Solvers for Software Reliability and Security

VIJAY GANESH
MIT
2011
The Software Reliability Problem

• Software is error-prone

• Significant and increasing costs

• Foundational research problem and opportunity
What is at the Core?
Logic Abstractions of Computation
What is at the Core?
Logic Abstractions of Computation
Why Logic for Program Reasoning

Logic Abstractions of Computation

Imperative Code: Operational view

from quickwiki.lib.base import *
from pylons.database import make_session

class PageController(BaseController):
  def before_(self):
    model.ctx.current = make_session()

  def index(self, title):
    page = model.Page.get_by(title=title)
    if page:
      c.content = page.get_wiki_content()
      return render_response('/page.myt')

e1f model.wikiwords.match(title):
  return render_response('/new_page.m abort(404)

  def edit(self, title):
    page = model.Page.get_by(title=title)

Logic Formula: Declarative View

\( \forall x. (P(x) \land Q(x)) \iff (\forall x. P(x)) \land (\forall x. Q(x)) \)
\( \forall x. (P(x) \lor Q(x)) \iff (\exists x. P(x)) \lor (\exists x. Q(x)) \)
\( \exists x. (P(x) \land Q(x)) \iff (\exists x. P(x)) \land (\exists x. Q(x)) \)
\( (\forall x. P(x)) \lor (\forall x. Q(x)) \iff (\forall x. P(x)) \lor (\forall x. Q(x)) \)
\( (\exists x. P(x)) \land (\forall x. Q(x)) \iff (\forall x. P(x)) \land (\forall x. Q(x)) \)

- Logic provides **abstractions of computation**
- Easy to work with abstractions
- Compact representation of desired properties
Reliability through Logical Reasoning
Engineering, Usability, Novelty

Program

Specification

Program Reasoning Tool

Logic Formulas

Solver

SAT/UNSAT

Program is Correct?
or Generate Counterexamples (Test cases)
What is at the Core?
The SAT/SMT Problem

- Rich logics (Modular arithmetic, Arrays, Strings,...)
- NP-complete, PSPACE-complete,...
- Practical, scalable, usable, automatic
- Enable novel software reliability approaches
So, What’s New?
From Reliability Problem to Solvers
So, What’s New?

From Reliability Problem to Solvers

Formal Methods

Program Analysis

SAT/SMT Solvers

- 1000+X improvement in 10 years
- Enabled completely new techniques
- Super-charged existing techniques

Monday, May 2, 2011
And, The Research Story is....
And, The Research Story is....

- Supercharge existing techniques
- Bounded model-checking
And, The Research Story is....

- Supercharge existing techniques
- Bounded model-checking
- Concolic testing
- Program analysis
And, The Research Story is....

- Solver-based languages
- Compiler optimizations using solvers
- Solver-based debuggers
- Solver-based type systems
- Solver-based concurrency bugfinding
- Solver-based synthesis

- Supercharge existing techniques
- Bounded model-checking

- Concolic testing
  - Program analysis

- 1,000 Constraints
- 10,000 Constraints
- 100,000 Constraints
- 1,000,000 Constraints

My Contributions

STP & HAMPI Solvers

Formal Methods
Program Analysis

Automatic Testing
Program Synthesis

STP & HAMPI Solvers
My Contributions

STP & HAMPI Solvers

- Can handle real-world formulas with millions of constraints
- **Enabled** completely new techniques (e.g., Concolic testing)
- Enable test million-line codes
- **Super-charged** existing techniques (e.g., Hardware bounded MC)
- Future is bright: Multicore, programming language, runtime systems
And, The Research Story is ...

- STP
- Enabled Concolic Testing
- EXE by Engler et al.
- BAP/BitBlaze by Song et al.
- Model checking by Dill et al.

- Solver-based languages (Alloy team)
- Solver-based debuggers
- Solver-based type systems
- Solver-based concurrency bugfinding

- HAMPI: String Solvers
- Ardilla by Ernst et al.
- Kudzu & Kaluza by Song et al.
- Klee by Engler et al.
- George Cadea’s Cloud 9 tester
- STP + HAMPI exceed 100+ projects

1,000,000 Constraints

100,000 Constraints

2005

2009

Today
## Key Contributions

<table>
<thead>
<tr>
<th>Name</th>
<th>Key Concept</th>
<th>Impact</th>
<th>Pubs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STP</strong></td>
<td>Bit-vector &amp; Array Solver(^1,2)</td>
<td>Abstraction-refinement for Solving</td>
<td>Concolic Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CAV 2007, CCS 2006, TISSEC 2008</td>
</tr>
<tr>
<td><strong>HAMPI</strong></td>
<td>String Solver(^1)</td>
<td>App-driven Bounding for Solving</td>
<td>Analysis of Web Apps</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ISSTA 2009(^3), TOSEM 2011 (Invited/in submission)</td>
</tr>
<tr>
<td><strong>(Un)Decidability</strong></td>
<td>results for Strings</td>
<td>Insights from Practical Applications</td>
<td>First results for strings+length</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>In submission</td>
</tr>
</tbody>
</table>

1. 100+ research projects use STP and HAMPI
2. STP won the SMTCOMP 2006 and 2010 competitions for bit-vector solvers
3. ACM Best Paper Award 2009
Rest of the Talk

• **STP** Bit-vector and Array Solver
  • Why Bit-vectors and Arrays?
  • How does STP scale: Abstraction-refinement
  • Impact: Concolic testing
  • Experimental Results

• **HAMPI** String Solver
  • Why Strings?
  • How does HAMPI scale: Bounding
  • Impact: String-based program analysis
  • Experimental Results

• Future Work
  • Multicore SAT
  • SAT-based Languages
  • Auto-tuning Solvers
  • Advice-based Solvers
Program Expressions → STP Solver

(x = z + 2 OR mem[i] + y <= 01)

• Bit-vector or machine arithmetic
• Arrays for memory
• C/C++/Java expressions
• NP-complete

SAT
UNSAT
Programs Reasoning & STP
Why Bit-vectors and Arrays

- STP logic tailored for software reliability applications
- Support *symbolic execution* / program analysis

<table>
<thead>
<tr>
<th>C/C++/Java/...</th>
<th>Bit-vectors and Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Var</td>
<td>32 bit variable</td>
</tr>
<tr>
<td>Char Var</td>
<td>8 bit variable</td>
</tr>
<tr>
<td>Arithmetic operation</td>
<td>Arithmetic function</td>
</tr>
<tr>
<td>(x+y, x-y, x*y, x/y,...)</td>
<td>(x+y, x-y, x*y, x/y,...)</td>
</tr>
<tr>
<td>assignments</td>
<td>equality</td>
</tr>
<tr>
<td>x = expr;</td>
<td>x = expr;</td>
</tr>
<tr>
<td>if conditional</td>
<td>if-then-else construct</td>
</tr>
<tr>
<td>if(cond) x = expr¹ else x = expr²</td>
<td>x = if(cond) expr¹ else expr²</td>
</tr>
<tr>
<td>inequality</td>
<td>inequality predicate</td>
</tr>
<tr>
<td>Memory read/write</td>
<td>Array read/write</td>
</tr>
<tr>
<td>x = *ptr + i;</td>
<td>ptr[i]; x = Read(ptr,i);</td>
</tr>
<tr>
<td>Structure/Class</td>
<td>Serialized bit-vector expressions</td>
</tr>
<tr>
<td>Function</td>
<td>Symbolic execution</td>
</tr>
<tr>
<td>Loops</td>
<td>Bounding</td>
</tr>
</tbody>
</table>

Monday, May 2, 2011
Problem: Automatically generate **crashing tests** given only the code.

![Diagram](image_url)

1. **Program**
   
2. **Symbolic Execution Engine with Implicit Spec**
   
3. **Formulas**
   - **SAT/UNSAT**
   - **STP**

4. **Crashing Tests**

5. **Automatic Tester**

---

*Monday, May 2, 2011*
How to Automate Testing?
Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player, ...

Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
        //1. Integer overflow causing NULL deref
        //2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}

• Formula captures computation
• Tester attaches formula to capture spec
Structured input processing code:
PDF Reader, Movie Player,...

Buggy_C_Program(int* data_field, int len_field) {

    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
        //1. Integer overflow causing NULL deref
        //2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}

Equivalent Logic Formula derived using symbolic execution

data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr : BITVECTOR(32); //symbolic

mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);

• Formula captures computation
• Tester attaches formula to capture spec
How to Automate Testing?
Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

Buggy_C_Program(int* data_field, int len_field) {
    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
        //1. Integer overflow causing NULL deref
        //2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}

Equivalent Logic Formula derived using symbolic execution

data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr : BITVECTOR(32); //symbolic
  .
  mem_ptr[ptr+i] = process_data(data_field[i]);
  mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
  .

• Formula captures computation
• Tester attaches formula to capture spec
How to Automate Testing?

Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

Buggy_C_Program(int* data_field, int len_field) {
    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
        //1. Integer overflow causing NULL deref
        //2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}

Equivalent Logic Formula derived using symbolic execution

data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr : BITVECTOR(32); //symbolic

mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);

//INTEGER OVERFLOW QUERY
0 <= j <= process(len_field);
ptr + i + j = 0?

• Formula captures computation
• Tester attaches formula to capture spec
How STP Works

Bird’s Eye View: Translate to SAT

Why Translate to SAT?
- Both theories NP-complete
- Non SAT approaches didn’t work
- Translation to SAT leverages solid engineering

Bit-vector & Array Formula
\[(x = z+2 \text{ OR } \text{mem}[i] + y \leq 01)\]

\[
\text{TranslateTo SAT} \quad \text{Boolean SAT Solver}
\]

STP

SAT

UNSAT
How STP Works

Rich Theories cause MEM Blow-up

- Making information explicit
  - Space cost
  - Time cost

Bit-vector & Array Formula

\[(x = z+2 \text{ OR } \text{mem}[i] + \gamma \leq 01)\]

...
Explicit Information causes Blow-up

Array Memory Read Problem

Logic Formula derived using symbolic execution

```latex
\begin{align*}
\text{data\_field, mem\_ptr : ARRAY;}
\text{len\_field : BITVECTOR(32); //symbolic}
\text{i, j, ptr : BITVECTOR(32); //symbolic}
\text{.}
\text{mem\_ptr[ptr+i] = process\_data(data\_field[i]);}
\text{mem\_ptr[ptr+i+1] = process\_data(data\_field[i+1]);}
\text{.}
\text{if(ptr+i = ptr+i+1) then mem\_ptr[ptr+i] = mem\_ptr[ptr+i+1];}
\end{align*}
```

//INTEGER OVERFLOW QUERY

\(0 \leq j \leq \text{process(len\_field);}\)

\(\text{ptr + i + j < ptr?}\)

• Array Aliasing is implicit
• Need to make information explicit during solving
• Cannot be avoided
How STP Works

Array-read MEM Blow-up Problem

- Problem: $O(n^2)$ axioms added, $n$ is number of read indices
- Lethal, if $n$ is large, say, $n = 100,000$; # of axioms is 10 Billion

\[
\begin{align*}
\text{Read(Mem,i}_0\text{)} &= \text{expr}_0 \\
\text{Read(Mem,i}_1\text{)} &= \text{expr}_1 \\
\text{Read(Mem,i}_2\text{)} &= \text{expr}_2 \\
&\quad\vdots \\
\text{Read(Mem,i}_n\text{)} &= \text{expr}_n
\end{align*}
\]
How STP Works
The Array-read Solution

• Key Observation
  • Most indices don’t alias in practice
  • Exploit locality of memory access in typical programs
  • Need only a fraction of array axioms for equivalence

Read(Mem,i_0) = expr_0
Read(Mem,i_1) = expr_1
Read(Mem,i_2) = expr_2
  ...
  ...
Read(Mem,i_n) = expr_n

v_0 = expr_0
v_1 = expr_1
  ...
  ...
v_n = expr_n
(i_0 = i_1) => (v_0 = v_1)
STP Key Conceptual Contribution
Abstraction-refinement Principle

Input Formula

Abstraction Step

Abstracted Formula

Boolean SAT Solver

Check Answer

Correct Answer

Refinement

Monday, May 2, 2011
## How STP Works

### What to Abstract & How to Refine?

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Less essential parts</td>
<td>1. Guided</td>
</tr>
<tr>
<td>2. Causes MEM blow-up</td>
<td>2. Must remember</td>
</tr>
<tr>
<td>Abstraction manages</td>
<td>Refinement manages</td>
</tr>
<tr>
<td>formula growth hardness</td>
<td>search-space hardness</td>
</tr>
</tbody>
</table>
How STP Works

Abstraction-refinement for Array-reads

Input
Read(A,i_0)=0
Read(A,i_1)=1
...
Read(A,i_n)=10,000
\( \Theta(i_0, i_1) \)

![Diagram of STP process]

Substitutions
Simplifications
Linear Solving
Array Abstraction
Conversion to SAT
Boolean SAT Solver

Refinement Loop

Result
How STP Works
Abstraction-refinement for Array-reads

Read(A, i₀) = 0
Read(A, i₁) = 1
...
Read(A, iₙ) = 10,000

θ'(i₀, i₁)

i₀ = i₁

Substitutions
Simplifications
Linear Solving

Array Abstraction

Conversion to SAT
Boolean SAT Solver

Result
How STP Works
Abstraction-refinement for Array-reads

Input

Abstracted Input
Array Axioms Dropped

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Result

Read(A,i_0)=0
Read(A,i_1)=1
...
Read(A,i_n)=10,000

v_0=0
v_1=1
...
v_n=10,000

θ'(i_0,i_1)
How STP Works

Abstraction-refinement for Array-reads

Input

Read(A,i₀)=0
Read(A,i₁)=1
...
Read(A,iₙ)=10,000
Θ(i₀,i₁)

Abstracted Input
Array Axioms Dropped

v₀=0
v₁=1
...
vₙ=10,000
Θ'(i₀,i₁)

Refinement Loop

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Result

Input Formula false in Assignment

Monday, May 2, 2011
How STP Works
Abstraction-refinement for Array-reads

Input

Read(A,i_0)=0
Read(A,i_1)=1
...
Read(A,i_n)=10,000
Θ(i_0,i_1)

Abstracted Input
Array Axioms Dropped

v_0=0
v_1=1
...
v_n=10,000
Θ'(i_0,i_1)

Refinement Loop

(i_0=i_1) → v_0=v_1

Add Axiom that is Falsified

i_0=0, i_1=0
v_0=0, v_1=1
...

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Result
How STP Works

Abstraction-refinement for Array-reads

Input

Read(A,i_0)=0
Read(A,i_1)=1
...
Read(A,i_n)=10,000
Θ(i_0,i_1)

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Refinement Loop

UNSAT
## STP vs. Other Solvers

<table>
<thead>
<tr>
<th>Testcase (Formula Size)</th>
<th>Result</th>
<th>Z3 (sec)</th>
<th>Yices (sec)</th>
<th>STP (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>610dd9c (~15K)</td>
<td>SAT</td>
<td>TimeOut</td>
<td>MemOut</td>
<td>37</td>
</tr>
<tr>
<td>Grep65 (~60K)</td>
<td>UNSAT</td>
<td>0.3</td>
<td>TimeOut</td>
<td>4</td>
</tr>
<tr>
<td>Grep84 (~69K)</td>
<td>SAT</td>
<td>176</td>
<td>TimeOut</td>
<td>18</td>
</tr>
<tr>
<td>Grep106 (~69K)</td>
<td>SAT</td>
<td>130</td>
<td>TimeOut</td>
<td>227</td>
</tr>
<tr>
<td>Blaster4 (~262K)</td>
<td>UNSAT</td>
<td>MemOut</td>
<td>MemOut</td>
<td>10</td>
</tr>
<tr>
<td>Testcase20 (~1.2M)</td>
<td>SAT</td>
<td>MemOut</td>
<td>MemOut</td>
<td>56</td>
</tr>
<tr>
<td>Testcase21 (~1.2M)</td>
<td>SAT</td>
<td>MemOut</td>
<td>MemOut</td>
<td>43</td>
</tr>
</tbody>
</table>

* All experiments on 3.2 GHz, 512 Kb cache
* MemOut: 3.2 GB (Memory used by STP much smaller), TimeOut: 1800 seconds
* Examples obtained from Dawn Song at Berkeley, David Molnar at Berkeley and Dawson Engler at Stanford
* Experiments conducted in 2007
STP vs. Other Leading Solvers

- STP vs. Boolector & MathSAT on 615 SMTCOMP 2007 - 2010 examples

* All experiments on 2.4 GHz, 1 GB RAM
* Timeout: 500 seconds/example
Impact of STP

• **Enabled** existing SE technologies to **scale**
  - Bounded model checkers, e.g., Chang and Dill

• **Easier to engineer** SE technologies
  - Formal tools (ACL2+STP) for verifying Crypto, Smith & Dill

• **Enabled new** SE technologies
  - Concolic testing (EXE,Klee,...) by Engler et al., Binary Analysis by Song et al.
## Impact of STP: Notable Projects

- Enabled Concolic Testing
- 100+ reliability and security projects

<table>
<thead>
<tr>
<th>Category</th>
<th>Research Project</th>
<th>Project Leader/Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal Methods</td>
<td>ACL2 Theorem Prover + STP</td>
<td>Eric Smith &amp; David Dill/Stanford</td>
</tr>
<tr>
<td></td>
<td>Verification-aware Design Checker</td>
<td>Jacob Chang &amp; David Dill/Stanford</td>
</tr>
<tr>
<td></td>
<td>Java PathFinder Model Checker</td>
<td>Mehlitz &amp; Pasareanu/NASA</td>
</tr>
<tr>
<td>Program Analysis</td>
<td>BitBlaze &amp; WebBlaze</td>
<td>Dawn Song et al./Berkeley</td>
</tr>
<tr>
<td></td>
<td>BAP</td>
<td>David Brumley/CMU</td>
</tr>
<tr>
<td>Automatic Testing Security</td>
<td>Klee, EXE</td>
<td>Engler &amp; Cadar/Stanford</td>
</tr>
<tr>
<td></td>
<td>SmartFuzz</td>
<td>Molnar &amp; Wagner/Berkeley</td>
</tr>
<tr>
<td></td>
<td>Kudzu</td>
<td>Saxena &amp; Song/Berkeley</td>
</tr>
<tr>
<td>Hardware Bounded Model-checking (BMC)</td>
<td>Blue-spec BMC</td>
<td>Katelman &amp; Dave/MIT</td>
</tr>
<tr>
<td></td>
<td>BMC</td>
<td>Haimed/NVIDIA</td>
</tr>
</tbody>
</table>
## Impact of STP

http://www.metafuzz.com

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Lines of Code</th>
<th>Number of Bugs Found</th>
<th>Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mplayer</td>
<td>~900,000</td>
<td>Hundreds</td>
<td>David Molnar/Berkeley &amp; Microsoft Research</td>
</tr>
<tr>
<td>Evince</td>
<td>~90,000</td>
<td>Hundreds</td>
<td>David Molnar/Berkeley &amp; Microsoft Research</td>
</tr>
<tr>
<td>Unix Utilities</td>
<td>1000s</td>
<td>Dozens</td>
<td>Dawson Engler et al./Stanford</td>
</tr>
<tr>
<td>Crypto Hash Implementations</td>
<td>1000s</td>
<td>Verified</td>
<td>Eric Smith &amp; David Dill/Stanford</td>
</tr>
</tbody>
</table>
Rest of the Talk

• **STP Bit-vector and Array Solver**
  - Why Bit-vectors and Arrays?
  - How does STP scale: Abstraction-refinement
  - Impact: Concolic testing
  - Experimental Results

• **HAMPI String Solver**
  - Why Strings?
  - How does HAMPI scale: Bounding
  - Impact: String-based program analysis
  - Experimental Results

• **Future Work**
  - Multicore SAT
  - SAT-based Languages
HAMPI String Solver

String Expressions → HAMPI Solver → SAT, UNSAT

- $X = \text{concat}(\text{"SELECT..."}, v) \text{ AND } (X \in \text{SQL\_grammar})$
- JavaScript and PHP Expressions
- Web applications, SQL queries
- NP-complete
What is the theory of Strings?

- Capture **String Expressions** in PHP, JavaScript, Perl, C/C++/Java
- Support **symbolic execution**/program analysis

<table>
<thead>
<tr>
<th>PHP/JavaScript/C++...</th>
<th>Theory of Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Var a; $a = 'name'</code></td>
<td><code>Var a : 12; //String variable of bounded-size</code>&lt;br&gt;<code>a = 'name'</code></td>
</tr>
<tr>
<td><code>a.&quot; is &quot;</code></td>
<td><code>Concat(a,&quot; is &quot;);</code></td>
</tr>
<tr>
<td><code>substr(a,1,3)</code></td>
<td><code>sub-string extraction</code></td>
</tr>
<tr>
<td><code>assignments/strcmp</code></td>
<td><code>equality</code>&lt;br&gt;<code>a = string_expr;</code></td>
</tr>
<tr>
<td><code>a = string_expr;</code></td>
<td><code>a in RE</code>&lt;br&gt;<code>a in SQL</code></td>
</tr>
</tbody>
</table>

Sanity check using regular expression RE<br>Expression in a suitable Language (e.g., SQL)
Hampi Use-case

String Operations in PHP, JavaScript,...

String Program Specification

Program Reasoning Tool

Logic Formulas

Program is Correct? or Generate Tests

HAMPI

SAT/UNSAT

Monday, May 2, 2011
Hampi Use-case

SQL Injection Vulnerabilities

Buggy PHP/JavaScript → Malicious SQL Query → Unauthorized Database Results → Backend Database
Hampi Use-case

SQL Injection Vulnerabilities

Web Vulnerabilities by Class
Q1-Q2 2009

- SQL Injection: 25%
- Cross-Site Scripting: 17%
- Authentication & Authorization: 14%
- Buffer Errors: 8%
- Path (Directory) Traversal: 8%
- Web Browser: 7%
- Code Injection: 4%
- Information Leak/Disclosure: 3%
- Cross-Site Request Forgery: 2%
Hampi Use-case

SQL Injection Vulnerabilities

Buggy Script

```sql
if (input in regexp(‘[0-9]+’))
query := “SELECT m FROM messages WHERE id=‘ ” + input + “ ‘”)
```

- `input` passes validation (regular expression check)
- `query` is syntactically-valid SQL
- `query` can potentially contain an attack substring (e.g., 1’ OR 1’ = ‘1)
Hampi Use-case
SQL Injection Vulnerabilities

if (input in regexp("[0-9]+"))
query := "SELECT m FROM messages WHERE id=' " + input + " '")

Should be: "^[0-9]+$"

• input passes validation (regular expression check)
• query is syntactically-valid SQL
• query can potentially contain an attack substring (e.g., 1’ OR ‘1’ = ‘1")
Hampi Use-case
SQL Injection Vulnerabilities

Input String

\[ \text{Var } v : 12; \]

SQL Grammar

\[ \text{cfg SqlSmall} := \text{"SELECT " [a-z]+ " FROM " [a-z]+ " WHERE " Cond}; \]
\[ \text{cfg Cond} := \text{Val} =\text{Val} \mid \text{Cond} \text{ OR } \text{Cond}; \]
\[ \text{cfg Val} := \text{[a-z]+} \mid \text{"" [a-z0-9]* ""} \mid \text{[0-9]+}; \]

SQL Query

\[ \text{val q} := \text{concat("SELECT msg FROM messages WHERE topicid="}, v, \text{""}); \]
\[ \text{assert v in [0-9]+}; \]
\[ \text{assert q in SqlSmall}; \]
\[ \text{assert q contains } \text{"OR \text{ ‘1’} \text{ ‘1’};} \]

"q is a valid SQL query"
"q contains an attack vector"

Hampi finds an attack input:
\[ v := 1' \text{' OR ‘I’ = ‘I} \]
\[ \text{SELECT msg FROM messages WHERE topicid=I’ OR ‘I’=’I’} \]
Hampi Key Contribution: Bounded Logics
Testing, Vulnerability Detection,...

• Finding satisfying assignment is key

• Short assignments are sufficient

• Hence, bounding strings is sufficient

• Bounded logics are easier to decide
# Hampi Key Conceptual Contribution

Bounding, expressiveness and efficiency

<table>
<thead>
<tr>
<th>$L_i$</th>
<th>Complexity of $\emptyset = L_1 \cap \ldots \cap L_n$</th>
<th>Current Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-free</td>
<td>Undecidable</td>
<td>n/a</td>
</tr>
<tr>
<td>Regular</td>
<td>PSPACE-complete</td>
<td>Quantified Boolean Logic</td>
</tr>
<tr>
<td>Bounded</td>
<td>NP-complete</td>
<td>SAT</td>
</tr>
</tbody>
</table>

Efficient in practice
How Hampi Works

Bird’s Eye View: Strings into Bit-vectors

Find a 4-char string v:
• (v) is in E
• (v) contains ()()

var v : 4;
cfg E := “()” | E E | (“ E “);
val q := concat(“(“, v, “)”);
assert q in E;
assert q contains “()”;

String Solution
v = )()(
How Hampi Works
Unroll Bounded CFGs into Regular Exp.

```
var v : 4;
cfg E := "()" | E E | "(" E ");
val q := concat("(" , v , ")");
assert q in E;
assert q contains "()";
```

```
Bound(E,6) → ([() + (())]) + ()[()] + [() + ()]()
```

```
string Solution
v = )()(
```

Diagram:
- **Hampi**
  - Normalizer
  - STP Encoder
  - STP Decoder
  - STP
  - Bit-vector Constraints
  - Bit-vector Solution
  - String Solution
How Hampi Works
Unroll Bounded CFGs into Regular Exp.

var v : 4;
cfg E := “()” | E E | (“ E “);
val q := concat(“(, v, ”));
assert q in E;
assert q contains“(())”;

Hampi

Normalizer

STP Encoder

STP Decoder

Bound(E,6) → ([() + ()]) + ()(() + ()) + [() + ()]()

Bit-vector Constraints

STP

Bit-vector Solution

String Solution

v = )()(
How Hampi Works
Converting Regular Exp. into Bit-vectors

Encode regular expressions recursively
• Alphabet \{ (, ) \} → 0, 1
• constant → bit-vector constant
• union + → disjunction ∨
• concatenation → conjunction ∧
• Kleene star * → conjunction ∧
• Membership, equality → equality

\[( v ) \in ( () [ ( ) ( ) + ( ( ) ) ] + [ ( ) ( ) + ( ( ) ) ] ( ) + ( [ ( ) ( ) + ( ( ) ) ] ) )\]

Formula \( \Phi_1 \lor Formula \Phi_2 \lor Formula \Phi_3 \)

How Hampi Works

Decoder converts Bit-vectors to Strings

Find a 4-char string v:
• (v) is in E
• (v) contains ()()

```
var v : 4;
cfg E := "()" | E E | "(" E ");
val q := concat("\"(, v, \")\");
assert q in E;
assert q contains "()";
```

```
String Solution
v = )()(
```
HAMPI: Result 1
Static SQL Injection Analysis

- 1367 string constraints from Wasserman & Su [PLDI’07]
- Hampi scales to large grammars
- Hampi solved 99.7% of constraints in < 1 sec
- All solvable constraints had short solutions
HAMPI: Result 2
Security Testing

• Hampi used to build Ardilla security tester [Kiezun et al., ICSE’09]

• 60 new vulnerabilities on 5 PHP applications (300+ kLOC)
  • 23 SQL injection
  • 37 cross-site scripting (XSS)

• 46% of constraints solved in < 1 second per constraint

• 100% of constraints solved in <10 seconds per constraint
HAMPI: Result 3
Comparison with Competing Tools

- **HAMPI vs. CFGAnalyzer (U. Munich):** HAMPI ~7x faster for strings of size 50+
- **HAMPI vs. Rex (Microsoft Research):** HAMPI ~100x faster for strings of size 100+
- **HAMPI vs. DPRLE (U. Virginia):** HAMPI ~1000x faster for strings of size 100+
## Impact of Hampi: Notable Projects

<table>
<thead>
<tr>
<th>Category</th>
<th>Research Project</th>
<th>Project Leader/Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Analysis</td>
<td>SQL-injection vulnerabilities</td>
<td>Wasserman &amp; Su/UC, Davis</td>
</tr>
<tr>
<td>Security Testing</td>
<td>Ardilla for PHP (SQL injections, cross-site scripting)</td>
<td>Kiezun &amp; Ernst/MIT</td>
</tr>
<tr>
<td>Concolic Testing</td>
<td>Klee, SAGE, Kudzu, NoTamper</td>
<td>Engler &amp; Cadar/Stanford, Godefroid/Microsoft Research, Saxena &amp; Song/Berkeley, Bisht &amp; Venkatakrishnan/U Chicago</td>
</tr>
<tr>
<td>New Solvers</td>
<td>Kaluza</td>
<td>Saxena &amp; Song/Berkeley</td>
</tr>
</tbody>
</table>
Rest of the Talk

• STP Bit-vector and Array Solver
  • Why Bit-vectors and Arrays?
  • How does STP scale: Abstraction-refinement
  • Impact: Concolic testing
  • Experimental Results

• HAMPI String Solver
  • Why Strings?
  • How does HAMPI scale: Bounding
  • Impact: String-based program analysis
  • Experimental Results

• Future Work
  • Multicore SAT
  • SAT-based Languages
Current Parallel SAT Approaches
Won’t Scale with more Nodes

• Portfolio or search-space split approach (ManySAT, pLingeling,...)
• Works ok on clusters
• Confirmed thru’ experimentation:
  • 12x speedup on a 128 node cluster
  • Not close to linear speedup
PSAT: Parallel SAT Approach

Partition SAT-Input into $k$ Pieces

- Didn’t work on clusters; much better prospects with multicore
- Latency much better on multicore than cluster
- Software engineering instances partition well
- Heuristics to minimize communication overhead
Imperative Language With SAT-based Declarative Primitives

Motivation:
- Declarative can be more robust
- Delegating the “how” to runtime

Combine imperative and SAT-based declarative language
- Efficient solvers evaluate and search
- Solvers leverage multicores

Examples
- Squander by Milicevic, Rayside and Daniel Jackson (MIT)
Related Work

• **Model Checking:**
  - Abstraction-refinement (Ed Clarke et al.)
  - Bounding (Ed Clarke, Daniel Jackson et al.)

• **Other SMT solvers**
  - Unsat core based approximations (Randy Bryant et al.)
  - Z3, CVC3, Boolector, BAT....

• **DPLL(T)**
  - Tinelli, Nieuwenhuis and Oliviera
Conclusions

• Logic formulas can capture meta-properties of software
  • The right logical abstraction (bit-vector and arrays, strings,...)

• Exploit meta-properties in solving formulas efficiently
  • Locality, modularity,...

• The more SMT solving, the less program analysis
  • Automation, ease-of-use,...
## Contributions at a Glance

- **STP* & HAMPI**
  (CAV 2007, TISSEC 2008, ISSTA 2009)

- Decidability/Undecidability results for strings
  (under submission)

- **BuzzFuzz: Directed Whitebox Fuzzing**
  (ICSE 2009)

- Concolic testers
  (JFuzz: NFM 2009)

- Solvers for integer linear arithmetic
  (FMCAD 2002, TACAS 2003)

- Retargetable compilers
  (DATE 1999)

## Future Work

- **Parallel SAT**

- **SAT-based programming languages**

- Program hardening

- Solvers for rich theories (attribute grammars, floating-point)

- **Auto-tuning SAT solvers**

- Advice-based SAT solvers

- Unsound and incomplete solvers

- **Solver-based concurrency bug-finding**

---

* 100+ research projects use STP and HAMPI (NSF funding $600,000.00)
* STP won the SMTCOMP 2006 and 2010 competitions for bit-vector solvers
* HAMPI paper won ACM Best Paper Award 2009