Industry tools rely on powerful verification engines.

- Boolean satisfiability (SAT) solvers.
- Binary decision diagrams (BDDs).

**Satisfiability Modulo Theories (SMT)**

- The next generation of verification engines.

- *SAT solvers + Theories*
  - Arithmetic
  - Arrays
  - Uninterpreted Functions

- Some problems are more naturally expressed in SMT.
- More automation.
\[ x + 2 = y \Rightarrow f(\text{read}(\text{write}(a, x, 3), y - 2)) = f(y - x + 1) \]
Example

\[ x + 2 = y \Rightarrow f\left(\text{read}(\text{write}(a, x, 3), y - 2)\right) = f(y - x + 1) \]

- Theory: *Arithmetic*
Example

\[ x + 2 = y \implies f(\text{read}(\text{write}(a, x, 3), y - 2)) = f(y - x + 1) \]

- Theory: Arrays
- Usually used to model the memory/heap.
- read: array access.
- write: array update.
Example

\[ x + 2 = y \Rightarrow f(\text{read}(\text{write}(a, x, 3), y - 2)) = f(y - x + 1) \]

- Theory: *Free functions.*
- Useful for abstracting complex operations.
Z3 is a new SMT solver developed at Microsoft Research.

Development/Research driven by internal customers.

Textual input & APIs (C/C++, .NET, OCaml).

Free for non-commercial use.

http://research.microsoft.com/projects/z3
SMT@Microsoft: Applications

- Test-case generation:
  
  *Pex, SAGE, and Vigilante.*

- Verifying Compiler:
  
  *Spec#/Boogie, HAVOC, and VCC.*

- Model Checking & Predicate Abstraction:
  
  *SLAM/SDV and Yogi.*

- Bounded Model Checking (BMC):
  
  *AsmL model checker.*

- Other: invariant generation, crypto, etc.
Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future
Test-case generation

- Test (correctness + usability) is 95% of the deal:
  - Dev/Test is 1-1 in products.
  - Developers are responsible for unit tests.
- Tools:
  - Annotations and static analysis (SAL, ESP)
  - File Fuzzing
  - Unit test case generation
Security is Critical

- Security bugs can be very expensive:
  - Cost due to worms (Slammer, CodeRed, Blaster, etc.): $Billions.
  - The real victim is the customer.

- Most security exploits are initiated via files or packets:
  - Ex: Internet Explorer parses dozens of files formats.

- Security testing: *hunting for million-dollar bugs*
  - Write A/V (always exploitable),
  - Read A/V (sometimes exploitable),
  - NULL-pointer dereference,
  - Division-by-zero (harder to exploit but still DOS attack), ...
Hunting for Security Bugs

- Two main techniques used by “black hats”:
  - Code inspection (of binaries).
  - *Black box fuzz testing.*

- **Black box** fuzz testing:
  - A form of black box random testing.
  - Randomly *fuzz* (=modify) a well formed input.
  - Grammar-based fuzzing: rules to encode how to fuzz.

- **Heavily** used in security testing
  - At MS: several internal tools.
  - Conceptually simple yet effective in practice
    
    *Has been instrumental in weeding out 1000 of bugs during development and test.*
Automatic Code-Driven Test Generation

Given program with a set of input parameters.
Generate inputs that maximize code coverage.
**Automatic Code-Driven Test Generation**

*Given* program with a set of input parameters.

*Generate* inputs that maximize code coverage.

**Example:**

Input \( x, y \)

\[ z = x + y \]

If \( z > x - y \) Then

Return \( z \)

Else

Error
Automatic Code-Driven Test Generation

**Given** program with a set of input parameters.

**Generate** inputs that maximize code coverage.

**Example:**

Input $x, y$

$z = x + y$

If $z > x - y$ Then

    Return $z$

Else

    Error

**Solve** $z = x + y \land z > x - y$
Automatic Code-Driven Test Generation

Given program with a set of input parameters.
Generate inputs that maximize code coverage.

Example:

Input $x, y$

$z = x + y$

If $z > x - y$ Then
    Return $z$

Else
    Error

Solve $z = x + y \land z > x - y$

$\implies x = 1, y = 1$
Automatic Code-Driven Test Generation

**Given** program with a set of input parameters.
**Generate** inputs that maximize code coverage.

**Example:**

Input \( x, y \)

\[ z = x + y \]

If \( z > x - y \) Then

    Return \( z \)

Else

    Error

**Solve** \[ z = x + y \land \neg (z > x - y) \]
**Automatic Code-Driven Test Generation**

**Given** program with a set of input parameters.

**Generate** inputs that maximize code coverage.

**Example:**

Input $x, y$

$z = x + y$

If $z > x - y$ Then

Return $z$

Else

Error

**Solve** $z = x + y \land \neg(z > x - y)$

$\implies x = 1, y = -1$
Method: Dynamic Test Generation

- Run program with random inputs.
- Collect constraints on inputs.
- Use SMT solver to generate new inputs.
- Combination with randomization: DART (Godefroid-Klarlund-Sen-05)
Method: Dynamic Test Generation

- *Run* program with *random* inputs.
- *Collect constraints* on inputs.
- *Use SMT solver* to generate new inputs.
- Combination with randomization: DART (Godefroid-Klarlund-Sen-05)

Repeat while finding new *execution paths*. 
DARTish projects at Microsoft

- **SAGE** (CSE) implements DART for x86 binaries and merges it with “fuzz” testing for finding security bugs.

- **PEX** (MSR-Redmond FSE Group) implements DART for .NET binaries in conjunction with “parameterized-unit tests” for unit testing of .NET programs.

- **YOGI** (MSR-India) implements DART to check the feasibility of program paths generated statically using a SLAM-like tool.

- **Vigilante** (MSR Cambridge) partially implements DART to dynamically generate worm filters.
**Initial Experiences with SAGE**

*25+ security bugs and counting.* (most missed by blackbox fuzzers)

- OS component X
  
  4 new bugs: “This was an area that we heavily fuzz tested in Vista”.

- OS component Y

  Arithmetic/stack overflow in y.dll

- Media format A

  Arithmetic overflow; DOS crash in previously patched component

- Media format B & C

  Hard-to-reproduce uninitialized-variable bug
Pex

- Pex monitors the execution of .NET application using the CLR profiling API.
- Pex dynamically checks for violations of programming rules, e.g. resource leaks.
- Pex suggests code snippets to the user, which will prevent the same failure from happening again.
- Very instrumental in exposing bugs in .NET libraries.
Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incremental: solve several similar formulas.
- “Small models”.
- Arithmetic $\times$ Machine Arithmetic.
Test-case generation & SMT

- Formulas are usually a big conjunction.
  - Pre-processing step.
  - Eliminate variables and simplify input formula.
  - *Significant performance impact.*

- Incremental: solve several similar formulas.

- “Small models”.

- Arithmetic × Machine Arithmetic.
Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incremental: solve several similar formulas.
  - New constraints can be asserted.
  - **push** and **pop**: (user) backtracking.
  - Reuse (some) lemmas.
- “Small models”.
- Arithmetic × Machine Arithmetic.
Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incremental: solve several similar formulas.
- “Small models”.
  - **Given** a set of constraints $C$, find a model $M$ that *minimizes* the value of the variables $x_0, \ldots, x_n$.
- Arithmetic $\times$ Machine Arithmetic.
Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incremental: solve several similar formulas.
- “Small models”.
  - **Given** a set of constraints $C$, find a model $M$ that minimizes the value of the variables $x_0, \ldots, x_n$.
  - **Eager (cheap) Solution:**
    - Assert $C$.
    - While satisfiable
      - Peek $x_i$ such that $M[x_i]$ is big
      - Assert $x_i < c$, where $c$ is a small constant
    - Return last found model
  - Arithmetic $\times$ Machine Arithmetic.
Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incremental: solve several similar formulas.
- “Small models”.

  - **Given** a set of constraints $C$, find a model $M$ that minimizes the value of the variables $x_0, \ldots, x_n$.

- **Refinement:**
  - Eager solution stops as soon as the context becomes unsatisfiable.
  - A “bad” choice (peek $x_i$) may prevent us from finding a good solution.
  - Use **push** and **pop** to retract “bad” choices.

- Arithmetic $\times$ Machine Arithmetic.
Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incremental: solve several similar formulas.
- “Small models”.
- Arithmetic $\times$ Machine Arithmetic.
  - Precision $\times$ Performance.
- SAGE has flags to abstract expensive operations.
Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future
A verifying compiler uses *automated reasoning to check the correctness* of a program that is compiles.

Correctness is specified by *types, assertions, … and other redundant annotations* that accompany the program.

Hoare 2004
Spec# Approach for a Verifying Compiler

- **Source Language**
  - C# + goodies = Spec#

- **Specifications**
  - method contracts,
  - invariants,
  - field and type annotations.

- **Program Logic**
  - Dijkstra’s weakest preconditions.

- **Automatic Verification**
  - type checking,
  - verification condition generation (VCG),
  - automatic theorem proving (SMT)
Spec# Approach for a Verifying Compiler

- Spec# (annotated C#) $\xrightarrow{}$ Boogie PL $\xrightarrow{}$ Formulas

- Example:

```csharp
class C {
    private int a, z;
    invariant z > 0
    public void M()
    {
        requires a != 0
        {
            z = 100/a;
        }
    }
}
```
Microsoft Hypervisor

- **Meta OS**: small layer of software between hardware and OS.
- **Mini**: 60K lines of non-trivial concurrent systems C code.
- **Critical**: must *guarantee isolation*.
- **Trusted**: a grand verification challenge.
Tool: A Verified C Compiler

- VCC translates an *annotated C program* into a *Boogie PL* program.
- Boogie generates verification conditions.
- A C-ish memory model
  - Abstract heaps
  - Bit-level precision
- The verification project has very recently started.
- It is a multi-man multi-year effort.
- More news coming soon.
Tool: HAVOC

- HAVOC also translates annotated C into Boogie PL.
- It allows the expression of richer properties about the program heap and data structures such as linked lists and arrays.
- Used to check NTFS-specific properties.
- Found 50 bugs, most confirmed.
  - 250 lines required to specify properties.
  - 600 lines of manual annotations.
  - 3000 lines of inferred annotations.
Verifying Compilers & SMT

- **Quantifiers, Quantifiers, ...**
  - Modeling the runtime.
  - Frame axioms (“what didn’t change”).
  - User provided assertions (e.g., the array is sorted).
  - Prototyping decision procedures (e.g., reachability, partial orders, ...).

- **Solver must be fast in satisfiable instances.**

- First-order logic is undecidable.

- Z3: pragmatic approach
  - **Heuristic Quantifier Instantiation.**
  - E-matching (i.e., matching modulo equalities).
E-matching is NP-hard.

The number of matches can be exponential.

In practice:

- Indexing techniques for fast retrieval: E-matching code trees.
- Incremental E-matching: Inverted path index.

It is not refutationally complete.
Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future
SLAM: device driver verification

- http://research.microsoft.com/slam/

- SLAM/SDV is a software model checker.

- Application domain: device drivers.

- Architecture

  - c2bp  C program $\rightarrow$ boolean program (predicate abstraction).
  - bebop  Model checker for boolean programs.
  - newton  Model refinement (check for path feasibility)

- SMT solvers are used to perform predicate abstraction and to check path feasibility.

- c2bp makes several calls to the SMT solver. The formulas are relatively small.
Predicate Abstraction: c2bp

- **Given** a C program $P$ and $F = \{p_1, \ldots, p_n\}$.
- **Produce** a boolean program $B(P, F)$
  - Same control flow structure as $P$.
  - Boolean variables $\{b_1, \ldots, b_n\}$ to match $\{p_1, \ldots, p_n\}$.
  - Properties true of $B(P, F)$ are true of $P$.
- **Example** $F = \{x > 0, x = y\}$. 
Abstracting Expressions via $F$

- $\text{Implies}_F(e)$
  - Best boolean function over $F$ that implies $e$

- $\text{ImpliedBy}_F(e)$
  - Best boolean function over $F$ that is implied by $e$
  - $\text{ImpliedBy}_F(e) = \neg \text{Implies}_F(\neg e)$
Computing $\text{Implies}_F(e)$

- minterm $m = l_1 \land \ldots \land l_n$, where $l_i = p_i$, or $l_i = \neg p_i$.

- $\text{Implies}_F(e)$ is the disjunction of all minterms that imply $e$.

- Naive approach
  - Generate all $2^n$ possible minterms.
  - For each minterm $m$, use SMT solver to check validity of $m \implies e$.

- Many possible optimizations.
Computing $\text{Implies}_F(e) : \text{Example}$

- $F = \{x < y, x = 2\}$
- $e : y > 1$
- Minterms over $P$
  - $x \geq y, x \neq 2$
  - $x < y, x \neq 2$
  - $x \geq y, x = 2$
  - $x < y, x = 2$
- $\text{Implies}_F(e) = \{x < y, x = 2\}$
Newton

- Given an error path \( \pi \) in the boolean program \( B \).
- Is \( \pi \) a feasible path of the corresponding C program?
  - Yes: found a bug.
  - No: find predicates that explain the infeasibility.
- Execute path symbolically.
- Check conditions for inconsistency using SMT solver.
Model Checking & SMT

- **All-SAT**
  
  Fast Predicate Abstraction.

- **Unsatisfiable Cores**
  
  Why the abstract path is not feasible?
Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future
Future work

- New theories:
  - Sets (HAVOC, VCC)
  - Partial orders (Spec#/Boogie)
  - Inductive data types (Pex)
  - Non linear arithmetic (Spec#/Boogie)

- Proofs (Yogi)

  Better support for quantifiers.
Better feedback when “potentially satisfiable”.

- Why is the “candidate model” not a model?
- Stream of “candidate models” (K. Claessen).

Decidable fragments:
- BSR class (no function symbols).
- Array property class (A. Bradley and Z. Manna).

Model finding by (unsound) reductions to decidable fragments.
Conclusion

- *SMT is hot at Microsoft.*
- Z3 is a new SMT solver.
- Main applications:
  - Test-case generation.
  - Verifying compiler.
  - Model Checking & Predicate Abstraction.