MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Spindle: Informed Memory Access Monitoring

By Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen (USENIX ATC '18)

Presented by Nada Abdalgawad, Luke Lesh, Ivris Raymond, and Ryan Hou

Group 4

What is Spindle?

A tool that performs ***static analysis*** in order to obtain regular and predictable patterns in ***memory accesses*** of a program

How does the memory access trace look like?

```
Load address1
Store address1
Load address2
Load address3
Load address4
Store address4
…
…
…
```

- Tells the sequence of memory addresses hence accesses
- Might have patterns > Spindle

Why is knowing memory accesses important?

Performance Optimization

- Locality analysis:
  - Temporal: access same memory location
  - Spatial: access nearby memory location
  - Improve cache utilization

- Cache efficiency:
  - Design algorithms and data structures
  - Exploit cache hierarchies
  - Reduce cache misses

Why is knowing memory accesses important?

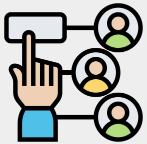
Memory prefetching

- Prefetching of Future Accesses:
  - Predict future memory accesses
  - Load data into cache before it is needed

Why is knowing memory accesses important?

Memory management

- Allocation and Deallocation:
  - Optimize memory allocation patterns
  - Reduce fragmentation



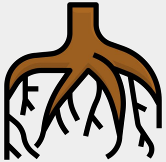
- Memory leak detection:
  - Anomalies: repeated allocations without corresponding deallocations
  - Maintains stability and reliability of an application

Why is knowing memory accesses important?

Debugging and Profiling

- Identifying Bottlenecks:
  - Identify performance bottlenecks due to memory usage
  - Focus optimization on crucial parts of the code

- Root Cause Analysis:
  - Diagnosing the root cause of problems
    - Segmentation faults
    - Out-of-memory errors
    - Inefficient memory usage



Why is knowing memory accesses important?

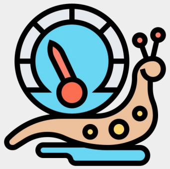
Resource Planning

- Resource Utilization:
  - Efficiency of system resources utilization
  - Useful in environments with constrained resources
    - Embedded systems
    - Cloud computing

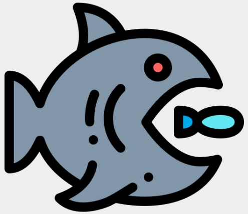
Other tools

- Valgrind and Intel PIN can produce a full list of memory access trace
- **Problems**

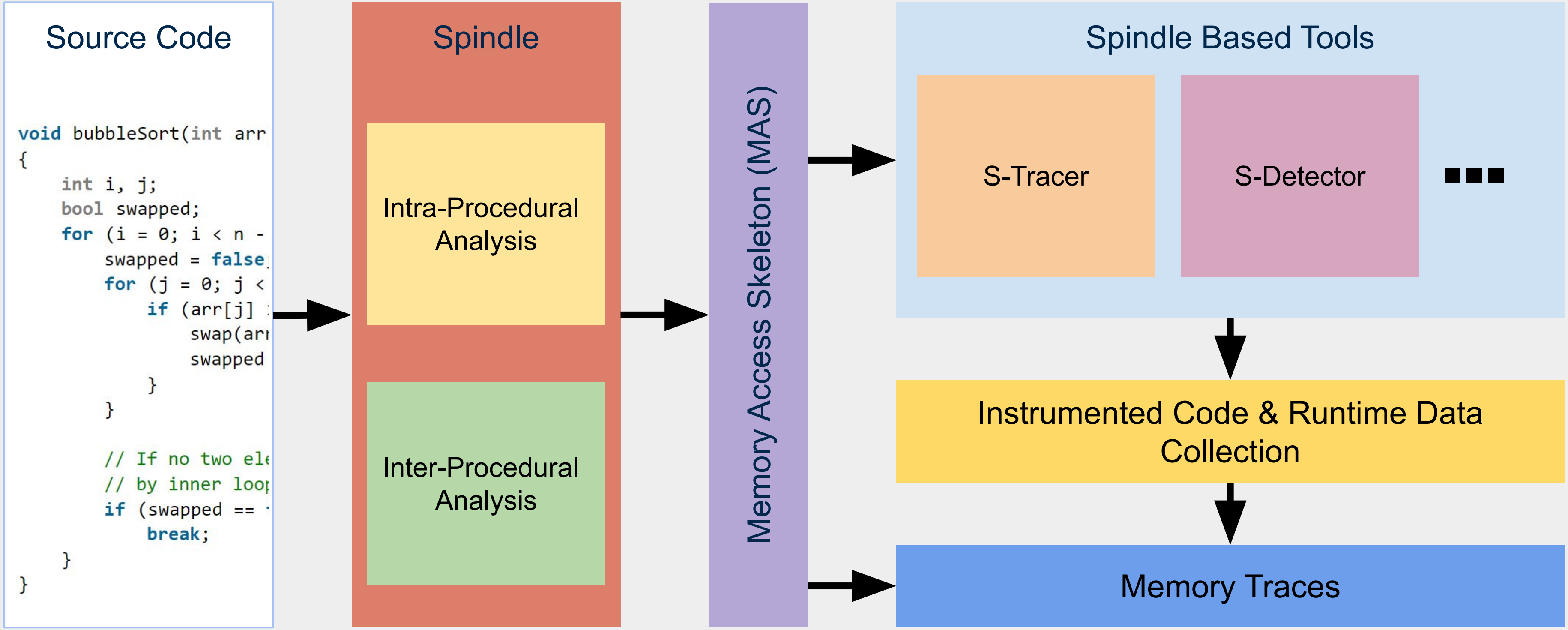
S l o w



Large Files

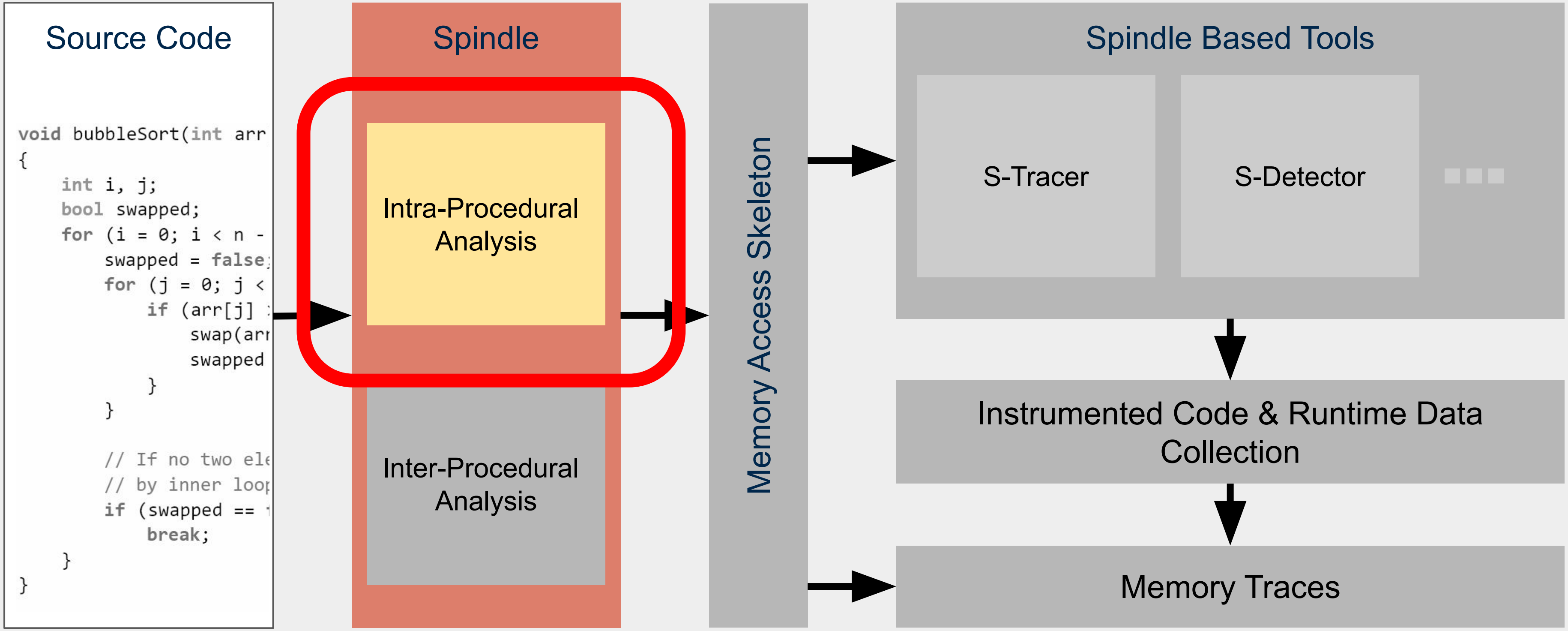


Spindle Overview



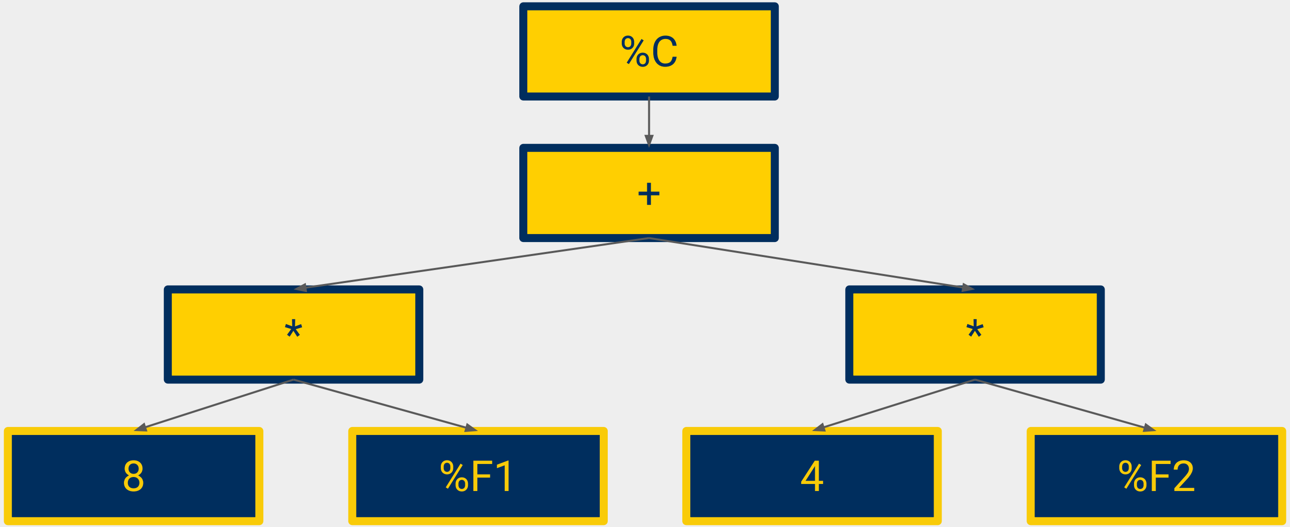
Cropped Bubble Sort Image: [GeeksforGeeks Bubble Sort Algorithm Article]

Intraprocedural Analysis



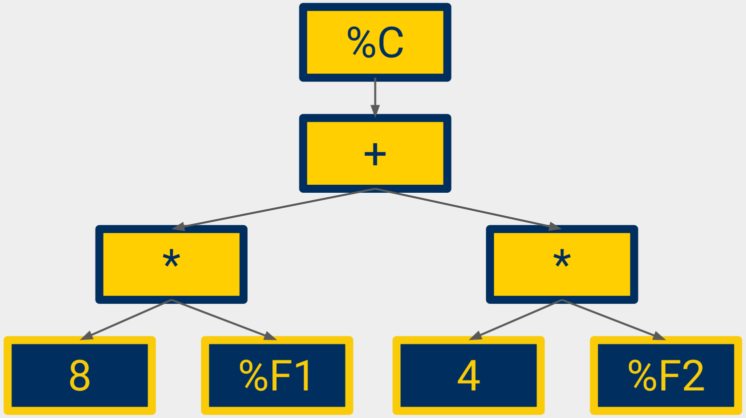
Memory Dependence Trees

%A = mul i32 8, %F1
%B = mul i32 4, %F2
%C = add i32 %B, %A
%D = load i32* %C

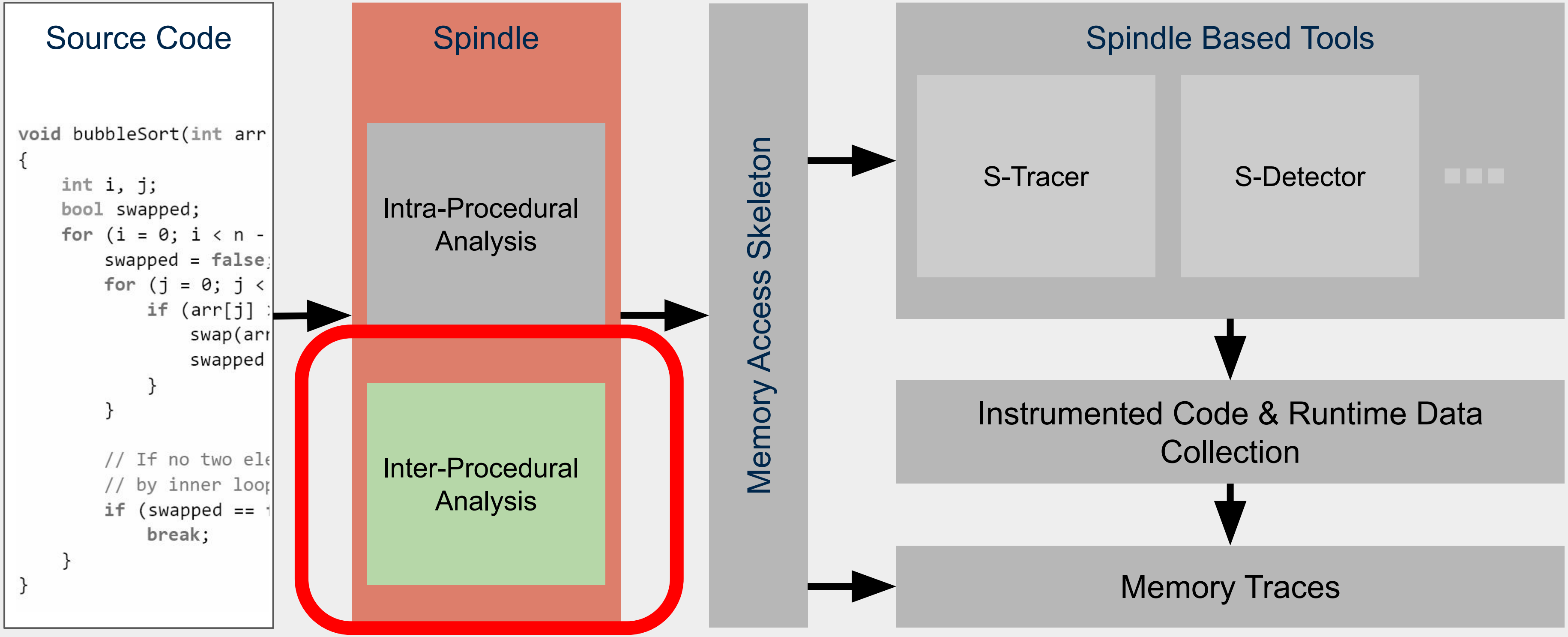


Types of Leaf Nodes

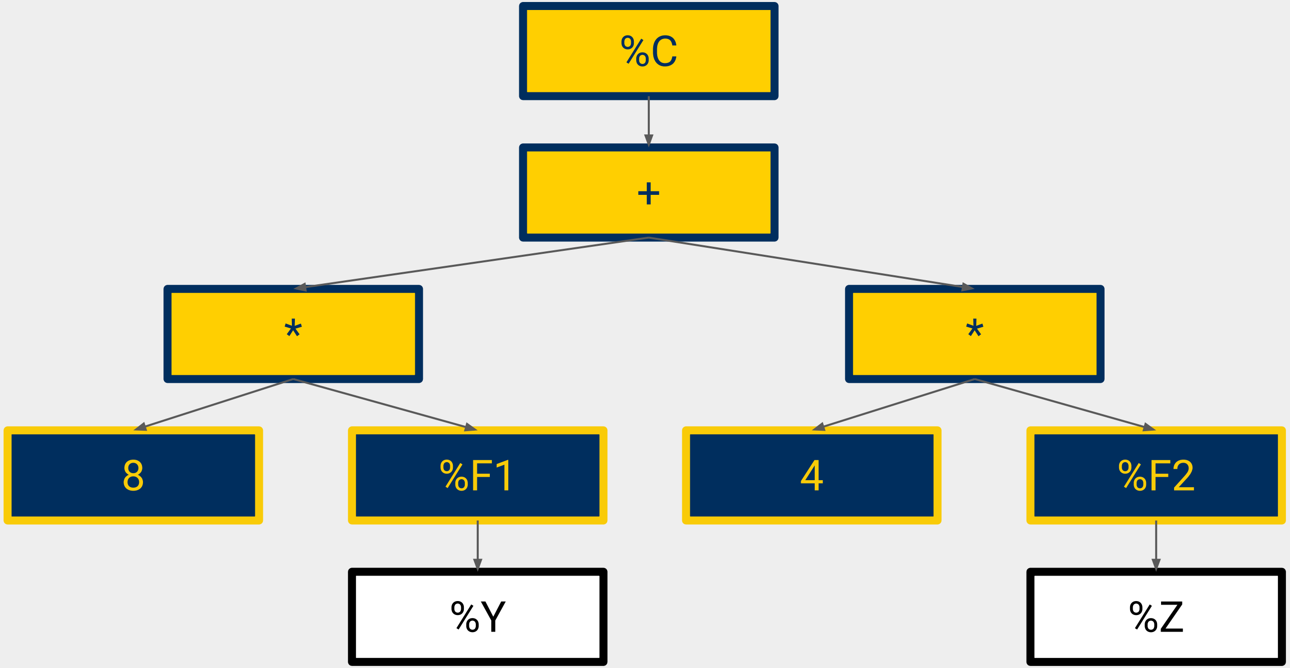
- **Constant value (compile time)**
- **Loop induction variable (compile time or runtime)**
- Base memory address (compile time or runtime)
- Function parameter (compile time or runtime)
- Data-dependent variable (runtime)
- Function return value (runtime)



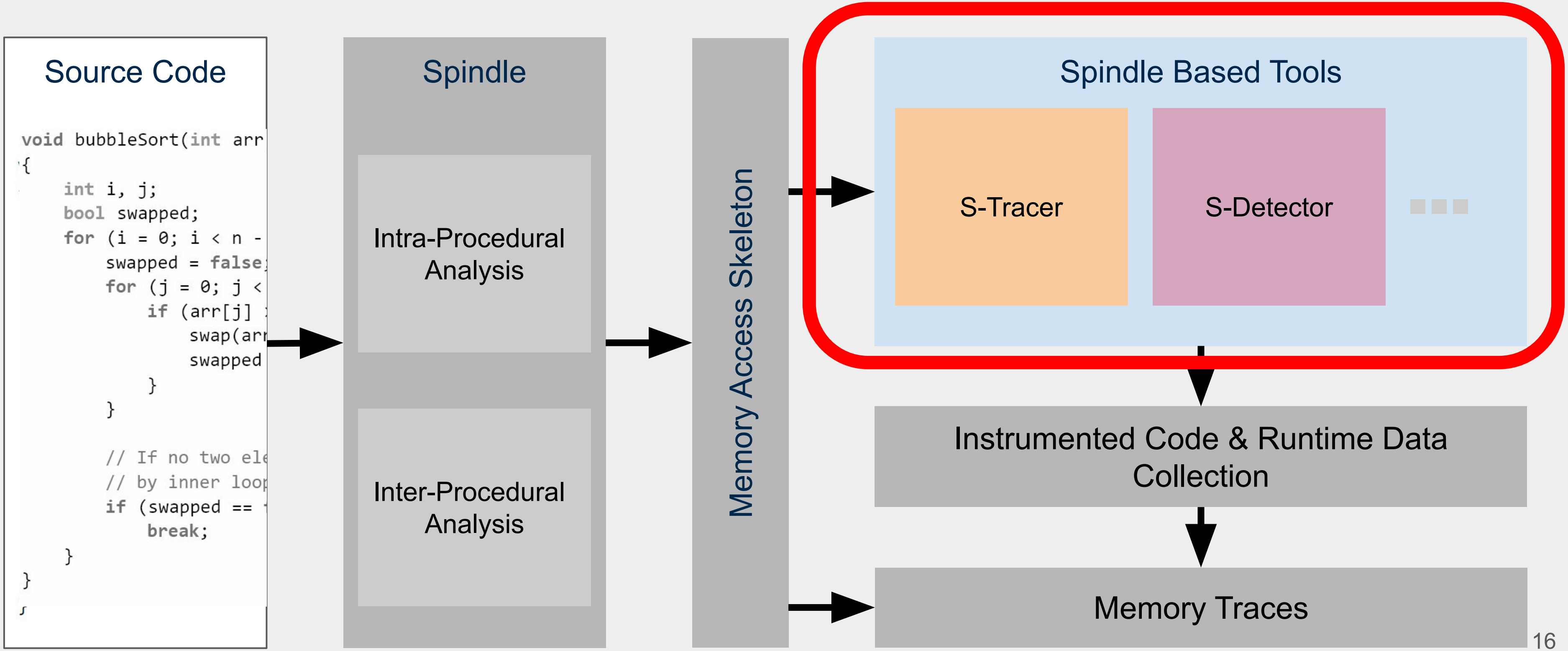
Interprocedural Analysis



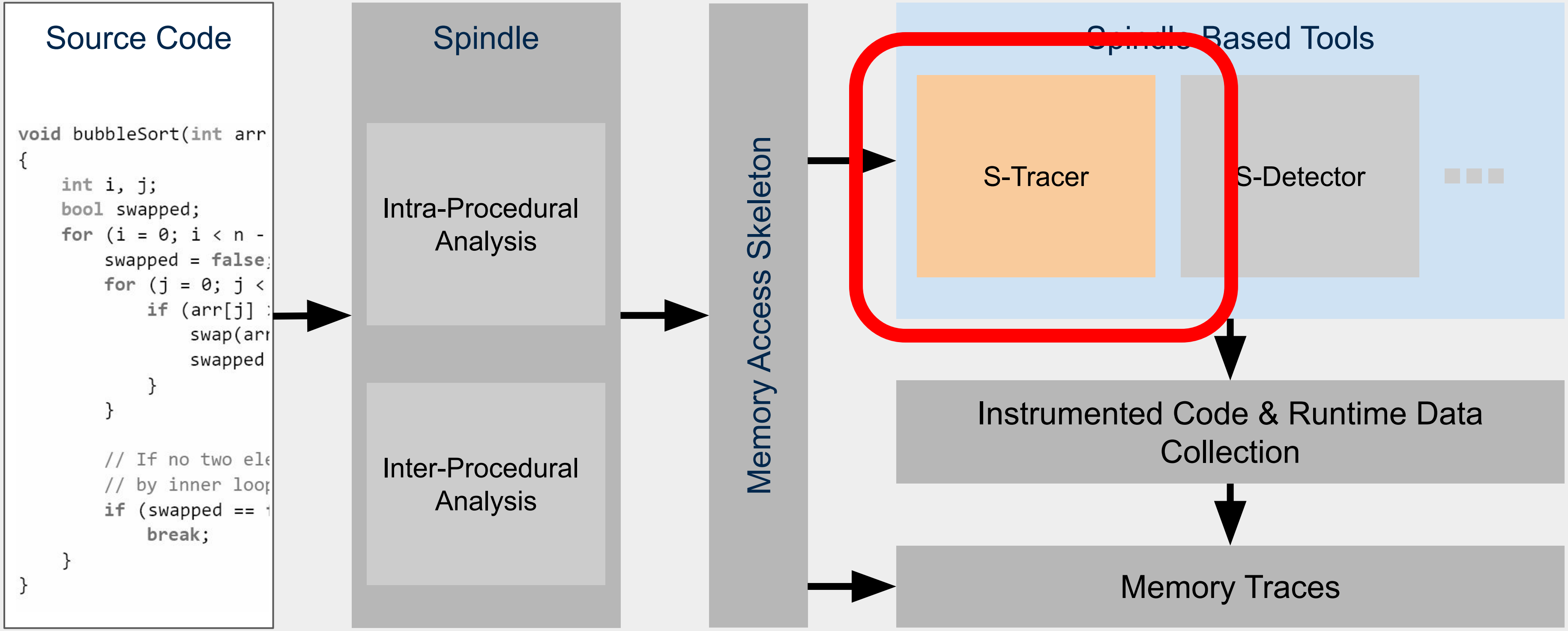
Interprocedural Analysis



Spindle-Based Tools



S-Tracer



Source Code
void bubbleSort(int arr
{
int i, j;
bool swapped;
for (i = 0; i < n -
swapped = false;
for (j = 0; j <
if (arr[j]
swap(arr
swapped
}
}
// If no two ele
// by inner loop
if (swapped ==
break;
}
}
Spindle
Intra-Procedural Analysis
Inter-Procedural Analysis
Memory Access Skeleton
Spindle-Based Tools
S-Tracer
S-Detector
Instrumented Code & Runtime Data Collection
Memory Traces

S-Tracer

- Memory **Trace** Collector
- Existing methods: record every memory access
  - Large memory trace size
  - Slowdown of program
- Use MAS and dynamically collected data:
  - Highly compressed memory traces
  - Lower runtime overhead

Static Trace

```
Function BubbleSort(dyn_A, dyn_N) {
 Loop0: L0, 0, dyn_N, 1 {
  Loop1: L1, L0, dyn_N, 1 {
   Load1: dyn_A+L0; Load2: dyn_A+L1;
    Branch: dyn_flag {
     Call Swap(dyn_A, L0, L1);
}}}}
Function Swap(S, i, j) {
 Load3 : S+i; Load4 : S+j;
 Store1: S+i; Store2: S+j;
}
```

Dynamic Trace

```
BubbleSort {
 dyn_A:
  0x7fffdfc58320;
 dyn_N:
  10;
 dyn_flag:
  {0,0,1,1,0,...,1,1};
}
```

S-Tracer Evaluation

- Evaluation workloads:
  - Regular memory accesses
  - Irregular graph algorithms
  - Multithreaded
- Compared against the PIN (Intel) tool
- Trace Size
  - Over **100x** trace size reduction for regular access patterns
  - Worst case size reduction: **6.93x**

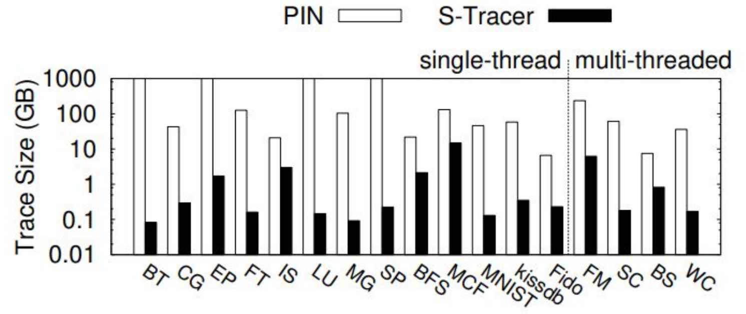


Figure 13: Trace size comparison

S-Tracer Evaluation

- Runtime Overhead
  - PIN: **502x** average slow down
  - S-Tracer: **6.5x** average slow down

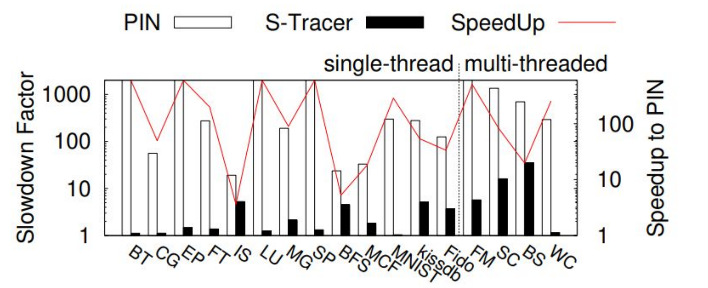
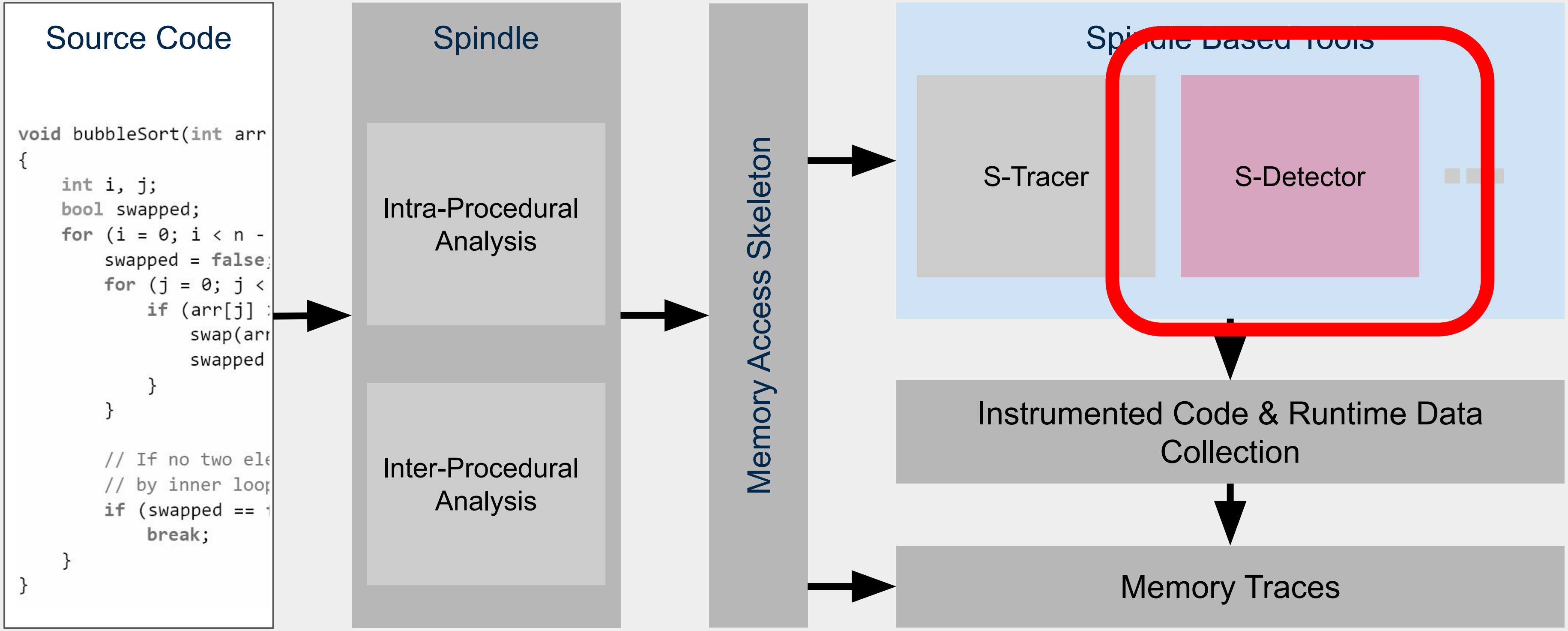


Figure 14: Application slowdown by S-Tracer and PIN with I/O (left) and S-Tracer speedup over PIN (right)

S-Detector



Source Code
void bubbleSort(int arr
{
int i, j;
bool swapped;
for (i = 0; i < n -
swapped = false;
for (j = 0; j <
if (arr[j]
swap(arr
swapped
}
}
// If no two ele
// by inner loop
if (swapped ==
break;
}
}
Spindle
Intra-Procedural Analysis
Inter-Procedural Analysis
Memory Access Skeleton
Spindle Based Tools
S-Tracer
S-Detector
Instrumented Code & Runtime Data Collection
Memory Traces

S-Detector

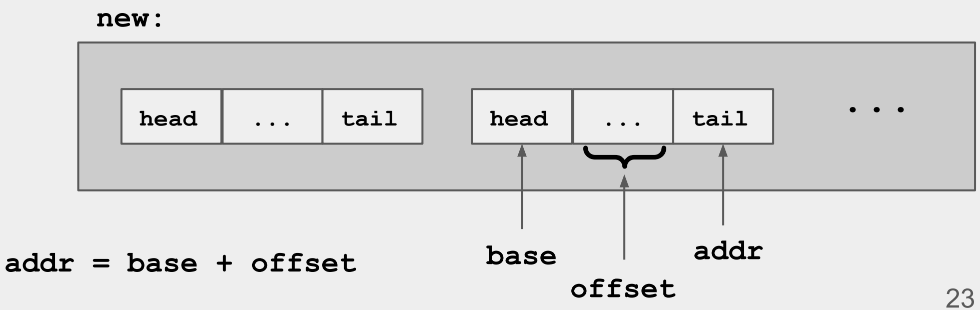
- Memory bug **detector**
  - Invalid Accesses
    - Out-of-bound array access, use after free
  - Memory Leaks
    - Unfreed allocated objects after termination

- Existing methods: Insert memory checking instructions
  - Problem: Significant program slowdown

S-Detector

- Use static information (eg MAS) to eliminate unnecessary instrumentation
- MAS informs us the coarse-grained memory accesses of object
    - Prune instruction-level checks by using object-level checks
    - Only need to check:
        - Valid Offset: `offset < struct_size`
        - Valid memory range of `[base, base + size]`

```
while (pos - 1 && red_cost >
              (cost_t)new[pos/2-1].flow){
   new[pos-1].tail = new[pos/2-1].tail;
   new[pos-1].head = new[pos/2-1].head;
   // Three more accesses to struct members
   // of new[pos-1] and new[pos/2-1].
   pos = pos/2;
   new[pos-1].tail = tail;
   // Four more accesses to struct members
   // of new[pos-1].
}
```



S-Detector Evaluation

- Average* runtime overhead
  - ASan: **66%**
  - Baseline: **184%**
  - Optimized: **26%**
- Avoided **64%** of runtime memory checks
- Evaluated on 11 C programs from the SPEC CPU 2006 benchmarks
- Compared to AddressSanitizer (Google) and Dr. Memory

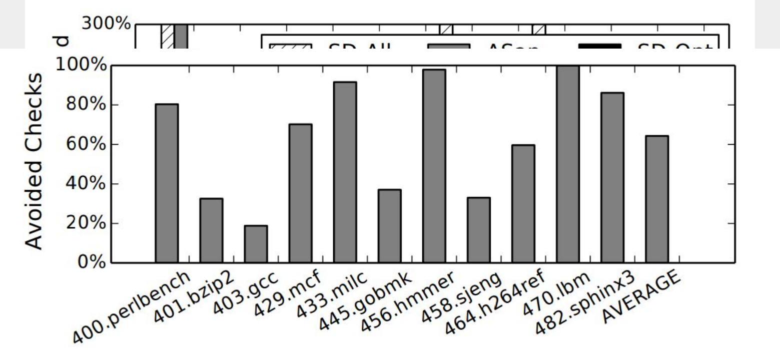


Figure 11: Reduction in runtime memory checks

Figure 10: Overhead comparison (bars over 300% truncated)

* Geometric mean of all test programs

Commentary

- S-Detector PoC only handles invalid accesses and memory leaks, but they chose to compare to tools that do not do static analysis
- Cannot capture dynamically linked libraries at the IR level
  - Requires fallback to dynamic instrumentation
- No quantitative analysis of the number of memory bugs caught by S-Detector compared to existing methods
- The MAS representation is currently limited to structured, predictable memory access components
- The MAS usage is not explained in the paper in very substantial detail

Thank you

Questions?

Outline

- Motivation (Nada)
  - Why do we care about tracing memory accesses?
  - Why is related work insufficient?
- Framework (Ivris)
  - M-CFG, computable and non computable types, sample trace
  - Adding static analysis helps us improve on current tools
- Static Analysis (Luke)
  - Intra-procedural
  - Inter-procedural
- Evaluation (Ryan)
  - S-Detector and Evaluation
  - S-Tracer and Evaluation
- Weaknesses and Improvements (Nada)

Contribution: Static Analysis to Reduce Runtime Overhead

- Reduce the runtime overhead of current memory analysis tools without compromising function by introducing static analysis
- Provide a representation of the memory accesses constructed from static analysis usable and modifiable later in instrumentation (MAS)
- Demonstrate flexibility of Spindle's analysis for constructing a variety of memory analysis tools

Memory Access Skeleton?

- Representation of the memory accesses for a given program
    - Needs to be usable in instrumentation phase and should be able to have blanks filled in for memory addresses not available at compile-time
- Segue into the intra and interprocedural analysis we use to construct this