XSTRESSOR: Automatic Generation of Large-Scale Worst-Case Test Inputs by Inferring Path Conditions

Charitha Saumya, Jinkyu Koo, Milind Kulkarni, and Saurabh Bagchi
Electrical and Computer Engineering
Purdue University
Outline

• Motivation
• Related work
• Method
• Evaluation
• Conclusion
Performance Inputs

• Understanding program behavior under worst-case load is critical for avoiding unexpected/ buggy program operation
Performance Inputs

- Understanding program behavior under worst-case load is critical for avoiding unexpected/buggy program operation.

Test Environment

Small scale Inputs → Expected program behavior

Deployed in actual environment

Large scale worst-case inputs → Program

- Long runtimes
  - Unnecessary/unexpected resource consumption
- Scale dependent bugs
### Why Performance Inputs?

- Algorithmic complexity attacks

| CVE-2012-3398 | Algorithmic complexity vulnerability in Moodle 1.9.x before 1.9.19, 2.0.x before 2.0.10, 2.1.x before 2.1.7, and 2.2.x before 2.2.4 allows remote authenticated users to cause a denial of service (CPU consumption) by using the advanced-search feature on a database activity that has many records. |
| CVE-2012-3287 | Poul-Henning Kemp md5crypt has insufficient algorithmic complexity and a consequently short runtime, which makes it easier for context-dependent attackers to discover cleartext passwords via a brute-force attack, as demonstrated by an attack using GPU hardware. |
| CVE-2012-2739 | Oracle Java SE before 7 Update 6, and OpenJDK 7 before 7u6 builds 2 and 8 before build 39, computes hash values without restricting the ability to trigger hash collisions predictably, which allows context-dependent attackers to cause a denial of service (CPU consumption) via crafted input to an application that maintains a hash table. |
| CVE-2012-2098 | Algorithmic complexity vulnerability in the sorting algorithms in the library compressing stream (BZip2CompressorOutputStream) in Apache Commons Compress before 1.4.1 allows remote attackers to cause a denial of service (CPU consumption) via a file with many repeating inputs. |
| CVE-2012-1588 | Algorithmic complexity vulnerability in the filterわかる function in the text filtering system (modules/filter/filter.module) in Drupal 7.x before 7.14 allows remote authenticated users with certain roles to cause a denial of service (CPU consumption) via a long email address. |
| CVE-2012-1150 | Python before 2.6.8, 2.7.x before 2.7.3, 3.x before 3.1.5, and 3.2.x before 3.2.3 computes hash values without restricting the ability to trigger hash collisions predictably, which allows context-dependent attackers to cause a denial of service (CPU consumption) via crafted input to an application that maintains a hash table. |
| CVE-2012-1035 | AdaCore Ada Web Services (AWS) before 2.10.2 computes hash values for form parameters without restricting the ability to trigger hash collisions predictably, which allows remote attackers to cause a denial of service (CPU consumption) by sending many crafted parameters. |
Why Performance Inputs?

• Some bugs manifests only in large scale (e.g. “Integer overflow bugs”)

  • Performance bug in one version of the parallel program library MPICH2 caused by a integer overflow
  • Bug manifests only when the parallel application works with massive amounts of data and processes

Source:
https://lists.mpich.org/pipermail/discuss/2015-October/004193.html

Correctness Testing is not enough!

Performance Testing at large scale is important to identify these bugs
Goal of XSTRESSOR

- Automatically generate large-scale performance inputs for programs with loops (worst-case inputs)

```
1 void insertion_sort(int* arr, int len){
2     int i = 1;
3     while(i < len){
4         int x = arr[i];
5         int j = i - 1;
6         while(j >= 0 && arr[j] > x){
7             arr[j+1] = arr[j];
8             j--;
9         }
10     } 
11     arr[j+1] = x;
12     i++;
13 } 
```
Goal of XSTRESSOR

• Automatically generate large-scale performance inputs for programs with loops (worst-case inputs)

```c
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
    arr[j+1] = x;
    i++;
}
```

Worst case at input size 10

10,9,8,7,6,5,4,3,2,1

Worst case at input size 1000

1000,999,..............3,2,1

Do this faster and more efficiently than existing techniques
Related Work

Fuzzing based approaches


Symbolic execution based approaches


Program

Feedback directed mutational fuzzing

Performance test suite

Seed inputs

No guarantee on finding the optimal worst-case input

Still relies on symbolic execution at large scale

Symbolic execution at small scale

Worst-case branch policy

Large scale worst-case input

Program

Related Work

Fuzzing based approaches


Symbolic execution based approaches


Can we avoid the scalability bottlenecks and still have the benefits of symbolic execution for generating worst-case inputs at scale?

No guarantee on finding the optimal worst-case input

Still relies on symbolic execution at large scale

XSTRESSOR Approach

Identify the hot loops
Symbolic execution in small scale Scale 1,2,...M
Worst-case path conditions [scale 1,2,...,M]

Large scale worst-case input at scale N (N>>M)

Worst-case path condition generator

SMT solver

Scale N
XSTRESSOR Approach

Model building phase

1. Identify the hot loops
2. Symbolic execution in small scale (Scale 1,2, ... M)
3. Worst-case path conditions (scale 1,2, ..., M)

Prediction phase

- Large scale worst-case input at scale N (N>>M)
- SMT solver
- Worst-case path condition generator
- Scale N

Does not require symbolic Execution at large scale
Insertion sort example

```c
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

True branch is taken in all iterations of inner loop

Symbolic computation tree for input scale 4
Insertion sort example

```c
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

Worst-case path condition input scale 4

\[ (arr[0] > arr[1]) \land \\
(arr[1] > arr[2]) \land \\
(\text{arr[0]} > \text{arr[2]}) \land \\
(\text{arr[2]} > \text{arr[3]}) \land \\
(\text{arr[1]} > \text{arr[3]}) \land \\
(\text{arr[0]} > \text{arr[3]}) \]

True branch is taken in all iterations of inner loop
Insertion sort example

```c
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

True branch is taken in all iterations of inner loop


Worst-case path condition input scale 5
Insertion sort example

```
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

True branch is taken in all iterations of inner loop

Worst-case path condition input scale 8

\[
\begin{align*}
(\text{arr}[0] &> \text{arr}[1]) \land (\text{arr}[6] > \text{arr}[7]) \\
(\text{arr}[1] &> \text{arr}[2]) \land (\text{arr}[5] > \text{arr}[7]) \\
(\text{arr}[0] &> \text{arr}[2]) \land (\text{arr}[4] > \text{arr}[7]) \\
(\text{arr}[2] &> \text{arr}[3]) \land (\text{arr}[3] > \text{arr}[7]) \\
(\text{arr}[1] &> \text{arr}[3]) \land (\text{arr}[2] > \text{arr}[7]) \\
(\text{arr}[0] &> \text{arr}[3]) \land (\text{arr}[1] > \text{arr}[7]) \\
(\text{arr}[3] &> \text{arr}[4]) \land (\text{arr}[0] > \text{arr}[7]) \\
(\text{arr}[2] &> \text{arr}[4]) \land (\text{arr}[7] > \text{arr}[8]) \\
(\text{arr}[1] &> \text{arr}[4]) \land (\text{arr}[6] > \text{arr}[8]) \\
(\text{arr}[0] &> \text{arr}[4]) \land (\text{arr}[5] > \text{arr}[8]) \\
(\text{arr}[4] &> \text{arr}[5]) \land (\text{arr}[4] > \text{arr}[8]) \\
(\text{arr}[3] &> \text{arr}[5]) \land (\text{arr}[3] > \text{arr}[8]) \\
(\text{arr}[2] &> \text{arr}[5]) \land (\text{arr}[2] > \text{arr}[8]) \\
(\text{arr}[1] &> \text{arr}[5]) \land (\text{arr}[1] > \text{arr}[8]) \\
(\text{arr}[0] &> \text{arr}[5]) \land (\text{arr}[0] > \text{arr}[8]) \\
(\text{arr}[5] &> \text{arr}[6]) \land (\text{arr}[5] > \text{arr}[6]) \\
(\text{arr}[4] &> \text{arr}[6]) \land (\text{arr}[3] > \text{arr}[6]) \\
(\text{arr}[2] &> \text{arr}[6]) \land (\text{arr}[1] > \text{arr}[6]) \\
(\text{arr}[0] &> \text{arr}[6]) \land (\text{arr}[0] > \text{arr}[6]) \\
(\text{arr}[7] &> \text{arr}[8]) \land (\text{arr}[7] > \text{arr}[8])
\end{align*}
\]
Insertion sort example

```
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 & arr[j] > x {  
            arr[j+1] = arr[j];
            j--;
    }
}
```

Can we identify the pattern and give a parametric representation for these constraints?

True branch is taken in all iterations of inner loop

Worst-case path condition input scale 8
**Insertion sort example**

```c
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

- **Inner loop induction variable:** `i`
- **Outer loop induction variable:** `i`
- **Worst-case path condition at input scale 4:**

- Variation of `j` sequence:
  - `[0],[1,0],[2,1,0]`
- Variation of `i` sequence:
  - `[1],[2,2],[3,3,3]`

"[" and "]" represent loop entry and exit.
Induction variable sequences

• Nested loops generate induction variable sequences with nested structure. Complex sequences are a combination of simpler sequences.

  Variation of j
  0,1,0,2,1,0

  Apply loop boundaries

  Variation of j
  [0],[1,0],[2,1,0]

• Simpler sequences fall into two categories

  **Increment sequences**
  • Variable is incremented/ decremented by a fixed amount
  • Example:
    [0,1,2,3,4,5]
  • Parameterized initial value, final value and a step

  **Constant sequences**
  • Variable remains constant
  • Example:
    [2,2,2,2]
  • Parameterized by a constant, no of repetitions
Induction variable sequences

• Nested loops generate induction variable sequences with nested structure. Complex sequences are a combination of simpler sequences

  Variation of $j$
  0,1,0,2,1,0

  Variation of $j$
  [0],[1,0],[2,1,0]

• Simpler sequences fall into two categories

  **Increment sequences**
  • Variable is incremented/ decremented by a fixed amount
  • Example:
    [0,1,2,3,4,5]
  • Parameterized initial value, final value and a step

  **Constant sequences**
  • Variable remains constant
  • Example:
    [2,2,2,2,2]
  • Parameterized by a constant, no of repetitions

XSTRESSOR uses a context free grammar to describe these sequences

**Induction variable sequence generators (ISG)**
Context Free Grammar to describe ISGs

\[ P \rightarrow I | C \]
\[ I \rightarrow \text{incre}(X, X, d) \]
\[ C \rightarrow \text{const}(X, X) \]
\[ X \rightarrow P | x \]

X – sequence of integers / integer
I – increment sequence
C – constant sequence
Context Free Grammar to describe ISGs

X – sequence of integers / integer
I – increment sequence
C – constant sequence

- $\text{inc}r$ function

Initial value
Final value

\[ [0,1,2,3,4] \rightarrow \text{inc}r(0,4,1) \]

- This is analogous to some variable incremented by a fixed amount inside a loop
Context Free Grammar to describe ISGs

\[
P \rightarrow I|C \\
I \rightarrow incre(X, X, d) \\
C \rightarrow const(X, X) \\
X \rightarrow P|x
\]

- $X$ – sequence of integers / integer
- $I$ – increment sequence
- $C$ – constant sequence

- \textit{const} function

\begin{align*}
\text{e.g. :} & \\
[2, 2, 2, 2] & \rightarrow \text{const}(2, 4)
\end{align*}

- This is analogous to a variable that remains constant in some number of iterations of a loop
Context Free Grammar to describe ISGs

\[ P \rightarrow I | C \]
\[ I \rightarrow \text{incre}(X, X, d) \]
\[ C \rightarrow \text{const}(X, X) \]
\[ X \rightarrow P | x \]

- Sequences itself can be arguments to \textit{const} and \textit{incre} functions

\[ \text{const}([0,1,2],[2,2,2]) \rightarrow \text{const}(0,2) \oplus \text{const}(1,2) \oplus \text{const}(2,2) \]

- This can represent the induction variable sequences generated by nested loops

\(\oplus\) - concatenation operator
How to construct an ISG

\[ P \rightarrow I | C \]
\[ I \rightarrow \text{incre}(X, X, d) \]
\[ C \rightarrow \text{const}(X, X) \]
\[ X \rightarrow P | x \]
How to construct an ISG

\[ P \rightarrow I|C \]
\[ I \rightarrow \text{incre}(X, X, d) \]
\[ C \rightarrow \text{const}(X, X) \]
\[ X \rightarrow P|x \]

\[ [0, 1, 0, 2, 1, 0, 3, 2, 1, 0] \]
\[ \text{incre}(0, 0, -1) \oplus \text{incre}(1, 0, -1) \oplus \text{incre}(2, 0, -1) \oplus \text{incre}(3, 0, -1) \]

\( \oplus \) - concatenation operator
How to construct an ISG

\[
P \rightarrow I | C \\
I \rightarrow \text{inc}(X, X, d) \\
C \rightarrow \text{const}(X, X) \\
X \rightarrow P | x
\]

\[\begin{align*}
[0], [1, 0], [2, 1, 0], [3, 2, 1, 0] \\
\text{inc}(0, 0, -1) \oplus \text{inc}(1, 0, -1) \oplus \text{inc}(2, 0, -1) \oplus \text{inc}(3, 0, -1) \\
\text{inc}([0, 1, 2, 3], [0, 0, 0, 0], -1)
\end{align*}\]

\(\oplus\) - concatenation operator
How to construct an ISG

\[
P \rightarrow I|C \\
I \rightarrow \text{incre}(X, X, d) \\
C \rightarrow \text{const}(X, X) \\
X \rightarrow P|x
\]

\[\oplus - \text{concatenation operator}\]

\[
[0],[1,0],[2,1,0],[3,2,1,0] \\
\text{incre}(0,0,-1) \oplus \text{incre}(1,0,-1) \oplus \text{incre}(2,0,-1) \oplus \text{incre}(3,0,-1) \\
\text{incre}([0,1,2,3],[0,0,0,0],-1) \\
\text{incre(\text{incre}(0,3,1), \text{const}(0,4),-1)}
\]

Induction variable sequence generator (ISG)
Back to motivating example

Variable “j”

True branch is taken in all iterations of inner loop

```
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

arr[j] > arr[i]

[0],[1,0],[2,1,0]

Scale 4

`incre( incre(0, 2, 1), const(0, 3), -1)```
Back to motivating example

True branch is taken in all iterations of inner loop

```c
void insertion_sort(int* arr, int len){
    int i = 0;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

Variable “j”

- $[0],[1,0],[2,1,0]$
  - Scale 4
  - incr( incr(0, 2, 1), const(0, 3), -1)
- $[0],[1,0],[2,1,0],[3,2,1,0]$
  - Scale 5
  - incr( incr(0, 3, 1), const(0, 4), -1)

$\text{arr}[j] > \text{arr}[i]$
Back to motivating example

A general ISG is learned using model fitting e.g. : polynomial model fitting

```c
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

Variable “j”

- True branch is taken in all iterations of inner loop

- **Scale 4**
  - $[0],[1,0],[2,1,0]$
  - $\text{incre( incre(0, 2, 1), const(0, 3), −1)}$

- **Scale 5**
  - $[0],[1,0],[2,1,0],[3,2,1,0]$
  - $\text{incre( incre(0, 3, 1), const(0, 4), −1)}$

- **Scale N**
  - $[0],[1,0],...,[N-2,N-1,...,1,0]$
  - $\text{incre( incre(0,N-2, 1), const(0,N-1), −1)}$
Back to motivating example

```c
void insertion_sort(int* arr, int len){
    int i = 1;
    while(i < len){
        int x = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > x){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = x;
        i++;
    }
}
```

True branch is taken in all iterations of inner loop

- Variable “i”
  - Scale 4: \[\text{const(incre}(1, 3, 1), \text{incre}(1, 3, 1))\]
  - Scale 5: \[\text{const(incre}(1, 4, 1), \text{incre}(1, 4, 1))\]
  - Scale N: \[\text{const(incre}(1, N - 1, 1), \text{incre}(1, N - 1, 1))\]

General ISG for variable “i”
Inferring conditions at large scale

\[ \text{arr}[j] > \text{arr}[i] \]

incre( incre(0,N−2, 1), const(0,N−1), −1)  
const(incre(1,N − 1, 1), incre(1,N − 1, 1))

G1

G2
Inferring conditions at large scale

For any scale $K$, $G1(K)$ and $G2(K)$

Worst case conditions for scale $K$
Evaluation Setup

- Compared against WISE, SPF-WCA
- 7 Micro benchmarks
- 2 case studies (*GNU grep* , *GNU cmp*)
- Various input scales
- 12 hours

Machine specs
- 8-core Intel Xeon 2.7 GHz 20MB L3 cache
- 192 GB ram
## Evaluation – micro benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (seconds)</th>
<th>Input scale 50</th>
<th>Time (seconds)</th>
<th>Input scale 500</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XSTRESSOR</td>
<td>WISE</td>
<td>SPF-WCA</td>
<td>XSTRESSOR</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>1.85</td>
<td>49.57</td>
<td>72.86</td>
<td>96.73</td>
</tr>
<tr>
<td>Sorted list (insert)</td>
<td>1.65</td>
<td>62.39</td>
<td>86.88</td>
<td>1.79</td>
</tr>
<tr>
<td>Merge sorted lists</td>
<td>2.16</td>
<td>OOT</td>
<td>29.96</td>
<td>2.57</td>
</tr>
<tr>
<td>Binary tree (search)</td>
<td>3.97</td>
<td>56.29</td>
<td>237.67</td>
<td>103.74</td>
</tr>
<tr>
<td>Dijkstra’s</td>
<td>6.32</td>
<td>9.68</td>
<td>1714.26</td>
<td>12830</td>
</tr>
<tr>
<td>Boolean matrix multiplication</td>
<td>107.87</td>
<td>960.03</td>
<td>OOT</td>
<td>OOT</td>
</tr>
<tr>
<td>Traveling salesman</td>
<td>OOT</td>
<td>OOM</td>
<td>OOT</td>
<td>OOT</td>
</tr>
</tbody>
</table>

XSTRESSOR can generate the worst-case inputs within seconds for most benchmarks.
# Evaluation – Time spent in each phase

<table>
<thead>
<tr>
<th>Program</th>
<th>Time statistic</th>
<th>Scales</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>Model building</td>
<td>9.28</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.10</td>
</tr>
<tr>
<td>Sorted list Insert</td>
<td>Model building</td>
<td>4.67</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.01</td>
</tr>
<tr>
<td>Merging sorted arrays</td>
<td>Model building</td>
<td>2.08</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.03</td>
</tr>
<tr>
<td>Binary tree search</td>
<td>Model building</td>
<td>10.98</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.10</td>
</tr>
<tr>
<td>Dijkstra’s</td>
<td>Model building</td>
<td>25.53</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.26</td>
</tr>
<tr>
<td>Boolean matrix multiplication</td>
<td>Model building</td>
<td>13.68</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>45.57</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>5.96</td>
</tr>
</tbody>
</table>

- Time spent in model building is a one-time thing
- Time spent in path prediction and solving the paths constraints increases with the input scale
Evaluation – case studies

- All three techniques perform well *in GNU cmp*
- For *GNU grep* WISE, SPF-WCA runs out of time,
  - Worst-case branch behavior is scale-dependent (take TRUE branch after taking \((N-1)\) FALSE branches)
  - XSTRESSOR’s ISGs are capable of capturing such behavior

<table>
<thead>
<tr>
<th>Application</th>
<th>Model building time (seconds)</th>
<th>Prediction time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
<td>I</td>
</tr>
<tr>
<td>GNU cmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNU grep</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

W – WISE  
I – SPF-WCA  
X – XSTRESSOR
Conclusion

• Complexity Testing in large scale is essential for resolving performance problems and algorithmic complexity attacks.

• XSTRESSOR avoids the drawbacks of existing white-box techniques for complexity testing by directly predicting the worst-case path condition using "Path generators (ISGs)".

• XSTRESSOR overperforms the existing white-box techniques by a reasonable margin and also scale to large input scales.
THANK YOU