Talk Outline

Topics covered in Lecture on SAT Solvers

- **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 on SMT Solvers

- **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**
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 (Many natural complexity classes have corresponding SATisfiability problems)

• 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
What is Logic

• A formal language for constructing mathematical formulas with an associated proof system

• Modern logic starts with the works of Boole, De Morgan, Frege, Cantor, Russell.

What is a formal language in the context of logic

• Well-defined rules for constructing formulas

• Formulas are defined inductively

• Universe of constant and variables

• Terms are constructed out of constants, variables and functions

• Atomic formulas are predicates applied to terms

• Formulas are Boolean combination of atomic formulas

• Appropriate quantification over variables
First-order logic

• Functions and predicates are uninterpreted

• FOL has equality

• Quantification only over variables (higher-order logics quantify over functions, predicates)

Soundness and completeness of first-order logic (Godel, 1930)

• Equivalence between provability and validity

• Axioms ⊢ A ⇔ Axioms ⊨ A

Undecidability of first-order logic (Turing, Church 1936)

• Hilbert’s 23rd problem

• No fully automatic proof system for first-order logic
Mathematical Theories

• Functions and predicates are interpreted (and appropriate axioms are added to FOL)

• Peano arithmetic (PA): +,-,*,/ are the functions. = and < the predicates.

• It is believed to be powerful enough to axiomatize number theory

Incompleteness theorem (Godel 1931)

• There are true statements that are not provable in a system as powerful as Peano arithmetic, assuming consistency of PA

• For the first time, this result distinguished truth from proof

• Huge impact on logic and computability

• Arithematization of syntax, Encode proofs as numbers, Diagonalize

• G: G is not provable in this Theory T
Mathematical Theories

• Peano arithmetic, Presburger, theories of strings, modular arithmetic, quantified Boolean logic

• Different complexity classes: From undecidable, doubly-exponential, all the way down to NP-complete

SMT problem refers to the satisfiability problems for such rich theories

• Satisfiability modulo-theories

• How do we solve the SAT problem for rich theories efficiently and practically

• Given the difficulty of solving these problems in general, what kind of heuristics are efficient

• Can we play with soundness and completeness in a controlled fashion?

• How do we combine such SATisfiability solvers into a solver for the combined theory

• Quantifiers

• How do we connect these solvers to practical software engineering applications
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:** \( \Phi_{\text{comb}} \Leftrightarrow (\Phi_{T1} \land \text{EQ}) \land (\Phi_{T2} \land \text{EQ}) \)

- **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

<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 ($\mathbb{R}$) 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**
- $f(e_1) = a$
- $f(x) = e_2$
- $f(y) = e_3$
- $f(e_4) = e_5$
- $x = y$

**R**
- $e_2 - e_3 = e_1$
- $e_4 = 0$
- $e_5 = a + 2$
- $e_2 = e_3$
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: \( \Phi_{\text{comb}} \iff (\Phi_{T_1} \land EQ) \land (\Phi_{T_2} \land EQ) \)

- 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>( e_2 = e_3 )</td>
</tr>
<tr>
<td>( x = y )</td>
<td></td>
</tr>
<tr>
<td>( e_1 = e_4 )</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:** \( \Phi_{comb} \iff (\Phi_{T1} \land EQ) \land (\Phi_{T2} \land EQ) \)

- **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
- UF says SAT, R says UNSAT. Combination returns UNSAT.

<table>
<thead>
<tr>
<th><strong>UF</strong></th>
<th><strong>R</strong></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>( e_5 = a )</td>
</tr>
<tr>
<td>( e_1 = e_4 )</td>
<td></td>
</tr>
</tbody>
</table>
Standard-issue SMT Solver Architecture
Combination of theories: Nelson-Oppen

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

- 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)

Output: SAT or UNSAT

Purify

DPLL(T)
(Handles Boolean Structure)

Theory 1

... 

Theory n
Standard-issue SMT Solver Architecture

DPLL(T)

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>DPLL(T)</strong></td>
<td>Barcelogic and Tinelli group</td>
<td>Oliveras, Nieuwenhuis &amp; Tinelli / UPC and Iowa / 2006</td>
</tr>
</tbody>
</table>
Talk Outline

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
The History of STP

- 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 Candea’s Cloud 9 tester
- STP + HAMPI exceed 100+ projects

1,000,000 Constraints

100,000 Constraints

2005

2009

Today
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 = expr^1 else x = expr^2</td>
<td>x = if(cond) expr^1 else expr^2</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[]; 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

Program

Symbolic Execution Engine with Implicit Spec

Formulas

Automatic Tester

SAT/UNSAT

Crashing Tests
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
**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

Vijay Ganesh
How STP Works

Rich Theories cause MEM Blow-up

- Making information explicit
  - Space cost
  - Time cost

Bit-vector & Array Formula

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

...
Explicit Information causes Blow-up
Array Memory Read Problem

Logic Formula derived using symbolic execution

```pascal
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\((\text{Mem},i_0)\) = expr\(_0\)
Read\((\text{Mem},i_1)\) = expr\(_1\)
Read\((\text{Mem},i_2)\) = expr\(_2\)
...
Read\((\text{Mem},i_n)\) = expr\(_n\)

Formula Growth

\[ v_0 = \text{expr}_0 \]
\[ v_1 = \text{expr}_1 \]
...
\[ v_n = \text{expr}_n \]

\((i_0 = i_1) \Rightarrow (v_0 = v_1)\)
\((i_0 = i_2) \Rightarrow (v_0 = v_2)\)
...
\((i_1 = i_2) \Rightarrow (v_1 = v_2)\)
...
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

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

\[
\begin{align*}
\nu_0 &= \text{expr}_0 \\
\nu_1 &= \text{expr}_1 \\
\ldots \\
\nu_n &= \text{expr}_n \\
(i_0 = i_1) \Rightarrow (\nu_0 = \nu_1)
\end{align*}
\]
STP Key Conceptual Contribution
Abstraction-refinement Principle

Input Formula → Abstraction Step → Abstracted Formula

Refinement → 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>
</tbody>
</table>

Abstraction manages formula growth hardness

Refinement manages search-space hardness
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)

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,i₀) = 0
Read(A,i₁) = 1

... Read(A,iₙ) = 10,000

Θ'(i₀,i₁)

i₀ = i₁
How STP Works

Abstraction-refinement for Array-reads

Input

\[ \text{Read}(A,i_0) = 0 \]
\[ \text{Read}(A,i_1) = 1 \]
\[ \ldots \]
\[ \text{Read}(A,i_n) = 10,000 \]
\[ \Theta(i_0,i_1) \]

Abstracted Input
Array Axioms Dropped

\[ v_0 = 0 \]
\[ v_1 = 1 \]
\[ \ldots \]
\[ v_n = 10,000 \]
\[ \Theta'(i_0,i_1) \]

Refinement Loop

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Result

Vijay Ganesh

Wednesday, 16 January, 13
How STP Works
Abstraction-refinement for Array-reads

Input

\[
\begin{align*}
\text{Read}(A,i_0) &= 0 \\
\text{Read}(A,i_1) &= 1 \\
\vdots \\
\text{Read}(A,i_n) &= 10,000 \\
\Theta(i_0,i_1)
\end{align*}
\]

Abstracted Input
Array Axioms Dropped

\[
\begin{align*}
v_0 &= 0 \\
v_1 &= 1 \\
\vdots \\
v_n &= 10,000 \\
\Theta'(i_0,i_1)
\end{align*}
\]

Refinement Loop

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Result

Input
Formula false in Assignment

Vijay Ganesh

Wednesday, 16 January, 13
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=i₁)⇒v₀=v₁

Add Axiom that
is Falsified

i₀=0,i₁=0
v₀=0,v₁=1
...

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

Result

Add Axiom that is Falsified

Vijay Ganesh
How STP Works

Abstraction-refinement for Array-reads

Input

\[
\begin{align*}
\text{Read}(A,i_0) &= 0 \\
\text{Read}(A,i_1) &= 1 \\
&\ldots \\
\text{Read}(A,i_n) &= 10,000 \\
\Theta(i_0,i_1)
\end{align*}
\]

Refinement Loop

Substitutions

Simplifications

Linear Solving

Array Abstraction

Conversion to SAT

Boolean SAT Solver

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><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</strong></td>
<td>Klee, EXE</td>
<td>Engler &amp; Cadar/Stanford</td>
</tr>
<tr>
<td><strong>Security</strong></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</strong></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>

Vijay Ganesh
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

- \( X = \text{concat}(\text{"SELECT..."}, v) \) 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/strcmp</td>
<td>equality</td>
<td>Equality Predicate</td>
</tr>
<tr>
<td>a = string_expr;</td>
<td>a = string_expr;</td>
<td></td>
</tr>
<tr>
<td>a /= string_expr;</td>
<td>a /= string_expr;</td>
<td></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></td>
</tr>
<tr>
<td>string_expr contains a sub_str</td>
<td>string_expr contains sub_str</td>
<td>Contains Predicate (Substring Predicate)</td>
</tr>
<tr>
<td>string_expr does not contain a sub_str</td>
<td>string_expr NOT?contains sub_str</td>
<td></td>
</tr>
</tbody>
</table>
Theory of Strings
The Hampi Language

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

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

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

SQL Injection Vulnerabilities

Web Vulnerabilities by Class
Q1-Q2 2009

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

SQL Injection Vulnerabilities

Buggy Script

```
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')
H Ampi Solver Motivating Example

**SQL Injection Vulnerabilities**

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’)
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

Var v : 12;

cfg SqlSmall := "SELECT " [a-zA-Z]+ " FROM " [a-zA-Z]+ " WHERE " Cond;

cfg Cond := Val "=" Val | Cond " OR " Cond;

cfg Val := [a-zA-Z]+ | "" [a-zA-Z0-9]* "" | [0-9]+;

SQL Grammar

SQL Query

val q := concat("SELECT msg FROM messages WHERE topicid='", v, "};

assert v in [0-9]+;

assert q in SqlSmall;

assert q contains "OR '1'='1'';

“q is a valid SQL query”

“q contains an attack vector”

SQLI attack conditions
### Hampi Key Conceptual Idea
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 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**

Var v : 12;

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

cfg Cond := Val "=" Val | Cond " OR " Cond;

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

**SQL Query**

val q := concat("SELECT msg FROM messages WHERE topicid=" v "");

assert v in [0-9]+;

assert q in SqlSmall;

assert q contains "OR ‘1’='1’;";

“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 ()()

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

STP Encoder

STP Decoder

Normalizer

Hampi

Bit-vector Constraints

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) → ([() + ()]) + ()[() + ()] + [() + ()]()
```

```
STP Encoder
```

```
Normalizer
```

```
STP Decoder
```

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

Vijay Ganesh

Wednesday, 16 January, 13
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 "()";
```

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

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

Unroll Bounded CFGs into Regular Exp.

**Step 1:**
```
var v : 4;
cfg E := "()" | E E | "( E "");
val q := concat("(, v, ")");
assert q in E;
assert q contains "()";
```

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

\([6,6]\)

**Step 2:**
cfg E := "()" | E E | "( E ""

Look for minimal length string

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

Step 3:
\[
\text{cfg E := "() | E E | "( E "\n}
\]
Length: 6
Recursively expand non-terminals:
Construct Partitions
[4,2]
[2,4]
[3,3]
[5,1]
[1,5]
[1,4,1]
Min. length constant: "()

Step 4:
\[
\text{cfg E := "() | E E | "( E "\n}
\]
Length: 6
Recursively expand non-terminals:
Construct RE
(()())
(()())
(((())
Min. length constant: "()

Wednesday, 16 January, 13
Unroll Bounded CFGs into Regular Exp.
Managing Exponential Blow-up

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

Recursively expand non-terminals:

Bound(E,6) \rightarrow ([() + (()])] + [()] + (())]()
Encode regular expressions recursively
• Alphabet \{ (, ) \} → 0, 1
• constant → bit-vector constant
• union + → disjunction ∨
• concatenation → conjunction ∧
• Kleene star * → conjunction ∧
• Membership, equality → equality

\[
( \lor ) \in ( ) [ ( ) ( ) + ( ( ) ) ] + [ ( ) ( ) + ( ( ) ) ] ( ) + ( [ ( ) ( ) + ( ( ) ) ] )
\]

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 ()()
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 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 and XSS

• 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)

  5 added to US National Vulnerability DB

• 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: 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

**Diagram:**
- **Program** flows to **Symbolic Execution Engine with Implicit Spec**
- **Symbolic Execution Engine with Implicit Spec** outputs **SAT/UNSAT**
- **STP** takes **Formulas** and outputs **SAT/UNSAT**
- **SAT/UNSAT** flows to **Crashing Tests**
- **Crashing Tests** flows to **Automatic Tester**

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, ...

```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 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

```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]);
.
.
```

• 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

```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]);
```

- 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));
    }
}

• Formula captures computation
• Tester attaches formula to capture spec

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?
HAMPI: Result 4
Helping KLEE Pierce Parsers

Mark Input
Symbolic

Parser

Semantic Core

Symbolic Execution
Engine
with Implicit Spec

Crashing Tests

Formulas

STP

SAT/UNSAT
HAMPI: Result 4
Helping KLEE Pierce Parsers

Generate Input Using HAMPI; Mark Partially Symbolic

Parser

Semantic Core

Symbolic Execution Engine with Implicit Spec

Formulas

KLEE

STP

SAT/UNSAT

Crashing Tests

Vijay Ganesh

Wednesday, 16 January, 13
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>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, Akhawee, 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>
<tbody>
<tr>
<td>Rex</td>
<td>Bjorner, Tillman, Vornkov et al. (Microsoft Research, Redmond)</td>
<td>• HAMPI + Length+Replace(s₁,s₂,s₃) - CFG • Translation to int. linear arith. (Z3)</td>
</tr>
<tr>
<td>Mona</td>
<td>Karlund et al. (U. of Aarhus)</td>
<td>• Can encode HAMPI &amp; Rex • User work • Automata-based • Non-elementary</td>
</tr>
<tr>
<td>DPRLE</td>
<td>Hooimeijer (U. of Virginia)</td>
<td>• Regular expression constraints</td>
</tr>
</tbody>
</table>
# 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)</td>
<td>Makanin result very difficult</td>
</tr>
<tr>
<td></td>
<td>Plandowski (1996, 2002/06)</td>
<td>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
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, CCS 2006, 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;, 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)