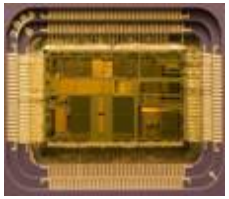
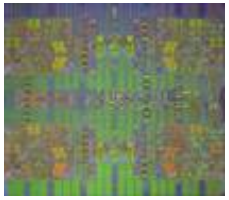
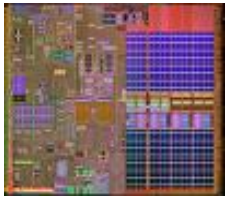
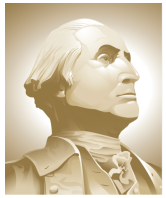


THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC


StatSym: Vulnerable Path Discovery through Statistics-Guided Symbolic Execution

Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue,

Tian Lan and Guru Venkataramani

Department of Electrical and Computer Engineering

The George Washington University

DSN 2017, Denver, Colorado

Outline

- Background
- Motivation
- *StatSym* Design
- *StatSym* Evaluation
- Case Study and Sensitivity Study
- Conclusion

Background

- **Software applications are growing rapidly**
  - Diversified application domains

  - Tremendous complexity in code base

Many man-months

Big SLOCs

Background – Cont.

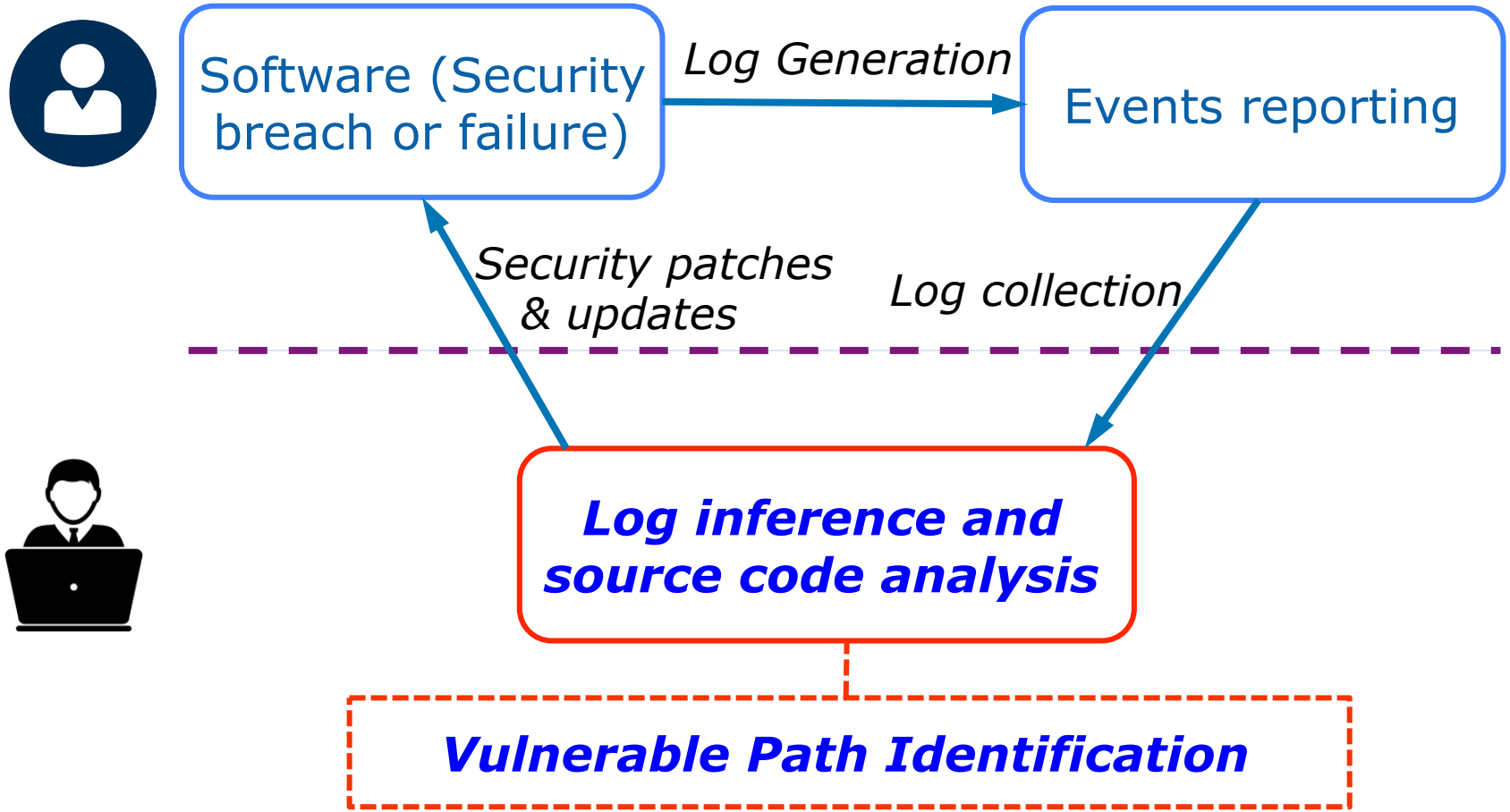
- **Software vulnerabilities are common**
  - 5~10 bug per 1000 lines of code
  - Number of Vulnerabilities increased 39% in the past 5 years[1]

- **Security breaches could be costly**
  - Recent security breach: WannaCry 2017[2]

Software vulnerability diagnosis is critical

1. FLEXERA, Vulnerability Review 2016, http://resources.flexerasoftware.com
2. WannaCry attack, Wikipedia, https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

Attack/Failure Diagnosis



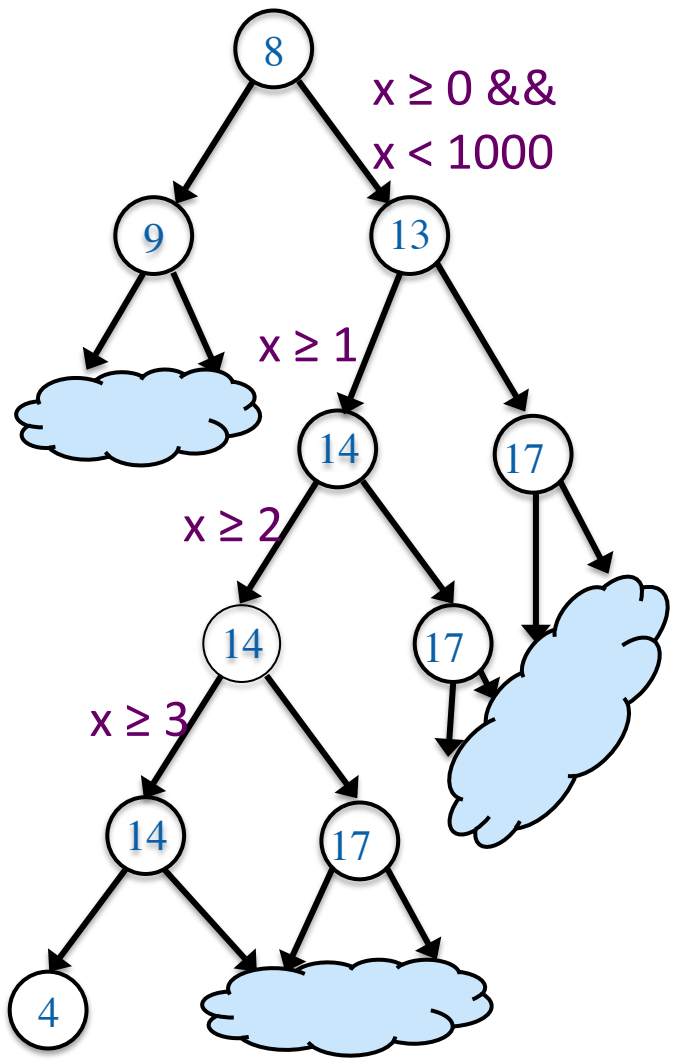
- **Vulnerable Path Identification is difficult**
  - Often involve substantial amount of manual efforts

Vulnerable Path Discovery Techniques

- **Statistical methods (*statistical debugging*)**
  - *E.g., Xu et al., DSN'16; Liblit et al., ICSE'09*
  - ➕ Fast
  - ➖ Maybe inaccurate
- **Formal method (*symbolic execution*)**
  - *E.g., KLEE, OSDI'08; S^2E ASPLOS'11*
  - ➕ Complete and accurate
  - ➖ Non-scalable due to state explosion
- **Combining statistical and formal method**
  - Statistics-guided symbolic execution

Motivational Example

```
void vulnerable(int a){ 1
   //...                 2
  if( a >=3 )            3
    vulnerability;       4
}                        5
                         6
void main(int x) {       7
    if(x>=1000 || x<0){  8
      ...                9
    }                    10
   else{                 11
     int i = 0;          12
     while(i<x){         13
       vulnerable(i);    14
       i++;              15
     }                   16
     ...                 17
  }                      18
}                        19
```

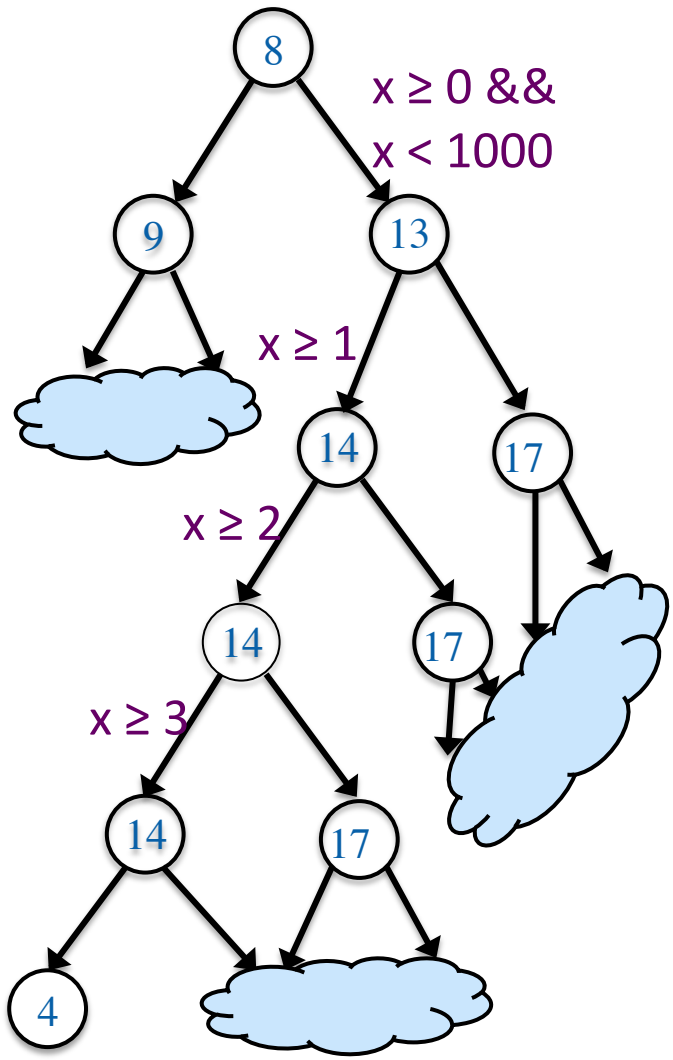


Pure Symbolic Execution:

Huge space has to be explored

Motivational Example-Cont.

```
void vulnerable(int a){ 1
   //...                 2
  if( a >=3 )            3
    vulnerability;       4
}                        5
                         6
void main(int x) {       7
    if(x>=1000 || x<0){  8
      ...                9
    }                    10
   else{                 11
     int i = 0;          12
     while(i<x){         13
       vulnerable(i);    14
       i++;              15
     }                   16
     ...                 17
  }                      18
}                        19
```



Statistical information:

Path:8 -> 14 -> 4

Condition: x>=3 at line 14

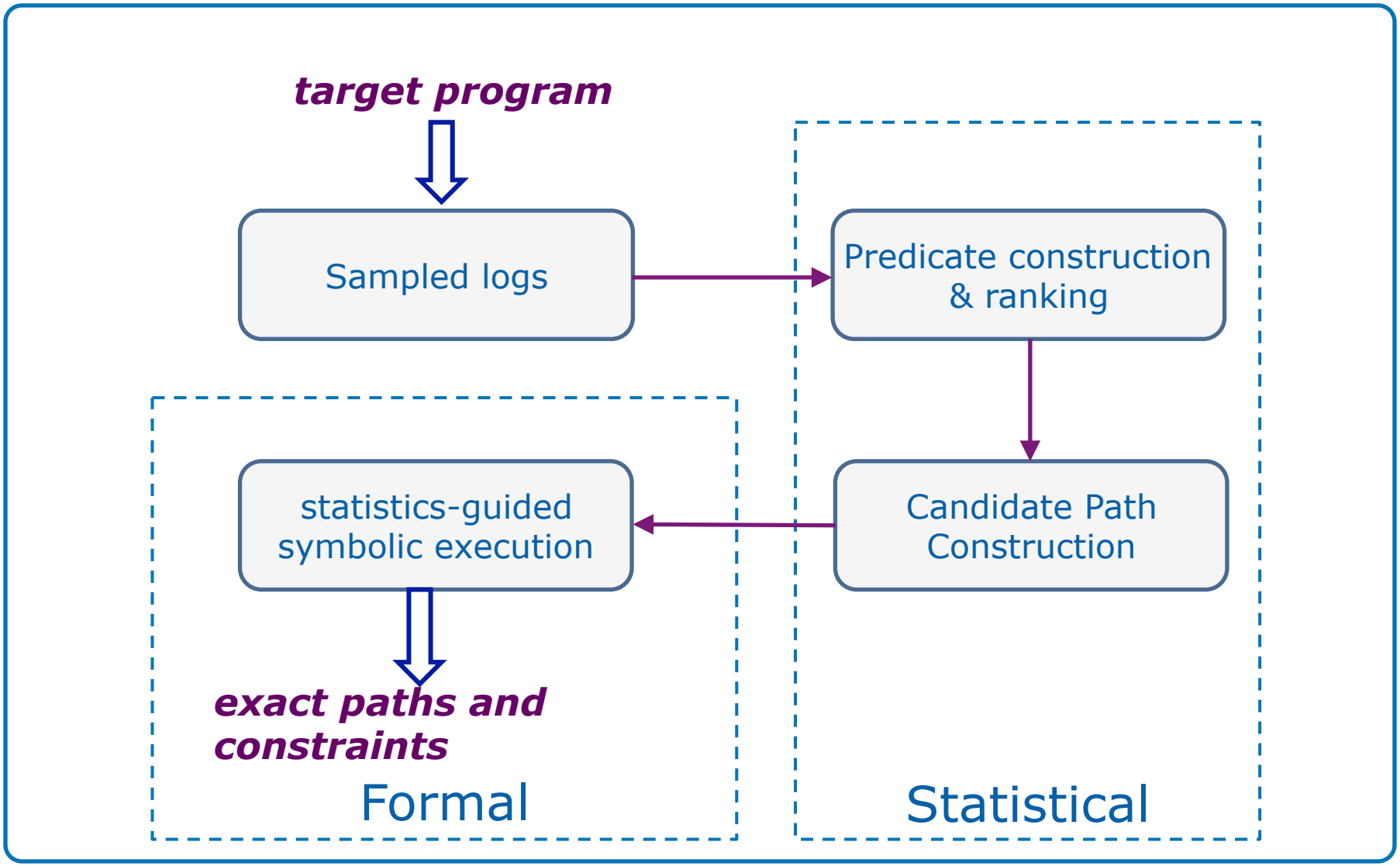
Symbolic execution with statistical guidance:

Search space is considerable pruned

Observations

- **Explosion of program states in symbolic execution**
  - ↗Exhaustive search of all paths (branches, loops, functions)
- **Runtime statistics can *guide* symbolic execution**
  - ↗Statistical paths trim off unnecessary sub-tree of state
  - ↗Predicates further reduce uninteresting search iterations

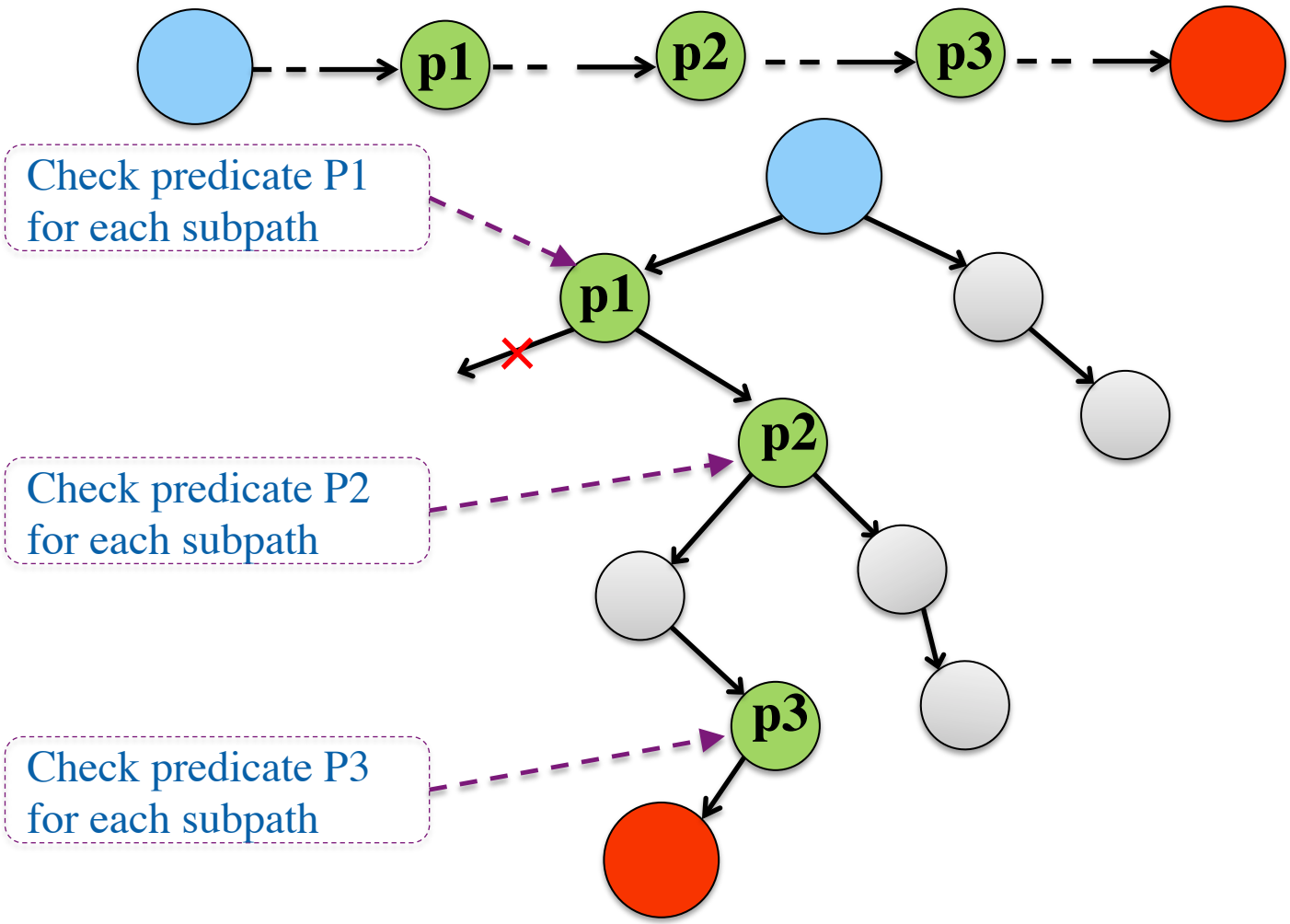
StatSym: Statistics-guided Symbolic Execution



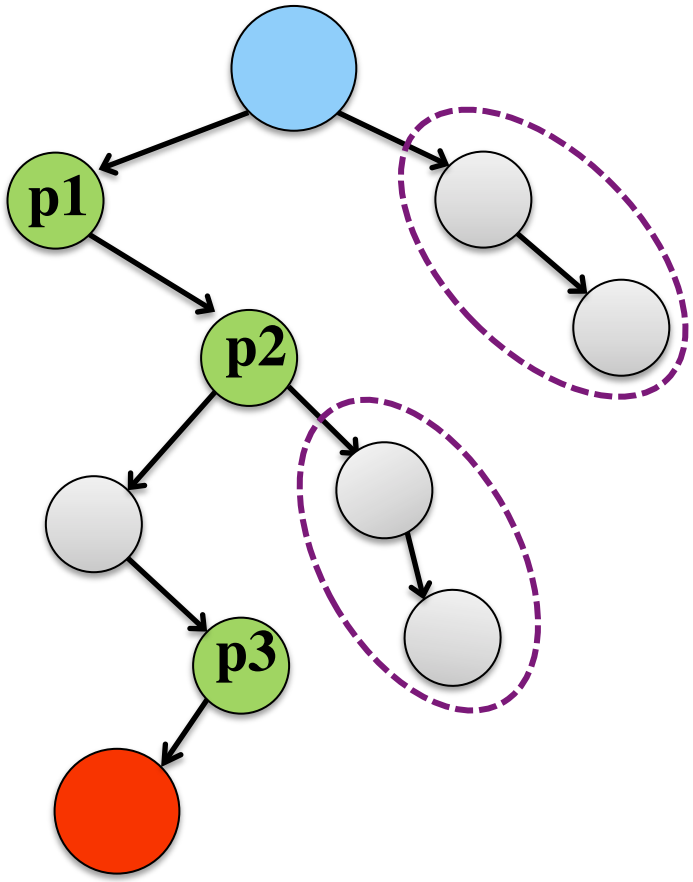
StatSym Design – Statistical Analysis

- **Predicate construction**
  - Construct predicate based on divergence between the value ranges
    for variable a, faulty run value set F, correct run value set C
    find a constraint x (predicate) that best separates the two sets
- **Predicate Ranking**
  give each predicate a score based on how x can distinguish F and C
  score of predicate: $s = |\mathbb{P}(x|\mathcal{C}) - \mathbb{P}(x|\mathcal{F})|.$
  P(x | C) and P(x | F): probabilities of predicate
- **Candidate Path Identification**
  - Traverse locations with highly ranked predicates
  - Generate *candidate paths* by connecting the nodes
  - Modeled as *graph covering problem*

StatSym Design – Guided Symbolic Execution

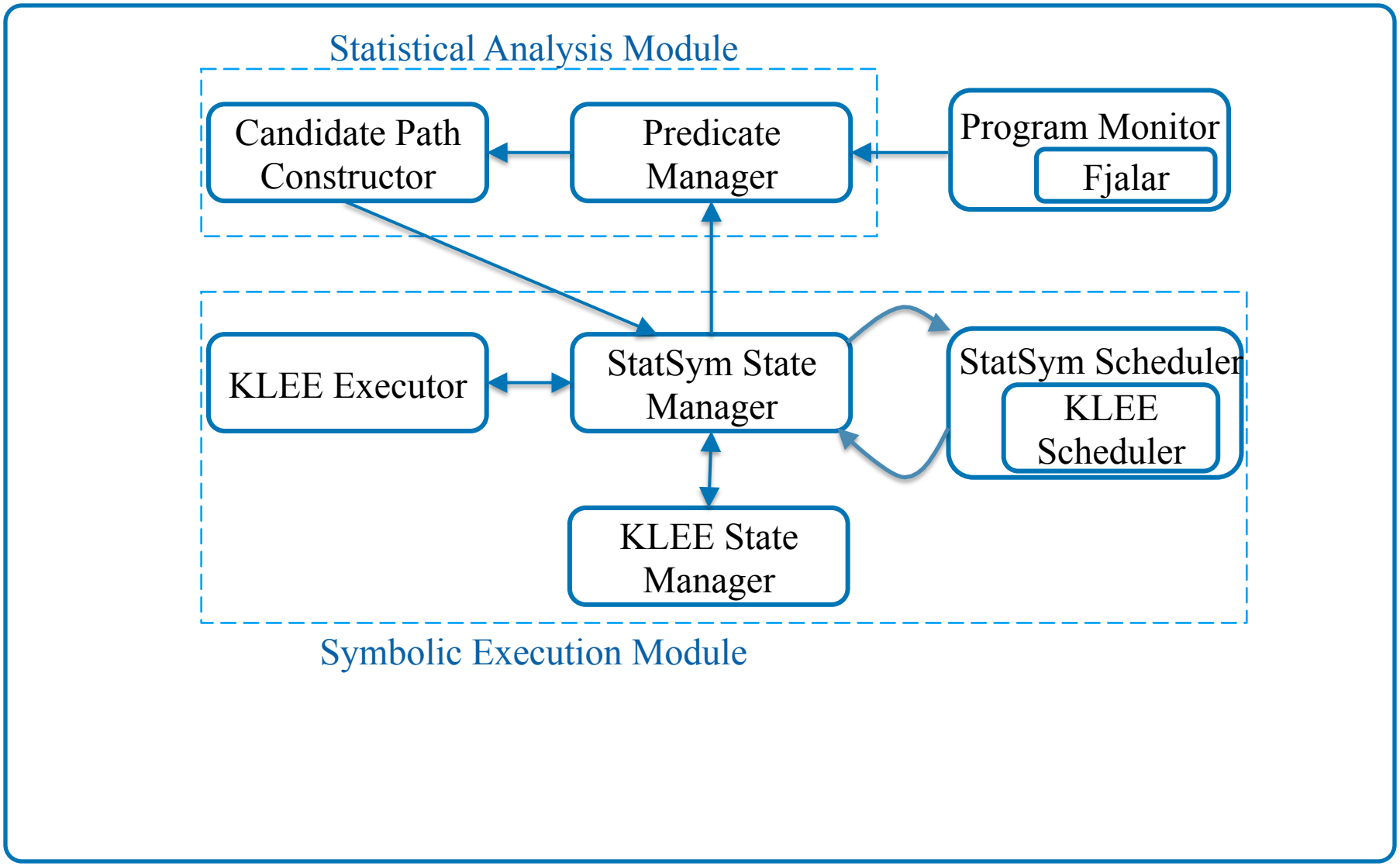


Statistics-guided Symbolic Execution: heuristics



- *High priority for paths that covers current node*
- *Set a threshold for number of diverted hops from current node*

StatSym Implementation



Experimental Setup

- **Benchmark Selection**
  - Four application from different application domains
    - *polymorph*, utility for file name conversion, *BugBench*
    - *Grep*, plain-text searching tool, from *STONESOUP*
    - *Ctree*, GNU tool for file system hierarchy display, *STONESOUP*
    - *Thttpd*, open source web server
- **Log Collection and Sampling**
  - Select a number of locations to instrument
  - Use a ranges of sampling rates
- **System Setup**
  - 4-core Intel Xeon E5405
  - 12GB DRAM

Evaluation (StatSym vs. KLEE)

	KLEE w/ *StatSym*		Pure Sym. Exec. w/ KLEE	
Benchmark	#paths	time(sec)	#paths	time(sec)
polymorph	63	214.6	8368	3252.0
CTree	112	45.6	17575	Failed
thttpd	5168	1691.0	17882	Failed
Grep	11462	563.0	38708	Failed

KLEE failed for 3 out of the four applications.
For smallest application, polymorph, StatSym speeds up KLEE by 15X

StatSym Execution Time Breakdown

	Statistical Analysis Module	Symbolic Execution Module
Benchmark	time(sec)	time(sec)
polymorph	1.9	180.6
CTree	58.4	1.6
thttpd	561.2	247
Grep	661.4	37.7

Time breakdown when sampling rate is 100%.

	Statistical Analysis Module	Symbolic Execution Module
Benchmark	time(sec)	time(sec)
polymorph	1.6	213
CTree	43.2	2.4
thttpd	428	1263
Grep	518.7	44.3

Time breakdown when sampling rate is 30%.

Case Study- Polymorph

Instrumented Locations:

L1: grok_commandLine():leave
L2: convert_fileName():enter
L3: is_fileHidden():leave
L4: does_nameHaveUppers():enter
L5: does_newnameExist():leave
L6: grok_commandLine():enter
L7: convert_fileName():leave
L8: main():enter
L9: does_newnameExist():enter
L10: main():leave
L11: is_fileHidden():enter
L12: does_nameHaveUppers():leave

Instrumented Variables:

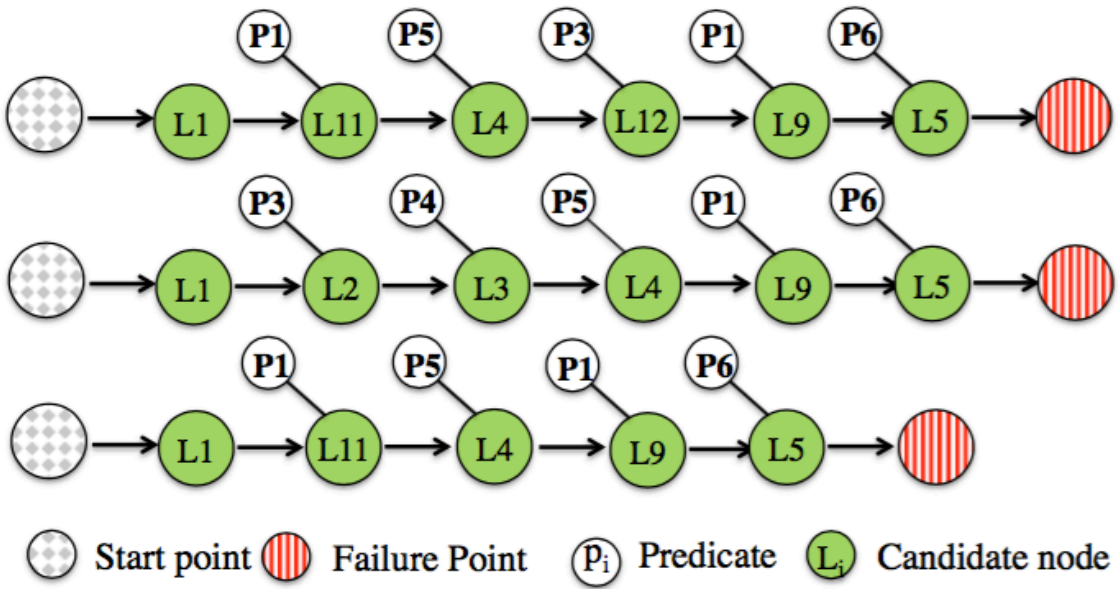
GLOBAL: *target*, *wd*, *hidden*, *track*, *clean*, *init_file*, *hidden_file*

FUNCPARAM: *argc*, *original*, *suspect*

No.	Predicate	Loc.
P1	len(suspect_FUNCPARAM) >536.5	L9
P2	len(original_FUNCPARAM)>518.0	L2
P3	len(suspect_FUNCPARAM)>535.0	L12
P4	len(suspect_FUNCPARAM)>517.5	L3
P5	len(suspect_FUNCPARAM)>526.0	L2
P6	len(suspect_FUNCPARAM)>497.5	L5
P7	track_GLOBAL<-infinity	L7
P8	wd_GLOBAL<-infinity	L7
P9	track_GLOBAL<-infinity	L10
P10	clean_GLOBAL<-infinity	L10

Instrumentation List and Predicate Lists

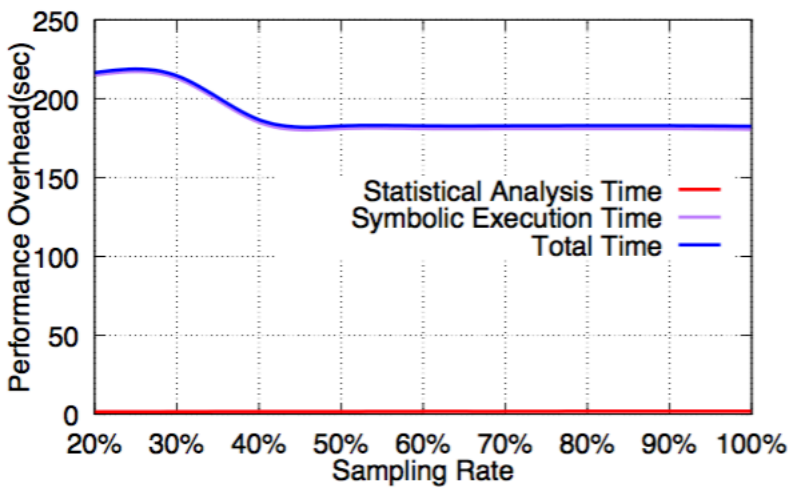
Case Study- Polymorph



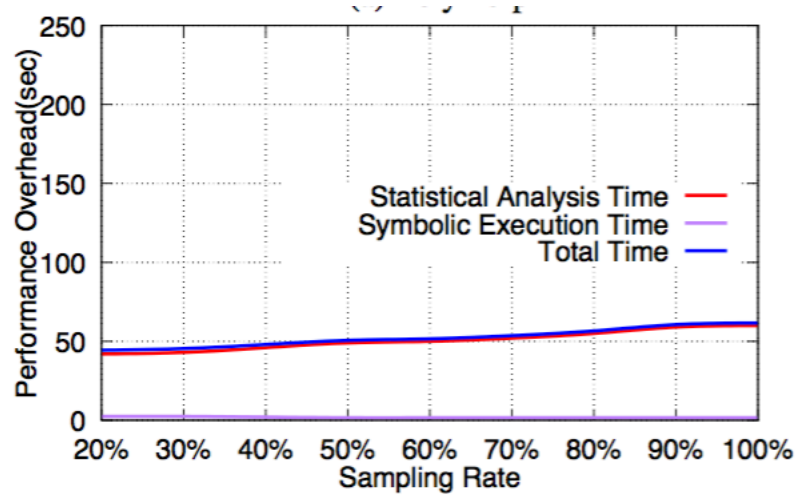
- **About 2s for generating candidate paths**
- **About 180s for symbolic executing the first path**
  - 63 paths explores by StatSym vs. 8368 paths explored by KLEE

Sensitivity Study

- **Sensitivity of StatSym to different sampling rates**



(a) Polymorph



(b) CTree

For Polymorph, symbolic execution time dominates the total time.
CTree, Statistics analysis time dominates the total time.
Overall, StatSym scales well with sample rates.

Conclusion

- **We propose StatSym, a framework for automatic vulnerable path discovery.**

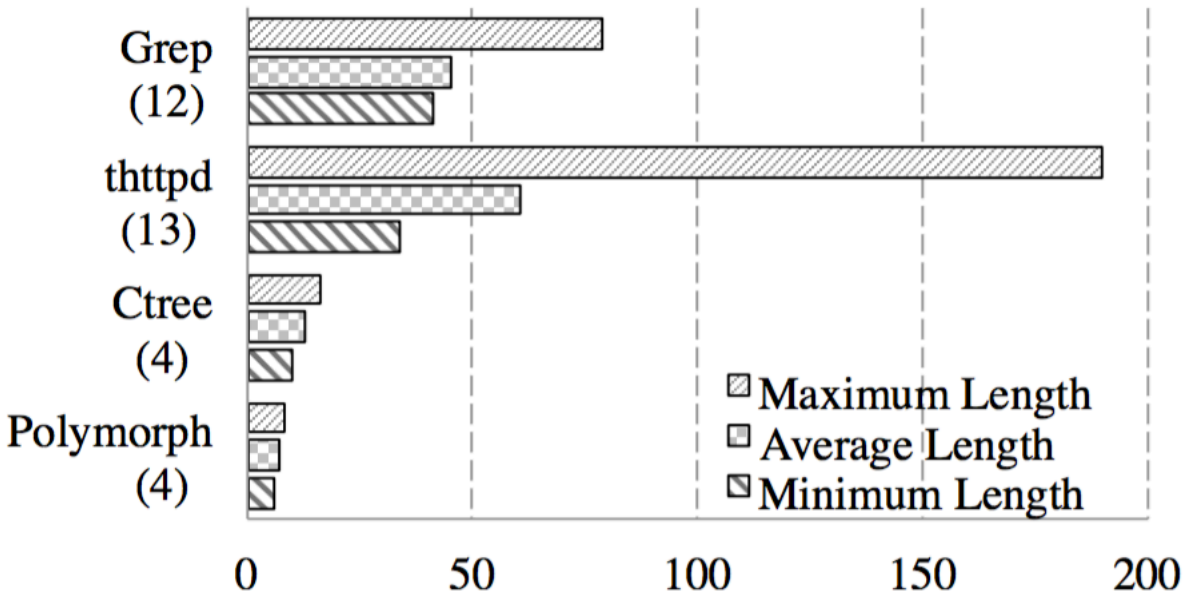
- **Our approach harnesses the scalability of statistical analysis and the rigorousness of symbolic execution**

- **We valuate StatSym using real-world applications**
  - Statsym achieves about 15X speedup

- **Acknowledgements**

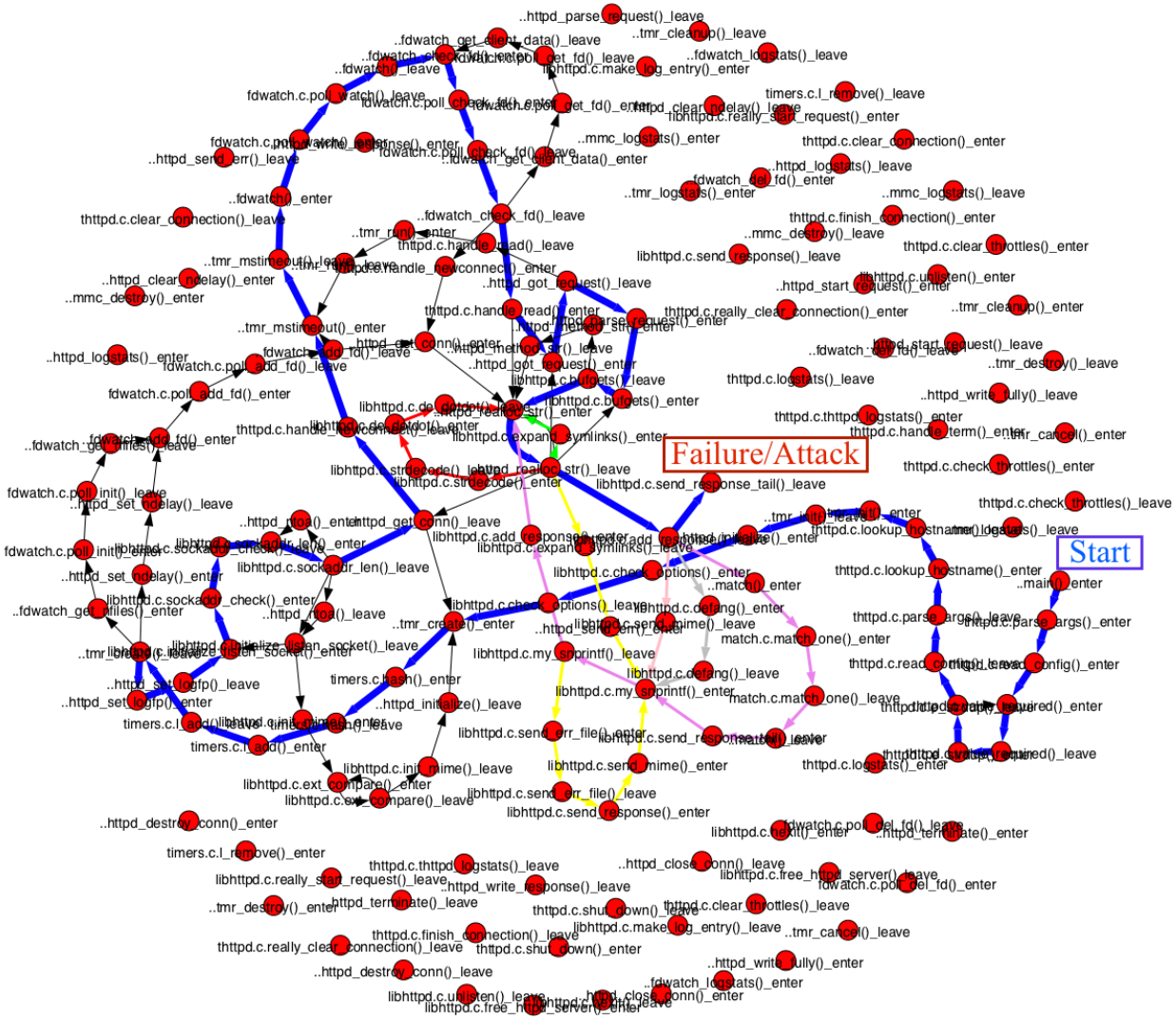
Email: albertyao@gwu.edu

Evaluation- Candidate Path Statistics

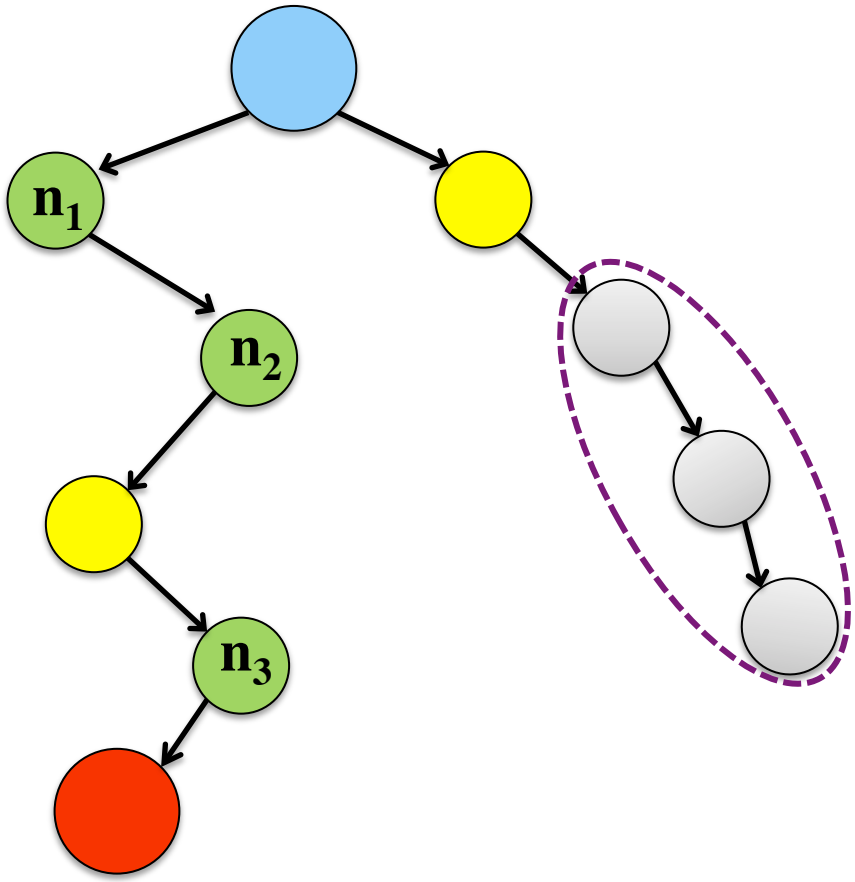


Maximum, average and minimum candidate path lengths

Full Candidate Path in thttpd



Statistics-guided Symbolic Execution: heuristics



- *High priority for paths that covers current node*
- *Set a threshold for number of diverted hops from current node*