From SAT To SMT: Part 2

Vijay Ganesh
MIT
Topics covered in Lecture 1

✓ Motivation for SAT/SMT solvers in software engineering
  • Software engineering (SE) problems reduced to logic problems
  • Automation, engineering, usability of SE tools through solvers

✓ High-level description of the SAT/SMT problem & logics
  • Rich logics close to program semantics
  • Demonstrably easy to solve in many practical cases

✓ Modern SAT solver architecture & techniques
  • DPLL search, shortcomings
  • Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn, backjump,
  • Termination, correctness
  • Big lesson: learning from mistakes

Topics covered in Lecture 2

• Modern SMT solver architecture & techniques
  • Rich logics closer to program semantics
  • DPLL(T), Combinations of solvers, Over/under approximations

• My own contributions: STP & HAMPI
  • Abstraction-refinement for solving
  • Bounded logics

• SAT/SMT-based applications

• Future of SAT/SMT solvers
Modern SMT Solvers
Are SAT Solvers Enough?

What is SMT

• Satisfiability Modulo Theories. Just a fancy name for a mathematical theory

Motivations: why we need SMT?

• A satisfiability solver for rich logics/natural theories
• Easier to encode program semantics in these theories
• Easier to exploit rich logic structure, greater opportunity for optimizations

SMT Logics

• Bit-vectors, arrays, functions, linear integer/real arithmetic, strings, non-linear arithmetic
• Datatypes, quantifiers, non-linear arithmetic, floating point
• Extensible, programmable

SAT & SMT is an explosive combo: incredible impact
Standard-issue SMT Solver Architecture

Combination of theories & DPLL(T)

- Input SMT Instance
- Core Solver (Detects Equivalent Terms)
- Purify
  - DPLL(T) (Handles Boolean Structure)
- Theory 1
- Theory n

Output: SAT or UNSAT
Problem Statement

• Combine theory solvers to obtain a solver for a union theory

Motivation

• Software engineering constraints over many natural theories

• Natural theories well understood

• Modularity

How

• Setup communication between individual theory solvers

• Communication over shared signature

• Soundness, completeness and termination
Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

\[ f(f(x) - f(y)) = a \]
\[ f(0) = a+2 \]
\[ x = y \]

**IDEA:** \( \Phi_{comb} \iff (\Phi_{T1} \land EQ) \land (\Phi_{T2} \land EQ) \)

- **First Step:** purify each literal so that it belongs to a single theory
- **Second Step:** check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars

<table>
<thead>
<tr>
<th>UF</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(e_1) = a )</td>
<td>( e_2 - e_3 = e_1 )</td>
</tr>
<tr>
<td>( f(x) = e_2 )</td>
<td>( e_4 = 0 )</td>
</tr>
<tr>
<td>( f(y) = e_3 )</td>
<td>( e_5 = a + 2 )</td>
</tr>
<tr>
<td>( f(e_4) = e_5 )</td>
<td></td>
</tr>
<tr>
<td>( x = y )</td>
<td></td>
</tr>
</tbody>
</table>
Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

\begin{align*}
f(f(x) - f(y)) &= a \\
f(0) &= a + 2 \\
x &= y
\end{align*}

**IDEA:** $\Phi_{\text{comb}} \iff (\Phi_{T1} \land \text{EQ}) \land (\Phi_{T2} \land \text{EQ})$

- **First Step:** purify each literal so that it belongs to a single theory
- **Second Step:** check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars

<table>
<thead>
<tr>
<th>UF</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(e_1) = a$</td>
<td>$e_2 - e_3 = e_1$</td>
</tr>
<tr>
<td>$f(x) = e_2$</td>
<td>$e_4 = 0$</td>
</tr>
<tr>
<td>$f(y) = e_3$</td>
<td>$e_5 = a + 2$</td>
</tr>
<tr>
<td>$f(e_4) = e_5$</td>
<td>$e_2 = e_3$</td>
</tr>
<tr>
<td>$x = y$</td>
<td></td>
</tr>
</tbody>
</table>
Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

\[
\begin{align*}
\text{f(f(x) - f(y))} & = a \\
f(0) & = a + 2 \\
x & = y
\end{align*}
\]

IDEA: \( \Phi_{\text{comb}} \iff (\Phi_{T1} \land \text{EQ}) \land (\Phi_{T2} \land \text{EQ}) \)

- **First Step**: purify each literal so that it belongs to a single theory
- **Second Step**: check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars

**UF**
- \( f(e_1) = a \)
- \( f(x) = e_2 \)
- \( f(y) = e_3 \)
- \( f(e_4) = e_5 \)
- \( x = y \)
- \( e_1 = e_4 \)

**R**
- \( e_2 - e_3 = e_1 \)
- \( e_4 = 0 \)
- \( e_5 = a + 2 \)
- \( e_2 = e_3 \)
Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

Example Constraint over Linear Reals (R) and Uninterpreted Functions (UF)

\[
\begin{align*}
  f(f(x) - f(y)) &= a \\
  f(0) &= a + 2 \\
  x &= y
\end{align*}
\]

IDEA: \( \Phi_{\text{comb}} \leftrightarrow (\Phi_{T1} \land \text{EQ}) \land (\Phi_{T2} \land \text{EQ}) \)

- **First Step:** purify each literal so that it belongs to a single theory
- **Second Step:** check satisfiability and exchange entailed equalities over shared vars (EQ)
- The solvers have to agree on equalities/disequalities between shared vars
- UF says SAT, R says UNSAT. Combination returns UNSAT.

**UF**

- \( f(e_1) = a \)
- \( f(x) = e_2 \)
- \( f(y) = e_3 \)
- \( f(e_4) = e_5 \)
- \( x = y \)
- \( e_1 = e_4 \)

**R**

- \( e_2 - e_3 = e_1 \)
- \( e_4 = 0 \)
- \( e_5 = a + 2 \)
- \( e_2 = e_3 \)
- \( e_5 = a \)
Standard-issue SMT Solver Architecture

Combination of theories: Nelson-Oppen

IDEA: $\Phi_{\text{comb}} \leftrightarrow (\Phi_{T_1} \land \text{EQ}) \land (\Phi_{T_2} \land \text{EQ})$

- Does NOT always work, i.e., does not always give a complete solver

- Example: Cannot combine $T_1$ with only finite models, and $T_2$ with infinite models

- Impose conditions on $T_1$ and $T_2$
  - Stably Infinite: If a T-formula has a model it has an infinite model
  - Examples: Functions, Arithmetic
  - Extensions proved to be artificial or difficult
  - Deep model-theoretic implications (Ghilardi 2006, G. 2007)
Standard-issue SMT Solver Architecture

Combination of theories & DPLL(T)

Input SMT Instance

Core Solver
(Detects Equivalent Terms)

Purify

DPLL(T)
(Handles Boolean Structure)

Theory 1

... Theory n

Output: SAT or UNSAT
Problem Statement

• Efficiently handle the Boolean structure of the input formula

Basic Idea

• Use a SAT solver for the Boolean structure & check assignment consistency against a T-solver

• T-solver only supports conjunction of T-literals

Improvements

• Check partial assignments against T-solver

• Do theory propagation (similar to SAT solvers)

• Conflict analysis guided by T-solver & generate conflict clauses (similar to SAT solvers)

• BackJump (similar to SAT solvers)
Standard-issue SMT Solver Architecture

DPLL(T)

Uninterpreted Functions formula

\((1)\ (g(a) = c) \land
(\neg 2 \lor 3)\ (f(g(a)) \neq f(c) \lor (g(a) = d)) \land
(\neg 4)\ (c \neq d)\)

Theory and Unit Propagation Steps by DPLL(T)

(Unit Propagate) (1)
(Unit Propagate) (\neg 4)
(Theory Propagate) (2)
(Theory Propagate) (3)
UNSAT
# History of SMT Solvers

<table>
<thead>
<tr>
<th>Category</th>
<th>Research Project</th>
<th>Researcher/Institution/Time Period</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Theorem Proving</strong></td>
<td>NuPRL</td>
<td>Robert Constable / Cornell / 1970’s-present</td>
</tr>
<tr>
<td></td>
<td>Boyer-Moore Theorem Prover</td>
<td>Boyer &amp; Moore / UT Austin / 1970’s-present</td>
</tr>
<tr>
<td></td>
<td>ACL2</td>
<td>Moore, Kauffmann et al. / UT Austin / 1980’s - present</td>
</tr>
<tr>
<td></td>
<td>PVS Proof Checker</td>
<td>Natarajan Shankar / SRI International / 1990’s-present</td>
</tr>
<tr>
<td><strong>SAT Solvers</strong></td>
<td>DPLL</td>
<td>Davis, Putnam, Logemann &amp; Loveland / 1962</td>
</tr>
<tr>
<td></td>
<td>Chaff &amp; zChaff</td>
<td>Zhang, Malik et al. / Princeton / 1997-2002</td>
</tr>
<tr>
<td></td>
<td>MiniSAT</td>
<td>Een &amp; Sorensson / 2005 - present</td>
</tr>
<tr>
<td><strong>Combinations</strong></td>
<td>Simplify</td>
<td>Nelson &amp; Oppen / DEC and Compaq / late 1980s</td>
</tr>
<tr>
<td></td>
<td>Shostak</td>
<td>Shostak / SRI International / late 1980’s</td>
</tr>
<tr>
<td></td>
<td>ICS</td>
<td>Ruess &amp; Shankar / SRI International / late 1990’s</td>
</tr>
<tr>
<td></td>
<td>SVC, CVC, CVC-Lite, CVC3 ...</td>
<td>Barrett &amp; Dill / Stanford U. / late 1990’s</td>
</tr>
<tr>
<td></td>
<td>Non-disjoint theories</td>
<td>Tinelli, Ghilardi,.., / 2000 - 2008</td>
</tr>
<tr>
<td><strong>DPLL(T)</strong></td>
<td>Barcelogic and Tinelli group</td>
<td>Oliveras, Nieuwenhuis &amp; Tinelli / UPC and Iowa / 2006</td>
</tr>
<tr>
<td><strong>Under/Over Approximations</strong></td>
<td>UCLID</td>
<td>Seshia &amp; Bryant / CMU / 2004 - present</td>
</tr>
<tr>
<td></td>
<td>STP</td>
<td>Ganesh &amp; Dill / Stanford / 2005 - present</td>
</tr>
<tr>
<td><strong>Widely-used SMT Solvers</strong></td>
<td>Z3</td>
<td>DeMoura &amp; Bjorner / Microsoft / 2006 - present</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>Barrett &amp; Tinelli / NYU and Iowa / early 2000’s - present</td>
</tr>
<tr>
<td></td>
<td>OpenSMT</td>
<td>Bruttomesso / USI Lugano / 2008 - present</td>
</tr>
<tr>
<td></td>
<td>Yices</td>
<td>Deuterre / SRI International / 2005 - present</td>
</tr>
<tr>
<td></td>
<td>MathSAT</td>
<td>Cimatti et al. / Trento / 2005 - present</td>
</tr>
<tr>
<td></td>
<td>STP</td>
<td>Deuterre / SRI International / 2005 - present</td>
</tr>
<tr>
<td></td>
<td>UCLID</td>
<td>Ganesh / Stanford &amp; MIT / 2005 - present</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Seshia / CMU &amp; Berkeley / 2004 - present</td>
</tr>
</tbody>
</table>
Topics covered in Lecture 1

✓ Motivation for SAT/SMT solvers in software engineering
  • Software engineering (SE) problems reduced to logic problems
  • Automation, engineering, usability of SE tools through solvers

✓ High-level description of the SAT/SMT problem & logics
  • Rich logics close to program semantics
  • Demonstrably easy to solve in many practical cases

✓ Modern SAT solver architecture & techniques
  • DPLL search, shortcomings
  • Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn, backJump,
  • Termination, correctness
  • Big lesson: learning from mistakes

Topics covered in Lecture 2

✓ Modern SMT solver architecture & techniques
  • Rich logics closer to program semantics
  • DPLL(T), Combinations of solvers, Over/under approximations

• My own contributions: STP & HAMPI
  • STP: Abstraction-refinement for solving
  • Applications to dynamic symbolic testing (aka concolic testing)
  • HAMPI: Bounded logics

• SAT/SMT-based applications

• Future of SAT/SMT solvers
STP Bit-vector & Array Solver

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
The History of STP

1,000,000 Constraints

100,000 Constraints

2005

2009

Today

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

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

Solver-based languages (Alloy team)
Solver-based debuggers
Solver-based type systems
Solver-based concurrency bugfinding
# 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 (x+y, x-y, x*y, x/y,...)</td>
<td>Arithmetic function (x+y,x-y,x*y,x/y,...)</td>
</tr>
<tr>
<td>assignments x = expr;</td>
<td>equality x = expr;</td>
</tr>
<tr>
<td>if conditional if(cond) x = expr¹ else x = expr²</td>
<td>if-then-else construct x = if(cond) expr¹ else expr²</td>
</tr>
<tr>
<td>inequality</td>
<td>inequality predicate</td>
</tr>
<tr>
<td>Memory read/write x = *ptr + i;</td>
<td>Array read/write ptr[]; 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>
How to Automatically Crash Programs?
Concolic Execution & STP

Problem: Automatically generate **crashing tests** given only the code
How to Automate Testing?
Concolic Execution & STP

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

```c
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
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
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));
    }
}

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, ...

```c
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**

```c
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

Translate To SAT

Boolean SAT Solver

Bit-vector
&
Array Formula
(x = z+2 OR
mem[i] + y <= 01)
...
How STP Works

Rich Theories cause MEM Blow-up

- Making information explicit
  - Space cost
  - Time cost

Bit-vector &
Array Formula
(x = z+2 OR
mem[i] + y <= 01)
...

Formula Growth

STP

Boolean SAT Solver

SAT

UNSAT
Explicit Information causes Blow-up
Array Memory Read Problem

Logic Formula derived using symbolic execution

```plaintext
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]);

if(ptr+i = ptr+j) then mem_ptr[ptr+i] = mem_ptr[ptr+j];

//INTEGER OVERFLOW QUERY
0 <= j <= process(len_field);
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

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

Formula Growth

$\forall_0 = expr_0$
$\forall_1 = expr_1$
...
$\forall_n = expr_n$

$\forall_{i_0 = i_1} \Rightarrow (\forall_0 = \forall_1)$
$\forall_{i_0 = i_2} \Rightarrow (\forall_0 = \forall_2)$
...
$\forall_{i_1 = i_2} \Rightarrow (\forall_1 = \forall_2)$
...
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

\[
\begin{align*}
    \text{Read}(\text{Mem},i_0) &= \text{expr}_0 \\
    \text{Read}(\text{Mem},i_1) &= \text{expr}_1 \\
    \text{Read}(\text{Mem},i_2) &= \text{expr}_2 \\
    \quad &\vdots \\
    \text{Read}(\text{Mem},i_n) &= \text{expr}_n
\end{align*}
\]

\[
\begin{align*}
    v_0 &= \text{expr}_0 \\
    v_1 &= \text{expr}_1 \\
    \quad &\vdots \\
    v_n &= \text{expr}_n \\
    (i_0 = i_1) &\Rightarrow (v_0 = v_1)
\end{align*}
\]
Abstraction-refinement Principle

1. **Input Formula**
   - Abstraction Step
     - Abstracted Formula
       - Boolean SAT Solver
         - Check Answer
           - Correct Answer
## 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 formula growth hardness</td>
<td>Refinement manages 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
\[ \Theta(i_0,i_1) \]

Refinement Loop

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Result
How STP Works
Abstraction-refinement for Array-reads

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

\( i_0 = i_1 \)

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

Refinement Loop
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

Refinement Loop

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

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
i₀=0,i₁=0
v₀=0,v₁=1
...

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

Input
Formula false in Assignment

Vijay Ganesh, Dagstuhl, Aug 8-12, 2011
How STP Works
Abstraction-refinement for Array-reads

<table>
<thead>
<tr>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A,i_0)=0</td>
</tr>
<tr>
<td>Read(A,i_1)=1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Read(A,i_n)=10,000</td>
</tr>
<tr>
<td>Θ(i_0,i_1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstracted Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>ν_0=0</td>
</tr>
<tr>
<td>ν_1=1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>ν_n=10,000</td>
</tr>
<tr>
<td>Θ'(i_0,i_1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Refinement Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_0=i_1) → ν_0=ν_1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Substitutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplifications</td>
</tr>
<tr>
<td>Linear Solving</td>
</tr>
<tr>
<td>Array Abstraction</td>
</tr>
<tr>
<td>Conversion to SAT</td>
</tr>
<tr>
<td>Boolean SAT Solver</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Axiom that is Falsified</td>
</tr>
<tr>
<td>i_0=0, i_1=0</td>
</tr>
<tr>
<td>ν_0=0, ν_1=1</td>
</tr>
<tr>
<td>...</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)

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

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

Bar chart comparing time in seconds for STP, Boolector, and MathSAT on 615 SMTCOMP 2007 - 2010 examples.
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><strong>Formal Methods</strong></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><strong>Program Analysis</strong></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><strong>Automatic Testing Security</strong></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><strong>Hardware Bounded Model-checking (BMC)</strong></td>
<td>Blue-spec BMC BMC</td>
<td>Katelman &amp; Dave/MIT 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

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

### The Hampi Language

<table>
<thead>
<tr>
<th>PHP/JavaScript/C++...</th>
<th>HAMPI: Theory of Strings</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var a;</td>
<td>Var a : 1...20;</td>
<td>Bounded String Variables</td>
</tr>
<tr>
<td>$a = 'name'</td>
<td>a = 'name’</td>
<td>String Constants</td>
</tr>
<tr>
<td>string_expr.” is ”</td>
<td>concat(string_expr,” is “);</td>
<td>Concat Function</td>
</tr>
<tr>
<td>substr(string_expr,1,3)</td>
<td>string_expr[1:3]</td>
<td>Extract Function</td>
</tr>
<tr>
<td>assignments/strncmp</td>
<td>equality</td>
<td>Equality Predicate</td>
</tr>
<tr>
<td>a = string_expr;</td>
<td>a = string_expr;</td>
<td>Equality Predicate</td>
</tr>
<tr>
<td>a /= string_expr;</td>
<td>a /= string_expr;</td>
<td>Equality Predicate</td>
</tr>
<tr>
<td>Sanity check in regular expression RE</td>
<td>string_expr in RE</td>
<td>Membership Predicate</td>
</tr>
<tr>
<td>Sanity check in context-free grammar CFG</td>
<td>string_expr in SQL</td>
<td>Membership Predicate</td>
</tr>
<tr>
<td>string_expr contains a sub_str</td>
<td>string_expr contains sub_str</td>
<td>Contains Predicate</td>
</tr>
<tr>
<td>string_expr does not contain a sub_str</td>
<td>string_expr NOT?contains sub_str</td>
<td>Contains Predicate</td>
</tr>
</tbody>
</table>

(Vijay Ganesh, Dagstuhl, Aug 8-12, 2011)
Theory of Strings
The Hampi Language

• \[ X = \text{concat}(\text{"SELECT msg FROM msgs WHERE topicid = ", v}) \]
  \[ \text{AND} \]
  \[ (X \in \text{SQL\_Grammar}) \]

• \[ \text{input} \in \text{RegExp}([0-9]+) \]

• \[ X = \text{concat (str\_term1, str\_term2, "c")}[1:42] \]
  \[ \text{AND} \]
  \[ X \text{ contains "abc"} \]
SELECT m FROM messages WHERE id='1' OR 1 = 1
HAMPI Solver Motivating Example

SQL Injection Vulnerabilities

Source: IBM Internet Security Systems, 2009
Source: Fatbardh Veseli, Gjovik University College, Norway
**HAMPI Solver Motivating Example**

**SQL Injection Vulnerabilities**

```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')
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')

Should be: `^[0-9]+$`
if (input in regexp("[0-9]+"))
query := "SELECT m FROM messages WHERE id='" + input + "'")
Rest of the Talk

• HAMPI Logic: A Theory of Strings

• Motivating Example: HAMPI-based Vulnerability Detection App

• How **HAMPI** works

• Experimental Results

• Related Work: Theory and Practice

• HAMPI 2.0

• SMTization: Future of Strings
Expressing the Problem in HAMPI

SQL Injection Vulnerabilities

**Input String**

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

**SQL Grammar**

\[\text{cfg SqlSmall} := "SELECT " [a-z]+ " FROM " [a-z]+ " WHERE " \text{Cond};\]

\[\text{cfg Cond} := \text{Val} "=\" \text{Val} | \text{Cond} \text{ OR } \text{Cond};\]

\[\text{cfg Val} := [a-z]+ | "" [a-z0-9]* "" | [0-9]+;\]

**SQL Query**

\[\text{val } q := \text{concat}("SELECT msg FROM messages WHERE topicid=" , v , ")";\]

\[\text{assert } v \text{ in [0-9]+;}\]

\[\text{assert } q \text{ in SqlSmall;}\]

\[\text{assert } q \text{ contains } \text{"OR \'1\'\'1";}\]

*“q is a valid SQL query”*

*“q contains an attack vector”*
Hampi Key Conceptual Idea
Bounding, expressiveness and efficiency

<table>
<thead>
<tr>
<th>Li</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 Efficient in practice</td>
</tr>
</tbody>
</table>
Hampi Key Idea: Bounded Logics
Testing, Vulnerability Detection,...

• Finding SAT assignment is key
• Short assignments are sufficient

• Bounding is sufficient
• Bounded logics easier to decide
Hampi Key Idea: Bounded Logics
Bounding vs. Completeness

• Bounding leads to incompleteness

• Testing (Bounded MC) vs. Verification (MC)

• Bounding allows trade-off (Scalability vs. Completeness)

• Completeness (also, soundness) as resources
### SQL Injection Vulnerabilities

**Input String**

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

**SQL Grammar**

\[
\begin{align*}
\text{cfg } \text{SqlSmall} & := \text{"SELECT } [a-z]+ \text{ FROM } [a-z]+ \text{ WHERE } \text{ Cond}; \\
\text{cfg } \text{Cond} & := \text{Val }"=" \text{Val } \text{ OR } \text{Cond}; \\
\text{cfg } \text{Val} & := [a-z]+ \text{ | '}' [a-z0-9]* '}' \text{ | [0-9]+; }
\end{align*}
\]

**SQL Query**

\[
\text{val } q := \text{concat("SELECT msg FROM messages WHERE topicid='' }, v, "")};
\]

**SQLI attack conditions**

\[
\begin{align*}
\text{assert } v \text{ in [0-9]+;} \\
\text{assert } q \text{ in SqlSmall;} \\
\text{assert } q \text{ contains "OR '1='1"};
\end{align*}
\]

"q is a valid SQL query"

"q contains an attack vector"
How Hampi Works

Bird’s Eye View: Strings into Bit-vectors

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

\[
\begin{align*}
\text{var } v & : 4; \\
\text{cfg } E & := "()" \mid E \ E \mid "(" \ E \ "")"; \\
\text{val } q & := \text{concat}("(" , v , ")")); \\
\text{assert } q & \text{ in } E; \\
\text{assert } q & \text{ contains } "()";
\end{align*}
\]

\[
\begin{align*}
\text{v} & = )()(
\end{align*}
\]
How Hampi Works
Unroll Bounded CFGs into Regular Exp.

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

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

```
Normalizer
```

```
STP Encoder
```

```
STP Decoder
```

```
String Solution
v = )()(
```

```
Hampi
```

```
STP
```

```
Bit-vector Constraints
```

```
Bit-vector Solution
```

```
Vijay Ganesh, Dagstuhl, Aug 8-12, 2011
```

54
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) -> ([() + ()]) + ([() + (())] + [() + (())])
```

```
Normalizer
```

```
STP Encoder
```

```
STP Decoder
```

```
String Solution: v = )()(
```

```
STP
```

```
bound Auto-derived
```

```
STP Encoder
```

```
Bit-vector Constraints
```

```
Bit-vector Solution
```

```
```
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 "()";
```
How Hampi Works
Unroll Bounded CFGs into Regular Exp.

Step 1:
\[
\text{var } v : 4;
\text{cfg } E := "()" | E E | "( E "");
\text{val } q := \text{concat}("(" , v , "));
\text{assert } q \text{ in } E;
\text{assert } q \text{ contains } "()";
\]

Auto-derive lower/upper bounds \([L,B]\) on CFG

[6,6]

Step 2:
\[
\text{cfg } E := "()" | E E | "( E ""
\]

Look for minimal length string

"()"
How Hampi Works

Unroll Bounded CFGs into Regular Exp.

Step 3:

\[\text{cfg } E := \text{"\((\)\)\" | } E \ E \ | \text{"\(( E \)\)\"}\]

Length: 6

Recursively expand non-terminals:

Min. length constant: 

\[\text{([4,2])}\]
\[\text{([2,4])}\]
\[\text{([3,3])}\]
\[\text{([5,1])}\]
\[\text{([1,5])}\]
\[\text{([1,4,1])}\]

Construct Partitions

Step 4:

\[\text{cfg } E := \text{"\((\)\)\" | } E \ E \ | \text{"\(( E \)\)\"}\]

Length: 6

Recursively expand non-terminals:

Min. length constant: 

\[(())()\]
\[(()())\]
\[(((())\]

Construct RE
Unroll Bounded CFGs into Regular Exp.
Managing Exponential Blow-up

Recursively expand non-terminals:

```
cfg E := “()” | E E | (“ E“)
```

Length: 6

Min. length constant: ”()”

Construct RE

• Dynamic programming style

• Works well in practice
Unroll Bounded CFGs into Regular Exp.
Managing Exponential Blow-up

\[
\text{cfg } E := \text{“(“} \mid E \ E \mid \text{“(“ } E \text{“)“}
\]

Recursively expand non-terminals:

Construct RE

\[
\text{Length: 6}
\]

\[
\text{Bound}(E, 6) \quad \rightarrow \quad ([() + ((())]) + ()[() + ((())] + [() + ((())])()
\]

Min. length constant: ”(“
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 \(\Phi_1\) ∨ Formula \(\Phi_2\) ∨ Formula \(\Phi_3\)

\[
\]
How Hampi Works
Converting Regular Exp. into Bit-vectors

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

Formula $\Phi_1 \lor$ Formula $\Phi_2 \lor$ Formula $\Phi_3$


- Constraint Templates
- Encode once, and reuse
- On-demand formula generation
How Hampi Works

Decoder converts Bit-vectors to Strings

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

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

String Solution
v = )()(
Rest of the Talk

• HAMPI Logic: A Theory of Strings

• Motivating Example: HAMPI-based Vulnerability Detection App

• How HAMPI works

• Experimental Results

• Related Work: Theory and Practice

• HAMPI 2.0

• SMTization: Future of Strings
HAMPI: Result I
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
• Attackers inject client-side script into web pages

• Somehow circumvent same-origin policy in websites

• `echo "Thank you $my_poster for using the message board";`

• Unsanitized `$my_poster`

• Can be JavaScript

• Execution can be bad
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

5 added to US National Vulnerability DB
HAMPI: Result 3

Comparison with Competing Tools

- HAMPI vs. CFGAnalyzer (U. Munich): HAMPI ~7x faster for strings of size 50+
HAMPI: Result 3
Comparison with Competing Tools

RE intersection problems

• HAMPI 100x faster than Rex (MSR)

• HAMPI 1000x faster than DPRLE (U.Virginia)

• Pieter Hooimeijer 2010 paper titled ‘Solving String Constraints Lazily’
How to Automatically Crash Programs?

KLEE: Concolic Execution-based Tester

Problem: Automatically generate crashing tests given only the code

Program

Symbolic Execution Engine with Implicit Spec

Automatic Tester

STP

Formulas

SAT/UNSAT

Crashing Tests

Vijay Ganesh, Dagstuhl, Aug 8-12, 2011
How to Automatically Crash Programs?
KLEE: Concolic Execution-based Tester

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
How to Automatically Crash Programs?
KLEE: Concolic Execution-based Tester

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));
  }
}

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 Automatically Crash Programs?
KLEE: Concolic Execution-based Tester

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

```c
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 Automatically Crash Programs?
KLEE: Concolic Execution-based Tester

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
HAMPI: Result 4
Helping KLEE Pierce Parsers

Symbolic Execution Engine with Implicit Spec

Parser
Semantic Core

Mark Input Symbolic

Formulas
STP

SAT/UNSAT

Crashing Tests
HAMI: Result 4
Helping KLEE Pierce Parsers

Generate Input Using HAMPI; Mark Partially Symbolic

Parser

Semantic Core

Symbolic Execution Engine with Implicit Spec

Formulas

SAT/UNSAT

STP

Crashing Tests

KLEE
HAMPI: Result 4
Helping KLEE Pierce Parsers

• Klee provides API to place constraints on symbolic inputs

• Manually writing constraints is hard

• Specify grammar using HAMPI, compile to C code

• Particularly useful for programs with highly-structured inputs

• 2-5X improvement in line coverage
## 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</td>
<td>Engler &amp; Cadar/Stanford</td>
</tr>
<tr>
<td></td>
<td>Kudzu</td>
<td>Saxena &amp; Song/Berkeley</td>
</tr>
<tr>
<td></td>
<td>NoTamper</td>
<td>Bisht &amp; Venkatakrishnan/U Chicago</td>
</tr>
<tr>
<td>New Solvers</td>
<td>Kaluza</td>
<td>Saxena &amp; Song/Berkeley</td>
</tr>
</tbody>
</table>
# Impact of Hampi: Notable Projects

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Description</th>
<th>Project Leader/Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kudzu</td>
<td>JavaScript Bug Finder &amp; Vulnerability Detector</td>
<td>Saxena, Akhawe, Hanna, Mao, McCamant, Song/Berkeley</td>
</tr>
<tr>
<td>NoTamper</td>
<td>Parameter Tamper Detection</td>
<td>Bisht, Hinrichs/U of Chicago, Skrupsky, Bobrowicz, Vekatakrishnan/ U. of Illinois, Chicago</td>
</tr>
</tbody>
</table>
Impact of Hampi: Notable Projects

NoTamper

- Client-side checks (C), no server checks
- Find solutions $S_1, S_2, \ldots$ to C, and solutions $E_1, E_2, \ldots$ to $\sim C$ by calling HAMPI
- $E_1, E_2, \ldots$ are candidate exploits
- Submit $(S_1, E_1), \ldots$ to server
- If server response same, ignore
- If server response differ, report error
## Related Work (Practice)

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Project Leader/Institution</th>
<th>Comparison with HAMPI</th>
</tr>
</thead>
</table>
| Rex       | Bjorner, Tillman, Vornkov et al. (Microsoft Research, Redmond) | • HAMPI  
  + Length+Replace(s₁,s₂,s₃)  
  - CFG  
  • Translation to int. linear arith. (Z3) |
| Mona      | Karlund et al. (U. of Aarhus) | • Can encode HAMPI & Rex  
  • User work  
  • Automata-based  
  • Non-elementary |
| DPRLE     | Hooimeijer (U. of Virginia) | • Regular expression constraints |
## Related Work (Theory)

<table>
<thead>
<tr>
<th>Result</th>
<th>Person (Year)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undecidability of Quantified Word Equations</td>
<td>Quine (1946)</td>
<td>Multiplication reduced to concat</td>
</tr>
<tr>
<td>Decidability (PSPACE) of QF Theory of Word Equations</td>
<td>Makanin (1977), Plandowski (1996, 2002/06)</td>
<td>Makanin result very difficult Simplified by Plandowski</td>
</tr>
<tr>
<td>Decidability (PSPACE-complete) of QF Theory of Word Equations + RE</td>
<td>Schultz (1992)</td>
<td>RE membership predicate</td>
</tr>
<tr>
<td>QF word equations + Length() (?)</td>
<td>Matiyasevich (1971)</td>
<td>Unsolved Reduction to Diophantine</td>
</tr>
<tr>
<td>QF word equations in solved form + Length() + RE</td>
<td>G. (2011)</td>
<td>Practical</td>
</tr>
</tbody>
</table>
Future of HAMPI & STP

• HAMPI will be combined with STP
  • Bit-vectors and Arrays
  • Integer/Real Linear Arithmetic
  • Uninterpreted Functions
  • Strings
  • Floating Point
  • Non-linear

• Additional features planned in STP
  • UNSAT Core
  • Quantifiers
  • Incremental
  • DPLL(T)
  • Parallel STP
  • MAXSMT?

• Extensibility and hackability by non-expert
Future of Strings

• Strings SMTization effort started
  • Nikolaj Bjorner, G.
  • Andrei Voronkov, Ruzica Piskac, Ting Zhang
  • Cesare Tinelli, Clark Barrett, Dawn Song, Prateek Saxena, Pieter Hooimeijer, Tim Hinrichs

• SMT Theory of Strings
  • Alphabet (UTF, Unicode,...)
  • String Constants and String Vars (parameterized by length)
  • Concat, Extract, Replace, Length Functions
  • Regular Expressions, CFGs (Extended BNF)
  • Equality, Membership Predicate, Contains Predicate

• Applications
  • Static/Dynamic Analysis for Vulnerability Detection
  • Security Testing using Concolic Idea
  • Formal Methods
  • Synthesis
Conclusions & Take Away

• SMT solvers essential for testing, analysis, verification,...

• Core SMT ideas
  • Combinations
  • DPLL(T)
  • Over/Under approximations (CEGAR,...)
  • SAT solvers

• Future of SMT solvers
  • SMT + Languages
  • SMT + Synthesis
  • Parallel SAT/SMT

• Demand for even richer theories
  • Attribute grammars
  • String theories with length
Modern SMT Solver References

These websites and handbook have all the references you will need


Topics Covered

Topics covered in Lecture 1

✓ Motivation for SAT/SMT solvers in software engineering
  • Software engineering (SE) problems reduced to logic problems
  • Automation, engineering, usability of SE tools through solvers

✓ High-level description of the SAT/SMT problem & logics
  • Rich logics close to program semantics
  • Demonstrably easy to solve in many practical cases

✓ Modern SAT solver architecture & techniques
  • DPLL search, shortcomings
  • Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn, backJump,
  • Termination, correctness
  • Big lesson: learning from mistakes

Topics covered in Lecture 2

✓ Modern SMT solver architecture & techniques
  • Rich logics closer to program semantics
  • DPLL(T), Combinations of solvers, Over/under approximations

✓ My own contributions: STP & HAMPI
  • Abstraction-refinement for solving
  • Bounded logics

✓ SAT/SMT-based applications
  • Dynamic systematic testing
  • Static, dynamic analysis for vulnerability detection

✓ Future of SAT/SMT solvers
# Key Contributions

http://people.csail.mit.edu/vganesh

<table>
<thead>
<tr>
<th>Name</th>
<th>Key Concept</th>
<th>Impact</th>
<th>Pubs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STP</strong> Bit-vector &amp; Array Solver&lt;sup&gt;1,2&lt;/sup&gt;</td>
<td>Abstraction-refinement for Solving</td>
<td>Concolic Testing</td>
<td>CAV 2007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CCS 2006</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TISSEC 2008</td>
</tr>
<tr>
<td><strong>HAMPI</strong> String Solver&lt;sup&gt;1&lt;/sup&gt;</td>
<td>App-driven Bounding for Solving</td>
<td>Analysis of Web Apps</td>
<td>ISSTA 2009&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TOSEM 2011 (CAV 2011)</td>
</tr>
<tr>
<td><strong>Taint-based Fuzzing</strong></td>
<td>Information flow is cheaper than concolic</td>
<td>Scales better than concolic</td>
<td>ICSE 2009</td>
</tr>
<tr>
<td><strong>Automatic Input Rectification</strong></td>
<td>Acceptability Envelope: Fix the input, not the program</td>
<td>New way of approaching SE</td>
<td>Under 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. HAMPI: ACM Best Paper Award 2009
4. Retargetable Compiler (DATE 1999)
5. Proof-producing decision procedures (TACAS 2003)
6. Error-finding in ARBAC policies (CCS 2011)