CERIAS

the center for education and research in information assurance and security

Structural Execution Indexing and Its Applications

Nick Sumner, Bin Xin, and Dr. Xiangyu Zhang {wsumner,xinb,xyzhang}@cs.purdue.edu

1) Context

Fundamental to dynamic program analyses is the notion of identity for a point within an execution. A lack of formalization has led to imprecise and heuristic approaches for representing execution points. This makes analyses inherently imprecise and less useful.

We present Execution Indexing[2] as a formal approach to unique identities within and between program executions and provide one intuitive approach to representing identity that has proven useful in improving the correctness or precision of real world analyses.

2) Introduction to Structural Execution Indexing

Execution Indexing is a general theory of uniquely identifying points of correspondence between program executions or points of correlation within a program's execution.

Structural Execution Indexing is a specific, novel approach to Execution Indexing utilizing equivalence and similarity between dynamic control dependences of any points in an execution.

One Program

```
1) ...
2)    while (p1) {
3)       get_input (buf);
4)    }
5)    get_input (buf);
6) ...
7) void get_input (char * buf)
8) {
9)    read (buf, 512);
10) }
```

Multiple Runs

Two Iterations

```
2) while (p1 true) {
3)    get_input (buf);
9)    read (buf,512);

2) while (p1 true) {
3)    get_input (buf);
9)    read (buf,512);

2) while (p1 false) {

5) get_input(buf);
9)    read (buf,512)
```

Zero Iterations

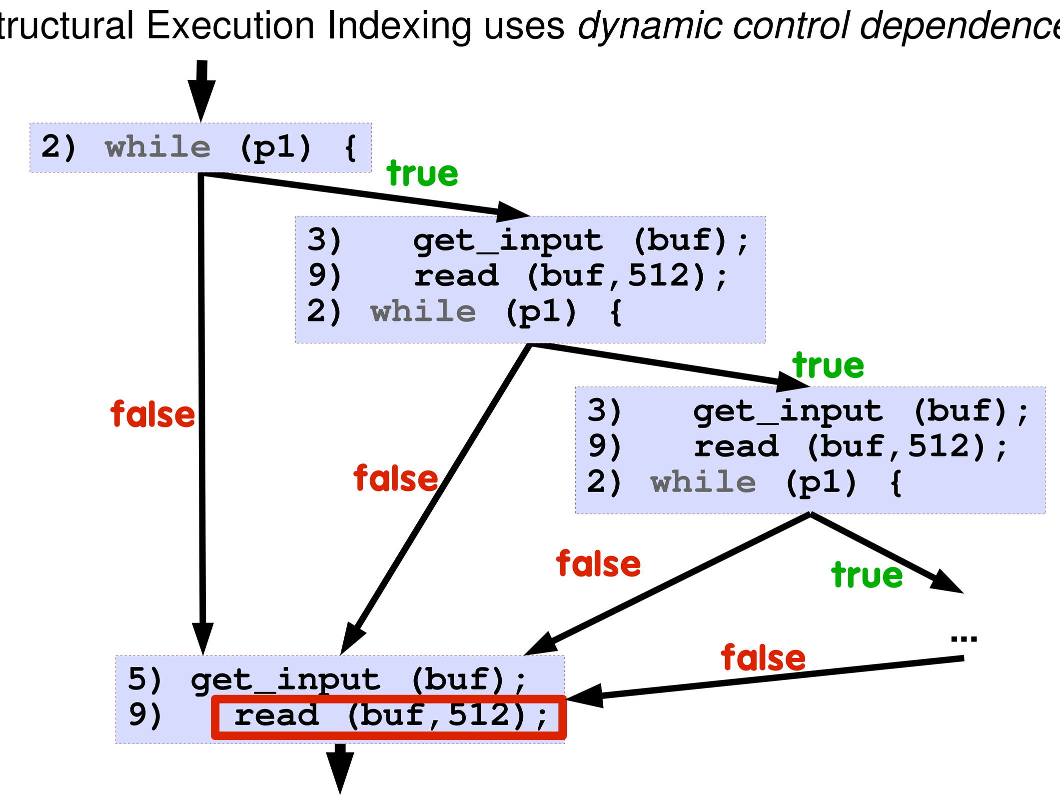
```
2) while (p1 false) {

5) get_input(buf);
9)    read (buf,512);
```

How Do We Know These Points Correspond?

Recognizing the same point between these executions can be useful in various analyses, but the meaning of "same" depends on context.

Structural Execution Indexing uses *dynamic control dependence*



3) Formalizing Structural Indices

An ***Execution Description Language*** is a context-free grammar that expresses *all* possible executions of a program. e.g.

Construct	EDL Representation	Examples
1) s1; 2) s2; 3) s3; 4) s4;	S → 1 2 3 4	1 2 3 4
1) if 2) s1; 3) else 4) s2;	S → 1 R_1 R_1 → 2 | 4	1 2 1 4
1) while (...) { 2) s1; 3) } 4) s2;	S → 1 R_1 4 R_1 → 2 1 R_1 | ε	1 2 1 4 1 2 1 2 1 4
1) void A () { 2) B (); 3) } 4) void B () { 5) s1; 6) }	S → 2 R_B R_B → 5	2 5

Note: unstructured control flow elided only for simplicity

Identifying EDL Terminals is Equivalent

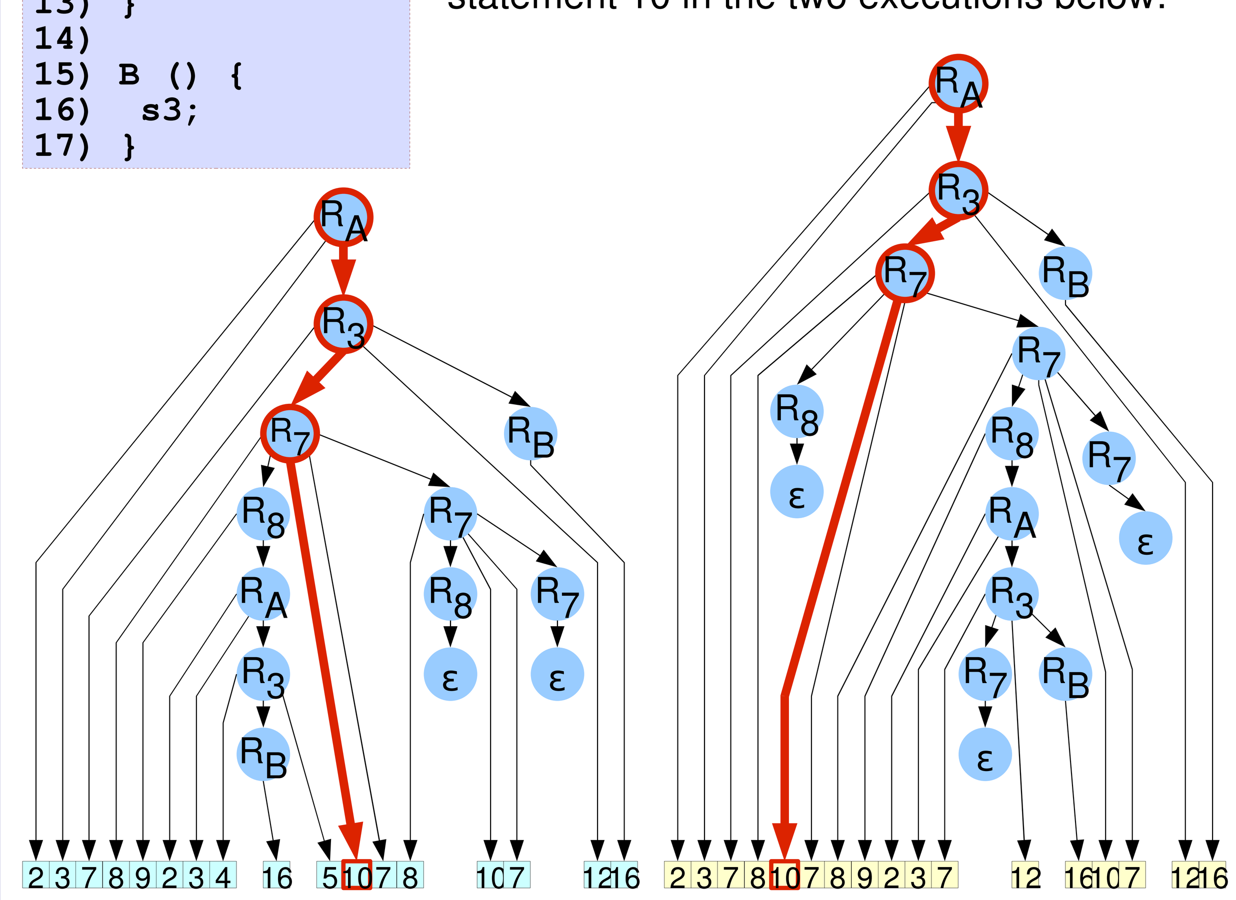
The structural index of a point is the path in the derivation tree from the root to the point in the execution. e.g.

```
1) A () {
2)   s1;
3)   if (C1) {
4)     B ();
5)     return;
6)   }
7)   while (C2) {
8)     if (C3)
9)       A ();
10)    s2;
11)  }
12)  B ();
13) }
14)
15) B () {
16)   s3;
17) }
```

Grammar Building →

R_A → 2 3 R_3
R_3 → 4 R_B 5 | 7 R_7 12 R_B
R_7 → 8 R_8 10 7 R_7 | ε
R_B → 16
R_8 → 9 R_A | ε

Consider the corresponding executions of statement 10 in the two executions below:



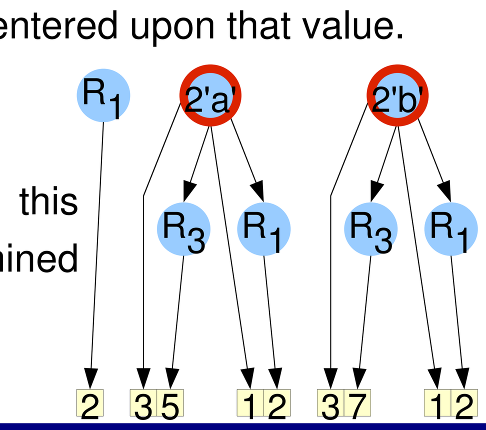
What If Data Determines Identity Semantics?

e.g. event based actions (on input 'a' do...)

Semantic Anchor Points allow the derivation tree to be re-rooted at a value for the duration of control flow centered upon that value.

```
1) while (p1) {
2)   val = getc ();
3)   switch (val) {
4)     case 'a':
5)       ...
6)     case 'b':
7)       ...
8)   }
9) }
```

Identity after this point is determined by `val`



4) Precise Debugging

Debugging often utilizes **breakpoints** to freeze an execution at certain specific times for analysis. The more precise these breakpoints are, the more precise and the faster the debugging takes place.

Particularly with the growth of **automated debugging**, where breakpoints may be set and analyzed by computers themselves, consistency and correctness are crucial.

5) Dynamic Extraction of Control Structures

Several modern analyses rely on dynamically extracting control structures to refine future actions or winnow aggregated information

```
1) while (p) {
2)   ...
3) }
```

When does the iteration we just started end?

This forms a crucial part of:

- **Hierarchical Dynamic Slicing**[1]
- **Execution Omission Error Detection**[3]
- **Potential Concurrency Profiling**
- **...**

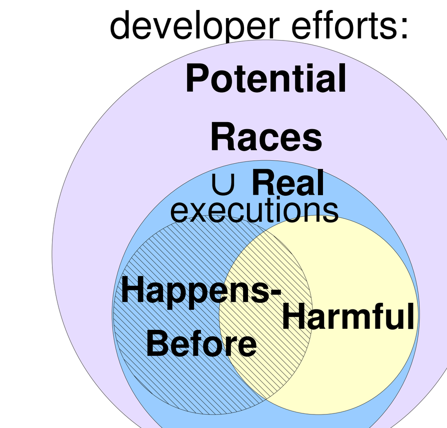
6) Data Race Classification

A ***data race***:
- 2 Memory accesses, at least 1 write
- Multiple threads
- No ordering constraints

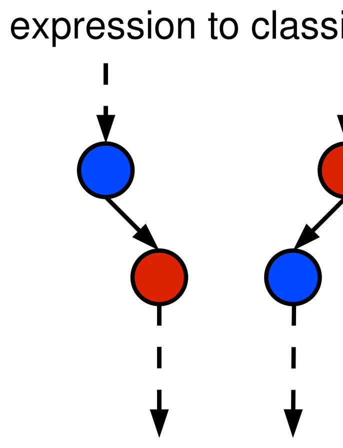
T1	T2
`if (x > 0) foo (x);`	`x = 0;`

They can cause nondeterministic execution.
Existing detection methods warn unnecessarily.

Classify to prioritize developer efforts:

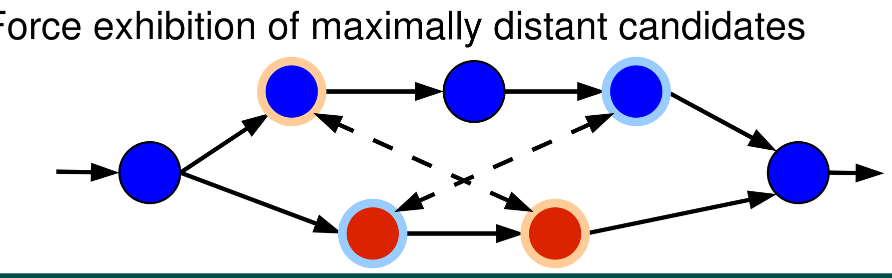


Re-execute and force race expression to classify:



Selecting Dynamic Accesses for Testing

Numerous dynamic memory accesses exhibit the same semantic race
- Infeasible to test them all
- Categorize by structure and data semantics
- Force exhibition of maximally distant candidates



References

[1] Tao Wang, Abhik Roychoudhury. Hierarchical dynamic slicing. ISSTA 2007: 228-238.
[2] Bin Xin, Nick Sumner, Xiangyu Zhang. Efficient program execution indexing. PLDI 2008: to appear.
[3] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, Rajiv Gupta. Towards locating execution omission errors. PLDI 2007: 415-424.