Modern Symbolic Execution: DART, EGT, CUTE, jCUTE, EXE, KLEE, CREST, CATG

Cristian Cadar
Department of Computing
Imperial College London

Koushik Sen
EECS Department
University of California, Berkeley
Scalability Challenges

Path exploration challenges

Constraint solving challenges
Path Exploration 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

- Depth-First Search
- Breadth-First Search
- Random state selection
- Coverage-optimized search (Best-First)
- Hybrid random-symbolic testing
- Random path search
- etc.
Search Heuristics

• Coverage-optimized search
  [EXE, KLEE, CREST implement diff. variations]
  • Select path closest to an uncovered instruction
  • Favor paths that recently hit new code

• Hybrid random-symbolic search [CUTE]
  • Interleaves random testing with systematic concolic testing
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
  - less constraints
- Avoid starvation
  - e.g. symbolic loop

[Implemented in CREST, KLEE]
One approach [KLEE]: use multiple heuristics in a round-robin fashion!

• Protects 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

[TACAS 2008]
data, arg1, arg2 = symbolic

flag = 0;

if (arg1 > 100)
  flag = 1;

if (arg2 > 100)
  flag = 1;

process(data, flag);
Many Redundant Paths

PCI driver (MINIX) - 1h runs

- Base
- Redundant path elimination
Lots of Redundant Paths

- bpf
- expat
- pcre
- tcpdump

- udhcpd
- sb16
- lance
Redundant Path Elimination

PCI driver (MINIX) - 1h runs

Branch coverage (%)

Generated tests

- Base
- Redundant path elimination
Statically Merging Paths

Default behaviour

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

Phi-Node Folding (when no side effects)

\[
\begin{align*}
\text{if } (a > b) & \quad \text{max} = \text{select}(a>b, a, b) \\
\text{else } & \quad \text{max} = b;
\end{align*}
\]
Statically Merging Paths

```c
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
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-inlude.t ... ok
./core.t ................ ok
./lowercase.t ........... ok
./mod-access.t .......... ok
...
# Regression Suites

<table>
<thead>
<tr>
<th><strong>PROS</strong></th>
<th><strong>CONS</strong></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>
Combining Regression Suites with Symbolic Execution

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

main(argv, argc)

- dangerous operations
- divergence points

Bounded symbolic execution

exit(0)
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
Our Constraint Solver: 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 (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); }
```

```
buf: ARRAY BITVECTOR(32)OF BITVECTOR(8)
SBVLT(buf[18], 0x00)
BVGE(buf[19]@buf[18]@buf[17]@buf[16], 0x00800000)
```
Constraint Solving: Performance

Constraint solving optimizations essential:

• Symbolic execution-driven optimizations implemented by constraint solvers

• Higher-level optimizations implemented outside constraint solvers

• Specialized constraint solvers (e.g., strings)
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

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

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

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

STP’s conversion of array terms to SAT is expensive

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

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

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

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

\[(a[2] = 1) \land (a[2] = 2) \land (a[2] = 3) \land (2+2+2 = 6)\]
Array-based Refinement in STP

STP’s conversion of array terms to SAT is expensive

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

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

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

\[
\begin{align*}
&w + z > 100 \\
&2w - 1 < 12345 \\
&x + y > 10 \\
&z &- z = z \\
&x < 10 ?
\end{align*}
\]

[Implemented in EXE, CUTE, etc.]
Caching Solutions

- Static set of branches: lots of similar constraint sets

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

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

Eliminating constraints cannot invalidate solution

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

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

Adding constraints often does not invalidate solution

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

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

[OSDI 2008]
Significant Speedup

Aggregated data over 73 applications

- Base
- Irrelevant Constraint Elimination
- Caching
- Irrelevant Constraint Elimination + Caching
EGT, EXE, KLEE and extensions

Joint work with Dawson Engler, Daniel Dunbar, Paul Marinescu, Peter Collingbourne, Paul Kelly, Junfeng Yang, Peter Pawlowski, Can Sar, Paul Twohey, Vijay Ganesh, David Dill, Peter Boonstoppel, JaeSeung Song, Peter Pietzuch
Experimental Results
(or what it’s good for)

High-coverage Test Generation

Generic Bug-Finding

Attack Generation

Semantic Error Detection via Crosschecking

Patch Testing
Experimental Results
(or what it’s good for)

High-coverage Test Generation

Generic Bug-Finding

Attack Generation

Semantic Error Detection via Crosschecking

Patch Testing
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>Coreutils, Busybox, Minix (over 450 apps)</td>
</tr>
<tr>
<td>UNIX file systems</td>
</tr>
<tr>
<td>ext2, ext3, JFS</td>
</tr>
<tr>
<td>Network servers</td>
</tr>
<tr>
<td>Bonjour, Avahi, udhcpd, lighttpd</td>
</tr>
<tr>
<td>Library code</td>
</tr>
<tr>
<td>libdwarf, libelf, PCRE, uClibc, Pintos</td>
</tr>
<tr>
<td>Packet filters</td>
</tr>
<tr>
<td>FreeBSD BPF, Linux BPF</td>
</tr>
<tr>
<td>MINIX device drivers</td>
</tr>
<tr>
<td>pci, lance, sb16</td>
</tr>
<tr>
<td>Kernel code</td>
</tr>
<tr>
<td>HiStar kernel</td>
</tr>
<tr>
<td>Computer vision code</td>
</tr>
<tr>
<td>OpenCV (filter, remap, resize, etc.)</td>
</tr>
<tr>
<td>OpenCL code</td>
</tr>
<tr>
<td>Parboil, Bullet, OP2</td>
</tr>
</tbody>
</table>

- Most bugs fixed promptly
Experimental Results
(or what it’s good for)

High-coverage Test Generation

Generic Bug-Finding

Attack Generation

Semantic Error Detection via Crosschecking

Patch Testing
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

01010110
11010100
01011100
00110101

[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 0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>08000</td>
<td>464A 3135 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08010</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08020</td>
<td>0000 0000 0100 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08030</td>
<td>E004 000F 0000 0000 0000 0000 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

= *

EXE/KLEE

recv( )

Network Server

10111001 01011100
10111001 01011100
10111001 01011100

[CCS 2006, ICCCN 2011]
**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</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 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
Experimental Results
(or what it’s good for)

**High-coverage Test Generation**

**Generic Bug-Finding**

**Attack Generation**

**Semantic Error Detection via Crosschecking**

**Patch Testing**
Semantic Bugs

- Bugs shown so far are all generic errors
- What about semantic bugs?
- Can find `assert()` violations
  - Can verify assert statements on a per-path basis
  - Unfortunately, we found few “interesting” asserts
Semantic Bugs via Crosschecking

Crosschecking (aka equivalence checking)
  • Successfully used in the past
  • Great match for symbolic execution

Lots of available opportunities:
  • Different implementations of the same functionality: e.g., libraries, servers, compilers
  • Optimized versions of a reference implementation
  • Refactored code
  • Reverse computations: e.g., compress and uncompress
We can find any mismatches in their behavior by:
1. Using symbolic execution to explore multiple paths
2. Comparing the path constraints across implementations
Crosschecking: Advantages

- No need to write any 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 proved to be equivalent
Crosschecking: Advantages

Many crosschecking queries can be *syntactically* proved 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).
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</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
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)
SIMD Optimizations

OpenCV: popular computer vision library from Intel and Willow Garage
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
Other Crosschecking Studies

UNIX utilities: desktop vs. embedded
[OSDI 2008]

GPU Optimizations: Scalar vs. GPGPU code
[HVC 2011]

DHCP servers: desktop vs. embedded
[WiP]
Experimental Results
(or what it’s good for)

High-coverage Test Generation

Generic Bug-Finding

Attack Generation

Semantic Error Detection via Crosschecking

Patch Testing
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\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
3. If stuck, identify the constraints that disallow execution to reach the patch, and backtrack.
Preliminary Results

Powers several popular sites such as YouTube and Wikipedia

<table>
<thead>
<tr>
<th>Revision</th>
<th>ELOC</th>
<th>Covered ELOC</th>
<th>Regression</th>
<th>KATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>2631</td>
<td>20</td>
<td>15 (75%)</td>
<td>20 (100%)</td>
<td></td>
</tr>
<tr>
<td>2660</td>
<td>33</td>
<td>9 (27%)</td>
<td>24 (72%)</td>
<td></td>
</tr>
<tr>
<td>2747</td>
<td>10</td>
<td>4 (40%)</td>
<td>10 (100%)</td>
<td></td>
</tr>
</tbody>
</table>
## Lighttpd r2631

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

Regression KATCH: 20 (100%)

http://zzz.example.com/

{KATCH} to

https://zz.example.com/
Bug reported and fixed promptly by developers
CREST and KLEE: Freely Available as Open-Source

CREST: http://code.google.com/p/crest/

- Extensible concolic execution tool for C
- 1500+ downloads since mid-2008 release
- Used to augment existing test suites, detect SQL injection vulnerabilities, modified to run distributed on a cluster for testing a flash storage platform, used in teaching courses at some universities, etc.
CREST and KLEE: Freely Available as Open-Source

KLEE:  http://klee.llvm.org

- Flexible symbolic execution tool based on the LLVM framework and the STP solver, primarily for C code
- Over 200 subscribers to the klee-dev mailing list
- Extended in many interesting ways by several research groups, in the areas of wireless sensor networks, automated debugging, schedule memoization in multithreaded code, exploit generation, online gaming, etc.
Modern Symbolic Execution:
DART, EGT, CUTE, jCUTE, EXE, KLEE, CREST, CATG

Cristian Cadar
Department of Computing
Imperial College London

Koushik Sen
EECS Department
University of California, Berkeley