow \mid *X_6] & X_8 &\rightarrow X_9 X_5 \\
& & X_9 &\rightarrow * \\
& & X_{10} &\rightarrow ] \\
& & X_{11} &\rightarrow 
\end{align*}
\]
Automaton approximation of the grammar

- Analyze cycles in productions
  - $X_1 \rightarrow X_2 X_6$
  - $X_2 \rightarrow X_{11} | X_2 X_7$
  - $X_7 \rightarrow (\quad$  
  - $X_6 \rightarrow X_5.\text{replace}([],))$
  - $X_5 \rightarrow X_{11} | X_8 X_{10}$
  - $X_8 \rightarrow X_9 X_5$
  - $X_9 \rightarrow ^*$
  - $X_{10} \rightarrow ]$
  - $X_{11} \rightarrow$

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:

- $A \rightarrow X \Rightarrow A \rightarrow X A'$
- $A \rightarrow B \Rightarrow A \rightarrow B, B' \rightarrow A'$
- $A \rightarrow X Y \Rightarrow A \rightarrow R A', R \rightarrow X Y$
- $A \rightarrow X B \Rightarrow A \rightarrow X B, B' \rightarrow A'$
- $A \rightarrow B X \Rightarrow A \rightarrow B, B' \rightarrow X A'$
- $A \rightarrow B C \Rightarrow A \rightarrow B, B' \rightarrow C, C' \rightarrow A'$
- $A \rightarrow \text{reg} \Rightarrow A \rightarrow R A', R \rightarrow \text{reg}$

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?
    - $X5 \rightarrow X5. \text{replace}([],))$
  - Current technique cannot handle it, use the closure of the character set of $X5$ as the approximation
    - $X5 \rightarrow \{*, )\}*$
 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 Transistor

- 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

\[
\begin{align*}
\text{String str} &= \text{tokens.nextToken()} \\
\text{String str1} &= \text{str.getToken()} \\
\text{String str2} &= \text{str.removeToken()}
\end{align*}
\]

FST for getToken

\[
\begin{align*}
\text{start} &\quad \text{S1} \\
\text{delim/} &\quad X/X \ (X=\Sigma) \\
\text{X/X} \ (X=\Sigma-\text{delim}) &\quad \text{S2}
\end{align*}
\]

FST for removeToken

\[
\begin{align*}
\text{start} &\quad \text{S1} \\
\text{delim/} &\quad X/X \ (X=\Sigma) \\
\text{X/} \ (X=\Sigma-\text{delim}) &\quad \text{S2}
\end{align*}
\]
FST for string operations

- Substring
  substring(1,2)

\[
\begin{align*}
S_1 \quad & \text{start} \\
S_2 \quad & X/ (X=\Sigma) \\
S_3 \quad & X/X (X=\Sigma) \\
& \quad X/ (X=\Sigma)
\end{align*}
\]
CFG-FST intersection

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

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

, 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

![Diagram of CFG-FST Intersection Algorithm](image-url)
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 \(A \rightarrow t\) in CFG, add \(A\) to the generating non-terminal set of \((s_1, s_2)\)

\[
\begin{align*}
S_1 & \xrightarrow{a/b} S_2 \\
& \Rightarrow \\
A & \rightarrow a
\end{align*}
\]
Solution of CFL-Reachability Problem, cont.

- For each non-terminal A on each pair of states \( <s_1, s_2> \), if \( B \in \text{generating-set}(s_2, s_x) \land C \rightarrow AB \in \text{Productions} \), add C to generating-set\( (s_1, s_x) \).

\[
\begin{align*}
S_1 & \rightarrow \{\ldots, A\} \\
S_2 & \rightarrow \{\ldots, B\} \\
S_x & \rightarrow \{\ldots\}
\end{align*}
\]

\( C \rightarrow AB \)
Solution of CFL-Reachability Problem, cont.

- For each non-terminal $A$ on each pair of states $<s_1, s_2>$, if $B \in \text{generating-set}(s_x, s_1) \land C \rightarrow BA \in \text{Productions}$, 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.

\[
\begin{align*}
S_1 & \rightarrow S_2 & b/c \{B, P, Q\} \\
S_2 & \rightarrow S_1 & b/b \{B, P, Q\} \\
S_1 & \rightarrow S_2 & b/c \{B, P, Q\} \\
S_2 & \rightarrow S_1 & b/b \{B, P, Q\} \\
\end{align*}
\]
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:

```
V1 V2
{…,A}
Vx
{…,B}
{…,C}
```

```latex
\[ C \rightarrow AB \]
```

```latex
\[ C_{1x} \rightarrow A_{12} B_{2x} \]
```
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:

  \[
  \begin{align*}
  X1 & \rightarrow X2X3 \\
  X2 & \rightarrow X4.\text{replace}(*,)) \quad \cdots \quad \text{op1} \\
  X4 & \rightarrow X5X6 \\
  X6 & \rightarrow X7.\text{replace}([,]) \\
  X7 & \rightarrow [X7]^+ \\
  X7 & \rightarrow [X7]^+
  \end{align*}
  \]

  input of op1: 
  \[
  \begin{align*}
  X4 & \rightarrow X5X6 \\
  X6 & \rightarrow X7.\text{replace}([,]) \\
  X7 & \rightarrow [X7]^+
  \end{align*}
  \]

  Resolve op2 first, then op1
Problems

- String operations in a deduction cycle
  - How to deal with the case below?
    - $X_7 \rightarrow X_7. \text{replace}([,])$

- Current technique cannot handle it, use the closure of the character set of $X_7$ as the approximation
  - $X_7 \rightarrow \{[, +]\}^*$
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.
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]));
    }
}
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

\[ V_1 \xrightarrow{\{\cdots, A\}} V_2 \xrightarrow{\{\cdots, B\}} V_x \]

\[ \{\cdots\} \]

\[ C(t) \rightarrow AB \]

\[ V_1 \xrightarrow{a/b} V_2 \]

\[ \{A\} \]

\[ A(t) \rightarrow a \]

\[ \Rightarrow \]

\[ V_1 \xrightarrow{C(t)} V_2 \xrightarrow{C_{1x}(t)} V_x \]

\[ \{\cdots, C\} \]

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

\[ V_1 \xrightarrow{a/b} V_2 \]

\[ \Rightarrow \]

\[ V_1 \xrightarrow{A(t)} V_2 \xrightarrow{A_{12}(t)} V_x \]

\[ \{A\} \]

\[ A_{12}(t) \rightarrow b \]
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 “>”)

```php
$a = ‘abc’;
$t = ‘f<br name=’;
echo $a.$t.’de’.’>’;
```
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
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
    \[
    \begin{align*}
    U + U &= U \\
    O + U &= O \\
    I + U &= I \\
    O + O &= O \\
    I + I &= I \\
    I + O &= C \\
    C + O &= C
    \end{align*}
    \]
Flag Propagation Algorithm

- **Four Propagation Rules**
  - **Neighboring Rule** (for neighboring variables)
    
    \[ S \rightarrow A(R)(L)B \]
    
    e.g.: \[ S \rightarrow A(O)(U)B \Rightarrow S \rightarrow A(O)(O)B \]
    
    \[ S \rightarrow A(U)(O)B \Rightarrow S \rightarrow A(O)(O)B \]
  
  - **Transitive Rule** (for terminals without ‘<’ and ‘>’)
    
    \[ S \rightarrow (L)'abc'(R) \]
    
    e.g.: \[ S \rightarrow (O)'abc'(U) \Rightarrow S \rightarrow (O)'abc'(O) \]
  
  - **Left Deducing Rule**
    
    \[ (L)S \rightarrow (L)AB \]
    
    e.g.: \[ (U)S \rightarrow (O)AB \Rightarrow (O)S \rightarrow (O)AB \]
  
  - **Right Deducing Rule**
    
    \[ S(R) \rightarrow AB(R) \]
Example CFG

- $A \rightarrow 'abc'$
- $T \rightarrow 'f<br name='$
- $D \rightarrow 'de'$
- $E \rightarrow '>$
- $S \rightarrow ATDE$

`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
Propagation

- \((U)A(\emptyset) \rightarrow (\emptyset)'abc' (\emptyset)\)
- \((\emptyset)T(\emptyset) \rightarrow (\emptyset)'f<\text{name=}'(I)\)
- \((\emptyset)D(\emptyset) \rightarrow (\emptyset)'de' (\emptyset)\)
- \((\emptyset)E(\emptyset) \rightarrow (I)'>' (\emptyset)\)
- \((U)S(\emptyset) \rightarrow (U)A(\emptyset) (\emptyset)T(\emptyset) (U)D(\emptyset) (U)E(\emptyset)\)

Left Deducing Rule
\((L)S \rightarrow (L)AB\)

Right Deducing Rule
\(S(R) \rightarrow AB(R)\)

Transitive Rule (for terminals without '<' and '>')
\(S \rightarrow (L)'abc'(R)\)

Neighboring Rule
\(S \rightarrow A(R)(L)B\)

\text{abcf}<\text{name=de}>
Final CFG with Differentiated Terminals

- \((O)A(O) \rightarrow (O)’abc’ (O)\)
- \((O)T(I) \rightarrow (O)’f<\text{br name=‘(I)}\)
- \((I)D(I) \rightarrow (I)’de’ (I)\)
- \((I)E(O) \rightarrow (I)’>’ (O)\)
- \((O)S(O) \rightarrow (O)A(O) (O)T(I) (I)D(I) (I)E(O)\)

\text{abcf}<\text{br name=de>
Conflict Cases

- **Code**
  
  ```
  $a = 'abc'
  echo $a.'<'.$a.'>'
  
  abc<abc>
  ```

- **CFG**
  
  (\(A(O) \rightarrow 'abc'\)  
  (O)B(I) \rightarrow (O)'<'(I)  
  (I)C(O) \rightarrow (I)'>'(O)  
  S(O) \rightarrow A(?)(O)B(I)  
  A(?)(I)C(O)

- **Final Result**
  
  A \rightarrow (C)'abc'(C)  
  B \rightarrow (O)'<'(I)  
  C \rightarrow (I)'>'(O)  
  S \rightarrow (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;
Outline

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

64
Why database applications?

- Database software projects depend 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:

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

- Possibly illegal dynamically generated SQL queries
  ```java
  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 $M(t) \cap MS' = \emptyset$. 

\[69\]
## Evaluation

### Evaluation on 9 programs

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

- **Total time in seconds**
- **Memory in Mbs**
- **Errors**
- **False Errors**
Limitations

- Sound but incomplete (may have false positives)
- Can find only syntax errors, cannot find runtime 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:
  
  ```python
  query = "SELECT * FROM accounts WHERE name=" + readName() + " AND password=" + readPassword();
  ```

- If input `' OR 'a'='a`, we get:
  
  ```sql
  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 on 5 real world projects

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

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 have false positives
- Cannot 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

<table>
<thead>
<tr>
<th>Experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExperimentId</td>
</tr>
<tr>
<td>VARCHAR(30)</td>
</tr>
<tr>
<td>req.</td>
</tr>
</tbody>
</table>

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

- Mark all the SQL queries that go 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 \( \Sigma^n \text{name} \Sigma^n \), which represents all strings containing the name.
- Intersect the automaton \( M(t) \) of each hot spot \( t \) and of each table column \( M(c) \).
- \( M(t) \cap M(c) \neq \emptyset \implies \) 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:

<table>
<thead>
<tr>
<th>Change</th>
<th>Predicted</th>
<th>True positives</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangeSc1</td>
<td>Added a column to a table</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>ChangeSc2</td>
<td>Added 3 columns to a table</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>ChangeSc3</td>
<td>Altered data type of a column</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>ChangeSc4</td>
<td>Added a new constraint to column</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>ChangeSp1</td>
<td>Added 3 new parameters to a stored proc.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ChangeSp2</td>
<td>Added new return columns to a stored proc.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ChangeSp3</td>
<td>Added new return columns to a stored proc.</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ChangeSp4</td>
<td>Added a new parameter in a stored proc.</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Predicted Changes vs. Real Changes:
Limitations

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

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

82
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:

```php
$x = $_Post[Color]
$content = $_Post[content]
if ($errMsg == "")
echo ("<p><h2><font color="".$x."">".$content.
"</font></h2><p>");
```
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
  ```php
  echo "<html>";
echo "<h1>";
if($head!="")
echo $head;
echo "</h1></html>";
  ```

- Possibly illegal dynamically generated SQL queries
  ```php
  echo "<html>";
echo "<h1>";
if($head!="")
echo $head.";
echo "</h1></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 $G \cap M' = \emptyset$. 
## Evaluation

### Evaluation on 6 programs

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

### Validation Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Depth</th>
<th>Bugs</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>webchess</td>
<td>9</td>
<td>1</td>
<td>123.33</td>
</tr>
<tr>
<td>schoolmate</td>
<td>17</td>
<td>14</td>
<td>7580.69</td>
</tr>
<tr>
<td>faqforge</td>
<td>10</td>
<td>30</td>
<td>45.64</td>
</tr>
<tr>
<td>phpwims</td>
<td>9</td>
<td>7</td>
<td>63.93</td>
</tr>
<tr>
<td>timeclock</td>
<td>14</td>
<td>11</td>
<td>145.61</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Files</th>
<th>Lines Per File</th>
<th>Total lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>std dev</td>
<td>max</td>
</tr>
<tr>
<td>Claroline</td>
<td>1144</td>
<td>148</td>
<td>248</td>
</tr>
<tr>
<td>FishCart</td>
<td>218</td>
<td>230</td>
<td>196</td>
</tr>
<tr>
<td>GecBBLite</td>
<td>11</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>PhPetition</td>
<td>17</td>
<td>159</td>
<td>75</td>
</tr>
<tr>
<td>PhPoll</td>
<td>40</td>
<td>144</td>
<td>112</td>
</tr>
<tr>
<td>Warp</td>
<td>44</td>
<td>554</td>
<td>520</td>
</tr>
<tr>
<td>Yapig</td>
<td>50</td>
<td>170</td>
<td>191</td>
</tr>
</tbody>
</table>

*Result of the detection*

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

*Caused by user input*

The information of subjects

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

All language specific code elements are externalized to property files.

- **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

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

- Externalized Code Elements

```java
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.manua=Manual
swing.menu.help=Help
swing.menu.about=About
swing.menu.quit=Quit

Risk_de.properties
swing.menu.options=Einstellung
swing.menu.manua=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

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

- 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.

Diagram:
- Constant Strings
- String Variables /Expressions
- GUI
Output API Methods

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

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

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

```java
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

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

```html
$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/';

$part_ref = 'http://www.sei.pku.edu.cn/';
$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>'
```

```
<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**
  
  ```php
  echo "and pressed 'refresh' on your browser.
       In this case, your responses have<br/>
  
  echo "already been saved."
  echo "</font></center><br /><br/>
  
  (from question.php, Lime Survey 0.97)
  
  ```

- **Constant Strings in value attribute of input tags**
  
  ```php
  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

  ```php
  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

PHP Code → String Taint Analysis → Context Free Grammar (CFG) Presenting Output

Output

Constant strings to be translated

CFG With Differentiated Terminals

Input Tag Checking

Our Approach

Flag Propagation

Constant String outside Tags:

abcf

Constant String in value attribute of input tags:
<input type=text value=search>

Constant String in output Tags:

Our Approach
Step 1 - String Taint Analysis

All Possible Contents of the output HTML

$\text{PHP Code} \rightarrow \text{String Taint Analysis} \rightarrow \text{CFG Presenting Output}

$\text{abc}$

$\text{t} = \text{f}<\text{br name='}$;

\text{echo}$ \text{a}.$\text{t}.'\text{de}'.>'$;

$\text{abc}$

$\text{A} \rightarrow 'abc'$

$\text{T} \rightarrow 'f}<\text{br name='}$

$\text{D} \rightarrow 'de'} \text{E->}'$

$\text{S} \rightarrow \text{ATDE}$

$\text{abcf}<\text{br name=de}>$

Static detection of SQL injection vulnerabilities, Wassermann and Su PLDI'07
Step 2 – Tag Range Analysis

CFG Presenting Output

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

Flag Propagation

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

CFG With Differentiated Terminals

abcf<br name=de>
Step 3- Input Tag Checking

```
<input type=text value=search>
```

1. **“input” inside tags**
   - Find `<` before ‘input’

2. **Determine scope of input tag**
   - “type”/”value” inside the scope of input tags
   - Determine scope of type/value attributes

3. **Terminals inside type/value attributes**
   - Output value
   - visible types
   - other

4. **Stop**
   - Find before/after terminal
   - No ‘<’
   - Stop

5. **Determining scopes**
Evaluation Subjects

- Three PHP projects
  - Lime Survey
  - Squirrel
  - Mrbs

<table>
<thead>
<tr>
<th>PJ/Ver</th>
<th>#LOC</th>
<th>#Constant Strings</th>
<th>#Need-to-Translate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lime Survey 0.97</td>
<td>11.3K</td>
<td>6493</td>
<td>290</td>
</tr>
<tr>
<td>Squirrel 0.2.1</td>
<td>4.0K</td>
<td>2457</td>
<td>184</td>
</tr>
<tr>
<td>MRBS 0.6</td>
<td>1.4K</td>
<td>704</td>
<td>57</td>
</tr>
</tbody>
</table>

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

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**

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

Most need to translate constant strings are outside tags which makes input tag checking very effective in reducing false negatives and false positives.
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)

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