Improving Software Reliability and Security via Symbolic Execution

Cristian Cadar
Department of Computing
Imperial College London

Joint work with Paul Marinescu, Peter Collingbourne, Paul Kelly, JaeSeung Song, Peter Pietzuch, Hristina Palikareva (Imperial)

Dawson Engler, Daniel Dunbar, Junfeng Yang, Peter Pawlowski, Can Sar, Paul Twohey, Vijay Ganesh, David Dill, Peter Boonstoppel (Stanford)
Writing Correct Software is Hard!

- Software complexity
  - Massive amounts of code
  - Tricky control flow
  - Complex dependencies
  - Intensive interaction w/ environment

- Code has to anticipate all possible interactions
  - Including malicious ones!
Dynamic Symbolic Execution

- Dynamic symbolic execution can automatically explore multiple paths through a program
  - Using a constraint solver to determine the feasibility of each explored path
- Before each dangerous operation, can check if there are any values that can cause an error
- For each path, can usually generate a concrete input triggering the path

Provides an automated way to reason about program behavior and the interaction with users and environment
Dynamic Symbolic Execution

- **Toy example**
- **Scalability challenges**
  - Path explosion, constraint solving challenges
- **Generic bug finding**
  - User-level utilities, kernel code, drivers, computer vision code, etc.
  - Attack generation against file systems and network servers
- **Symbolic race detection for GPGPU code**
- **Semantic errors via crosschecking**
  - Server interoperability, SIMD optimizations, GPU optimizations
- **Patch testing**
  - Testing six years of patches
Toy Example

```c
int bad_abs(int x) {
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}
```

Each path is explored separately!
Implicit checks before each dangerous operation
- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

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
- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

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!
Mixed Concrete/Symbolic Execution

All operations that do not depend on the symbolic inputs are (essentially) executed as in the original code

**Advantages:**

– Ability to interact with the outside environment
  * E.g., system calls, uninstrumented libraries
– Can partly deal with limitations of constraint solvers
  * E.g., unsupported theories
– Only relevant code executed symbolically
  * Without the need to extract it explicitly
Three tools: EGT, EXE, KLEE

C code

Constraint Solver (STP)

x ≥ 0
x ≠ 1234
x = 3

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

Path exploration challenges

Constraint solving challenges
Naïve exploration can easily get “stuck”

- Employing search heuristics
- Dynamically eliminating redundant paths
- Statically merging paths
- Using existing regression test suites to prioritize execution
- etc.
Search Heuristics

- Coverage-optimized search
  - Select path closest to an uncovered instruction
  - Favor paths that recently hit new code
- Best-first search
- Random path search
- etc.

[CCS'06, OSDI'08, ICSE'11, etc.]
Random Path Selection

- Maintain a binary tree of active paths
- Subtrees have equal prob. of being selected, irresp. of size
- NOT random state selection
- Favors paths high in the tree
  - fewer constraints
- Avoid starvation
  - e.g. symbolic loop
Our latest tool KLEE uses multiple heuristics in a round-robin fashion, to protect against individual heuristics getting stuck in a local maximum.
Eliminating Redundant Paths

- If two paths reach the same program point with the same constraint sets, we can prune one of them.
- We can discard from the constraint sets of each path those constraints involving memory which is never read again.
data, arg1, arg2 = *

flag = 0;

if (arg1 > 100)
    flag = 1;

if (arg2 > 100)
    flag = 1;

process(data, flag);
Many Redundant Paths

PCI driver (MINIX) - 1h runs
- Blue: Base
- Red: Redundant path elimination

Non-redundant explored states

Generated tests
Lots of Redundant Paths

- **bpf**
- **expat**
- **pcre**
- **tcpdump**

- **udhcpd**
- **sb16**
- **lance**
Redundant Path Elimination

PCI driver (MINIX) - 1h runs

- Blue line: Base
- Red line: Redundant path elimination

Branch coverage (%) vs. Generated tests

Coverage increases with generated tests.
Statically Merging Paths

Default behaviour

if (a > b)
    max = a;
else max = b;

Phi-Node Folding (when no side effects)

if (a > b)
    max = a;
else max = b;

max = select(a>b, a, b)

[EuroSys'11]
Statically Merging Paths

```
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

(which are often optimized for conjunctions of constraints)
Using Existing Regression Suites

- Most applications come with a manually-written regression test suite

$ cd lighttpd-1.4.29
$ make check
...
./cachable.t ............ ok
./core-404-handler.t .. ok
./core-condition.t .... ok
./core-keepalive.t .... ok
./core-request.t ...... ok
./core-response.t ..... ok
./core-var-include.t .. ok
./core.t .............. ok
./lowercase.t ........... ok
./mod-access.t ........ ok
...
## Regression Suites

<table>
<thead>
<tr>
<th>PROS</th>
<th>CONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Designed to execute interesting program paths</td>
<td>• Execute each path with a single set of inputs</td>
</tr>
<tr>
<td>• Often achieve good coverage of different program features</td>
<td>• Often exercise the general case of a program feature, missing corner cases</td>
</tr>
</tbody>
</table>
1. Use the paths executed by the regression suite to bootstrap the exploration process (to benefit from the coverage of the manual test suite and find additional errors on those paths)

2. Incrementally explore paths around the dangerous operations on these paths, in increasing distance from the dangerous operations (to test all possible corner cases of the program features exercised by the test suite)
Multipath Analysis

Bounded symbolic execution

main(argv, argc)

exit(0)

✓

• dangerous operations
• divergence point

Bounded symbolic execution
Scalability Challenges

Path exploration challenges

Constraint solving challenges
Constraint Solving Challenges

1. **Accuracy**: need 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 related to pointer/integer casting and arithmetic overflows

2. **Performance**: real programs generate many expensive constraints
STP Constraint Solver [Ganesh, Dill]

- 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 (BVs)** and **arrays of BVs**
- We model each memory block as an array of 8-bit BVs
- We can translate all C expressions into STP constraints with bit-level accuracy
  - Main exception: floating-point
Constraint Solving: Accuracy

- Mirror the (lack of) type system in C
  - Model each memory block as an array of 8-bit BVs
  - Bind types to expressions, not bits

```c
char buf[N]; // symbolic
struct pkt1 { char x, y, v, w; int z; } *pa = (struct pkt1*) buf;
struct pkt2 { unsigned i, j; } *pb = (struct pkt2*) buf;
if (pa[2].v < 0) { assert(pb[2].i >= 1<<23); }
```

<table>
<thead>
<tr>
<th>buf: ARRAY BITVECTOR(32)OF BITVECTOR(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBVLT(buf[18], 0x00)</td>
</tr>
<tr>
<td>BVGE(buf[19]@buf[18]@buf[17]@buf[16], 0x00800000)</td>
</tr>
</tbody>
</table>
Constraint Solving: Performance

• Inherently expensive (NP-complete)
• Invoked at every branch

• Key insight: exploit the characteristics of constraints generated by symex
Some Constraint Solving Statistics
[after optimizations]

<table>
<thead>
<tr>
<th>Application</th>
<th>Instrs/s</th>
<th>Queries/s</th>
<th>Solver %</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>695</td>
<td>7.9</td>
<td>97.8</td>
</tr>
<tr>
<td>base64</td>
<td>20,520</td>
<td>42.2</td>
<td>97.0</td>
</tr>
<tr>
<td>chmod</td>
<td>5,360</td>
<td>12.6</td>
<td>97.2</td>
</tr>
<tr>
<td>comm</td>
<td>222,113</td>
<td>305.0</td>
<td>88.4</td>
</tr>
<tr>
<td>csplit</td>
<td>19,132</td>
<td>63.5</td>
<td>98.3</td>
</tr>
<tr>
<td>dircolors</td>
<td>1,019,795</td>
<td>4,251.7</td>
<td>98.6</td>
</tr>
<tr>
<td>echo</td>
<td>52</td>
<td>4.5</td>
<td>98.8</td>
</tr>
<tr>
<td>env</td>
<td>13,246</td>
<td>26.3</td>
<td>97.2</td>
</tr>
<tr>
<td>factor</td>
<td>12,119</td>
<td>22.6</td>
<td>99.7</td>
</tr>
<tr>
<td>join</td>
<td>1,033,022</td>
<td>3,401.2</td>
<td>98.1</td>
</tr>
<tr>
<td>ln</td>
<td>2,986</td>
<td>24.5</td>
<td>97.0</td>
</tr>
<tr>
<td>mkdir</td>
<td>3,895</td>
<td>7.2</td>
<td>96.6</td>
</tr>
<tr>
<td><strong>Avg:</strong></td>
<td><strong>196,078</strong></td>
<td><strong>675.5</strong></td>
<td><strong>97.1</strong></td>
</tr>
</tbody>
</table>

1h runs using KLEE with DFS and no caching

UNIX utilities (and many other benchmarks)
- Large number of queries
- Most queries <0.1s
- Most time spent in the solver (before and after optimizations!)
Constraint Solving Optimizations

Implemented at several different levels:

• SAT solvers
• SMT solvers
• Symbolic execution tools
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

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

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

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

Simplified formula

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

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

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

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

\[i_1 = 2\]
\[i_2 = 2\]
\[i_3 = 2\]
\[v_1 = e_1 = 1\]
\[v_2 = e_2 = 2\]
\[v_3 = e_3 = 3\]
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
Eliminating Irrelevant Constraints

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

\[ w + z > 100 \]
\[ 2 \times w \times 1 < 12345 \]
\[ x + y > 10 \]
\[ z \& z = z \]

if \((x < 10)\) {
  ...
}

\[ x < 10 \]
Caching Solutions

- Static set of branches: lots of similar constraint sets

\[
\begin{align*}
2 \times y &< 100 \\
x &> 3 \\
x + y &> 10
\end{align*}
\]

\[
\begin{align*}
x = 5 \\
y = 15
\end{align*}
\]

\[
\begin{align*}
2 \times y &< 100 \\
x + y &> 10
\end{align*}
\]

Eliminating constraints cannot invalidate solution

\[
\begin{align*}
x = 5 \\
y = 15
\end{align*}
\]

\[
\begin{align*}
2 \times y &< 100 \\
x &> 3 \\
x + y &> 10 \\
x &< 10
\end{align*}
\]

Adding constraints often does not invalidate solution

\[
\begin{align*}
x = 5 \\
y = 15
\end{align*}
\]
Speedup

**Aggregated data over 73 applications**

- Base
- Irrelevant Constraint Elimination
- Caching
- Irrelevant Constraint Elimination + Caching
More on Caching: Instrs/Sec

<table>
<thead>
<tr>
<th>Application</th>
<th>No caching</th>
<th>Caching</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>3,914</td>
<td>695</td>
<td>0.17</td>
</tr>
<tr>
<td>base64</td>
<td>18,840</td>
<td>20,520</td>
<td>1.08</td>
</tr>
<tr>
<td>chmod</td>
<td>12,060</td>
<td>5,360</td>
<td>0.44</td>
</tr>
<tr>
<td>comm</td>
<td>73,064</td>
<td>222,113</td>
<td>3.03</td>
</tr>
<tr>
<td>csplit</td>
<td>10,682</td>
<td>19,132</td>
<td>1.79</td>
</tr>
<tr>
<td>dircolors</td>
<td>8,090</td>
<td>1,019,795</td>
<td>126.05</td>
</tr>
<tr>
<td>echo</td>
<td>227</td>
<td>52</td>
<td>0.22</td>
</tr>
<tr>
<td>env</td>
<td>21,995</td>
<td>13,246</td>
<td>0.60</td>
</tr>
<tr>
<td>factor</td>
<td>1,897</td>
<td>12,119</td>
<td>6.38</td>
</tr>
<tr>
<td>join</td>
<td>12,649</td>
<td>1,033,022</td>
<td>81.66</td>
</tr>
<tr>
<td>ln</td>
<td>13,420</td>
<td>2,986</td>
<td>0.22</td>
</tr>
<tr>
<td>mkdir</td>
<td>25,331</td>
<td>3,895</td>
<td>0.15</td>
</tr>
<tr>
<td><strong>Avg:</strong></td>
<td><strong>16,847</strong></td>
<td><strong>196,078</strong></td>
<td><strong>11.63x</strong></td>
</tr>
</tbody>
</table>

- Instrs/sec on ~1h runs, using DFS, w/ and w/o caching
- Need for better, more adaptive caching algorithms!

[CAV'13]
Portfolio of SMT Solvers

- C code
- KLEE
- metaSMT
- x = 3
- x = -2
- x = 1234
- x = 3
- C code constraints: $x \geq 0$, $x \neq 1234$

SMT Solvers:
- Boolector
- STP
- Z3

[CAV'13]
EGT, EXE, KLEE

Successfully used our tools to:

• Automatically generate high-coverage test suites

• Find bugs and security vulnerabilities in complex software

• Perform bounded verification
Bug Finding with EGT, EXE, KLEE: Focus on Systems and Security Critical Code

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX utilities</td>
</tr>
<tr>
<td>UNIX file systems</td>
</tr>
<tr>
<td>Network servers</td>
</tr>
<tr>
<td>Library code</td>
</tr>
<tr>
<td>Packet filters</td>
</tr>
<tr>
<td>MINIX device drivers</td>
</tr>
<tr>
<td>Kernel code</td>
</tr>
<tr>
<td>Computer vision code</td>
</tr>
<tr>
<td>OpenCL code</td>
</tr>
</tbody>
</table>

• Most bugs fixed promptly
### Coreutils Commands of Death

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>md5sum -c t1.txt</code></td>
<td><code>pr -e t2.txt</code></td>
</tr>
<tr>
<td><code>mkdir -Z a b</code></td>
<td><code>tac -r t3.txt t3.txt</code></td>
</tr>
<tr>
<td><code>mkfifo -Z a b</code></td>
<td><code>paste -d|abcdefgijklmnopqrstuvwxyz</code></td>
</tr>
<tr>
<td><code>mknod -Z a b p</code></td>
<td><code>ptx -F|abcdefgijklmnopqrstuvwxyz</code></td>
</tr>
<tr>
<td><code>seq -f %0 1</code></td>
<td><code>ptx x t4.txt</code></td>
</tr>
<tr>
<td><code>printf %d </code></td>
<td><code>cut -c3-5,8000000- --output-d: file</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t1.txt:</th>
<th>t3.txt:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t \tMD5(</td>
<td>\n</td>
</tr>
<tr>
<td>t2.txt: \b\b\b\b\b\b\b\b\b\t</td>
<td>t4.txt: A</td>
</tr>
</tbody>
</table>

[OSDI 2008, ICSE 2012]
Some modern operating systems allow untrusted users to mount regular files as disk images!
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

[Oakland 2006]
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 . .</td>
</tr>
</tbody>
</table>

- 64th sector of a 64K disk image
- Mount it and PANIC your kernel
Attack Generation: Network Servers

Network

\[ = * \]

Network Server

recv()
Bonjour: Packet of Death

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hex Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0010</td>
<td>003E 0000 4000 FF11 1BB2 7F00 0001 E000</td>
</tr>
<tr>
<td>0020</td>
<td>00FB 0000 14E9 002A 0000 0000 0000 0000 0001</td>
</tr>
<tr>
<td>0030</td>
<td>0000 0000 0000 055F 6461 6170 045F 7463</td>
</tr>
<tr>
<td>0040</td>
<td>7005 6C6F 6361 6C00 000C 0001</td>
</tr>
</tbody>
</table>

- Causes Bonjour to abort, potential DoS attack
- Confirmed by Apple, security update released
Semantic Errors via Crosschecking (Equivalence Checking)

Lots of available opportunities as code is:

- Optimized frequently
- Refactored frequently
- Different implementations of the same interface

We can find any mismatches in their behavior by:

1. Using symbolic execution to explore multiple paths
2. Comparing the path constraints or input/output pairs across implementations
Crosschecking: Advantages

• Can find semantic errors without the need for specifications!

• Constraint solving queries can be solved faster

• Can support constraint types not (efficiently) handled by the underlying solver, e.g., floating-point

Many crosschecking queries can be *syntactically* proven to be equivalent
Crosschecking: Advantages

Many crosschecking queries can be *syntactically* proven to be equivalent
ZeroConf Protocol

- Enables devices to automatically configure themselves and their services and be discovered without manual intervention
- Two popular implementations: Avahi (open-source), and Bonjour (open-sourced by Apple)

[ICCCN 2011]
Server Interoperability
Bonjour vs. Avahi

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hex Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0010</td>
<td>003E 0000 4000 FF11 1BB2 7F00 0001 E000</td>
</tr>
<tr>
<td>0020</td>
<td>00FB 0000 14E9 002A 0000 0000 0002 0001</td>
</tr>
<tr>
<td>0030</td>
<td>0000 0000 0000 055F 6461 6170 045F 7463</td>
</tr>
<tr>
<td>0040</td>
<td>7005 6C6F 6361 6C00 000C 0001</td>
</tr>
</tbody>
</table>

- mDNS specification (§18.11):
  “Multicast DNS messages received with non-zero Response Codes MUST be silently ignored.”
- Avahi ignores this packet, Bonjour does NOT
New Platforms, New Code

- Recent years have seen the emergence of new computing platforms which provide many opportunities for optimizations.

- Code is often adapted manually to benefit from these platforms, which is error-prone, as any manual process.
Most processors offer support for SIMD instructions

- Can operate on multiple data concurrently
- Many algorithms can make use of them (e.g., computer vision algorithms)

[EuroSys 2011]
OpenCV

Popular computer vision library from Intel and Willow Garage

Computer vision algorithms were optimized to make use of SIMD

[Corner detection algorithm]
OpenCV Results

• Crosschecked 51 SIMD-optimized versions against their reference scalar implementations
  • Proved the bounded equivalence of 41
  • Found mismatches in 10
• Most mismatches due to tricky FP-related issues:
  • Precision
  • Rounding
  • Associativity
  • Distributivity
  • NaN values
Surprising find: min/max not commutative nor associative!

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

\[a < b \text{ (ordered)} \rightarrow \text{always returns false if one of the operands is NaN}\]

\[
\min(\text{NaN}, 5) = 5
\]
\[
\min(5, \text{NaN}) = \text{NaN}
\]

\[
\min(\min(5, \text{NaN}), 100) = \min(\text{NaN}, 100) = 100
\]
\[
\min(5, \min(\text{NaN}, 100)) = \min(5, 100) = 5
\]
GPU Optimizations

Scalar vs. GPGPU code

[HVC 2011]
General Purpose GPU Computing

(2006) NVIDIA CUDA

(2008) OpenCL
General Purpose GPUs (GPGPUs) are a programmable platform for highly parallel computation.

New programming model:
- Large number of threads
- Hierarchical execution and memory model
OpenCL

- Open Computing Language (OpenCL): an open standard for parallel computation
  - Targets both CPUs and GPGPUs
- OpenCL C language is a dialect of C99
- An OpenCL C program consists of one or more OpenCL C kernels which are executed on a device (such as a GPU)
- OpenCL C kernels may be invoked in parallel by the host using multiple work-items
OpenCL Example

```c
__kernel void arr_sqrt(global float *a) {
    size_t i = get_global_id(0),
    N = get_local_size(0);
    float r0 = i > 0 ? a[i-1]:0;
    float r1 = a[i];
    float r2 = i < N ? a[i+1]:0;
    a[i] = (r0+r1+r2)/3;
}
```

```
i = get_global_id(0)      N = get_local_size(0)
    0  1  ...  i  ...  ...  ...  ...  N-1
```
__kernel void arr_sqrt(global float *a) {
    size_t i = get_global_id(0),
    N = get_local_size(0);
    float r0 = i > 0 ? a[i-1]:0;
    float r1 = a[i];
    float r2 = i < N ? a[i+1]:0;
    a[i] = (r0+r1+r2)/3;
}
__kernel void arr_sqrt(global float *a) {
    size_t i = get_global_id(0),
    N = get_local_size(0);
    float r0 = i > 0 ? a[i-1]:0;
    float r1 = a[i];
    float r2 = i < N ? a[i+1]:0;
    barrier(CLK_GLOBAL_MEM_FENCE);
    a[i] = (r0+r1+r2)/3;
}

barrier() blocks until all work-items (in the same work-group) reached the call
__kernel void arr_sqrt(global float *a) {
    size_t i = get_global_id(0),
    N = get_local_size(0);
    float r0 = i > 0 ? a[i-1]:0;
    float r1 = a[i];
    float r2 = i < N ? a[i+1]:0;
    a[i] = (r0+r1+r2)/3;
}

• Wid: Work-item that accessed byte
• R: whether byte was read
• W: whether byte was written
__kernel void arr_sqrt(global float *a) {
  size_t i = get_global_id(0),
  N = get_local_size(0);
  float r0 = i > 0 ? a[i-1]:0;
  float r1 = a[i];
  float r2 = i < N ? a[i+1]:0;
  a[i] = (r0+r1+r2)/3;
}

• Wid: Work-item that accessed byte
• R: whether byte was read
• W: whether byte was written
Race Checking for GPGPUs

We model barrier() by resetting the MAR before continuing execution any of the work-items past the barrier.
Race Checking for GPGPUs

We model barrier() by resetting the MAR before continuing execution any of the work-items past the barrier.

```c
__kernel void arr_sqrt(global float *a) {
    size_t i = get_global_id(0),
          N = get_local_size(0);
    float r0 = i > 0 ? a[i-1]:0;
    float r1 = a[i];
    float r2 = i < N ? a[i+1]:0;
    barrier(CLK_GLOBAL_MEM_FENCE);
    a[i] = (r0+r1+r2)/3;
}
```

<table>
<thead>
<tr>
<th>Wid</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>

MAR(a[0]): work item 1
Symbolic Races

Work-item $i$

\[ a[i] = \ldots \]

Work-item $j$

\[ a[j] = \ldots \]

Write-after-write race

if $i = j$ satisfiable
GPGPU (OpenCL) Optimizations

• Parboil:
  – GPU benchmark suite, originally written in CUDA

• OP2
  – Library for applications on unstructured grids

• Bullet open-source physics library
  – Popular library used movie studios and professional game developers
  – Analyzed soft body engine
Several bugs and mismatches:

- 2 mismatches between C and OpenCL code
  - Incorrect FP associativity and distributivity assumptions (CP in Parboil)
- 3 memory errors
  - Buffer overflows (MRI-Q&MRI-FHD in Parboil)
  - Use-after-free: incorrect synchronization between host and kernel code (MRI-Q in Parboil)
  - Uninitialized memory (MRI-FHD in Parboil)
- 1 race condition
  - Missing synchronization barrier (OP2)
- 1 compiler bug
  - NVidia compiler bug (incorrect optimization)
Semantic mismatches not always errors
  – Underspecified behavior

Two (anecdotal) insights:

1. Provide developers the ability to add “assumptions” eg:
   – Floating-point associativity holds:
     • A+(B+C) = (A+B)+C
   – Disregard the difference between 0⁻ and 0⁺:
     • A+0 = A

2. All things being equal, developers prefer to keep the behavior of the reference implementation
   – Particularly if we can provide some guarantees
     • bounded equivalence
KATCH: High-Coverage Symbolic Patch Testing

--- klee/trunk/lib/Core/Executor.cpp 2009/08/01 22:31:44 77819
+++ klee/trunk/lib/Core/Executor.cpp 2009/08/02 23:09:31 77922
@@ -2422,8 +2424,11 @@
     info << "none\n";
 } else {
     const MemoryObject *mo = lower->first;
+   std::string alloc_info;
+   mo->getAllocInfo(alloc_info);
     info << "object at " << mo->address
-   << " of size " << mo->size << "\n";
+   << " of size " << mo->size << "\n"
+   << "\t" << alloc_info << "\n";
Symbolic Patch Testing

1. Select the regression input closest to the patch (or partially covering it)
Symbolic Patch Testing

2. Greedily drive exploration toward uncovered statements in the patch
Symbolic Patch Testing

3. If stuck, identify the constraints that disallow execution to reach the patch, and backtrack
void log(char input) {
    int file = open("access.log"...);
    if (input >= ' ' && input <= '~') {
        // printable characters
        write(file, &input, 1);
    } else {
        char escinput = escape(input);
        write(file, &escinput, 1);
    }
    close(file);
}
void log(char input) {
    int file = open("access.log"...);
    if (input >= ' ' && input <= '~') {
        // printable characters
        write(file, &input, 1);
    } else {
        char escinput = escape(input);
        write(file, &escinput, 1);
    }
    close(file);
}

Available input: “t” (or any printable char)

1. Greedy step: choose the symbolic branch point whose unexplored side is closest to the patch.
2. Explore this side!
void log(char input) {
    if (input >= ' ' && input <= '~') {
        ...
    } else {
        ...
    }
}

if (0 == strcmp(request, "GET")
    ...
for (char* p = request; *p; p++)
    log(*p);

Available input: “GET”

Greedy step fails!

1. Backtrack to the last symbolic branch that disallows this side to be executed
2. Explore the other side of that branch

enum escape_t escape;
void log(char input) {
    if (escape == ESCAPE_ALL) {
        . . .
    }
    opt = getopt_long(argc, argv, ...);
    switch (opt) {
    case 'a': escape = ESCAPE_SPACE;
               break;
    . . .
    case 'b': escape = ESCAPE_ALL;
    . . .
    log(...);

Available test: opt = ‘a’
Patch guarded by concrete branch
Backtracking step fails!

1. Find all reaching definitions for the variables involved and try to cover another one.
2. Favors definitions that can be statically shown to satisfy target, or unexecuted definitions
Naïve solution: calculate the context-sensitive static distance between the path executed by an input and the patch code

```c
if (x < 100)
    f(x);
else
    if (x > 200)
        f(x+1);
void f(int x) {
    if (x % 2 == 0)
        PATCH;
    ...
```
Naïve solution: calculate the context-sensitive static distance between the path executed by an input and the patch code

```c
void f(int x) {
    if (x == 999)
        PATCH;
    ...
}
```

```c
if (x < 100)
    f(x);
else
    if (x > 200)
        f(x+1);

x = 55  ➔  1
x = 155  ➔  2
```
For which basic block in the program, compute a necessary condition for reaching the target.

Prune CFG edges which make the target unreachable.

```
if (x < 100)
  f(x);
else
  if (x > 200)
    f(x+1);

void f(int x) {
  if (x == 999)
    PATCH;
  ...
```
Example: Lighttpd r2631

Powers several popular sites such as YouTube and Wikipedia

<table>
<thead>
<tr>
<th>Revision</th>
<th>ELOC</th>
<th>Covered ELOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>Regression</td>
</tr>
<tr>
<td>2631</td>
<td></td>
<td>15 (75%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>KATCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20 (100%)</td>
</tr>
</tbody>
</table>

http://zzz.example.com/  KATCH  https://zzz.example.com/
## Lighttpd r2660

<table>
<thead>
<tr>
<th>Revision</th>
<th>ELOC</th>
<th>Covered ELOC</th>
<th>Regression</th>
<th>KATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>2660</td>
<td>33</td>
<td>9 (27%)</td>
<td></td>
<td>24 (72%)</td>
</tr>
</tbody>
</table>

```c
165 if (str->ptr[i] >= ' ' && str->ptr[i] <= '~') {
166   /* printable chars */
167   buffer_append_string_string_len(dest,&str->ptr[i],1);
168 } else switch (str->ptr[i]) {
169   case ' "':
170     BUFFER_APPEND_STRING_STRING_CONST(dest, "\\\"");
171     break;
```

Bug reported and fixed promptly by developers
Extended Evaluation (WiP)

Key evaluation criteria: **no cherry picking!**

- choose all patches for an application over a contiguous time period

<table>
<thead>
<tr>
<th>Suite Description</th>
<th>ELOC</th>
<th>Patches Written</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindUtils suite (FU)</td>
<td>12,648</td>
<td>125</td>
</tr>
<tr>
<td>find, xargs, locate</td>
<td></td>
<td>over ~26 months</td>
</tr>
<tr>
<td>DiffUtils suite (DU)</td>
<td>55,655</td>
<td>175</td>
</tr>
<tr>
<td>s/diff, diff3, cmp</td>
<td>+ 280,000 in libs</td>
<td>over ~30 months</td>
</tr>
<tr>
<td>BinUtils suite (BU)</td>
<td>81,933</td>
<td>181</td>
</tr>
<tr>
<td>ar, elfedit, nm, etc.</td>
<td>+ 800,000 in libs</td>
<td>over ~16 months</td>
</tr>
</tbody>
</table>
Patch Coverage (basic block level)

Standard symbolic execution (30min/BB) only added +1.2% to FU
Patch Coverage
(current results – ongoing work)

**FU:**
- TEST: 63% covered, 0% uncovered
- + KATCH: 87% covered, 100% covered
- 10min/BB

**DU:**
- TEST: 35% covered, 73% covered
- + KATCH: 100% covered
- 10min/BB

**BU:**
- TEST: 18% covered, 27% covered
- + KATCH: 100% covered
- 15min/BB

*Standard symbolic execution (30min/BB) only added +1.2% to FU*
Binutils: Coverage+Bugs
(current results)

- Found 14 distinct crash bugs, all in BU
  - All unreachable by standard symbolic execution
- 12 bugs still present in latest version
  - Reported (some already fixed) by developers
- 10 bugs found in the patch code itself or affected by patch code
KLEE – Demo

http://klee.llvm.org

- bad_abs
- xcheck_abs
- squares
Dynamic Symbolic Execution

• Automatically reasons about program behavior and the interaction with users and environment
• Can generate inputs exposing both generic and semantic bugs in complex software
  • Including file systems, library code, utility applications, network servers, device drivers, computer vision code
KLEE: Freely Available as Open-Source

http://klee.llvm.org

• Over 250 subscribers to the klee-dev mailing list
• Extended in many interesting ways by several research groups, in the areas of:
  • wireless sensor networks/distributed systems
  • schedule memoization in multithreaded code
  • automated debugging
  • exploit generation
  • client-behavior verification in online gaming
  • GPU testing and verification
  • etc.