Motivation

Providing functional specifications for large software is very hard

- Functional specification: desired properties of inputs and outputs
- E.g., C compiler

> 600 pages in English
**An Idea**

- Input
  - C Compiler
  - Other C Compilers
- Buggy
  - different
- Compare Output
  - same
- Correct

**Differential Testing**

**Cross-check**
- Execute different implementations of the same functionality (e.g., GCC and LLVM) with the same inputs
- Compare their outputs
- Report any anomalies as bugs
Success Stories

• Compilers

```c
int foo() {
    signed char x = 1;
    unsigned char y = 255;
    return x > y;
}
```

1. gcc in Ubuntu 8.04 - Buggy
2. Other compilers - Correct

• Neural Nets

1. One version of Nvidia self-driving car system - Buggy
2. Other versions - Correct

Challenges

How do we generate good inputs?

– **Concise**: Avoids illegal and redundant tests

– **Diverse**: Gives good coverage of discrepant parts
Approaches

• Unguided Approach
  – Generate test inputs independently across iterations
  – e.g., Csmith
• Guided Approach
  – Generate test inputs by observing program behavior for past inputs
  – e.g., NEZHA, DeepXplore

Unguided Testing for C Compilers

CSmith: Unguided Differential Testing tool for C Compilers

[Diagram showing the process of generating random C programs, compiling them using GCC, and comparing the outputs of multiple binaries]
Input Generation

- Based on random testing
  - Randomly generate C programs

- Considering domain-specific knowledge
  - Well-formedness (C syntax)
  - Well-definedness (C semantics)

Found Compiler Bugs

<table>
<thead>
<tr>
<th>LLVM Version</th>
<th>1.9</th>
<th>2.0</th>
<th>2.1</th>
<th>2.2</th>
<th>2.3</th>
<th>2.4</th>
<th>2.5</th>
<th>2.6</th>
<th>2.7</th>
<th>2.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Bugs</td>
<td>27</td>
<td>20</td>
<td>18</td>
<td>22</td>
<td>22</td>
<td>12</td>
<td>13</td>
<td>7</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GCC Version</th>
<th>3.0.0</th>
<th>3.1.0</th>
<th>3.2.0</th>
<th>3.3.0</th>
<th>3.4.0</th>
<th>4.0.0</th>
<th>4.1.0</th>
<th>4.2.0</th>
<th>4.3.0</th>
<th>4.4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Bugs</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>11</td>
<td>7</td>
<td>6</td>
<td>14</td>
<td>5</td>
</tr>
</tbody>
</table>
Limitations of Unguided Testing

Generally, highly inefficient to find discrepancies, because:

- Randomly generates inputs
- Ignores any information from past inputs

Guided Testing for Binaries

**NEZHA**: Guided Differential Testing tool for Binaries

- Exploit behavioral discrepancies between multiple binary executables
- Evolve an input corpus that is guided based on runtime information (obtained by doing binary instrumentation)
Guided Testing

• Starting from input seeds, keep generating mutations until discrepancies are found

• Instead of maintaining all mutants, only keep “promising” ones
  – Mutants of inputs that led to combinations of states unseen so far
Example

Input corpus: \{ 7, 0, 1 \}

Exercised input:

Exercised edges: \{ \}

Example

Input corpus: \{ 7, 0, 1 \}

Exercised input: 7

Exercised edges:

\{ E1, E2', E3' \}
Example

Input corpus: \{7, 0, 1\}

Exercised input: 0

Exercised edges:
\{
  \{E1, E2', E3'\},
  \{E2, E3, E1'\}
\}

---

Example

Input corpus: \{7, 0, 1\}

Exercised input: 1

Exercised edges:
\{
  \{E1, E2', E3'\},
  \{E2, E3, E1'\}
\}

Not added and discarded
Example

```c
int checker_A (int v) {
    if (v % 2 != 0)
        return -1;
    if (v < 1 || v > 7)
        return -2;
    return 0;
}

int checker_B (int v) {
    if (v < 3 || v > 7)
        return -2;
    if (v % 2 != 0)
        return -1;
    return 0;
}
```

Input corpus: { 7, 0, 1 }

Exercised input: 2

(pick 1 and mutate it by adding 1)

Exercised edges:

```
{E1, E2, E3},
{E2, E3, E1'},
{E1, E1'},
{E3, E4, E1'}
```

Intuition

• Inputs that exercise different code regions in the two apps might imply differences in handling logic

• Such inputs are likely to find discrepancies
Case Studies

<table>
<thead>
<tr>
<th>Application</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSL Libraries</td>
<td>OpenSSL, LibreSSL, BoringSSL, GnuTLS, wolfSSL, mbedTLS</td>
</tr>
<tr>
<td>PDF Readers</td>
<td>Evince PDF, MuPDF, Xpdf</td>
</tr>
<tr>
<td>ELF Parser</td>
<td>ClamAV, binutils</td>
</tr>
<tr>
<td>XZ Parser</td>
<td>ClamAV, XZ</td>
</tr>
</tbody>
</table>

Discrepancies and Bugs Found

<table>
<thead>
<tr>
<th>Type</th>
<th>SSL Certificate</th>
<th>PDF File</th>
<th>ELF Binary</th>
<th>XZ Archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrepancies</td>
<td>764</td>
<td>7</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Errors and Crashes</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Guided Testing for DNNs

DeepXplore: guided differential testing tool for Deep Neural Networks (DNNs)
Input Generation

- Iteratively changing a seed input
- Two criteria:
  1) Maximizing differential behaviors
  2) Maximizing neuron coverage
- Formally, maximizing the following objective function

\[
\text{obj}(x) = (\sum_{k \neq j} F_k(x)[c] - \lambda_1 \cdot F_j(x)[c]) + \lambda_2 \cdot f_n(x)
\]

Maximizing Differential Behavior

- Suppose we have \( n \) DNNs \( F_1 \ldots F_n \)
- Let \( F_k(x)[c] \) be the class probability that \( F_k(x) \) predicts \( x \) to be \( c \)
- Randomly select one DNN \( F_k \)
- Maximize this objective function:

\[
\text{obj}_1(x) = \sum_{k \neq j} F_k(x)[c] - \lambda_1 \cdot F_j(x)[c]
\]

- Easily solved using gradient ascent
Maximizing Neuron Coverage

- Iteratively pick an inactivated neuron $f$
- Modify the input such that a neuron becomes activated, that is, maximize this objective function for neuron $n$:

$$\text{obj}_2(x) = f_n(x)$$

- Easily solved using gradient ascent

Empirical Evaluation

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Description</th>
<th>DNN Description</th>
<th>DNN Name</th>
<th>#Neurons</th>
<th>#Disciplines / #Seeds</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>Hand-written digits</td>
<td>LeNet variations</td>
<td>MNI_C1</td>
<td>52</td>
<td>1,073 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MNI_C2</td>
<td>148</td>
<td>1,968 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MNI_C3</td>
<td>268</td>
<td>827 / 2,000</td>
</tr>
<tr>
<td>Imagenet</td>
<td>General images</td>
<td>State-of-the-art image classifiers</td>
<td>IMG_C1</td>
<td>14,888</td>
<td>1,969 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IMG_C2</td>
<td>16,166</td>
<td>1,976 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IMG_C3</td>
<td>94,059</td>
<td>1,996 / 2,000</td>
</tr>
<tr>
<td>Driving</td>
<td>Driving video frames</td>
<td>Nvidia DAVE self-driving systems</td>
<td>DRV_C1</td>
<td>1,560</td>
<td>1,720 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DRV_C2</td>
<td>1,560</td>
<td>1,866 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DRV_C3</td>
<td>844</td>
<td>1,930 / 2,000</td>
</tr>
<tr>
<td>Contagio/Virus total</td>
<td>PDFs</td>
<td>PDF malware detectors</td>
<td>PDF_C1</td>
<td>402</td>
<td>1,103 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDF_C2</td>
<td>602</td>
<td>789 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDF_C3</td>
<td>802</td>
<td>1,253 / 2,000</td>
</tr>
<tr>
<td>Drebin</td>
<td>Android apps</td>
<td>Android app malware detectors</td>
<td>APP_C1</td>
<td>402</td>
<td>2,000 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>APP_C2</td>
<td>102</td>
<td>2,000 / 2,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>APP_C3</td>
<td>212</td>
<td>2,000 / 2,000</td>
</tr>
</tbody>
</table>
Empirical Evaluation

What Have We Learned?

- Feeding same inputs to different implementations
- Observing behavioral differences
- Reporting discrepancies as bugs
- Input generation methods
  - Unguided vs. Guided
  - Tools: Csmith, NEZHA, DeepXPlore