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

Test Environment

Small scale Inputs

Program

Expected program behavior

Performance Inputs

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

Test Environment

Small scale Inputs

Program

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 Kamp 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 build 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 bzip2 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_url 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.

Denial of Service Attacks using Crafted inputs

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

Integer overflow bug in Boeing 737 software (2015)
Source : www.theguardian.com

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)

```c
void inserion_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++;
    }
}
```

XSTRESSOR

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

Goal of XSTRESSOR

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

```c
void inserion_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++;
    }
}
```

XSTRESSOR

Worst case at input size 10

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

Worst case at input size 1000

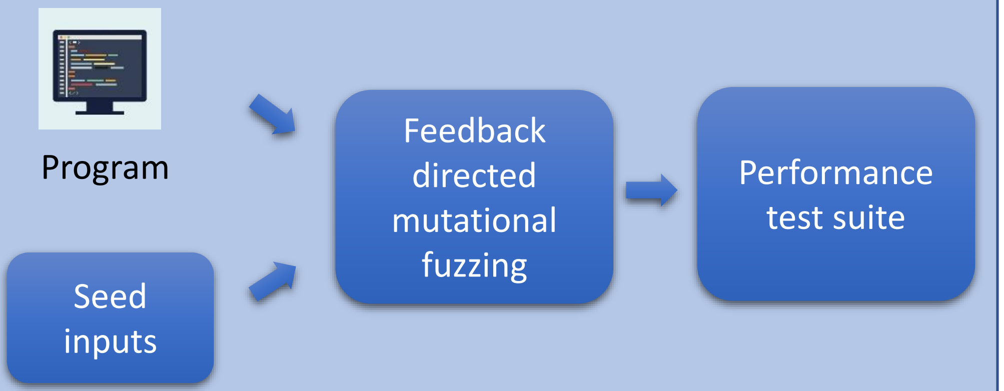
1000,999,…………..3,2,1

Do this faster and more efficiently than existing techniques

Related Work

Fuzzing based approaches

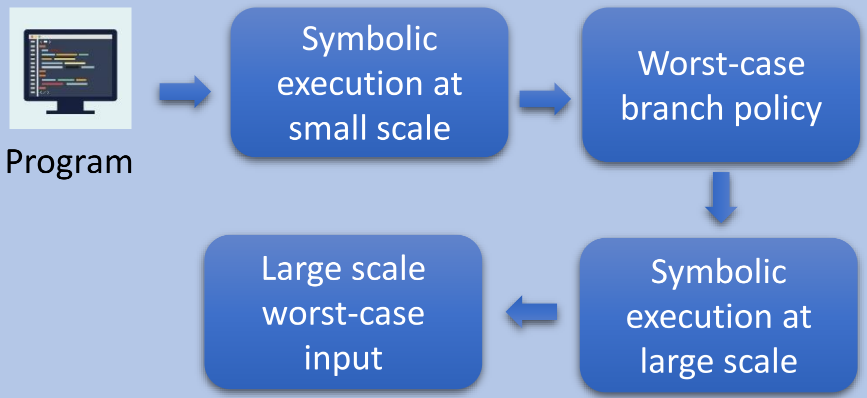
- SlowFuzz [3] – CCS 2017
- PerfFuzz [4] – ISSTA 2018



No guarantee on finding the optimal worst-case input

Symbolic execution based approaches

- WISE [1] – ICSE 2009
- SPF-WCA [2] – ICST 2017



Still relies on symbolic execution at large scale

[1] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. ICSE '09
[2] Kasper Luckow, Rody Kersten, and Corina Pasareanu. Symbolic complexity analysis using context-preserving histories. ICST'17
[3] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. CCS '17
[4] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: Automatically generating pathological inputs. ISSTA 2018

Related Work

Fuzzing based approaches

- SlowFuzz [3] – CCS 2017
- PerfFuzz [4] – ISSTA 2018

No guarantee on finding the optimal worst-case input

Symbolic execution based approaches

- WISE [1] – ICSE 2009
- SPF-WCA [2] – ICST 2017

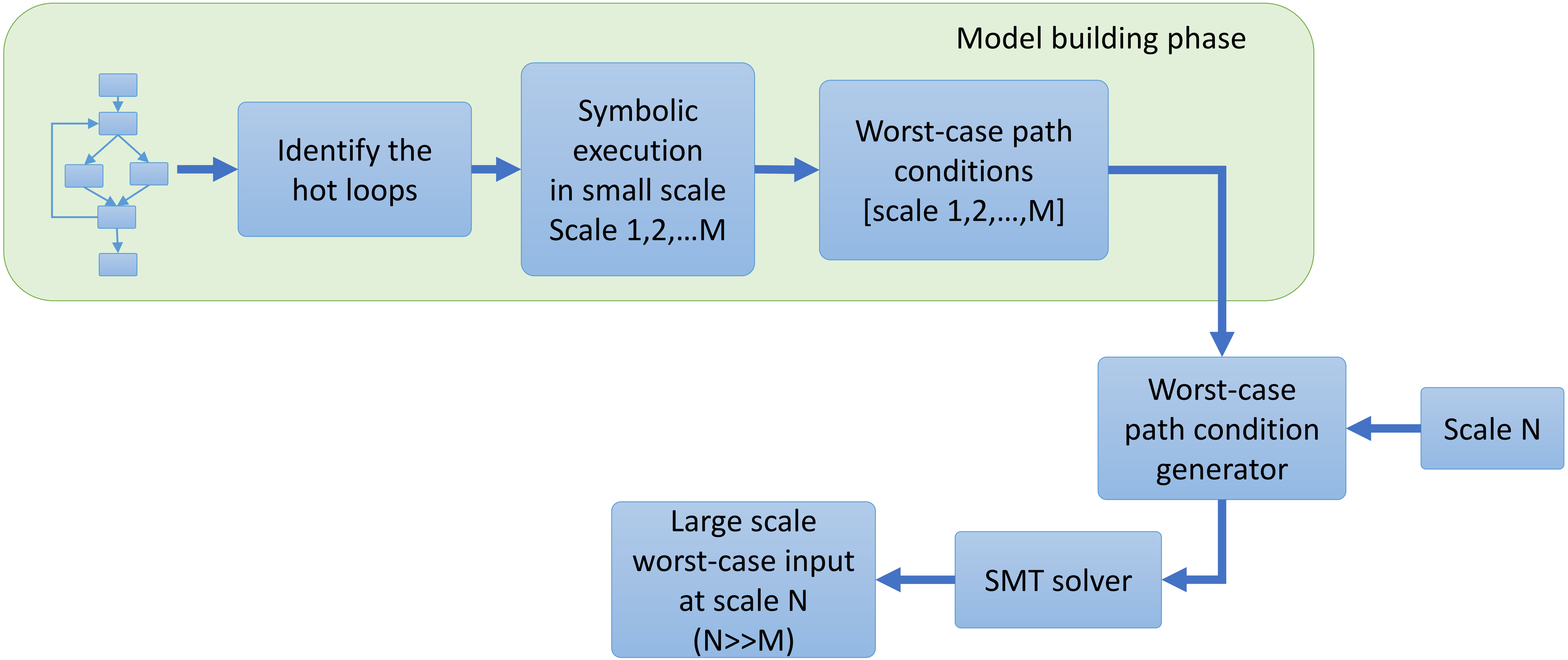
Symbolic

Still relies on symbolic execution at large scale

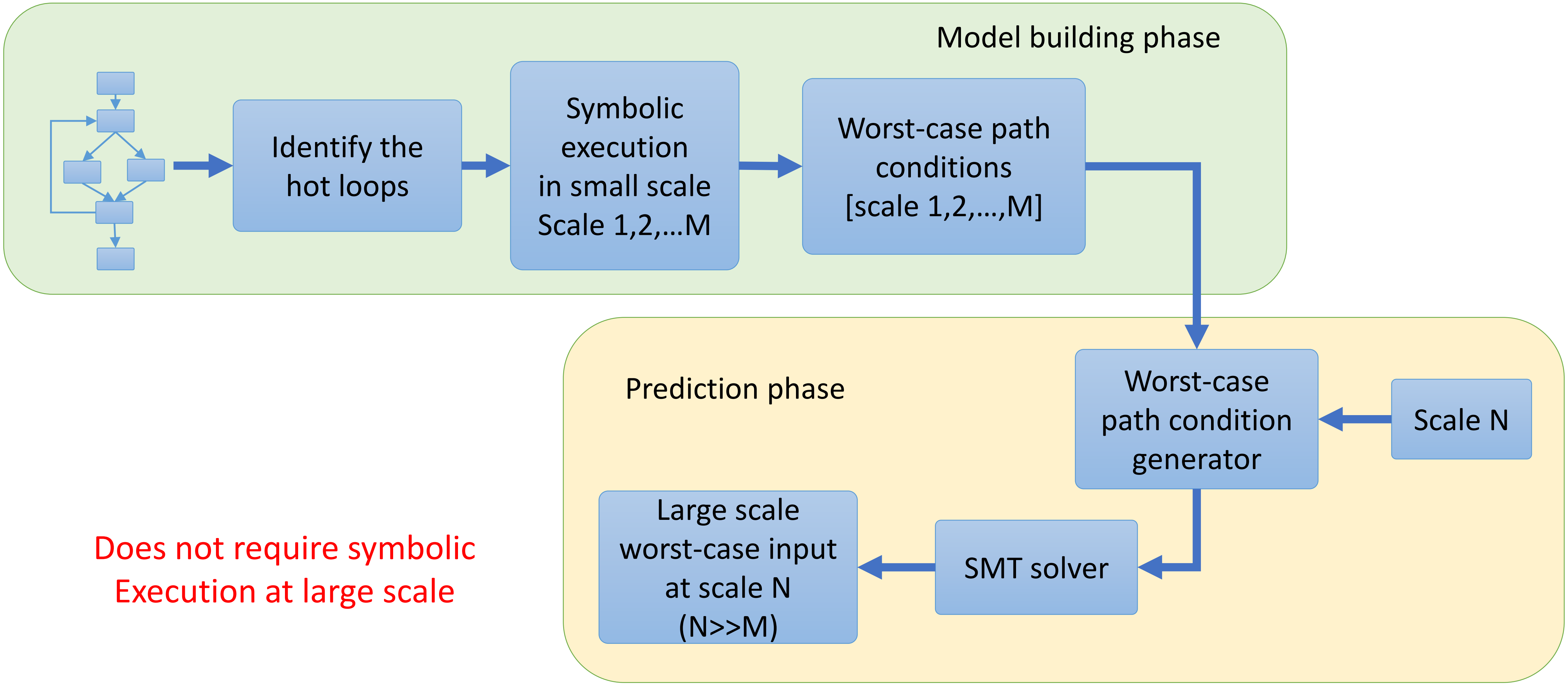
Can we avoid the scalability bottlenecks and still have the benefits of symbolic execution for generating worst-case inputs at scale?

[1] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. ICSE ’09
[2] Kasper Luckow, Rody Kersten, and Corina Pasareanu. Symbolic complexity analysis using context-preserving histories. ICST’17
[3] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. CCS ’17
[4] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: Automatically generating pathological inputs. ISSTA 2018

XSTRESSOR Approach



XSTRESSOR Approach



Does not require symbolic Execution at large scale

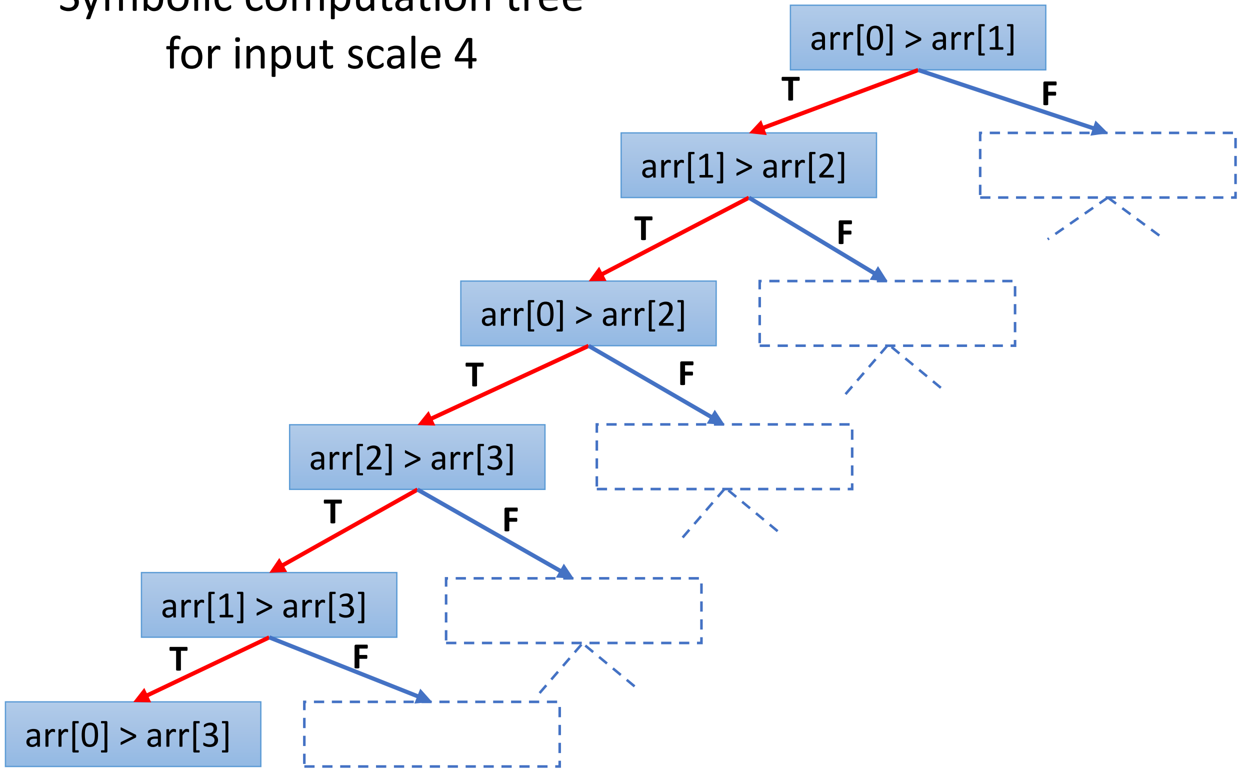
Insertion sort example

```c
void inserion_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 inserion_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

(arr[0] > arr[1])∧
(arr[1] > arr[2])∧
(arr[0] > arr[2])∧
(arr[2] > arr[3])∧
(arr[1] > arr[3])∧
(arr[0] > arr[3])

Worst-case path condition input scale 4

Insertion sort example

```c
void inserion_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

$$
\begin{aligned}
&(arr[0] > arr[1]) \land \\
&(arr[1] > arr[2]) \land \\
&(arr[0] > arr[2]) \land \\
&(arr[2] > arr[3]) \land \\
&(arr[1] > arr[3]) \land \\
&(arr[0] > arr[3]) \land \\
&(arr[3] > arr[4]) \land \\
&(arr[2] > arr[4]) \land \\
&(arr[1] > arr[4]) \land \\
&(arr[0] > arr[4])
\end{aligned}
$$

Worst-case path condition input scale 5

Insertion sort example

```c
void inserion_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

$(arr[0] > arr[1]) \land$
$(arr[1] > arr[2]) \land$
$(arr[0] > arr[2]) \land$
$(arr[2] > arr[3]) \land$
$(arr[1] > arr[3]) \land$
$(arr[0] > arr[3]) \land$
$(arr[3] > arr[4]) \land$
$(arr[2] > arr[4]) \land$
$(arr[1] > arr[4]) \land$
$(arr[0] > arr[4]) \land$
$(arr[4] > arr[5]) \land$
$(arr[3] > arr[5]) \land$
$(arr[2] > arr[5]) \land$
$(arr[1] > arr[5]) \land$
$(arr[0] > arr[5]) \land$
$(arr[5] > arr[6]) \land$
$(arr[4] > arr[6]) \land$
$(arr[3] > arr[6]) \land$
$(arr[2] > arr[6]) \land$
$(arr[1] > arr[6]) \land$
$(arr[0] > arr[6]) \land$
$(arr[6] > arr[7]) \land$
$(arr[5] > arr[7]) \land$
$(arr[4] > arr[7]) \land$
$(arr[3] > arr[7]) \land$
$(arr[2] > arr[7]) \land$
$(arr[1] > arr[7]) \land$
$(arr[0] > arr[7]) \land$
$(arr[7] > arr[8]) \land$
$(arr[6] > arr[8]) \land$
$(arr[5] > arr[8]) \land$
$(arr[4] > arr[8]) \land$
$(arr[3] > arr[8]) \land$
$(arr[2] > arr[8]) \land$
$(arr[1] > arr[8]) \land$
$(arr[0] > arr[8])$

Worst-case path condition input scale 8

Insertion sort example

```c
void inserion_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--;
```

True branch is taken in all iterations of inner loop

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

$(arr[0] > arr[1]) \land$
$(arr[1] > arr[2]) \land$
$(arr[0] > arr[2]) \land$
$(arr[2] > arr[3]) \land$
$(arr[1] > arr[3]) \land$
$(arr[0] > arr[3]) \land$
$(arr[3] > arr[4]) \land$
$(arr[2] > arr[4]) \land$
$(arr[1] > arr[4]) \land$
$(arr[0] > arr[4]) \land$
$(arr[4] > arr[5]) \land$
$(arr[3] > arr[5]) \land$
$(arr[2] > arr[5]) \land$
$(arr[2] > arr[6]) \land$
$(arr[1] > arr[6]) \land$
$(arr[0] > arr[6]) \land$

$(arr[6] > arr[7]) \land$
$(arr[5] > arr[7]) \land$
$(arr[4] > arr[7]) \land$
$(arr[3] > arr[7]) \land$
$(arr[2] > arr[7]) \land$
$(arr[1] > arr[7]) \land$
$(arr[0] > arr[7]) \land$
$(arr[7] > arr[8]) \land$
$(arr[6] > arr[8]) \land$
$(arr[5] > arr[8]) \land$
$(arr[4] > arr[8]) \land$
$(arr[3] > arr[8]) \land$
$(arr[2] > arr[8]) \land$

Worst-case path condition input scale 8

Insertion sort example

```c
void inserion_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

Inner loop induction variable

$$\mathrm{arr}[j] > \mathrm{arr}[i]$$

Outer loop induction variable

$$(\mathrm{arr}[0] > \mathrm{arr}[1]) \wedge$$
$$(\mathrm{arr}[1] > \mathrm{arr}[2]) \wedge$$
$$(\mathrm{arr}[0] > \mathrm{arr}[2]) \wedge$$
$$(\mathrm{arr}[2] > \mathrm{arr}[3]) \wedge$$
$$(\mathrm{arr}[1] > \mathrm{arr}[3]) \wedge$$
$$(\mathrm{arr}[0] > \mathrm{arr}[3])$$

Worst-case path condition at input scale 4

Inner loop induction variable → Variation of j [0],[1,0],[2,1,0]

Outer loop induction variable → Variation of i [1],[2,2],[3,3,3]

Induction variable sequences

"[" 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 :
  [2,2,2,2,2]
- 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	Apply loop boundaries →	Variation of j
0,1,0,2,1,0		[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

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

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

Context Free Grammar to describe ISGs

$$P \rightarrow I|C$$

$$\boxed{I \rightarrow incre(X, X, d)}$$

$$C \rightarrow const(X, X)$$

$$X \rightarrow P|x$$

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

- *incre* function

Initial value

step

e.g. :

[0,1,2,3,4] ⟶ *incre*(0,4,1)

Final value

- This is analogous to some variable incremented by a fixed amount inside a loop

Context Free Grammar to describe ISGs

$$P \rightarrow I|C$$
$$I \rightarrow incre(X, X, d)$$
$$\boxed{C \rightarrow const(X, X)}$$
$$X \rightarrow P|x$$

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

- *const* function

Const value

repetitions

e.g. :

[2,2,2,2] $\longrightarrow$ *const*(2,4)

- This is analogous to a variable that remains constant in some number of iterations of a loop

Context Free Grammar to describe ISGs

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

⊕ - concatenation operator

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

- Sequences itself can be arguments to *const* and *incre* functions

const([0,1,2],[2,2,2]) $\longrightarrow$ *const*(0,2) ⊕*const*(1,2) ⊕ *const*(2,2)

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

How to construct an ISG

$$P \rightarrow I|C$$

$$I \rightarrow incre(X, X, d)$$

$$C \rightarrow const(X, X)$$

$$X \rightarrow P|x$$

[0],[1,0],[2,1,0,],[3,2,1,0]

How to construct an ISG

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

[0],[1,0],[2,1,0,],[3,2,1,0]

$\downarrow$

incre(0,0,-1) ⊕ incre(1,0,-1) ⊕ incre(2,0,-1) ⊕ incre(3,0,-1)

⊕ - concatenation operator

How to construct an ISG

$$P \rightarrow I|C$$

$$I \rightarrow incre(X, X, d)$$

$$C \rightarrow const(X, X)$$

$$X \rightarrow P|x$$

[0],[1,0],[2,1,0,],[3,2,1,0]

$\downarrow$

incre(0,0,-1) ⊕ incre(1,0,-1) ⊕ incre(2,0,-1) ⊕ incre(3,0,-1)

$\downarrow$

incre([0,1,2,3], [0,0,0,0], -1)

⊕ - concatenation operator

How to construct an ISG

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

[0],[1,0],[2,1,0,],[3,2,1,0]

↓

incre(0,0,-1) ⊕ incre(1,0,-1) ⊕ incre(2,0,-1) ⊕ incre(3,0,-1)

↓

incre([0,1,2,3], [0,0,0,0], -1)

↓

incre(incre(0,3,1), const(0,4), -1)

Induction variable sequence generator (ISG)

⊕ - concatenation operator

Back to motivating example

```
void inserion_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

arr$\boxed{[j]}$ > arr[i]

Variable "j"

[0],[1,0],[2,1,0] →

Scale 4

incre(incre(0, 2, 1), const(0, 3), −1)

Back to motivating example

```
void inserion_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

arr[j] > arr[i]

Variable "j"

Scale 4

[0],[1,0],[2,1,0] → incre(incre(0, 2, 1), const(0, 3), −1)

Scale 5

[0],[1,0],[2,1,0],[3,2,1,0] → incre(incre(0, 3, 1), const(0, 4), −1)

Back to motivating example

```c
void inserion_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

arr[j] > arr[i]

Variable "j"

Scale 4

[0],[1,0],[2,1,0] → incre(incre(0, 2, 1), const(0, 3), −1)

Scale 5

[0],[1,0],[2,1,0],[3,2,1,0] → incre(incre(0, 3, 1), const(0, 4), −1)

Scale N

[0],[1,0],....[N-2,N-1,....,1,0] → incre(incre(0,N−2, 1), const(0,N−1), −1)

A general ISG is learned using model fitting

e.g. : polynomial model fitting

Back to motivating example

```
void inserion_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

arr[j] > arr[i]

Variable "i"

Scale 4

[1],[2,2],[3,3,3] → const(incre(1, 3, 1), incre(1, 3, 1))

Scale 5

[1],[2,2],[3,3,3],[4,4,4,4] → const(incre(1, 4, 1), incre(1, 4, 1))

Scale N

[1],[2,2],....[N-1,N-1,....,N-1] → const(incre(1,N – 1, 1), incre(1,N – 1, 1))

General ISG for variable "i"

Inferring conditions at large scale

$arr\boxed{[j]} > arr\boxed{[i]}$

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

G1

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

G2

Inferring conditions at large scale

arr$\boxed{[j]}$ > arr$\boxed{[i]}$

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

G1

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

G2

For any scale K,

G1(K)

G2(K)

arr[j] > arr[i]

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

Input scale 50			
Benchmark	Time (seconds)		
	XSTRESSOR	WISE	SPF-WCA
Insertion sort	1.85	49.57	72.86
Sorted list (insert)	1.65	62.39	86.88
Merge sorted lists	2.16	OOT	29.96
Binary tree (search)	3.97	56.29	237.67
Dijkstra's	6.32	9.68	1714.26
Boolean matrix multiplication	107.87	960.03	OOT
Traveling salesman	OOT	OOM	OOT

Input scale 500			
Benchmark	Time (seconds)		
	XSTRESSOR	WISE	SPF-WCA
Insertion sort	96.73	OOT	OOM
Sorted list (insert)	1.79	OOT	OOM
Merge sorted lists	2.57	OOT	46.62
Binary tree (search)	103.74	OOT	OOM
Dijkstra's	12830	3099.41	OOT
Boolean matrix multiplication	OOT	OOM	OOT
Traveling salesman	OOT	OOT	OOT

XSTRESSOR can generate the worst-case inputs within seconds for most benchmarks

Evaluation – Time spent in each phase

Program	Time statistic	Scales		
		40	50	100
Insertion sort	Model building	9.28	9.28	9.28
	Path prediction	0.52	0.66	2.01
	Solver	0.10	0.15	0.84
Sorted list Insert	Model building	4.67	4.67	4.67
	Path prediction	0.01	0.02	0.03
	Solver	0.01	0.01	0.02
Merging sorted arrays	Model building	2.08	2.08	2.08
	Path prediction	0.04	0.06	0.13
	Solver	0.03	0.03	0.06
Binary tree search	Model building	10.98	10.98	10.98
	Path prediction	0.33	0.49	1.90
	Solver	0.10	0.16	0.88
Dijkstra's	Model building	25.53	25.53	25.53
	Path prediction	1.92	3.13	17.52
	Solver	0.26	0.41	1.95
Boolean matrix multiplication	Model building	13.68	13.68	13.68
	Path prediction	45.57	92.92	1102.13
	Solver	5.96	14.41	69.32

- 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

W – WISE
I – SPF-WCA
X – XSTRESSOR

Application	Model building time (seconds)			Prediction time(seconds)								
				50			100			500		
	W	I	X	W	I	X	W	I	X	W	I	X
GNU cmp	2.98	1.72	4.234	1.40	1.81	1.86	3.31	1.75	4.07	41.24	2.49	26.77
GNU grep	16.99	OOT	22.70	OOT	OOT	96.54	OOT	OOT	674.27	OOT	OOT	29825.23

- 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

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