Automating Software Engineering Tasks with Automated Logic Solvers

Tevfik Bultan
Verification Lab (VLab),
Computer Science Department
University of California, Santa Barbara
bultan@cs.ucsb.edu
http://www.cs.ucsb.edu/~vlab
Software is eating the world

• Commerce, entertainment, social interaction

• We will rely on software more in the future

• Winning formula: apps + cloud
One major road block

• Apps are not trustworthy
  – They are notorious for security vulnerabilities and unreliable behavior

• Apps are hard to program
  – Distributed behavior, interaction among many components and many languages

• Apps are hard to test
  – Highly dynamic behavior and concurrency

• Testing accounts for 50% of the development costs and software failures cost USA 60 billion annually
VLab Research Agenda

• Improve dependability of software
  – How?
    • By eliminating bugs
      – How?
        » Use automated logic solvers

• By eliminating bugs I do not mean debugging
  – Debugging is done after the bug is found
    • Finding the bug is the hard part

• Our goal is to find the bugs automatically
  – and eliminate them before the software is deployed
    • we do this using automated logic solvers
We have a hammer

Automated Logic Solvers
Actually, we have several hammers

**SAT solvers**: Satisfiability for Boolean logic formulas

**SMT solvers**: Satisfiability for combination of theories: linear arithmetic, arrays, strings, etc.

**Automata-based constraint solvers**: All solutions for Boolean, numeric, string constraints

**Model counting constraint solvers**: Counting solutions for Boolean, numeric, string constraints

Automated Logic Solvers
Unfortunately, life is complicated

Software engineering problems

Automated logic solvers
Automated bug finding is hard

- It is hard because software systems are too complex
- In order to make automated bug finding feasible
  - we need to focus our attention
Making automated bug finding feasible

• We focus our attention by
  – *Abstraction*
    • Hide details that do not involve the things we are checking
  – *Modularity*
    • We focus on one part of the system at a time
  – *Separation of concerns*
    • We focus on one property at a time
  • It turns out these are also the main principles of software design
    – We should be able to exploit the design structure in automated verification!
Separation of concerns

• First, we need to identify our concerns
  – What should we be concerned with if we want to eliminate the bugs in web applications

• Here are some concerns:
  – *Input validation*
    • Errors in input validation are a major cause of security vulnerabilities
  – *Access control*
    • Many applications unintentionally disclose users’ data
  – *Data model*
    • Integrity of the data model is essential for correctness of all applications
Why do we care about input validation?

- SQL Injection, XSS, File Inclusion as percentage of all computer security vulnerabilities (extracted from the CVE repository)
Why do we care about input validation?

**OWASP Top Ten 2007**
1. Injection Flaws
2. XSS
3. Malicious File Execution

**OWASP Top Ten 2010**
1. Injection Flaws
2. XSS

**OWASP Top Ten 2013**
1. Injection Flaws
2. Broken Auth. Session Management
3. XSS
Why care about data model?

- The data in the back-end database is the most important resource for most applications.
- Integrity and consistency of this data is crucial for dependability of the application.
Why care about access control?

14 million Verizon subscribers' details leak from crappily configured AWS S3 data store

US telco giant insists only infosec bods saw the info

By Iain Thomson in San Francisco 12 Jul 2017 at 19:34

Updated Another day, another leaky Amazon S3 bucket. This time, one that exposed account records for roughly 14 million Verizon customers to anyone online curious enough to find it.
Model-View-Controller (MVC) architecture

- MVC consists of three modules
  - Model represents the data
  - View is its presentation to the user
  - Controller defines the way the application reacts to user input
Web Application Architecture

- Model View Controller (MVC) pattern:
  Ruby on Rails, Zend for PHP, CakePHP, Struts for Java, Django for Python, ...
- Object Relational Mapping (ORM)
  ActiveRecord, Hibernate, ...

Desktop Client

Mobile Client

Internet

Data Model

ORM

OOP

Rel Db

RESTful Controller

View
Exploiting MVC for verification

• Modularity provided by MVC can be exploited in verification

• For checking access control properties
  – Focus on controllers

• For checking input validation
  – Focus on controllers

• For checking data model properties
  – Focus on the model
Three Step Analysis Process

Three step process

1. Using modularity, separation of concerns and abstraction principles, generate a model of the code for analysis
   – For example: Extract the input validation code from the application

2. Translate analysis questions about the extracted model to logic queries
   – For example: Convert questions about input validation vulnerabilities to questions about string constraints

3. Use a logic solver to answer the query
   – For example: Use a string constraint solver to find bugs in the input validation code
Three Applications

Separation of concerns
+ modularity
+ abstraction

Input validation

Data model

Access control

String + numeric constraint solver
SAT or SMT solvers
SAT or SMT solvers
A Simple Example

- Another PHP Example:

```php
<?php
$www = $_GET["www"]; 
$l_otherinfo = "URL";
echo "<td>" . $l_otherinfo . ": " . $www . "</td>";
?>
```

- The `echo` statement in line 4 is a sensitive function
- It contains a Cross Site Scripting (XSS) vulnerability
Is It Vulnerable?

A simple **taint analysis** can report this segment vulnerable using taint propagation

```php
1: <?php
2: $www = $_GET["www"];
3: $l_otherinfo = "URL";
4: echo "<td> $l_otherinfo : $www </td>";
5: ?>
```

- **echo** is tainted → script is **vulnerable**
How to Fix it?

- To fix the vulnerability we added a sanitization routine at line s. Taint analysis will assume that $www is *untainted* and report that the segment is *NOT* vulnerable

```php
tainted
1: <?php
2: $www = $_GET["www"];
3: $l_otherinfo = "URL";
untainted
s: $www = ereg_replace("[^A-Za-z0-9 .-@://]","",$www);
untainted
4: echo "<td>" . $l_otherinfo . ": " . $www. "</td>";
5: ?>
```
1: <?php
2: $www = $_GET["www"];
3: $l_otherinfo = "URL";
5: ?>
Sanitization Routines can be Erroneous

- The sanitization statement is not correct!

```php
ereg_replace("[^A-Za-z0-9 .-@://]", ",", $www);
```
- Removes all characters that are not in `{A-Za-z0-9 .-@:/}

- `.@` denotes **all characters between “.” and “@”**
  (including “<” and “>”)
- “.-@” should be “.-@”

- This example is from a buggy sanitization routine used in
  MyEasyMarket-4.1 (line 218 in file trans.php)
Vulnerabilities Can Be Tricky

- Input `<!sc+rip!t ...>` does not match the attack pattern
  - but it can cause an attack

```php
1: <?php
2: $www = $_GET['www'];
3: $l_otherinfo = "URL";
4: $www = ereg_replace("[^A-Za-z0-9 .-@://]", "", $www);
5: echo "<td> $l_otherinfo : $www </td>";
6: ?>
```
String Analysis

- If string analysis determines that the intersection of the attack pattern and possible inputs of the sensitive function is empty
  - then we can conclude that the program is secure

- If the intersection is not empty, then we can again use string analysis to generate a **vulnerability signature**
  - characterizes all malicious inputs

- Given $\Sigma^*<\text{script}\Sigma^*$ as an attack pattern:
  - The vulnerability signature for $_GET["www"]$ is $\Sigma^*<\alpha*sa*ca*ra*ia*pa*t\Sigma^*$
  where $\alpha \notin \{ A-Za-z0-9 -@:/ \}$
Automatically generated patch

- Our string analysis generates a vulnerability signature that characterizes all possible input strings that can exploit the vulnerability
- Our patch generation algorithm automatically generates a patch that removes the vulnerability

```php
1: <?php
P:  if(preg_match('/[^[^ <]*<.*/','$_GET["www"]'))
    $_GET["www"] = preg_replace(<,"",$_GET["www"]));
2:  $www = $_GET["www"];
3:  $l_otherinfo = "URL";
4:  $www = preg_replace("[^A-Za-z0-9 .-@://],"",$www);
5:  echo "<td>" . $l_otherinfo . ": " . $www. "</td>";
6: ?>```
**GOAL:** To automatically detect and repair vulnerabilities that are caused by input validation and sanitization errors (such as XSS and SQL Injection).

1) **Sanitizer Extraction**
- Static and dynamic program analysis to extract input validation and sanitization operations.

2) **String Analysis**
- Automata based string constraint solving.

3) **Bug Detection and Repair**
- Differential or policy directed (using attack patterns) bug detection and repair.
- Bug reports (attack strings) and code patches.
Min – Max Input Validation Policies
Differential Analysis: No Specification

```
attachEmailFieldFixer: function () {
  var fn_get_email = function (x) {
    return (x.tagName.toUpperCase() == "INPUT" && x.type == "email");
  }

  var fn_fix_email = function () {
    var e = this;
    if (e && e.value.length > 0) {
      e.value = e.value.replace(/\s/g, ' ');
    }
  }

  for (i = 0, len = forms.length; i < len; i += 1) {
    var j,
        j_len,
        elements = forms[i].elements,
        nodes = FUNBB.common.arrayOfMatched(fn_get_email, elements);

    for (j = 0, j_len = nodes.length; j < j_len; j += 1) {
      nodes[j].onblur = fn_fix_email;
    }
  }
};

<?php

valid_email = function ($email) {
  $return = ($hook = get_hook( 'em_fn_is_valid_email_start' )) ? eval($hook) : null;
  if ($return != null)
    return $return;

  if (strlen($email) > 80)
    return false;

  return preg_match( '/^-([<>()\[\]\.,;:\s@"\']\+\.[^<>()\[\]\.,;:\s@"\']\+)*(["\"']\+)+@((\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}|([a-zA-Z]\d-]+\.[a-zA-Z]{2,}))\$/i', $email);
}
```

Client-side

Server-side
Some Experiments

We applied our analysis to three open source PHP applications
• Webches 0.9.0 (a server for playing chess over the internet)
• EVE 1.0 (a tracker for player activity for an online game)
• Faqforge 1.3.2 (a document management tool)

<table>
<thead>
<tr>
<th>Application</th>
<th>#Files</th>
<th>LOC</th>
<th># XSS Sinks</th>
<th># SQLI Sinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Webchess 0.9.0</td>
<td>23</td>
<td>3375</td>
<td>421</td>
<td>140</td>
</tr>
<tr>
<td>EVE 1.0</td>
<td>8</td>
<td>906</td>
<td>114</td>
<td>17</td>
</tr>
<tr>
<td>Faqforge 1.3.2</td>
<td>10</td>
<td>534</td>
<td>375</td>
<td>133</td>
</tr>
</tbody>
</table>

• Attack patterns:
  – $\Sigma^* \text{<script}} \Sigma^*$ (for XSS)
  – $\Sigma^*$ or 1=1 $\Sigma^*$ (for SQLI)
Analysis Results

- We use (single, 2, 3, 4) indicates the number of detected vulnerabilities that have single input, two inputs, three inputs and four inputs

<table>
<thead>
<tr>
<th>Type</th>
<th># Vulnerabilities (single, 2, 3, 4)</th>
<th>Time (s) total</th>
<th>Memory (KB) average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. XSS SQL</td>
<td>(24, 3, 0, 0) (43, 3, 1, 2)</td>
<td>46.08 110.7</td>
<td>16850 136790</td>
</tr>
<tr>
<td>2. XSS SQL</td>
<td>(0, 0, 8, 0) (8, 3, 0, 0)</td>
<td>288.50 23.9</td>
<td>125382 17280</td>
</tr>
<tr>
<td>3. XSS SQL</td>
<td>(20, 0, 0, 0) (0, 0, 0, 0)</td>
<td>7.87 6.7</td>
<td>9948 &lt;1</td>
</tr>
</tbody>
</table>
A Case Study

- Schoolmate 1.5.4
  - Number of PHP files: 63
  - Lines of code: 8181

- Analysis results:

<table>
<thead>
<tr>
<th>Time</th>
<th>Memory</th>
<th>Number of XSS sensitive sinks</th>
<th>Number of XSS Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 minutes</td>
<td>281 MB</td>
<td>898</td>
<td>153</td>
</tr>
</tbody>
</table>

- After *manual* inspection we found the following:

<table>
<thead>
<tr>
<th>Actual Vulnerabilities</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
<td>48</td>
</tr>
</tbody>
</table>
Case Study – False Positives

- Why false positives?
  - **Path insensitivity:** 39
    - Path to vulnerable program point is not feasible
  - **Un-modeled built in PHP functions:** 6
  - **Unfound user written functions:** 3
    - PHP programs have more than one execution entry point
  - We can remove all these false positives by extending our analysis to a path sensitive analysis and modeling more PHP functions
Case Study - Sanitization

- We patched all actual vulnerabilities by adding sanitization routines

- We ran our analysis second time
  - and proved that our patches are correct with respect to the attack pattern we are using
String Analysis @ UCSB VLab

• Symbolic String Verification: An Automata-based Approach [Yu et al., SPIN’08]
• Symbolic String Verification: Combining String Analysis and Size Analysis [Yu et al., TACAS’09]
• Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses [Yu et al., ASE’09]
• Stranger: An Automata-based String Analysis Tool for PHP [Yu et al., TACAS’10]
• Relational String Verification Using Multi-Track Automata [Yu et al., CIAA’10, IJFCS’11]
• String Abstractions for String Verification [Yu et al., SPIN’11]
• Patching Vulnerabilities with Sanitization Synthesis [Yu et al., ICSE’11]
• Verifying Client-Side Input Validation Functions Using String Analysis [Alkhalaf et al., ICSE’12]
• ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies [Alkhalaf et al., ISSTA’12]
• Automata-Based Symbolic String Analysis for Vulnerability Detection [Yu et al., FMSD’14]
• Semantic Differential Repair for Input Validation and Sanitization [Alkhalaf et al. ISSTA’14]
• Automated Test Generation from Vulnerability Signatures [Aydin et al., ICST’14]
• Automata-based model counting for string constraints [Aydin et al., CAV’15]
© 2017

String Analysis for Software Verification and Security

Authors: Bultan, T., Yu, F., Alkhalaf, M., Aydin, A.

This is the first existing book focusing on string analysis
Three Applications

Separation of concerns
+ modularity
+ abstraction

Input validation

Data model

Access control

String + numeric constraint solver

SAT or SMT solvers

SAT or SMT solvers
Bug Finding in Data Models

• We worked on checking data model properties in Ruby-on-Rails applications
• Rails uses active records for object-relational mapping
• There are many options in active records declarations that can be used to specify constraints about the data model
• Our goal:
  – Automatically analyze the active records files in Rails applications to check if the data model satisfies some properties that we expect it to satisfy
• Approach:
  – Bounded (SAT based) or unbounded (SMT-based) bug detection
  – We automatically search for data model errors within a given scope for bounded case
An Example Rails Data Model

Static Data Model

class User < ActiveRecord::Base
  has_many :todos
  has_many :projects
end

class Project < ActiveRecord::Base
  belongs_to :user
  has_many :todos
  has_many :notes
end

class Todo < ActiveRecord::Base
  belongs_to :user
  belongs_to :project
end

class Note < ActiveRecord::Base
  belongs_to :project
end

Data Model Updates: Actions

class ProjectsController < ApplicationController
  def destroy
    @project = Project.find(params[:project_id])
    @project.notes.each do |note|
      note.delete
    end
    @project.delete
    respond_to(...)
  end
end
More Realistic Data Model
Properties to Check

Example properties for an online book store

• Relationship cardinality
  – *Is it possible to have two accounts for one user?*

• Transitive relations
  – *A book’s author should be the same as the book’s edition’s author*

• Deletion properties
  – *Deleting a user should not create orphan orders*
GOAL: To automatically detect and repair data model errors in web apps written using MVC based frameworks (such as Ruby on Rails)

1) Model Extraction
   Static analysis + instrumented execution

2) Property Inference
   Search for property patterns in data model

3) Logic Translation
   Encoding in First Order Logic (unbounded) or Boolean logic (bounded)

4) Bug Detection
   Bug detection via SMT or SAT solvers

5) Data Model Repair
   Automated repair based on property patterns

Rails code

Formal data model

User specified invariants

Bug reports (property violating instances)

Code patches
Model Extraction

Extraction is hard for actions
• Dynamic type system
• Metaprogramming
• Eval
• Ghost Methods

Observations
• The schema is static
• Action declarations are static
• ORM classes and methods do not change their semantic during execution
  – even if the implementation code is generated dynamically
Translation of Statements to FOL

• An action is a sequential composition of statements

• Statements
  – A state is represented with a predicate denoting all entities that exist in a state
  – A statement is a migration between states

• Loops
  – Use quantification to represent loop semantics
Inductive Verification

\( Inv(s) \) is a formula denoting that all invariants hold in state \( s \)

\( Action(s, s') \) is a formula denoting that the action may transition from state \( s \) to state \( s' \)

Check if: \( \forall s, s': Inv(s) \land Action(s, s') \rightarrow Inv(s') \)
# Experiments

<table>
<thead>
<tr>
<th></th>
<th>FatFreeCRM</th>
<th>Tracks</th>
<th>KandDan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>30358</td>
<td>18023</td>
<td>2173</td>
</tr>
<tr>
<td># ADS Nodes</td>
<td>85447</td>
<td>37755</td>
<td>907</td>
</tr>
<tr>
<td># Nodes after optimization</td>
<td>1611</td>
<td>1483</td>
<td>280</td>
</tr>
<tr>
<td># Classes</td>
<td>30</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td># Actions</td>
<td>167</td>
<td>70</td>
<td>35</td>
</tr>
<tr>
<td># Invariants</td>
<td>8</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td># Empty actions</td>
<td>107</td>
<td>52</td>
<td>31</td>
</tr>
<tr>
<td>Avg. # of predicates</td>
<td>286</td>
<td>205</td>
<td>69</td>
</tr>
<tr>
<td>Theorem prover peak memory</td>
<td>243Mb</td>
<td>203Mb</td>
<td>126Mb</td>
</tr>
<tr>
<td>Avg. time per action/invariant</td>
<td>3.1 sec</td>
<td>40.5 sec</td>
<td>10.5 sec</td>
</tr>
<tr>
<td># Action/invariant pairs</td>
<td>480</td>
<td>180</td>
<td>20</td>
</tr>
<tr>
<td>Verified</td>
<td>468</td>
<td>133</td>
<td>17</td>
</tr>
<tr>
<td>Falsified</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Inconclusive</td>
<td>8</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>False positives</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Detected Exceptions</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
Data Model Analysis @ UCSB VLab

- Ivan Bocic, Tevfik Bultan: Data Model Bugs. NFM 2015: 393-399
- Ivan Bocic, Tevfik Bultan: Efficient Data Model Verification with Many-Sorted Logic. ASE 2015: 42-52
- Ivan Bocic, Tevfik Bultan: Coexecutability for Efficient Verification of Data Model Updates. ICSE 2015: 744-754
- Ivan Bocic, Tevfik Bultan: Symbolic model extraction for web application verification. ICSE 2017: 724-734
Three Applications

Separation of concerns
+ modularity
+ abstraction

Input validation

Data model

Access control

String + numeric constraint solver

SAT or SMT solvers

SAT or SMT solvers
About 10 years ago we developed a technique for checking access control policies.

Basic idea:
- To check a complicated access control policy, compare it to a simple policy.
- For example, you may want to check that the complex policy is at least as restrictive as some default simple policy.
Access Control Checking for XACML

• Given two XACML policies P1 and P2:
  – Check if P1 is at least as strong as P2

• We showed that this type of differential policy check can be converted checking satisfiability of a Boolean logic formula

• We implemented a XACML policy checker using a SAT solver
Access Control Checking for Rails

- Rails developers use libraries such as CanCan, CanCanCanCan or Pundit for access control
- We develop a technique where we automatically extract the access control policy from the Rails code
- Then we check if the access control policy is correctly enforced in actions
- We showed that this type of check can be converted to checking satisfiability of an SMT formula
- We implemented a Rails access control policy checker using an SMT solver

• Graham Hughes, Tevfik Bultan: Automated verification of access control policies using a SAT solver. STTT 10(6): 503-520 (2008)

• Ivan Bocic, Tevfik Bultan: Finding access control bugs in web applications with CanCheck. ASE 2016: 155-166
Zelkova: Access Control at Amazon

• Zelkova uses automated reasoning to analyze policies and the future consequences of policies.
  – This includes AWS Identity and Access Management (IAM) policies, Amazon Simple Storage Service (S3) policies, and other resource policies.

• Zelkova translates policies into precise mathematical language and then uses automated reasoning tools to check properties of the policies.
  – These tools include automated reasoners called Satisfiability Modulo Theories (SMT) solvers, which use a mix of numbers, strings, regular expressions, dates, and IP addresses to prove and disprove logical formulas.
Three Applications

Separation of concerns
+ modularity
+ abstraction

- Input validation
  - String + numeric constraint solver

- Data model
  - SAT or SMT solvers

- Access control
  - SAT or SMT solvers
Conclusions

• Software dependability is a crucial problem for future of human civilization

• Using automated techniques that rely on automated logic solvers we can find and remove errors applications before they are deployed

• In order to develop feasible and scalable techniques we need to exploit the architecture of the software and the principles of modularity, abstraction and separation of concerns