Static Detection of Security Vulnerabilities in Scripting Languages

Research by Yichen Xie, Alex Aiken of Stanford University
Presented by Adam Bergstein
Outline

• Background
  – PHP
  – SQL Injection
  – Basic Blocks
  – Symbolic Execution
  – Static Analysis Basics

• Xie’s Analysis Tool (XAT)
  – CFG and Basic Blocks
  – Symbolic Analysis
  – Summarization Approach
  – Recap of XAT
  – Correlating Static Analysis Concepts

• My Thoughts
Background

There are some key concepts used before diving into this static analysis approach.
PHP

• Scripting languages are different
  – $_GET and $_POST user input
  – Stateless execution

• Dynamic native functionality and constructs
  – Dynamic includes
    • Mimics cut and paste of code into a script
    • Inherits runtime state of program at time of include
  – Dynamic variable types
  – Dynamic hash tables
  – Extract function
  – Eval function for implicit execution
PHP Code Examples

• Some strings are dynamic, some are not
  – $var = "$other_var"; $var = '$other_var';
• This function creates different variables based on run-time user input
  – extract($_GET);
• This block loads an include file based on run-time user input
  – $operation = $_GET['operation'];
    include("/includes/$operation.include");
  – Operation include could contain trusted functionality
• Hash table using string variable keys
  – $field = 'first_name';
    $field_value = $_GET[$first_name];
• Possibly unmediated eval call
  – $string = $_GET['string'];
    eval("echo $string;");
  – Could contain a value like: ‘NULL; mysql_query("delete from users")"
SQL Injection

• Unintended user input in database queries
• PHP has native functionality for databases
  – Makes it easier to produce vulnerabilities
  – No native prepared statement and object type integration like Java
• Strings are used in queries
  – String segments can be composed of one or more strings
  – One string may have influence of many variables, including user input
SQL Injection Examples

• Code
  – $whatever = $_GET['condition'];
  – mysql_query("select * from users where name='$whatever'")

• Retrieving information
  – Requests to page.php?condition=nothing’ or 1=1
  – Exposes all user information

• Altering information
  – Requests to page.php?condition=nothing’; delete from users;
  – Truncates data in users table
Basic Blocks

• One entry point and one exit point
  – Block comprised of one or more lines of code in between

• Basic blocks must terminate on “jumps”
  – IF statements, exit command, return command, exceptions
  – Calls and returns with functions

• A maximal basic block cannot be extended to include adjacent blocks without violating a basic block
  – The smallest basic block can be one line of code
  – Maximal basic blocks create blocks for as many lines of code as possible until it violates the rules of a basic block
Control Flow Graph: CFG

Definitions

Basic Block \(\equiv\) a sequence of statements (or instructions) \(S_1 \ldots S_n\) such that execution control must reach \(S_1\) before \(S_2\), and, if \(S_1\) is executed, then \(S_2 \ldots S_n\) are all executed in that order (unless one of the statements causes the program to halt)

Leader \(\equiv\) the first statement of a basic block

Maximal Basic Block \(\equiv\) a maximal-length basic block

CFG \(\equiv\) a directed graph (usually for a single procedure) in which:

- Each node is a single basic block
- There is an edge \(b_1 \rightarrow b_2\) if block \(b_2\) may be executed after block \(b_1\) in some execution

NOTE: A CFG is a conservative approximation of the control flow! Why?

Homework: Read Section 9.4 of Aho, Sethi & Ullman: algorithm to partition a procedure into basic blocks.
Symbolic Execution

- Applying a symbol to all variables and maintain state throughout all program paths
- Useful for determining how variables change throughout a program
- It is a means of simulating the execution of a block of code
Static Analysis Concept Review

• Abstract domains
  – How the behavior of the program is modeled

• Control flow graphs (ICFG or CFG)
  – Program statements and conditions modeled as nodes
  – ICFG is a collection of CFGs accounting for procedures

• Context sensitivity
  – Join over all paths versus join over all valid paths
  – Accounting for differences of calls to the same procedure instead of summarizing behavior across all the calls

• Flow sensitivity
  – Differentiating between control-flow paths

• Lattice and transition functions
  – Specific transitions of the CFG that alter lattice within a path

• Concretization function
  – Mapping actual values to the abstract model

• Sinks and sink sources
  – Identifying areas of the code that are meaningful to the analysis

• Summary functions (may/must, Sharir/Pnueli)
  – A means of generalizing behavior of reused code, especially useful in interprocedural data flow
int y;

void main() {
  n1: int a = 5;
  n2: y = 1;
  n3, n4: f(a);
  n5: if(...) {
    n6: a = 2;
    n7, n8: f(a);
  }
  n9: ...
}

void f(int b) {
  n10: if(...) 
    n11: y = 2;
  else
    n12: y = b;
}

CFG Example from Book
Xie’s Analysis Tool (XAT)

This presents a summarization approach that utilizes some of the traditional static analysis concepts we have looked at in class.
Fundamental Workflow

Code → AST → CFG → Symb. Ex. → Summary → Analysis
Code to AST

• XAT authors wrote or found a tool to convert the PHP source code into an abstract syntax tree
• Specific to PHP 5.0.5
• AST is then used to produce a control flow graph (CFG)
CFG in XAT

• The CFG in the previous example used basic blocks as nodes
  – These were not maximal basic blocks but still sensitive to jumps
  – More nodes allow for a more precise analysis of the graph by reasoning about the impact of every line

• XAT uses *maximal basic blocks* for nodes of a CFG
  – Each node can represent multiple lines of code
  – The code within the block is summarized by symbolic execution
  – Edges still mimic control flow within graph
  – Seems to be motivated by Harvard’s SUIF CFG Library
    • [http://www.eecs.harvard.edu/hube/software/v130/cfg.html](http://www.eecs.harvard.edu/hube/software/v130/cfg.html)

• There are multiple CFGs prepared as functions are found
  – Parsing main will uncover function calls
  – Each function is parsed into an AST and gets its own CFG
  – The CFG is then used in the creation of a summary, described later
How are the CFGs prepared?

• Start with the primary script, labeled main
  – Parse main into an AST
    • Document user-defined functions found
  – CFG for main is produced by extracting the maximal basic blocks from the AST
    • Edges are the control flow between blocks (jumps)
    • Conditional edges are labeled with the branch predicate
    • Functions are represented by a single node within a calling CFG
      – This references the intraprocedural summary described later
  – Unique CFGs are created for each user-defined function
    •Parsed into an AST and converted into a CFG
    • Also leverages maximal basic blocks
    • Recursive – if functions are found, they too are added in the queue and processed in a similar fashion
Example Code of a “main” script

```plaintext
Function foo($x)
{
...
}

Function bar($x, $y)
{
....
}

$var1 = ‘string value’;
$var2 = ‘string value’; //block 1
$var3 = foo($var1); //block 2
$var4 = bar($var, $var2); //block 3
if($var3 === TRUE){ //branch 1
    $var5 = foo($var4); //block 4
    $var6 = foo($var2); //block 5
    $var7 = bar($var5, $var6); //block 6
}
$var8 = ‘string value’;
...
Exit(); //block 7
```
Example of CFG

CFG for function MAIN

BAR

FOO

VAR2 = TRUE

VAR3 = TRUE

MAIN

BLOCK1

BLOCK2

BLOCK3

BLOCK4

BLOCK5

BLOCK6

BLOCK7
Symbolic Analysis in XAT

- Processes each maximal basic block found in the CFG
  - Sequential execution that starts at first block of main
  - Stops on end of block, return, exit, or call to a user-defined function that exits
- As the analysis progresses, each location is tracked using a simulation state
  - A location is a variable or entry in a hash table and has a value
    \[
    \text{State } (\Gamma) : \text{Loc} \rightarrow \text{Value}
    \]
    - Example: Location X maps to an initial value $X_0$
    - Each hash table entry is tracked uniquely based on key
- Analysis updates each location’s simulation state until the end of the block
  - The end state of the block is captured within the block summary described later
Language Constructs

Type \((\tau)\) ::= str | bool | int | \perp
Const \((c)\) ::= string | k | true | false | null
L-val \((lv)\) ::= \(x\) | Arg\#i | \(l[e]\)
Expr \((e)\) ::= \(c\) | \(lv\) | \(e\) binop \(e\) | unop \(e\) | \((\tau)e\)
Stmt \((S)\) ::= \(lv \leftarrow e\) | \(lv \leftarrow f(e_1, \ldots, e_n)\)
| return \(e\) | exit | include \(e\)

binop \(\in\) \{+,-,concat,\text{==},!,\text{==},<,>,\ldots\}
unop \(\in\) \{-,\neg\}

Figure 3: Language Definition
Reasoning about data types

• The symbolic execution accounts for differences in data types within the analysis

• String, boolean, integer, and unknown
  – Input parameters often start out as unknown types

• Strings are the most fundamental data type
  – User input is assumed to be a string when used within a query
  – String concatenation operation consists of other string segments
    • Each segment potentially composed of multiple variable values
  – Particularly useful in analysis of SQL injection to determine what variables influence a query
Boolean and Integer Types

• Boolean variables are useful for sanitization functions
  – Conditionally, a bool can influence sanitizing one or more other variables
  – Untaint(F-set, T-set) maps to each bool variable
    • F-set defines the list of sanitized variables when the boolean is false
    • T-set defines the list of sanitized variables when boolean is true

• Integers are tracked but “less emphasized”
  – Really only useful for when casting as a string or boolean
  – Of note: True = 1, False = 0
Data Type Value Representation

RECALL:

State (Γ) : Loc → Value

LIST OF POSSIBLE VALUES:

\[
\begin{align*}
\text{Loc} (l) & ::= x \mid l[\text{string}] \mid l[\bot] \\
\text{Init-Values} (o) & ::= l_0 \\
\text{Segment} (β) & ::= \text{string} \mid \text{contains}(σ) \\
\text{String} (s) & ::= [β_1, \ldots, β_n] \\
\text{Boolean} (b) & ::= \text{true} \mid \text{false} \mid \text{untaint}(σ_0, σ_1) \\
\text{Loc-set}(σ) & ::= \{l_1, \ldots, l_n\} \\
\text{Integer} (i) & ::= k \\
\text{Value} (v) & ::= s \mid b \mid i \mid o \mid \bot
\end{align*}
\]
Hash Tables Case Study

PROGRAM:

1. $hash = \texttt{$_POST;}$
2. $key = \texttt{‘userid’;}
3. $userid = \texttt{hash[\$key];}$

INITIALIZE:

$$\Gamma = \{\text{hash} \Rightarrow \text{hash}_0, \text{key} \Rightarrow \text{key}_0, \texttt{\_POST} \Rightarrow \texttt{\_POST}_0, \texttt{\_POST[userid]} \Rightarrow \texttt{\_POST[userid]}_0\}$$

SYMBOLIC EXECUTION (Black Magic):

- hash \rightarrow \texttt{\_POST}_0
- key \rightarrow \texttt{‘userid’}
- Hash[key] \rightarrow \texttt{\_POST[userid]}_0
- userid \rightarrow \texttt{\_POST[userid]}_0
Include Files

- This is a special case, specific to scripting languages
- Dynamically inserting code into a program
  - Inherits variable scope at the point of include statement
  - Like a “cut and paste” of code into current location
- An include file is processed by... (Draw on board)
  - Parse as an AST and convert into a CFG
  - Extract new user defined functions and process them with their own AST and CFG
  - Remove include statement from the original code and split block into two at point of include (splice operation)
  - Create an edge from the first original calling block to the first block of the include CFG
  - Create an edge for all return blocks of the include CFG to the original second calling block
  - Remove all return statements from blocks produced from include
Summarization Concept

• Should now have an idea of the running program represented as CFGs

• Can now run the analysis using the simulation state tracking of locations and values
  – Analysis tracks information about data throughout each block

• Input to analysis: Source code, query functions, sanitization functions
  – User defined input is assumed to be not sanitized

• Goal is to track sanitization of variables
  – Analyze simulation state throughout entire execution of the program and across procedure calls
Summarization Approach

• XAT summarizes the relevant information for SQL Injection
  – Starts at the first block of the main CFG and traverses through using symbolic execution
  – Updates the simulation state as the analysis progresses
  – Function calls trigger the interprocedural analysis
    • Main calls foo, foo calls bar, etc...

• Interprocedural Analysis
  – The current simulation state of main passed to an instance of the particular intraprocedural summary
  – If no intraprocedural summary exists, it is created and then analysis continues

• Intraprocedural Summary
  – A summary of all block summaries that belong to a function
  – If no block summaries exist, they are created and then analysis continues

• Block Summary
  – Summary of a maximal basic block (node in a CFG)
Block Summary

• Characterizes a CFG node

• Six Tuple: <E, D, F, T, R, U>
  – E (Error Set): Locations that flow into a query and need to be sanitized before entering the block
  – D (Definitions): Locations defined in current block
  – F (Value flow): Substring concept, pair of memory locations <L₁, L₂> where L₁ is a substring of L₂ on exit of the block
  – T (Termination): A true/false value if the block exits or if the block contains a call to a function that exits
  – R (Return value): The return value or undefined
  – U (Untaint set): Analyze each successor of a block. Define the set of sanitized values for each successor
Intraprocedural Summary

• Summarize each of the block summaries within a procedure
• Four Tuple: <E, R, S, X>
  – E (Error set): Locations that flow into a query and need to be sanitized before calling the function
    • Backward reachability analysis, start with each return block and traverse to the first block of the procedure
    • Leverage E, D, F, U of block summary to calculate a global E across all blocks in procedure
    • Main must not include any user input
  – R (Return set): Set of locations that correspond to the segments of the string returned
    • Only returns a set if it is a string
  – S (Sanitization set): Set of parameters or global variables sanitized within the function
    • Forward reachability analysis, start with first block and traverse to each return block
    • Intersection of each path corresponds to the sanitization set (flow sentivity)
  – X (Program exit): True/false value if this terminates across all paths
Intraprocedural Summary

FOO
(Intraprocedural Summary)

block summary

block summary

block summary

BAR
(Intraprocedural Summary)

block summary

block summary

block summary

block summary

block summary

block summary
Interprocedural Analysis

• Instances of function calls map the current simulation state to the parameters used in intraprocedural summaries
• Function $f$ has a summary tuple $<E,S,R,X>$ which maps to an actual call $f(e_1, e_2, ..., e_n)$ in a block
• This is the concretization function, which substitutes simulation state values to the summaries (abstract domain)
• Simulation state reflects the current state at the location the function is called
More Interprocedural Details

• Pre-conditions: Map simulation state to elements in E based on the parameters of the specific function call
  – All members of E must be sanitized before calling function, errors thrown if any global variable or parameter is not sanitized before call
  – Warnings thrown on unknown types due to inability to sanitize
• Exit condition: Block marked as an exit block, outgoing edges removed
• Post-condition: Identify and mark sanitized parameters or global variables after execution
  – If there is conditional sanitization, the intersection of the untaint set is used
  – This is useful for the analysis of the next block
• Return value: This is based on the data type of returned variable
  – Boolean: return untaint true and false sets based on actual parameters or global values
  – String: return the actual parameters or global values that correlate to the segments of the string returned
  – Transfers sanitized data back to the block that called and its simulation state is updated accordingly
Recap of XAT

• Parse source files into ASTs for main and functions
• Convert ASTs into CFGs for functions and main
  – Maximal basic block for nodes
  – “Cut and paste” splice for include files
• Run analysis on the CFGs
  – Maintain simulation state through symbolic analysis
  – Trigger interprocedural summaries
  – Trigger intraprocedural summaries for each procedure called
  – Trigger block summaries for all blocks in a procedure called
• Analysis should report errors for all non-sanitized data
  – Warnings returned for unknown data type variables used in queries
## Results

<table>
<thead>
<tr>
<th></th>
<th>Err</th>
<th>Msgs</th>
<th>Bugs (FP)</th>
<th>Warn</th>
</tr>
</thead>
<tbody>
<tr>
<td>e107</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>News Pro</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>myBloggie</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>DCP Portal</td>
<td>39</td>
<td>39</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>PHP Webthings</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>99</td>
<td>99</td>
<td>0</td>
<td>115</td>
</tr>
</tbody>
</table>

Table 1: Summary of experiments. **Err Msgs**: number of reported errors. **Bugs**: number of confirmed bugs from error reports. **FP**: number of false positives. **Warn**: number of unique warning messages for variables of unresolved origin (uninspected).
PHP Fusion

• Use of extract function created a lot of undefined data type variables in the analysis
  – This generated a lot of warnings

• Regular expressions created a difficulty in modeling
Correlating Static Analysis Concepts

• Sinks and sink sources
  – Database query functions and user-defined input, respectively
  – User-defined input is assumed to be tainted
• Sanitization functions
• Lattice: sanitized or not sanitized
• Abstract domains: summarization tuples and mapping to simulation state
• Soundness: It is sound since it returns errors for known issues (known data types) and warnings for issues it could not reason about (unable to model data type or dynamic functionality)
  – Sanitization set intersection of intraprocedural analysis could cause false positives though
• Completeness: Not complete; Authors admitted to struggles modeling all dynamic functionality (regular expressions, unknown data types)
  – Regular expression difficulties
More Static Analysis Concepts

• Context-sensitivity
  – It is fundamentally not context-sensitive since it does not process each function call uniquely – it uses summaries
  – This analysis does account for differences between different calls to functions due to the mapping of the simulation state and the ability to return different sanitization sets
  – Does the summarization remove data critical to context-sensitivity? Yes, according to the post-condition of the interprocedural analysis
  – JOP versus JOVP

• Flow sensitivity
  – It is not flow sensitive since the intraprocedural summary generalizes all of the control-flow paths of the blocks
  – This is seen in the intersection of the untaint set of boolean returns in intraprocedural summaries
My Thoughts

• Ease of coding and dynamic functionality make PHP very difficult to model
  – A lot of dynamic functionality
  – Heavy reliance on run-time data
  – I believe that XAT was fairly effective at trying to reason about this
• Neglected evaluated code
  – This is a logical extension of the sanitized/unsanitized string processing done in paper
  – Eval("$r = mysql_query("delete from $table")");
  – This is not an explicit function call
• Left out native PHP functions
  – How are they modeled?
• Left out PHP constants and DEFINE statements
  – Mimics variables but uses non-traditional syntax
  – Can be used within strings
More Thoughts

• PHP 5.x has object orientation
  – PHP 5.3 includes namespaces
  – No mention of any of this

• No mention of association of data type to specific sanitization function
  – Does not make any sense to run \textit{is\_numeric} on a string
  – Add\_slashes for a number, not validated

• This approach would work well across database platforms, since different functions can be passed for sanitization and for database queries
Questions?