Constraint Solving Challenges in Dynamic Symbolic Execution

Cristian Cadar
Department of Computing
Imperial College London

Joint work with Dawson Engler, Daniel Dunbar
Peter Collingbourne, Paul Kelly, Vijay Ganesh, David Dill, Junfeng Yang
Writing Correct Software Is Hard

• Software complexity
  – Massive amounts of code
  – Tricky control flow
  – Complex dependencies
  – Abusive use of pointer operations
  – Intensive interaction w/ environment
    • E.g., data from OS, network, etc.

• Current testing approaches are insufficient
  – Most projects still use only manual (expensive) and/or random testing (often ineffective)
Dynamic Symbolic Execution

- Let code to generate its own (complex) test cases!

- Automatically generated high coverage test suites
  - Over 90% on average on ~160 user-level apps

- Found bugs and security vulnerabilities in complex software
  - Including file systems, device drivers, computer vision code, utilities, network servers, packet filters
int bad_abs(int x) {
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}
All-Value Checks

Implicit checks before each dangerous operation

- Null-pointer dereferences
- Buffer overflows
- Division/modulo by zero
- Assert violations

All-value checks!

- Errors are found if any buggy values exist on that path!

```c
int foo(unsigned k) {
    int a[4] = {3, 1, 0, 4};
    k = k % 4;
    return a[a[k]];
}
```
All-Value Checks

Implicit checks before each dangerous operation
- Null-pointer dereferences
- Buffer overflows
- Division/modulo by zero
- Asserts violations

All-value checks!
- Errors are found if any buggy values exist on that path!

```c
int foo(unsigned k) {
    int a[4] = {3, 1, 0, 4};
    k = k % 4;
    return a[a[k]];
}
```

Buffer overflow!
Dynamic (vs. Static) SymEx

• Each path explored separately as in regular testing
  – EXE uses `fork()` system call to fork execution!

• Mixed concrete/symbolic execution
  – All operations that do not depend on the symbolic inputs are (essentially) executed as in the original code!
    • E.g., `malloc(5)` allocates object on the heap in EXE
Dynamic (vs. Static) SymEx

**Advantages:**
- Ability to interact with the outside environment
  - System calls, uninstrumented libraries
- Only relevant code executed symbolically
  - Without the need to extract it explicitly

**...and disadvantages:**
- Can only explore a finite number of paths!
  - Important to prioritize most “interesting” ones
Three tools: EGT, EXE, KLEE

C code

Constraint Solver (STP)

x ≥ 0
x ≠ 1234
x = 3

EGT/EXE/KLEE

x = -2
x = 1234
x = 3
Scalability Challenges

Path exploration challenges

Constraint solving challenges
1. **Accuracy:** need constraint solver that allows bit-level modeling of memory:
   - Systems code often observes the same bytes in different ways: e.g., using pointer casting to treat an array of chars as a network packet, inode, etc.
   - Bugs in systems code are often triggered by corner cases such as arithmetic overflows

2. **Performance:** real programs generate expensive constraints
STP

• Modern constraint solver, based on *eager* translation to SAT (uses MiniSAT)
• Developed at Stanford by Ganesh and Dill, initially targeted to (and driven by) EXE

• Two data types: **bitvectors** and **arrays of bitvectors**
• We model each memory block as an array of bitvectors
• We can translate all C expressions into STP constraints with bit-level accuracy
  – Main exception: floating-point
Constraint solving optimizations essential:

• STP optimizations
• Higher-level optimizations
Reasoning about Arrays in STP

- Many programs generate large constraints involving arrays with symbolic indexes

- STP handles this via *array-based refinement*
Reasoning about Arrays in STP

STP’s conversion of array terms to SAT is expensive

\[(a[i_1] = e_1) \land (a[i_2] = e_2) \land (a[i_3] = e_3) \land (i_1+i_2+i_3=6)\]

\[(v_1 = e_1) \land (v_2 = e_2) \land (v_3 = e_3) \land (i_1+i_2+i_3=6)\]

\[(i_1 = i_2 \implies v_1 = v_2) \land (i_1 = i_3 \implies v_1 = v_3) \land (i_2 = i_3 \implies v_2 = v_3)\]

Expands each formula by \(n\cdot(n-1)/2\) terms, where \(n\) is the number of syntactically distinct indexes
Array-based Refinement in STP

STP’s conversion of array terms to SAT is expensive

\[(a[i_1] = e_1) \land (a[i_2] = e_2) \land (a[i_3] = e_3) \land (i_1+i_2+i_3=6)\]

\[(v_1 = e_1) \land (v_2 = e_2) \land (v_3 = e_3) \land (i_1+i_2+i_3=6)\]

\[(i_1 = i_2 \rightarrow v_1 = v_2) \land (i_1 = i_3 \rightarrow v_1 = v_3) \land (i_2 = i_3 \rightarrow v_2 = v_3)\]

Under-approximation

UNSATISFIABLE

Original formula

UNSATISFIABLE
Array-based Refinement in STP

STP’s conversion of array terms to SAT is expensive

\[(a[i_1] = e_1) \land (a[i_2] = e_2) \land (a[i_3] = e_3) \land (i_1 + i_2 + i_3 = 6)\]

\[(v_1 = e_1) \land (v_2 = e_2) \land (v_3 = e_3) \land (i_1 + i_2 + i_3 = 6)\]

\[(i_1 = i_2 \rightarrow v_1 = v_2) \land (i_1 = i_3 \rightarrow v_1 = v_3) \land (i_2 = i_3 \rightarrow v_2 = v_3)\]
Array-based Refinement in STP

STP’s conversion of array terms to SAT is expensive

\[(a[i_1] = e_1) \land (a[i_2] = e_2) \land (a[i_3] = e_3) \land (i_1+i_2+i_3=6)\]

\[(v_1 = e_1) \land (v_2 = e_2) \land (v_3 = e_3) \land (i_1+i_2+i_3=6)\]

\[(i_1 = i_2 \rightarrow v_1 = v_2) \land (i_1 = i_3 \rightarrow v_1 = v_3) \land (i_2 = i_3 \rightarrow v_2 = v_3)\]

\[
\begin{align*}
  i_1 &= 2 \\
i_2 &= 2 \\
i_3 &= 2 \\
v_1 &= e_1 = 1 \\
v_2 &= e_2 = 2 \\
v_3 &= e_3 = 3 \\
\end{align*}
\]

\[
\begin{align*}
  (a[2] = 1) \land (a[2] = 2) \land (a[2] = 3) \land (2+2+2 = 6)
\end{align*}
\]
Array-based Refinement in STP

STP’s conversion of array terms to SAT is expensive

\[(a[i_1] = e_1) \land (a[i_2] = e_2) \land (a[i_3] = e_3) \land (i_1+i_2+i_3=6)\]

\[(v_1 = e_1) \land (v_2 = e_2) \land (v_3 = e_3) \land (i_1+i_2+i_3=6)\]

\[(i_1 = i_2 \implies v_1 = v_2) \land (i_1 = i_3 \implies v_1 = v_3) \land (i_2 = i_3 \implies v_2 = v_3)\]

\[
\begin{align*}
i_1 &= 2 \\
i_2 &= 2 \\
i_3 &= 2 \\
v_1 &= e_1 = 1 \\
v_2 &= e_2 = 2 \\
v_3 &= e_3 = 3
\end{align*}
\]

\[
\begin{align*}
(a[2] &= 1) \land (a[2] = 2) \land \bar{a[2]} \land
\end{align*}
\]

\[
\begin{align*}
(a[2] &= 3) \land (2+2+2 = 6)
\end{align*}
\]
## Evaluation

<table>
<thead>
<tr>
<th>Solver</th>
<th>Total time (min)</th>
<th>Timeouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>STP (baseline)</td>
<td>56</td>
<td>36</td>
</tr>
<tr>
<td>STP (array-based refinement)</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

8495 test cases from our symbolic execution benchmarks

- Timeout set at 60s (which are added as penalty), underestimates performance differences
Higher-Level Constraint Solving Optimizations

- Two simple and effective optimizations
  - Eliminating irrelevant constraints
  - Caching solutions
    - Dramatic speedup on our benchmarks
Eliminating Irrelevant Constraints

- In practice, each branch usually depends on a small number of variables

```plaintext
...                           x + y > 10
...                           z & -z = z
if (x < 10) {
    ...                         x < 10 ?
}                             }
```
Caching Solutions

- Static set of branches: lots of similar constraint sets

```
2 * y < 100
x > 3
x + y > 10
```

```
2 * y < 100
x + y > 10
```

```
2 * y < 100
x > 3
x + y > 10
x < 10
```

Eliminating constraints cannot invalidate solution

Adding constraints often does not invalidate solution

x = 5
y = 15
Dramatic Speedup

Aggregated data over 73 applications

- Base
- Irrelevant Constraint Elimination
- Caching
- Irrelevant Constraint Elimination + Caching
Statically Merging Paths

Default behaviour

\[
\begin{align*}
\text{if } (a > b) \\
& \text{max} = a; \\
\text{else } & \text{max} = b;
\end{align*}
\]

Phi-Node Folding (when no side effects)

\[
\text{if } (a > b) \\
& \text{max} = a; \\
\text{else } & \text{max} = b;
\]
Statically Merging Paths

```plaintext
for (i=0; i < N; i++) {
    if (a[i] > b[i])
        max[i] = a[i];
    else max[i] = b[i];
}
```

- Default: $2^N$ paths
- Phi-node folding: 1 path

**morph** computer vision algorithm: $2^{256} \rightarrow 1$

Path merging $\equiv$ Outsourcing problem to constraint solver

(especially problematic for solvers optimized for conjunctions of constraints)
Evaluation

• Motivation and Overview
• Example and Basic Architecture
• Constraint Solving Challenges

• Evaluation
  – Coverage results
  – Bug finding
  – Crosschecking
  – Attack generation
GNU Coreutils Suite

- Core user-level apps installed on many UNIX systems
- 89 stand-alone (i.e. excluding wrappers) apps (v6.10)
  - File system management: `ls`, `mkdir`, `chmod`, etc.
  - Management of system properties: `hostname`, `printenv`, etc.
  - Text file processing: `sort`, `wc`, `od`, etc.
  - …

Variety of functions, different authors, intensive interaction with environment

Heavily tested, mature code
Coreutils ELOC (incl. called lib)
Methodology

• Fully automatic runs
• Run KLEE one hour per utility, generate test cases
• Run test cases on *uninstrumented* version of utility
• Measure line coverage using \texttt{gcov}
  – Coverage measurements not inflated by potential bugs in our tool
High Line Coverage
(Coreutils, non-lib, 1h/utility = 89 h)

Overall: 84%, Average 91%, Median 95%

Coverage (ELOC %)

Apps sorted by KLEE coverage
Beats 15 Years of Manual Testing

<table>
<thead>
<tr>
<th>Avg/utility</th>
<th>KLEE</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>91%</td>
<td>68%</td>
</tr>
</tbody>
</table>

Manual tests also check correctness
Evaluation

• Motivation and Overview
• Example and Basic Architecture
• Constraint Solving Challenges
• Evaluation
  – Coverage results
  – Bug finding
  – Crosschecking
  – Attack generation
## Bug Finding Summary

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UNIX file systems</strong></td>
</tr>
<tr>
<td><strong>UNIX utilities</strong></td>
</tr>
<tr>
<td><strong>MINIX device drivers</strong></td>
</tr>
<tr>
<td><strong>Library code</strong></td>
</tr>
<tr>
<td><strong>Packet filters</strong></td>
</tr>
<tr>
<td><strong>Networking servers</strong></td>
</tr>
<tr>
<td><strong>Operating Systems</strong></td>
</tr>
<tr>
<td><strong>Computer vision code</strong></td>
</tr>
</tbody>
</table>

- Most bugs fixed promptly
GNU Coreutils Bugs

• Ten crash bugs
  – More crash bugs than approx previous three years combined
  – KLEE generates actual command lines exposing crashes
# Ten command lines of death

<table>
<thead>
<tr>
<th>Command 1</th>
<th>Command 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>md5sum -c t1.txt</td>
<td>pr -e t2.txt</td>
</tr>
<tr>
<td>mkdir -Z a b</td>
<td>tac -r t3.txt t3.txt</td>
</tr>
<tr>
<td>mkfifo -Z a b</td>
<td>paste -d\ abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>mknod -Z a b p</td>
<td>ptx -F\ abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>seq -f %0 1</td>
<td>ptx x t4.txt</td>
</tr>
</tbody>
</table>

**t1.txt:** \t \tMD5(

**t2.txt:** \bb\bb\bb\bb\bb|t

**t3.txt:** \n
**t4.txt:** A
Experimental Evaluation

- Motivation and Overview
- Example and Basic Architecture
- Constraint Solving Challenges
- Results
  - Coverage results
  - Bug finding
  - Crosschecking
  - Attack generation
Assume $f(x)$ and $f'(x)$ implement the same interface

1. Make input $x$ symbolic
2. Run tool on `assert(f(x) == f'(x))`
3. Find mismatches!
What to Crosscheck?

Lots of available opportunities

- Different implementations of the same functionality
  - e.g., libraries, servers, compilers
- Optimized versions of reference implementations
- Refactorings
- Reverse computation
  - e.g., compress/uncompress
Coreutils vs. Busybox

UNIX utilities should conform to *IEEE Std. 1003.1*

- Crosschecked pairs of Coreutils and Busybox utilities
  - Busybox: implementation for embedded devices
- Found lots of mismatches
## Mismatches Found

<table>
<thead>
<tr>
<th>Input</th>
<th>Busybox</th>
<th>Coreutils</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>comm t1.txt t2.txt</code></td>
<td>[doesn't show diff]</td>
<td>[shows diff]</td>
</tr>
<tr>
<td><code>tee -</code></td>
<td>[copies once to stdout]</td>
<td>[copies twice]</td>
</tr>
<tr>
<td><code>tee &quot;&quot; &lt;t1.txt</code></td>
<td>[infinite loop]</td>
<td>[terminates]</td>
</tr>
<tr>
<td><code>cksum /</code></td>
<td>&quot;4294967295 0 /&quot;</td>
<td>&quot;/: Is a directory&quot;</td>
</tr>
<tr>
<td><code>split /</code></td>
<td>&quot;/: Is a directory&quot;</td>
<td></td>
</tr>
<tr>
<td><code>tr</code></td>
<td>[duplicates input]</td>
<td>&quot;missing operand&quot;</td>
</tr>
<tr>
<td><code>[ 0 &quot;&lt;&quot; 1 ]</code></td>
<td></td>
<td>&quot;binary op. expected&quot;</td>
</tr>
<tr>
<td><code>tail -2l</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td><code>unexpand -f</code></td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
<tr>
<td><code>split -</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
</tbody>
</table>

```
t1.txt: a  t2.txt: b  (no newlines!)
```
SSE Optimizations in Computer Vision Algorithms

- **Computer vision algorithms** often optimized to use SSE instructions
  - Operate on multiple data concurrently
  - Provide significant speedup
- Translation to SSE is usually done manually
  - Starting from a reference scalar implementation

\[ \begin{align*}
\text{SSE-optimized computer vision algorithm} & \quad \text{crosscheck} \quad \text{Original computer vision algorithm}
\end{align*} \]
Computer Vision Algorithms and Floating Point Operations

- Computer vision algorithms make intensive use of floating-point
- No constraint solvers for floating-point available (IEEE 754 standard not pretty!)
  - Recent development: FP internal support in CMBC
  - Any other solvers that we can try?
Computer Vision Algorithms and Floating Point Operations

• To ensure equality, the optimized SSE version needs to build FP values in roughly the same way
  • Observed developers try to mimic the scalar code using SSE
• Usually can cheaply prove/disprove equivalence via

  $\frac{44}{55}$

  \begin{align*}
  \text{expression} & \quad + \quad \text{syntactical} \\
  \text{canonicalization} & \quad \text{expression matching}
  \end{align*}
SSE Optimizations in OpenCV

**OpenCV**: popular open-source computer vision library from Intel and Willow Garage

Corner detection algorithm

[from wikipedia.org]
OpenCV Results

• Crosschecked 51 SSE/scalar pairs
  • Proved the bounded equivalence of 41
  • Found mismatches in 10
• Most mismatches due to tricky FP-related issues:
  • Precision
  • Rounding
  • Associativity
  • Distributivity
  • NaN values
Example Source of Mismatches

min/max not commutative nor associative!

\[
\text{min}(a,b) = a < b \ ? \ a : b
\]

\(a < b\) (ordered) \(\rightarrow\) always returns false if one of the operands is NaN

\[
\begin{align*}
\text{min}(\text{NaN}, 5) &= 5 \\
\text{min}(5, \text{NaN}) &= \text{NaN}
\end{align*}
\]

Could lead to arbitrarily large differences:

\[
\begin{align*}
\text{min}(\text{min}(5, \text{NaN}), 100) &= \text{min}(\text{NaN}, 100) = 100 \\
\text{min}(5, \text{min}(\text{NaN}, 100)) &= \text{min}(5, 100) = 5
\end{align*}
\]
Experimental Evaluation

- Motivation and Overview
- Example and Basic Architecture
- Constraint Solving Challenges
- Evaluation
  - Coverage results
  - Bug finding
  - Crosschecking
  - Attack generation
Allow untrusted users to mount regular files as disk images!

Trend in modern operating systems:

Attack Generation – File Systems
Attack Generation – File Systems

- Mount code is executed by the kernel!
- Attackers may create malicious disk images to attack a system
Attack Generation – File Systems

mount( )

ext2 / ext3 / JFS

EXE

01010110
11010100

1011001
01011100

01010111
00110101

...
## Disk of death (JFS, Linux 2.6.10)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hex Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>08000</td>
<td>464A 3135 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08010</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08020</td>
<td>0000 0000 0100 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08030</td>
<td>E004 000F 0000 0000 0002 0000 0000 0000</td>
</tr>
<tr>
<td>08040</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10000</td>
<td></td>
</tr>
</tbody>
</table>

64\textsuperscript{th} sector of a 64K file. Mount. And **PANIC** your kernel!
Our techniques and tools can effectively:

- Generate high coverage test suites
  - Over 90% on average on Coreutils and Busybox utilities
- Generate inputs exposing bugs and security vulnerabilities in complex software
  - Including file systems, device drivers, library code, utility applications, network tools, packet filters
- Find semantic bugs via crosschecking
  - Crosschecked Coreutils and Busybox utilities, checked correctness of SSE optimizations
Symbolic Execution: Related Work

Symbolic execution for program testing introduced in the 1970s:

- James C. King. A new approach to program testing
  International Conference on Reliable Software, April 1975

Dynamic symbolic execution for automatic test case generation:

- EGT paper: [Cadar and Engler 2005]
- Independent work at Bell Labs on DART [Godefroid, Klarlund, Sen 2005]

Very active area of research, e.g:

- SAGE, Pex @ Microsoft Research
- JPF-SE, Symbolic JPF @ NASA Ames
- CREST @ UC Berkeley
- S2E, Cloud9, Oasis @ EPFL
- BitBlaze, WebBlaze @ UC Berkeley
KLEE: Available as Open-Source

http://klee.llvm.org

Already used and extended in many interesting ways by several research groups, in the areas of:

- wireless sensor networks
- schedule memoization in multithreaded code
- automated debugging
- exploit generation
- online gaming, etc.