S3 : A Symbolic String Solver for Vulnerability Detection in Web Applications

Minh-Thai Trinh, Duc-Hiep Chu, Joxan Jaffar
National University of Singapore (NUS)
Web applications

• **Usually:**
  • take *string values* as inputs,
  • manipulate *string values*, and then
  • use *string values* to construct database queries.

  ![Diagram of username and password inputs]

• "SELECT ... where user='\$user' and password='\$pwd'"
Vulnerabilities in web applications

- From OWASP, the most serious web security vulnerabilities:
  - #1: Injection flaws such as SQL injection
  - #3: Cross Site Scripting (XSS) flaws

Due to inadequate sanitization and inappropriate use of input strings provided by users
Dynamic Symbolic Execution (DSE)

- Current trend to detect vulnerabilities in web applications (Saxena [SP'10], Brumley [SP'10, ICSE'14])
- How does it work?
  - Symbolic execution for high coverage of program execution space
  - But concretize when necessary to avoid false positive
    - Event space
    - Loops
    - Hard-to-solve constraints such as non-linear constraints
function validateEmail(email) {
    ...
    // break email into 3 parts
    // local part
    // @ character
    // domain part
    if (domain.equals("nus.edu.sg")) {
        var reg = new RegExp("^[a-zA-Z][0-9]*$" );
        var test1 = reg.test(local);
        var test2 = local.length == 8;
        return test1 && test2;
    }
    else if (domain.equals("comp.nus.edu.sg"))
        return local.length >= 4;
    else
        return false;
}
Server-side PHP code

```php
$email = $_POST['email'];
$pwd = $_POST['password'];
$stm = "SELECT ... where email='$email' and password='$pwd'";
$result = mysql_query($stm);
```
SQL injection?

- To detect SQL injection, we may want to test whether $eml$ contains the string:

  \[ ' \text{OR} \ 1=1--' \]

- The attack specification (e.g. above) is given by security experts
Dynamic Symbolic Execution (DSE)

- First express all the input email addresses that can be validated by using the symbolic constraints
  - So that we know the form of $eml$ at the server side
  - Combine with the attack specification (on $eml$) to decide if the JavaScript code is vulnerable to SQL injection
function validateEmail(email) {
  ...
  // break email into 3 parts
  // local part
  // @ character
  // domain part
  if (domain.equals("nus.edu.sg")) {
    var reg = new RegExp("^[a-zA-Z][0-9]*$"),
    var test1 = reg.test(local),
    var test2 = local.length == 8;
    return test1 && test2;
  }
  else if (domain.equals("comp.nus.edu.sg"))
    return local.length >= 4;
  else
    return false;
}
Vulnerability Detection \(\rightarrow\) Constraint Solving

- Email address that passes the validation
  - if PC1 or PC2 is satisfiable

- Email address that leads to SQL injection
  - if It passes the validation and leads to $\text{eml}$ which contains the string
    - ' OR 1=1--
Checking satisfiability of formulae

- From vulnerability detection to checking the satisfiability of the following formulae:

**PC1’**

- email = local .“@” . domain /
- domain = “nus.edu.sg” /
- reg = /^[a-zA-Z][0-9]*$/ /
- local in reg /
- len(local) = 8 /
- email = $eml /
- $eml contains ““ OR 1=1--”

**PC2’**

- email = local .“@” . domain /
- domain = “comp.nus.edu.sg” /
- len(local) >= 4 /
- email=$eml /
- email contains ““ OR 1=1--”

---

S3: A Symbolic String Solver (CCS’14 talk)
Traditional Random Testing

- Test with concrete inputs
  - To exploit the SQL injection vulnerability, the input email addresses need to be validated first
    - In order to reach $\text{eml}$ at the server side
  - E.g. $'\ OR\ 1=1--$ Does not pass the validation test
  - **Unlikely** to test with the interesting case:
    - $'\ OR\ 1=1--@comp.nus.edu.sg$
S3: A Robust and Efficient String Solver
S3 Language

- Independent of input languages, e.g. PHP, JavaScript, etc.
- Non-string constraints
  - E.g., constraints of int-sort, bool-sort, ...
  - Length constraints
- String constraints over multiple string variables:
  - String equations
  - Membership predicates
  - String operations
    - ReplaceAll
- Regular expressions:
  - Constructed from Constant Strings using Union, Concatenation, Kleene star operations
  - S3 also supports character classes, escaped sequences, repetition operators, sub-match extraction using capturing parentheses, etc.
Comparison with Kaluza

- Kaluza is the representative for the state-of-the-art
  - Supports the most expressive constraint language so far
  - Is the underlying solver for a DSE framework (Kudzu[SP’10]) to detect vulnerabilities in JavaScript programs
  - Can also be used in other vulnerability analyses (NoTamper[CCS'10], WAPTEC[CCS'11])

- S3 is even more expressive:
  - Unbounded strings
  - High-level string operations such as ReplaceAll
    - Used frequently in sanitization
  - S3 has better performance, better robustness
function validateFields(p1,p2) {
    var re1 = /^\(ab\)*$/;
    var re2 = /^\(bc\)*$/;
    var t1 = re1.test(p1);
    var t2 = re2.test(p2);
    var t3 = p2.length > 0;
    return (t1 && t2 && t3)
}
Constraint Solving

JavaScript Code

```javascript
function validateFields(p1, p2)
{
    var re1 = /^(ab)*$/;
    var re2 = /^(bc)*$/;
    var t1 = re1.test(p1);
    var t2 = re2.test(p2);
    var t3 = p2.length > 0;
    return (t1 && t2 && t3)
}
```

Generated Constraints

\[
p1 \in ("ab")^* \land \\
p2 \in ("bc")^* \land \\
\text{length}(p2) > 0 \land \\
p1 \cdot p2 = "abababababababcc"
\]

Check for satisfiability. If UNSAT then the program is SAFE.

p1 • p2 ≠ "abababababababcc"
Star representation

Generated Constraints

\[
\begin{align*}
p_1 & \in ("ab")^* \land \\
p_2 & \in ("bc")^* \land \\
\text{length}(p_2) & > 0 \land \\
\end{align*}
\]

Our Internal Representation

\[
\begin{align*}
p_1 & = \text{star}("ab", n_1) \land \\
p_2 & = \text{star}("bc", n_2) \land \\
\text{length}(p_2) & > 0 \land \\
\end{align*}
\]

\[
\begin{align*}
p_1 \cdot p_2 & = "abababababababcc" \\
p_1 \cdot p_2 & = "ababababababababcc"
\end{align*}
\]
Regular Expression to String Equation

- \( p_1 \in ("ab")^* \rightarrow p_1 = \text{star}("ab", n_1) \)
- \( n_1 \) is used to represent the number of repeating “ab”
- \( n_1 \) is a variable, not a constant
- \( n_1 \) is a fresh variable and generated automatically
- Specifically, \( \text{star}("ab", n_1) \) can be interpreted as:
  - \((p_1 = \"") \land n_1=0) \lor p_1 = "ab" \cdot \text{star}("ab", n_1-1)
  - \((p_1 = \"") \land n_1=0) \lor p_1 = \text{star}("ab", n_1-1) \cdot "ab"
  - \((p_1 = \"") \land n_1=0) \lor p_1 = "ab" \cdot \text{star}("ab", n_1-2) \cdot "ab"
- Guided by the current context
Incremental Solving of S3

Kaluza (Generate and Test)

- Generate all possible length assignments for p1 and p2:
- \{pair | pair=(\text{len}(p1),\text{len}(p2))\} = \{(0, 12), (2, 10), (4, 8), (6, 6), (8, 4), (10, 2)\}

6 times of testing

- \text{star}(\text{“ab”},n1) . \text{star}(\text{“bc”},n2) = \text{“ababababababcc”}
- \text{star}(\text{“ab”},n1) . \text{star}(\text{“bc”},n2-1) . \text{“bc”} = \text{“ababababababcc”}

Note that \text{len}(p2) > 0

S3

- UNSAT (since “bc” \neq “cc”)

20/11/14
In summary

• **Kaluza**: *generate and test* approach
  • Generates all possible length assignments
  • For each length assignment, test if any string assignment satisfies the given formula.
  • Suffers from the combinatorial explosion

• **S3**: *incremental solving* approach
Implementation

- Is built on top of Z3-str (FSE’13) to exploit Z3’s infrastructure
  - Lemma generation
  - Non-string constraints
- S3 is more expressive than Z3-str:
  - Regular expressions (e.g. /a*b*/)
  - Membership predicates (e.g. x is in /a*b*/)
  - String operations that work on regular expression (e.g. replaceAll, match, split, etc.)
Experimental Evaluation

- Kaluza benchmarks: 50000+ test cases
  - Generated from the vulnerability analysis of Kudzu[SP’10]
- Classified by Kaluza into 2 categories
  - SAT Category: 21819 benchmarks
  - UNSAT Category: 33230 benchmarks
Interpreting the solver’s conclusions

- **SAT:**
  - The formula is satisfiable
  - Can generate the test input to exploit the vulnerabilities

- **UNSAT:**
  - The formula is unsatisfiable
  - Cannot generate any test input to exploit the vulnerabilities

- **MAYBE:**
  - Inconclusive
  - Need further investigation
Experimental Evaluation

S3 vs. Kaluza on SAT Category (21819 benchmarks)

<table>
<thead>
<tr>
<th></th>
<th>Kaluza</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>21808</td>
<td>21813</td>
</tr>
<tr>
<td>UNSAT</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>ERROR</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

S3: A Symbolic String Solver (CCS'14 talk)
### Experimental Evaluation

#### S3 vs. Kaluza on UNSAT Category (33230 benchmarks)

<table>
<thead>
<tr>
<th>Category</th>
<th>S3</th>
<th>Kaluza</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSAT</td>
<td>14877</td>
<td>7124</td>
</tr>
<tr>
<td>SAT</td>
<td>18210</td>
<td>2894</td>
</tr>
<tr>
<td>ERROR</td>
<td>12691</td>
<td>22653</td>
</tr>
<tr>
<td>TO (1 min)</td>
<td>143</td>
<td>559</td>
</tr>
</tbody>
</table>

**Table:**
- **S3:** A Symbolic String Solver (CCS'14 talk)
Experimental Evaluation

<table>
<thead>
<tr>
<th></th>
<th>#Files</th>
<th>Time(s)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>K</td>
<td>S3</td>
</tr>
<tr>
<td>SAT/Small</td>
<td>19984</td>
<td>5190</td>
<td>267</td>
<td>19.4x</td>
</tr>
<tr>
<td>SAT/Big</td>
<td>1835</td>
<td>3165</td>
<td>166</td>
<td>19.0x</td>
</tr>
<tr>
<td>UNSAT/Small</td>
<td>11761</td>
<td>4532</td>
<td>173</td>
<td>26.2x</td>
</tr>
</tbody>
</table>

Table 8: Timing Comparison: S3 vs. Kaluza
Conclusion

- A string solver
  - Support a rich set of constraints,
    - Generated from vulnerability analysis of web applications
  - Robust and efficient
- A modular contribution to any hypothetical DSE end-to-end system
- The tool is available soon
Future Work

- Strengthening the tool
  - Conflict clause learning in the string theory

- Integrating into an advanced DSE framework