EXE: Automatically Generating Inputs of Death

Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, Dawson Engler

Stanford University

CCS 2006, Alexandria, VA
What is EXE?

- Goal: generate inputs that explore (ideally) all paths of real C systems code
What is EXE?

- Goal: generate inputs that explore (ideally) all paths of real C systems code

1. Bug-finding tool
   - Produces concrete inputs that trigger attacks
PCRE – expressions of death

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^[^[0^-][&quot;\]]*?-?</code> {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
<tr>
<td><code>[^-</code>[^]][&quot;]]*?-?` {0</td>
<td></td>
</tr>
</tbody>
</table>

[^0^-]: [^0^-]
What is EXE?

- Goal: generate inputs that explore (ideally) all paths of real C systems code

1. Bug-finding tool
   - Produces concrete inputs that trigger attacks

2. Test case generator
   - Good statement/block, branch, path coverage
EXE vs. random (BPF)
Basic idea

- Use the code itself to construct its input
- Symbolic execution = collect constraints on inputs marked as *symbolic*
Example (simplified BPF code)

```c
static inline void *skb_header_pointer(struct sk_buff *skb,
                                      int offset,
                                      int len) {
    if (offset + len <= skb->len) {
        return skb->data + offset;
        exit(1);
    }
}
...
exe_make_symbolic(&offset);
...
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```
static inline void *skb_header_pointer(struct sk_buff *skb, int offset, int len) {
    if (offset + len <= skb->len) {
        return skb->data + offset;
        exit(1);
    }
}

exe_make_symbolic(&offset);

... u16* p = skb_header_pointer(skb, offset, 4);
    u32 A = *p;
Example (simplified BPF code)

```c
static inline void *skb_header_pointer(struct sk_buff *skb, int offset, int len) {
    if (offset + len <= skb->len)
        return skb->data + offset;
    exit(1);
}
...
exe_make_symbolic(&offset);
...
```

```c
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```
Example (simplified BPF code)

```c
static inline void *skb_header_pointer(struct sk_buff *skb, int offset, int len) {
    if (offset + len <= skb->len) {
        return skb->data + offset;
        exit(1);
    }
}
```

```c
... exe_make_symbolic(&offset);
... u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```
Example (simplified BPF code)

```c
static inline void *skb_header_pointer(struct sk_buff *skb, int offset, int len) {
    if (offset + len <= skb->len) {
        return skb->data + offset;
    }
    exit(1);
}
...
exe_make_symbolic(&offset);
...
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```
### Example (simplified BPF code)

```c
static inline void *skb_header_pointer(struct sk_buff *skb, int offset, int len) {
    if (offset + len <= skb->len) {
        return skb->data + offset;
    }

    exit(1);
}
```

```c
exe_make_symbolic(&offset);
```

```c
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```
Example (simplified BPF code)

```c
static inline void *skb_header_pointer(struct sk_buff *skb, int offset, int len) {
    if (offset + len <= skb->len) {
        return skb->data + offset;
        exit(1);  
    }

    ...  
    exe_make_symbolic(&offset);

    ...  
    u16* p = skb_header_pointer(skb, offset, 4);
    u32 A = *p;
```
Example (simplified BPF code)

```c
static inline void *skb_header_pointer(struct sk_buff *skb, int offset, int len) {
    if (offset + len <= skb->len) {
        return skb->data + offset;
        exit(1);
    }
}
...
exe_make_symbolic(&offset);
...
u16* p = skb_header_pointer(skb, offset, 4);
u32 A = *p;
```
Running EXE

% exe-cc bpf.c
% ./a.out
EXE execution

```
if (offset + len <= skb->len)
    TRUE
    offset+ 4 <= 128
    add(offset + 4 <= 128)
    return 0xdeadbeef + offset
    A = *(0xdeadbeef + offset)
    A = *p;
    121
    test1.out

FALSE
add(offset + 4 > 128)
return skb->data + offset
exit(1)
2147483513
test2.out
```
EXE execution

add(offset + 4 <= 128)

return 0xdeadbeef + offset

A = *(0xdeadbeef+offset)

121

test1.out

if (offset + len <= skb->len)

offset + 4 <= 128

TRUE

FALSE

add(offset + 4 > 128)

return skb->data + offset

A = *p;

exit(1)

2147483513

test2.out
EXE execution

Running the values on the uninstrumented code makes the code follow the exact path on which the values were generated.
Implicit checks
Arbitrary checks

- By default, EXE looks for generic errors
- But, can check arbitrary properties:

```c
assert(compress(uncompress(x)) == x);
```
Big challenge no. 1

- Systems code often observes the same bytes in different ways
  - Simple casts: signed to unsigned, int to char etc.
  - Pointer casting: treating array of bytes as: network packets, inodes, packet filters etc.

```c
char buffer[N];
struct sk_buff *skb = (struct sk_buff*) buffer;
hlen = skb->len - skb->data_len;
```
Modeling of memory in EXE

- Mirror the (lack of) C type system
  - Untyped memory
    - Bind types to expressions, not bits
  - Bit-level accuracy

- Need constraint solver that has untyped memory and bit-level accuracy
Modern constraint solver, based on SAT

Eagerly translates high-level constraints to SAT formula, using straightforward transformations
  - E.g., a 32-bit add is implemented as a ripple-carry adder

Uses off-the-shelf SAT solver (MiniSAT)

Declared the co-winner of the bitvector division of SMTLIB, held during CAV 2006
Bitvectors

- Untyped memory + bit-level accuracy
  - Bitvector data type:
    - Fixed length sequence of bits
      - Ex: 0110 is a constant, 4-bit bitvector

- Arrays of bitvectors
Bitvectors

- Bitvectors have all operations on integers
  - including multiplication, division, modulo

- EXE can translate all C expressions into STP constraints with bit-level precision
  - Except floating-point
Big challenge no. 2

- Exponential space
  - Goal: find bugs, achieve good coverage
  - Efficient exploration of the search space
    - Especially in the presence of loops
- Search heuristics
Search heuristics

- DFS used by default
- Best First search
  - Each forked EXE process calls into a server with its current state
  - Server chooses the next process to run based on some heuristic
Best first heuristic

- Current best first search heuristic
  - Pick the process at the line of code run the fewest number of times
  - Run it in DFS mode for a while, then iterate
  - Good statement/block coverage
Big challenge no. 3

- Reasoning about arrays in STP
- Example:
  - Symbolic index $i$, $0 \leq i < n$
  - $(a[i] = 7)$
Big challenge no. 3

- Reasoning about arrays in STP
- Example:
  - Symbolic index $i$, $0 \leq i < n$
  - $(a[i] = 7) \iff (a[0] = 7) \lor (a[1] = 7) \lor \ldots \lor (a[n-1] = 7)$
Converting arrays to SAT

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

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

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

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

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

\[(i_1 = i_2 \Rightarrow v_1 = v_2) \land (i_1 = i_3 \Rightarrow v_1 = v_3) \land (i_2 = i_3 \Rightarrow v_2 = v_3)\]
Converting arrays to SAT

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

Array elimination expands each formula by \(n(n-1)/2\) terms, where \(n\) is the number of syntactically distinct indexes.
Array-based refinement

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

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

\[(i_1 = i_2 \Rightarrow v_1 = v_2) \land (i_1 = i_3 \Rightarrow v_1 = v_3) \land (i_2 = i_3 \Rightarrow v_2 = v_3)\]
Array-based refinement

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

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

\[(i_1 - i_2 \Rightarrow v_1 - v_2) \land (i_1 - i_3 \Rightarrow v_1 - v_3) \land (i_2 - i_3 \Rightarrow v_2 - v_3)\]
Array-based refinement

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

Under-approximation
UNSATISFIABLE

Original formula
UNSATISFIABLE
Array-based refinement

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

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

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

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

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

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

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

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

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

\[
(a[2] = 1) \land (a[2] = 2) \land (a[2] = 3) \land (2 + 2 + 2 = 6)
\]
## Evaluation

<table>
<thead>
<tr>
<th>Solver</th>
<th>Total time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVCL</td>
<td>1006</td>
</tr>
<tr>
<td>STP (baseline)</td>
<td>56</td>
</tr>
<tr>
<td>STP (array-based refinement)</td>
<td>10</td>
</tr>
</tbody>
</table>

- 8495 test cases from our benchmarks
- Timeout set at 60 s.

- 100 x faster than CVCL
- 5 x faster than base STP
### Evaluation

<table>
<thead>
<tr>
<th>Solver</th>
<th>Total time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVCL</td>
<td>1006</td>
</tr>
<tr>
<td>STP (baseline)</td>
<td>56</td>
</tr>
<tr>
<td>STP (array-based refinement)</td>
<td>10</td>
</tr>
<tr>
<td>STP (all optimizations)</td>
<td>2</td>
</tr>
</tbody>
</table>

- 8495 test cases from our benchmarks
- Timeout set at 60 s.

- 100 x faster than CVCL
- 5 x faster than base STP
Results

- Berkeley Packet Filter
- Perl Compatible Regular Expressions Library
- udhcpd DHCPD server
- Linux file systems
Berkeley Packet Filter (BPF)

- Allows programmers to specify what network packets they want to receive
- Did not hope to find bugs
- Checked the FreeBSD and Linux implementations
BPF – Results

- Buffer overflows in both FreeBSD and Linux versions

**FreeBSD filter of death:**

```c
s[0].code = BPF_STX;
s[0].k    = 0xffffffff0UL;
s[1].code = BPF_RET;
s[1].k    = 0xffffffff0UL;
```

**Linux filter of death:**

```c
s[0].code = BPF_LD|BPF_B|BPF_ABS;
s[0].k    = 0xffffffffUL;
s[1].code = BPF_RET;
s[1].k    = 0xffffffff0UL;
```
Perl Compatible Reg Exp (PCRE)

- Used by popular open-source projects
  - Apache, PHP, Postfix
- Found buffer overflows which crash PCRE
  - In pcre_compile, which compiles a pattern string into a regular expression
- Author notified, and promptly fixed the bug
PCRE – regex’s of death

<table>
<thead>
<tr>
<th>Regex 1</th>
<th>Regex 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^[^\[\0^\0]*-?]</code></td>
<td><code>[-\</code>[\0^\0]`-?]`</td>
</tr>
<tr>
<td><code>[^\</code>[\0^\0]`-?]`</td>
<td><code>[^\</code>[\0^\0]`-?]`</td>
</tr>
<tr>
<td><code>[^\</code>[\0^\0]`-?]`</td>
<td><code>[^\</code>[\0^\0]`-?]`</td>
</tr>
<tr>
<td><code>(?#)\[\[\0\0\]\-]</code></td>
<td><code>(?#)\[\[\0\0\]\-]</code></td>
</tr>
<tr>
<td><code>(?#)\[\[\0\0\]\[\0\0\]\]</code></td>
<td><code>(?#)\[\[\0\0\]\[\0\0\]\]</code></td>
</tr>
<tr>
<td><code>(?#)\[\[\0\0\]\[\0\0\]\]</code></td>
<td><code>(?#)\[\[\0\0\]\[\0\0\]\]</code></td>
</tr>
<tr>
<td><code>(?#)\[\[\0\0\]\[\0\0\]^\0\]</code></td>
<td><code>(?#)\[\[\0\0\]^\0\]</code></td>
</tr>
<tr>
<td><code>(?#)\[\[\0\0\]^\0\]</code></td>
<td><code>(?#)\[\[\0\0\]^\0\]</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Regex 3</th>
<th>Regex 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[-\</code>[\0^\0]`</td>
<td><code>[^\[\0^\0\]\</code>-?]`</td>
</tr>
<tr>
<td><code>[^\</code>[\0^\0]`-?]`</td>
<td><code>[^\</code>[\0^\0]`-?]`</td>
</tr>
<tr>
<td><code>[^\</code>[\0^\0]`-?]`</td>
<td><code>[^\</code>[\0^\0]`-?]`</td>
</tr>
</tbody>
</table>
udhcpd 0.9.8

- Clean, well-tested user-level DHCPD server
- Marked its input packet as symbolic, and changed its network read call to return symbolic data
- Found five memory errors
Linux file systems

- Generated disk images for ext2, ext3, JFS
- Found bugs in all systems – generated real disk images which when mounted, compromise or crash the Linux kernel
- *Automatically generating malicious disks using symbolic execution*  
  J. Yang, C. Sar, P. Twohey, C. Cadar, D. Engler  
  IEEE Security 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 3153  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 0002 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>08040</td>
<td>0000 0000 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>
Related Work

- DART system [Godefroid, Klarlund, Sen]
- CUTE system [Sen, Marinov, Aga]
- CBMC [Clarke, Kroening]
  - Limitations in terms of handling systems code
Eager translation to SAT
- UCLID, Cogent, Saturn

Nelson-Oppen solvers
- CVCL, Yices, SVC, Barcelogic Tools

Hard to do side-by-side comparison
- No common benchmarks
- No common syntax
Summary

- EXE generates inputs that expose bugs and achieve good coverage
- STP constraint solver which enables EXE to solve constraints fast
- Systems code benchmarks
  - Found bugs in all of them
  - Generated inputs that trigger the bugs discovered
Questions?