How to Crash Your Code
Using Dynamic Symbolic Execution

Cristian Cadar
Department of Computing
Imperial College London

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
Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

Abstract. This paper presents a technique that uses code to automatically generate its own test cases at run-time by using a combination of symbolic and concrete (i.e., regular) execution. The input values to a program (or software component) provide the standard interface of any testing framework with the program it is testing, and generating input values that will explore all the “interesting” behavior in the tested program remains an important open problem in software testing research. Our approach works by turning the problem on its head: we lazily generate, from within the program itself, the input values to the program (and
Dynamic Symbolic Execution

Automated technique for generating high-coverage test suites, and finding bugs in software systems

• Received significant interest in the last few years
• Many dynamic symbolic execution/concolic tools available as open-source:
  – CREST, KLEE, SYMBOLIC JPF, etc.
• Started to be adopted by the industry:
  – Microsoft (SAGE, PEX)
  – IBM (APOLLO)
  – Fujitsu (KLEE/KLOVER, SYMBOLIC JPF)
  – etc.
Toy Example

```c
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
• 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]];
}
```
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!
Dynamic Symbolic Execution

- Each path is (essentially) explored separately
  - As in regular testing

- 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
  • 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

EGT/EXE/KLEE

x ≥ 0
x ≠ 1234

x = 3

Constraint Solver (STP)

x = -2
x = 1234
x = 3
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

• **Coverage-optimized search**
  – Select path closest to an uncovered instruction
  – Favor paths that recently hit new code

• **Best-first search**

• **Random path search**

• **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
  - less constraints
- Avoid starvation
  - e.g. symbolic loop
Our latest system 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

- Base
- 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

- Base
- Redundant path elimination

Branch coverage (%)

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)
```
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

```bash
$ 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><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>
ZESTI: Using Existing Regression Suites

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)
eixt(0)

✓

dangerous operations
divergence points

Bounded symbolic execution

exit(0)
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><strong>UNIX utilities</strong></td>
</tr>
<tr>
<td>Coreutils, Busybox, Minix (over 450 apps)</td>
</tr>
<tr>
<td><strong>UNIX file systems</strong></td>
</tr>
<tr>
<td>ext2, ext3, JFS</td>
</tr>
<tr>
<td><strong>Network servers</strong></td>
</tr>
<tr>
<td>Bonjour, Avahi, udhcpd, lighttpd</td>
</tr>
<tr>
<td><strong>Library code</strong></td>
</tr>
<tr>
<td>libdwarf, libelf, PCRE, uClibc, Pintos</td>
</tr>
<tr>
<td><strong>Packet filters</strong></td>
</tr>
<tr>
<td>FreeBSD BPF, Linux BPF</td>
</tr>
<tr>
<td><strong>MINIX device drivers</strong></td>
</tr>
<tr>
<td>pci, lance, sb16</td>
</tr>
<tr>
<td><strong>Kernel code</strong></td>
</tr>
<tr>
<td>HiStar kernel</td>
</tr>
<tr>
<td><strong>Computer vision code</strong></td>
</tr>
<tr>
<td>OpenCV (filter, remap, resize, etc.)</td>
</tr>
<tr>
<td><strong>OpenCL code</strong></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!
• 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
10111001 01011100
01010111 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 0000</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>08000</td>
<td>464A 3135 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08010</td>
<td>1000 0000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08020</td>
<td>0000 0000 0100 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08030</td>
<td>E004 000F 0000 0000 0000 0002 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

[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</td>
</tr>
<tr>
<td>0010</td>
<td>003E 0000 4000 FF11 1BB2</td>
</tr>
<tr>
<td>0020</td>
<td>00FB 0000 14E9 002A 0000</td>
</tr>
<tr>
<td>0030</td>
<td>0000 0000 0000 055F 6461</td>
</tr>
<tr>
<td>0040</td>
<td>7005 6C6F 6361 6C00 000C</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

Option 1: Use manually-written specifications!
Crosschecking (Equivalence Checking)

Option 2: Crosschecking!

• 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
Crosschecking

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</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

[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
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
[Marinescu and Cadar, SPIN 2012]

```cpp
--- 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)
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.
## Preliminary Results

Powers several popular sites such as YouTube and Wikipedia

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

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

http://zzz.example.com/  →  KATCH  →  https://zz.example.com/
Lighttpd r2660

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

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

Bug reported and fixed promptly by developers
Dynamic Symbolic Execution

- Automatically explores paths through a program
- 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

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

http://klee.llvm.org