Spindle:
Informed Memory Access Monitoring

ATC @ Boston, USA, 2018.07.12

Haojie Wang*†, Jidong Zhai*, Xiongchao Tang*†
Bowen Yu*, Xiaosong Ma†, Wenguang Chen*

https://github.com/thu-pacman/Spindle



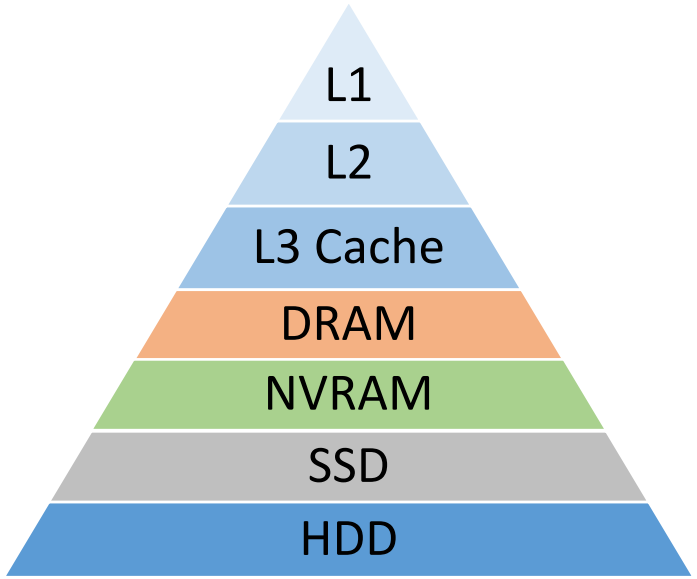
清華大學
Tsinghua University



* *Tsinghua University*
† *Qatar Computing Research Institute*

Memory access monitoring: imperative

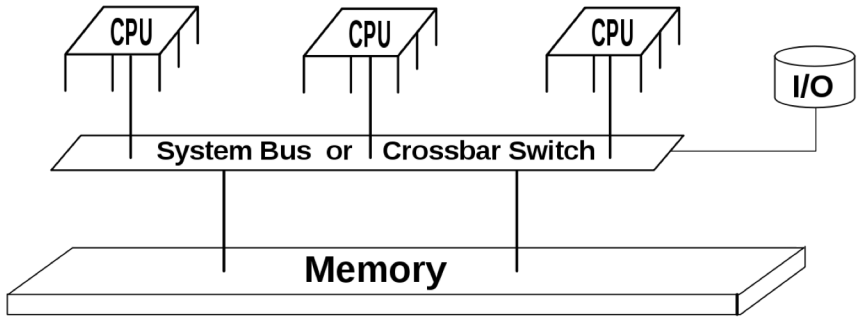
- Memory access is error prone
  - Uninitialized read, write after free, data race, deadlock
- Memory access is a key factor of performance
  - Big gap between CPU and memory, complex cache hierarchy
- Memory access has high security risk
  - Buffer overflow



Memory Hierarchy

- We need memory access monitoring for *bug detection*, *performance optimization*, and *malware analysis*

Full monitoring for a long-running program introduces large overhead



Shared memory system

Conventional one-by-one checking is slow

- Dynamic tools
  - PIN, Valgrind, DynamoRIO, Dr.Memory
  - Modify instruction flow at runtime to insert checking functions

```
1  for (int i=0; i<N; ++i) {
2    a[i] = i;
3    checkAccess(&a[i]);
4  }
```

Overhead >100x

Trace size ~10GB/s



Check all memory access one by one

Record all memory access for tracing

- Static + dynamic hybrid tools
  - Google Address Sanitizer, Memory Sanitizer
  - Add checking functions at compile time

Still doing redundant work

Overhead ~ 2x

Use access pattern to avoid redundant checks

- Memory access addresses often ***predictable***

```
1  for (int i=0; i<N; ++i) {
2    a[i] = i;
3  }
```

Learn memory access pattern from source code

```
a, a+4, a+8, a+12, ……, a+4*(N-1)
```

- Only starting address **a** and loop count **N** are unknown
  - Record at runtime

Instrumentation code

```
1  recordArrayBase(a);
2  recordLoopFinal(N);
3  for (int i=0; i<N; ++i){
4    a[i] = i;
5  }
```

Knowing a and N, we can compute each a[i]
No more checking is needed

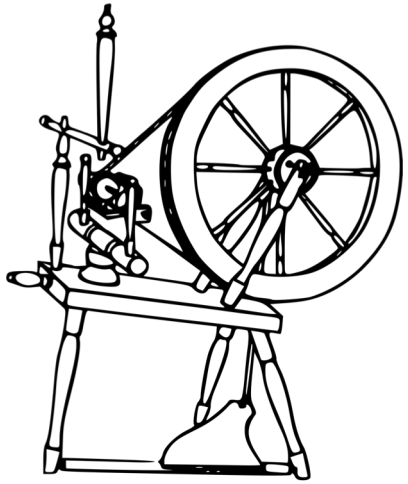
N checks → 2 checks

Solution: Compute what is computable, record the rest

Spindle: Informed Memory Access Monitoring

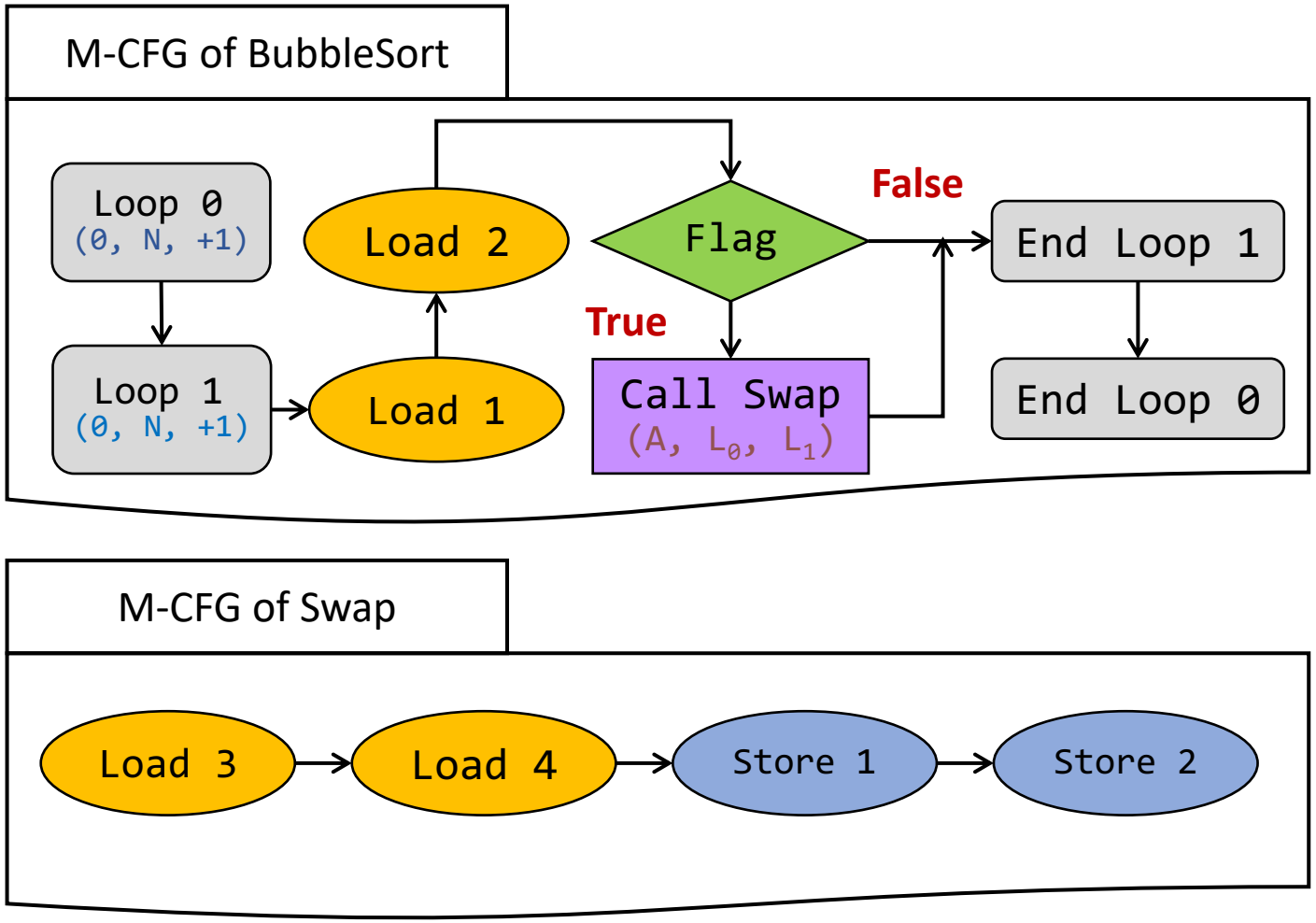
- Enabling ***judicious memory access monitoring***
  - Extracts *code structure* and memory access *patterns* in static analysis
    - Loop, branch, array index, structure offset, …
  - Dramatically reduces runtime instrumentation overhead
    - *Non-computable accesses:* Program input, heap variable address, stack address, …
- ***General framework*** for building memory monitoring tools
  - For trace generation, bug detection, security check, …
  - *Tool-specific analysis* module supplied by tool developers
- Two proof of concept tool
  - *S-Detector for* efficient memory bug detection
  - *S-Tracer* for low-overhead memory tracing

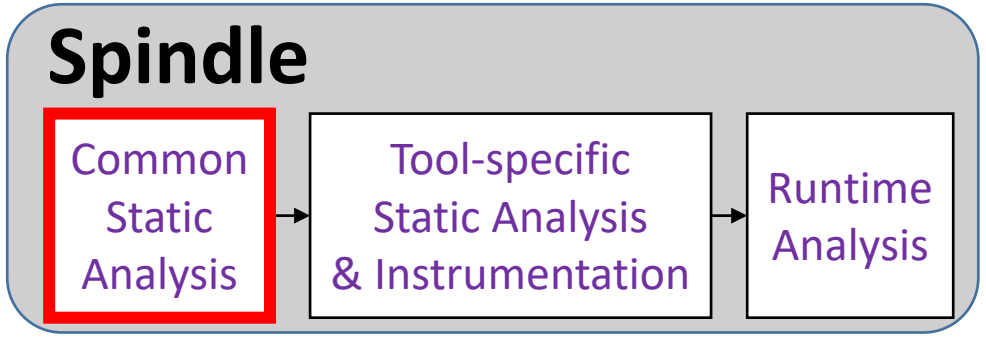
Spindle overview



Spindle common analysis

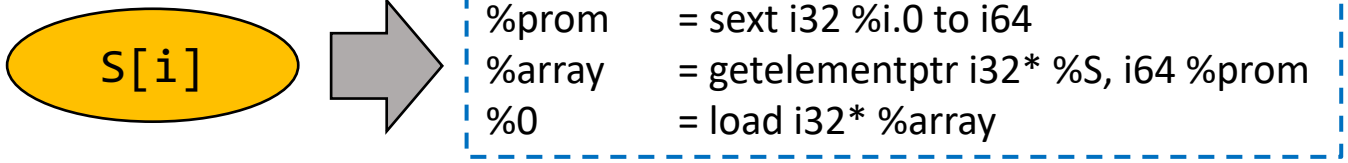
- Control flow analysis

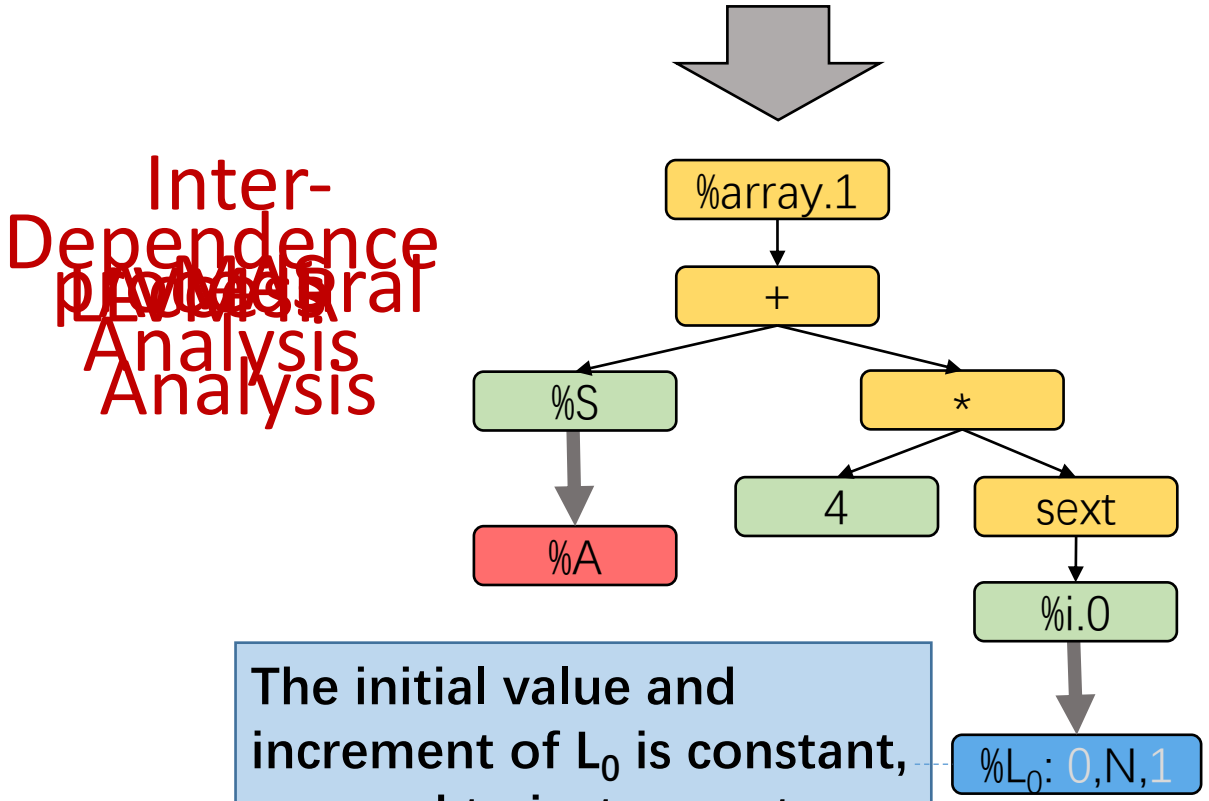




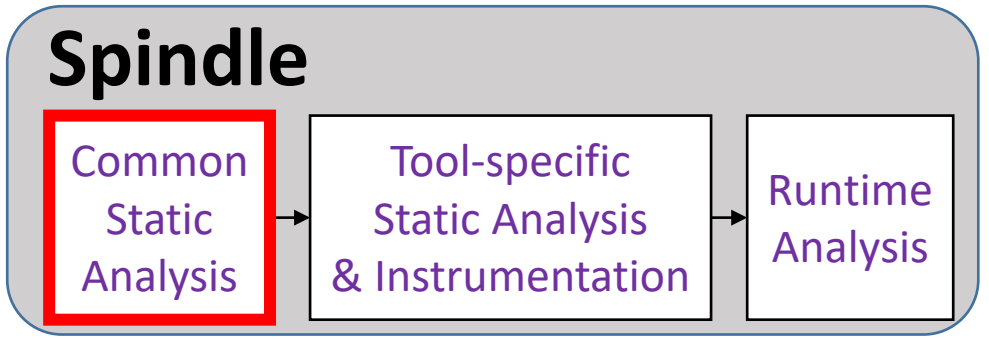

```
1  void BubbleSort(int *A, int N) {
2    for (int i=0; i<N; ++i) {
3      for (int j=i+1; j<N; ++j) {
4        bool flag = (A[i]>A[j]);
5        if (flag) {
6          Swap(A, i, j);
7  }}}}
8
9  void Swap(int *S, int i, int j) {
10   int tmp = S[i];
11   S[i] = S[j];
12   S[j] = tmp;
13 }
```

Spindle common analysis





The initial value and increment of L_0 is constant, no need to instrument.

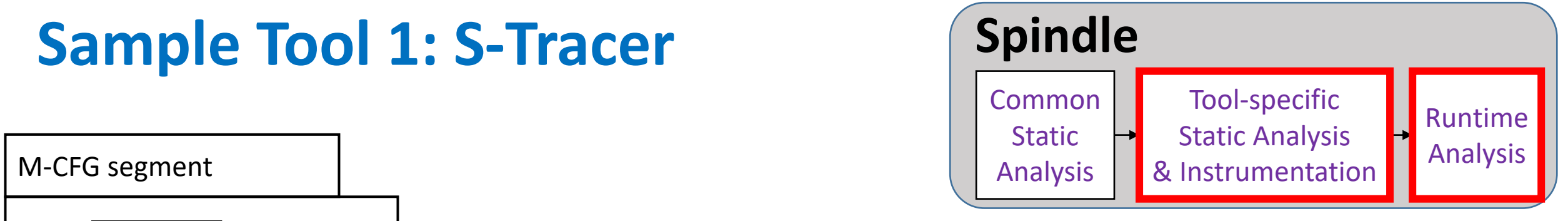


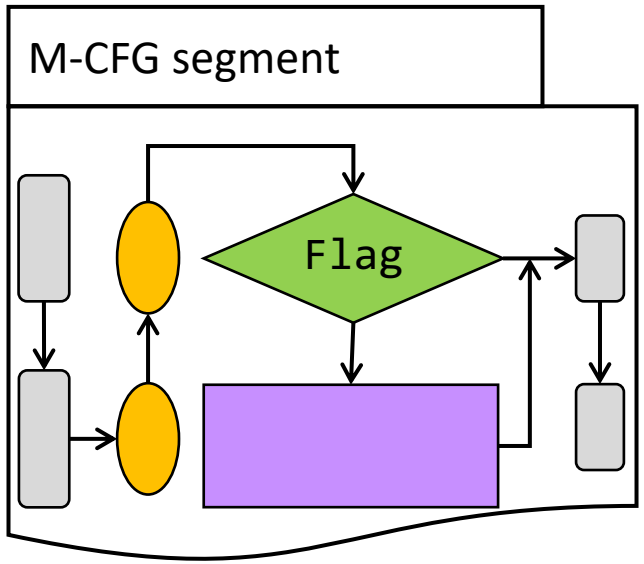

```
1  void BubbleSort(int *A, int N) {
2    for (int i=0; i<N; ++i) {
3      for (int j=i+1; j<N; ++j) {
4        bool flag = (A[i]>A[j]);
5        if (flag) {
6          Swap(A, i, j);
7  }}}} Branch: dyn_flag {
8          Call Swap(dyn_A, L0, L1);
9  }}}}void Swap(int *S, int i, int j) {
10   int tmp = S[i];
11   S[i] = S[j];
12   S[j] = tmp;
13 }
```

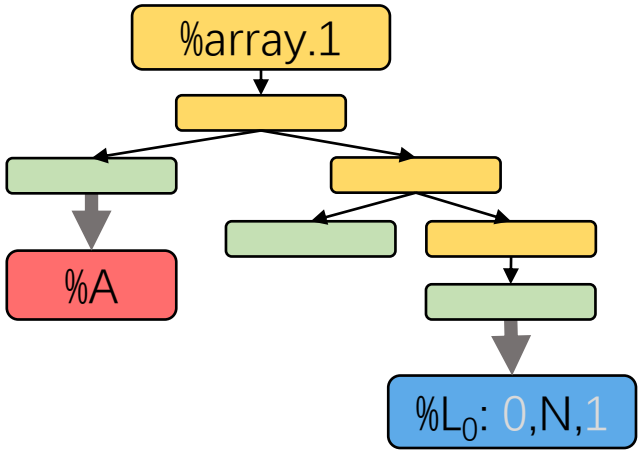
Memory Access Skeleton

```
Function BubbleSort(dyn_A, dyn_N) {
  L0: 0, dyn_N, 1 {
    L1: L0, dyn_N, 1 {
      Load1: dyn_A+L0*4; Load2: dyn_A+L1*4;
Function Swap(S, i, j) {
  Load3 : S+i*4; Load4 : S+j*4;
  Store1: S+i*4; Store2: S+j*4;
}
```

Sample Tool 1: S-Tracer





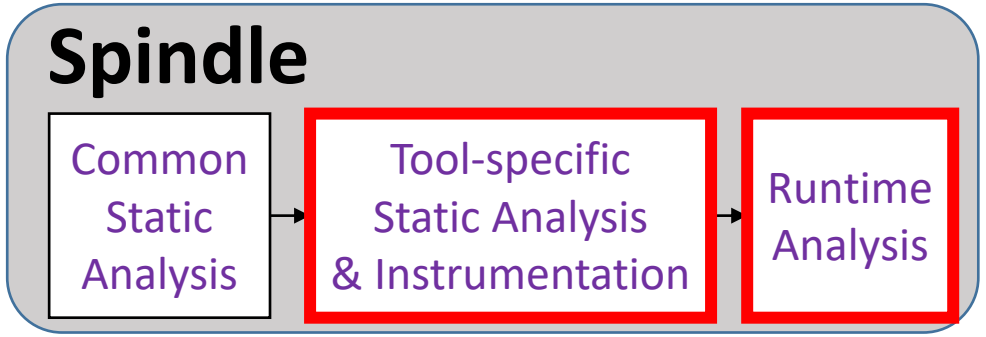



```
1  void BubbleSort(int *A, int N) {
2    recordAddr(A, variable_id);
3    recordLoop(N, loop_id);
4    for (int i=0; i<N; ++i) {
5      for (int j=i+1; j<N; ++j) {
6        bool flag = (A[i]>A[j]);
7        recordPath(flag, path_id);
8          if (flag) {
9            Swap(A, i, j);
10 }}}}
11 // Omit function Swap
```

Dynamic Trace

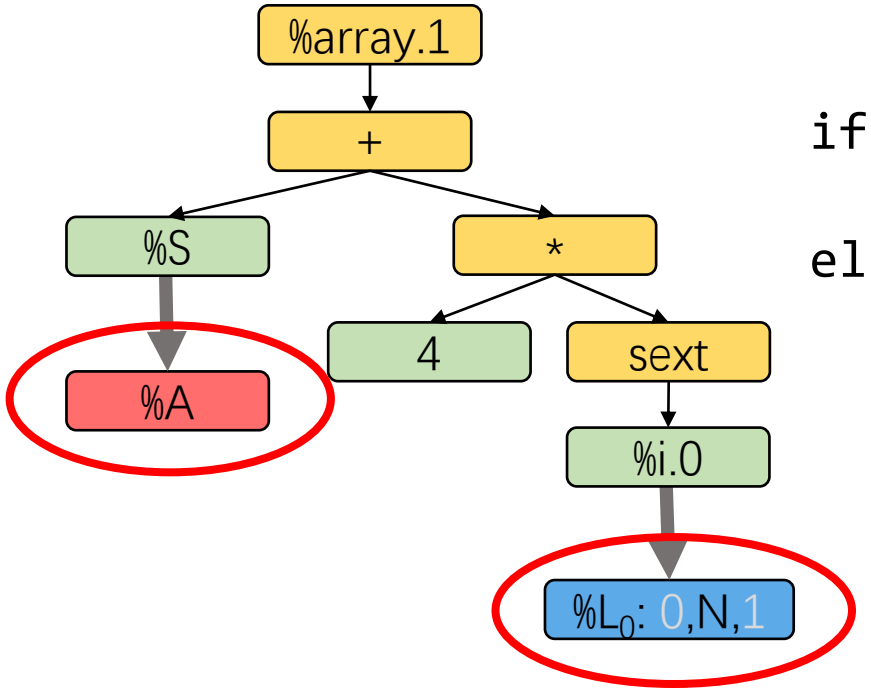
```
BubbleSort {
 dyn_A:
  0x7fffdfc58320;
 dyn_N:
  10;
 dyn_flag:
  {0,0,1,1,0,...,1,1};
}
```

S-Detector Example




```
1  void BubbleSort(int *A, int N){
2    checkValid(access_id,
3               N<=sizeof(A));
4    for (int i=0; i<N; ++i) {
5      for (int j=i+1; j<N; ++j){
6        bool flag = (A[i]>A[j]);
7        if (flag){
8          Swap(A, i, j);
9  }}}}
10
11 void Swap(int *S, int i, int j){
12   int tmp = S[i];
13   S[i] = S[j];
14   S[j] = tmp;
15 }
```

$S[i]$ is valid ⟵ $i < sizeof(S)$



if $N \leq sizeof(A)$:
    access $S[i]$ is valid
else:
    access $S[i]$ may not be valid

Only A and N need to be checked!

Challenges and Solutions

- Multi-threaded programs
  - Record when the thread is ***created*** and ***joined***, then apply **per-thread analysis** as the single-thread programs
- Register spilling & Call convention
  - IR level register allocation simulator & call convention simulator
- External libraries without source code
  - Use dynamic tools to collect the memory accesses from the libraries, and merge them to Spindle

Test Platform and Programs

- Platform
  - Intel Xeon E7-8890 (v3), running CentOS 7.1
  - 128GB of DDR3 memory
  - 1TB SATA-2 hard disk
- Benchmarks
  - NPB: BT, CG, EP, FT, IS, LU, MG, SP
  - SPEC CPU: 400.perlbench, 401.bzip2, 403.gcc, 429.mcf, 433.milc, 445.gobmk, 456.hmmer, 458.sjeng, 464.h264ref, 470.lbm, 482.sphinx3
  - Parsec: streamcluster (SC), freqmine (FQ), blackscholes (BS)
- Applications
  - BFS from Graph 500
  - MNIST, Kissdb, Fido, Mapreduce word count (WC)

Spindle and Spindle-tool Usage

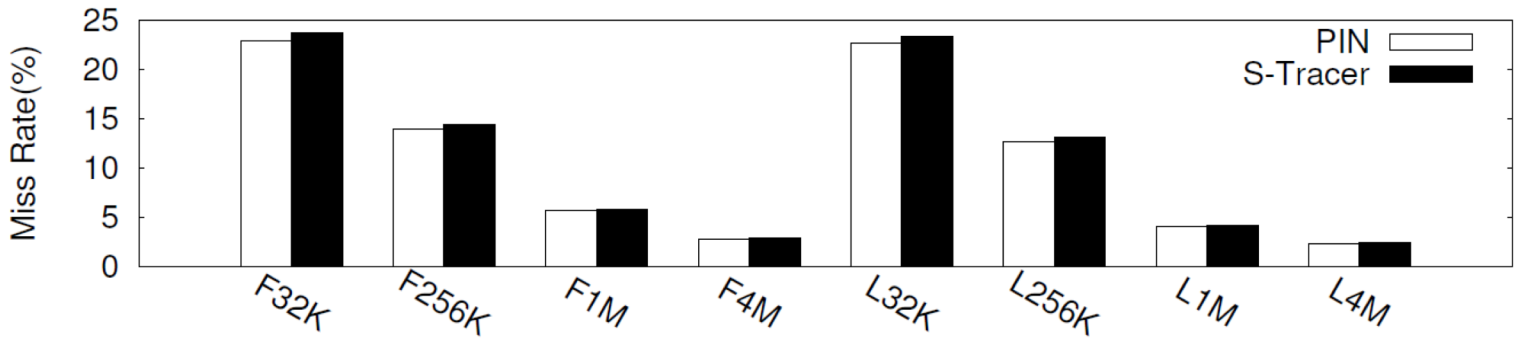
- For Spindle-based tool developers
  - LLVM plugin: tool-specific static analysis
    - For S-Tracer it dumps the MAS as static trace and adds instrumentation functions
    - For S-Detector it analyzes the object boundaries and validation before reading/writing
  - Runtime library: tool-specific runtime instrumentation
    - For S-Tracer it records compilation time unknown values as dynamic trace
    - For S-Detector it creates shadow memory to check the boundaries and validation
  - S-Tracer and S-Detector each implemented in < 500 Lines of C++ code

- For end-users of Spindle-based tools (LLVM plugin):

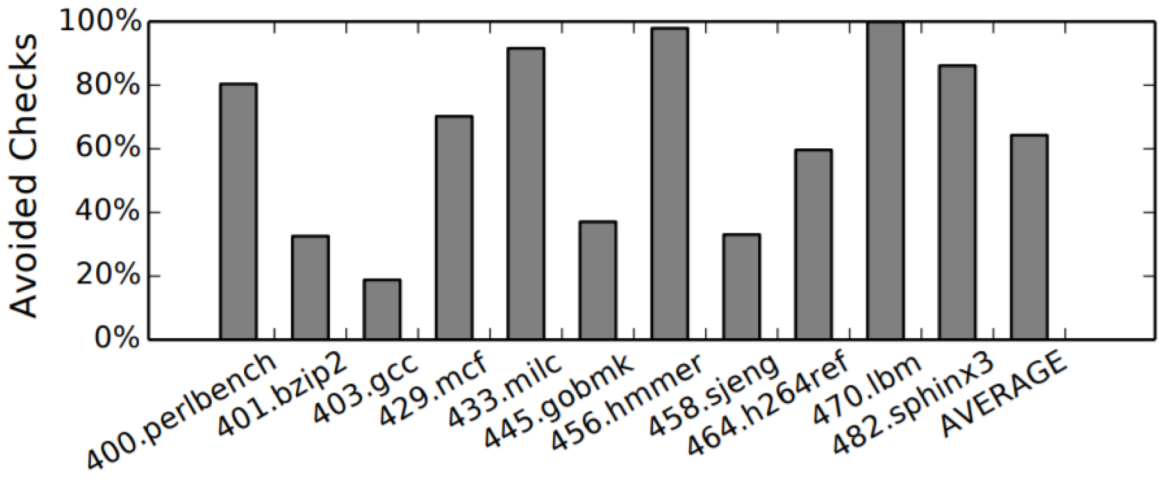
```
clang –O3 -m64 -Xclang -load -Xclang libSTool.so source.c && ./a.out
```

Evaluation

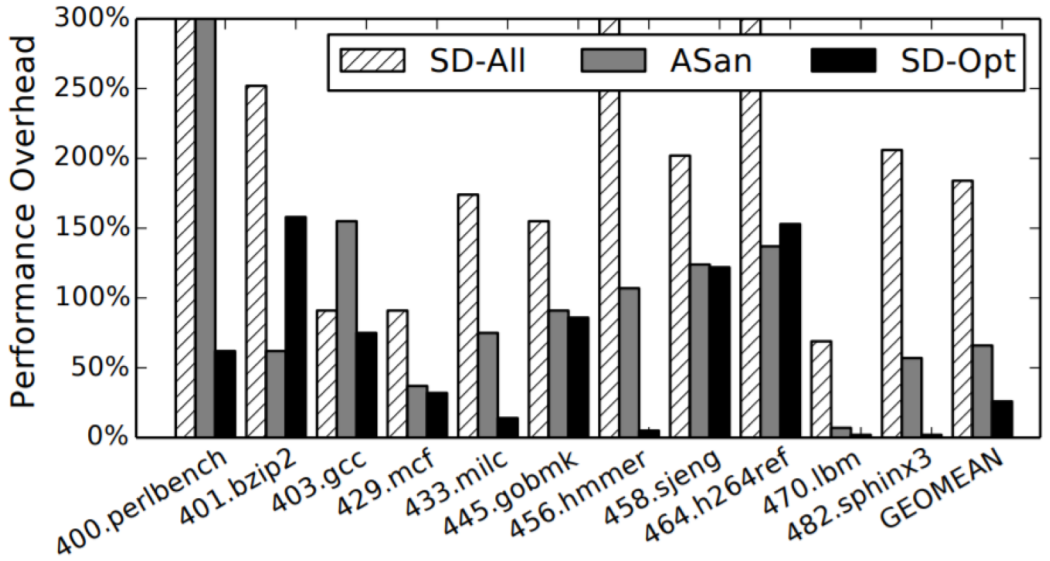
- Compilation overhead:
  - 2% to 35% of the original LLVM compilation cost (average at 10%)
- Correctness:
  - Full trace size difference between 0.5% and 6% compared with PIN (median at 3.2%)
  - Heap trace difference ratio between 0.0% and 4.7% compared with PIN (median at 1.5%)
- Cache simulation for worst case: BFS



Evaluation: S-Detector



Reduction in runtime memory checks

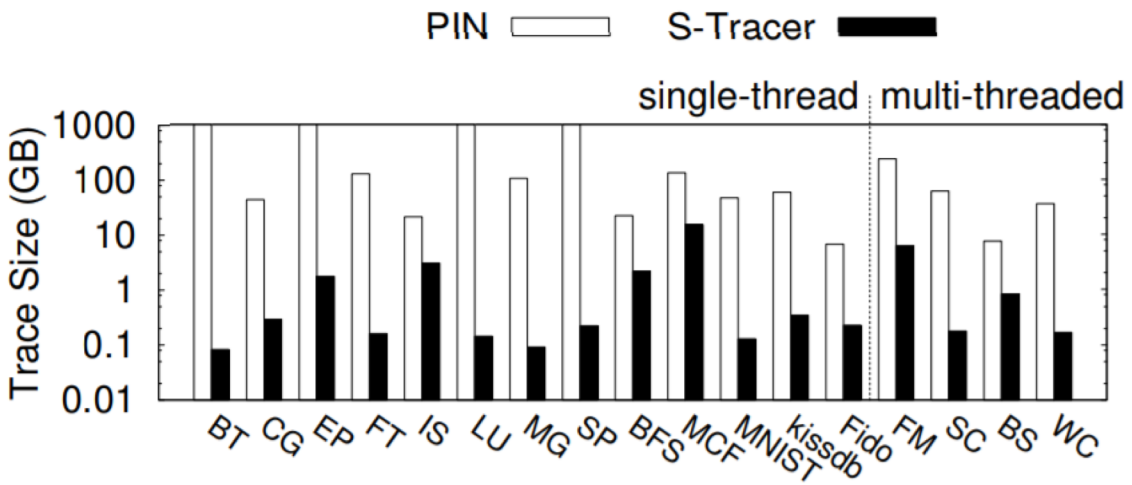


Overhead comparison (bars over 300% truncated)

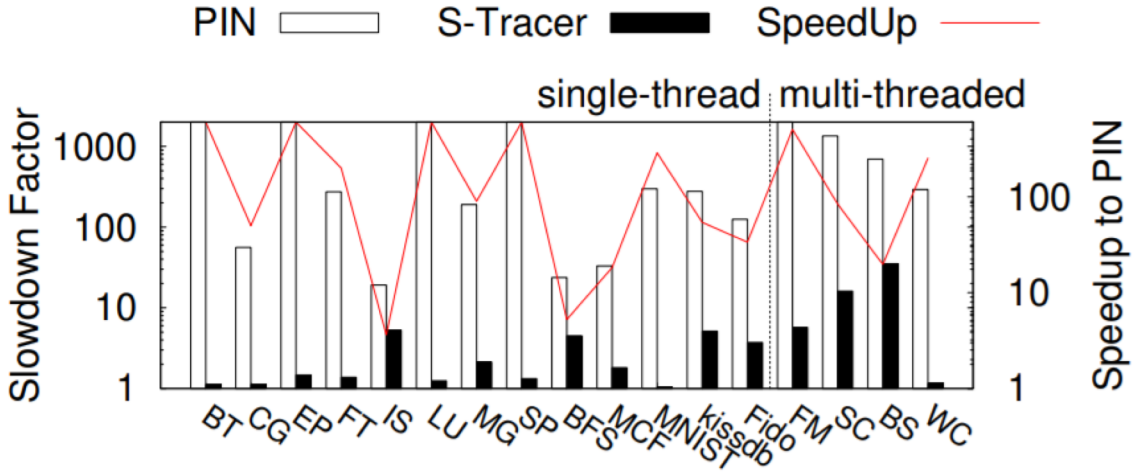
Spindle enables S-Detector to cut runtime memory checks by 64% on average

S-Detector brings down runtime overhead to 26% on average compared with ASan

Evaluation: S-Tracer



Trace size comparison



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

S-Tracer achieves reduction by orders of magnitude in trace size from the PIN baseline

S-Tracer reduces slowdown from PIN by a factor of 61x on average

Conclusion

- Spindle framework for memory access monitoring
  - **Computing** rather than **collecting** memory accesses
  - Falling back to conventional runtime instrumentation when necessary
  - General underlying mechanism supporting diverse tools

- Two sample Spindle-based tools: **S-Detector** and **S-Tracer**
  - Illustrated significant trimming of instrumentation
  - Greatly reduced runtime overhead and trace size

- Future work
  - Support C++ codes
  - Provide richer APIs for static analyzing

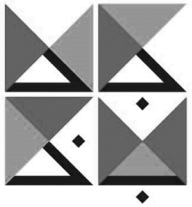
Q&A

Thank you!



PACMAN

pacman.cs.tsinghua.edu.cn



ds.qcri.org

Backup Slides

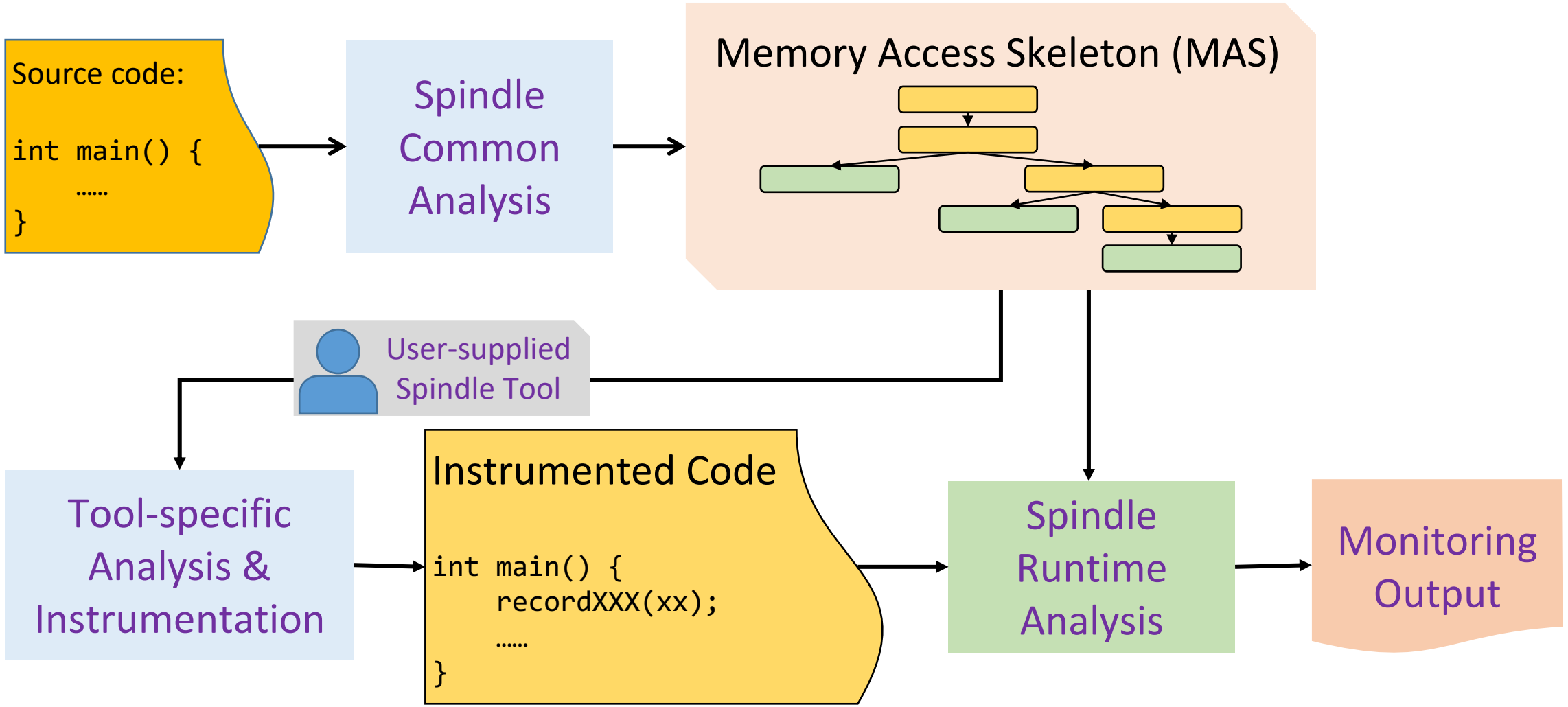
Multi-threaded programs

```
void thread1() {
    recordThreadID(pthread_self());
    // do some thing
}}

int main() {
    recordThreadID(pthread_self());
    pthread_t tid;
    int err;
    err = pthread_create(&tid, NULL, thread1, NULL);
    recordCreateThread(pthread_self(), tid, thread1);
    if (0 != err) {
        printf("Can't create thread\n");
    }
    int **ret = NULL;
    pthread_join(tid, (void **)ret);
    recordJoinThread(pthread_self(), tid);
    return 0;
}
```

For multi-threaded programs, for example, pthread programs, Spindle will record when the thread is ***created***, when it is ***joined***, along with the ***thread ID*** for each function.

Then Spindle can apply **per-thread analysis** as the single-thread programs.

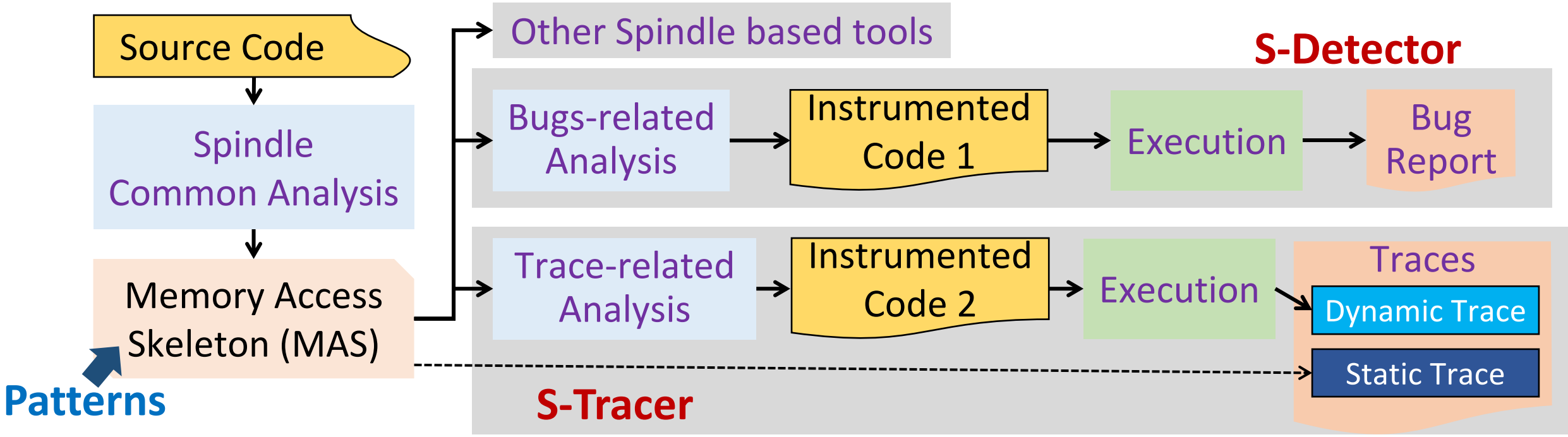

Source code:
int main() {
……
}
Spindle Common Analysis
Memory Access Skeleton (MAS)
User-supplied Spindle Tool
Tool-specific Analysis & Instrumentation
Instrumented Code
int main() {
recordXXX(xx);
……
}
Spindle Runtime Analysis
Monitoring Output

Spindle: Informed Memory Access Monitoring

S-Tracer for memory tracing

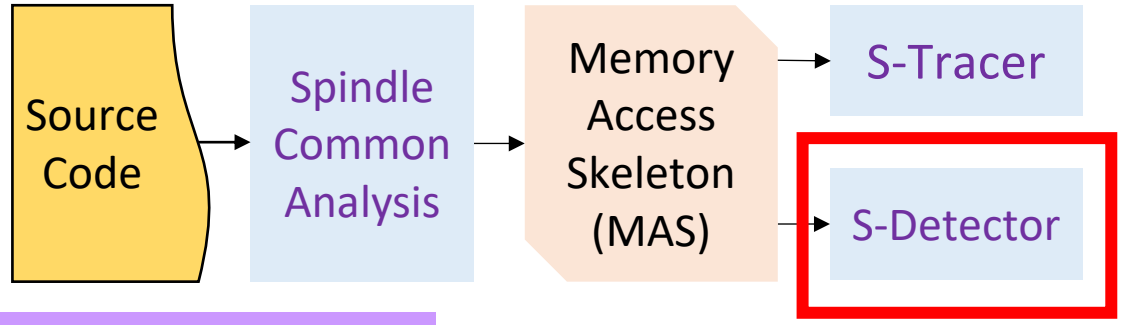
S-Detector for memory bug detection

- Analyze code to find memory access patterns at compile time
  - Loop, branch, array index, structure offset
- Record non-computable parameters at runtime
  - Program input, heap variable address



S-Detector Example

```
1  while (pos-1 && red_cost >
2          (cost_t)new[pos/2-1] .flow) {
3      new[pos-1].tail = new[pos/2-1].tail;
4      new[pos-1].head = new[pos/2-1].head;
5      // More codes.
6      pos = pos/2;
7      new[pos-1].tail = tail;
8      // More codes
9  }
```



In-struct accesses

Struct type $cost_t$'s range:

$$[struct_base, struct_base + struct_size)$$

Struct access to array new's member:

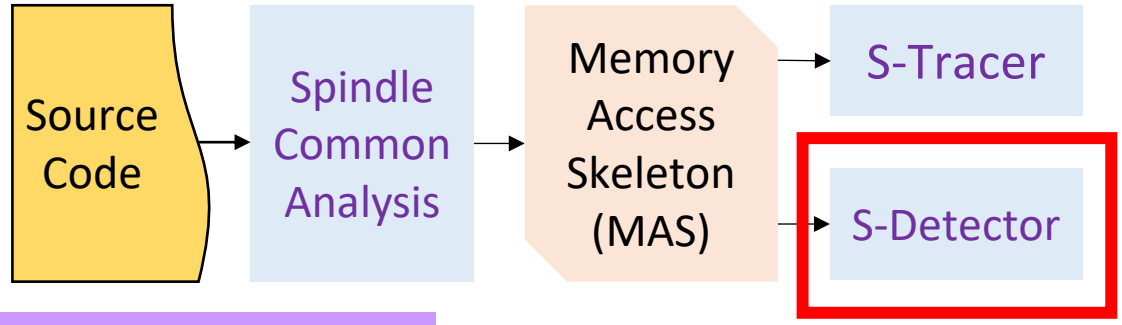
$$addr = struct_base + const_offset$$

Such accesses are valid only if:

$$const_offset < struct_size$$

and accesses to array new is valid.

S-Detector Example




```
1  while (pos-1 && red_cost >
2          (cost_t)new[pos/2-1] .flow) {
3      new[pos-1].tail = new[pos/2-1].tail;
4      new[pos-1].head = new[pos/2-1].head;
5      // More codes.
6      pos = pos/2;
7      new[pos-1].tail = tail;
8      // More codes
9  }
```

Only have to record: ***pos_init***, ***pos_end***, ***new_size***.
(***struct_size*** is known at compilation time)

In-struct accesses

Struct type $cost_t$'s range:

$$[struct_base, struct_base + struct_size)$$

Struct access to array new's member:

$$addr = struct_base + const_offset$$

Such accesses are valid only if:

$$const_offset < struct_size$$

and accesses to array new is valid.

In-loop accesses

Loop induction variable pos's range:

$$[pos_init, pos_final)$$

Array new's size:

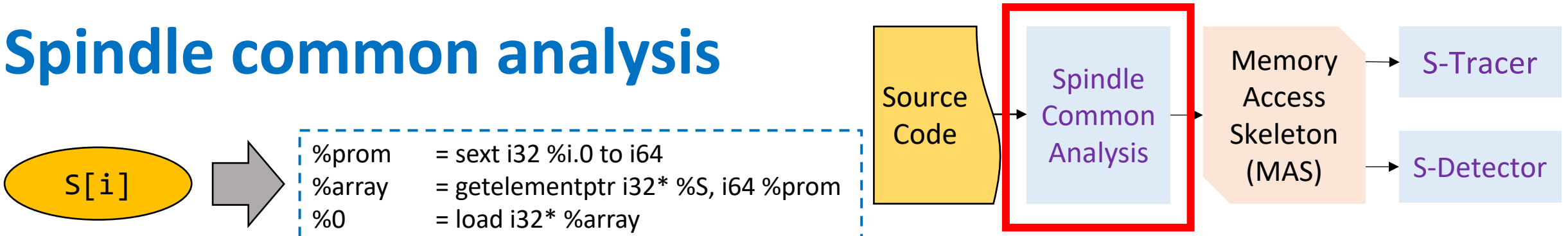
$$new_size$$

Array accesses valid only if:

$$(pos_init - 1) * struct_size < new_size \quad (1)$$

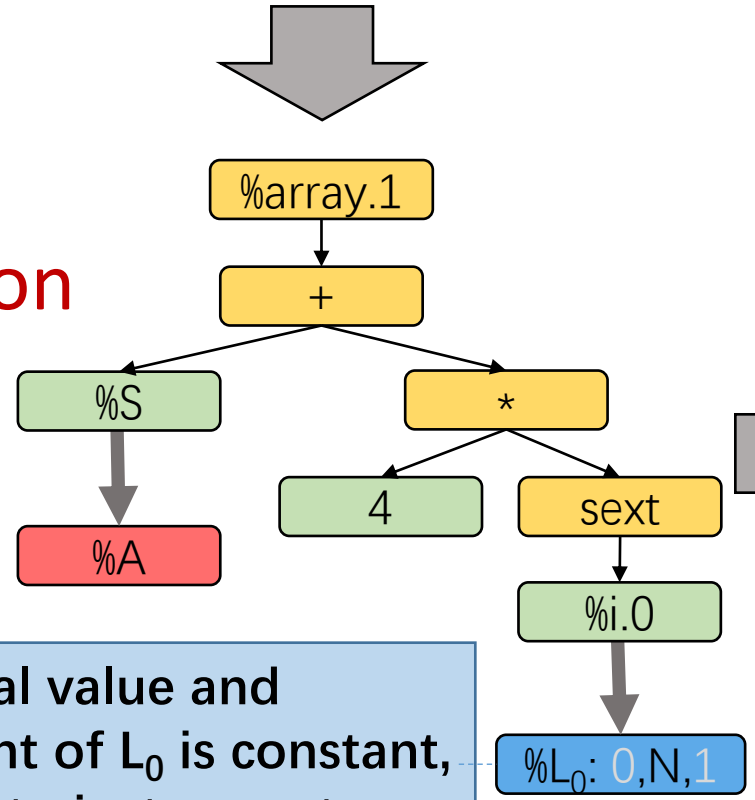
$$\&\& \quad pos_end/2 - 1 \geq 0 \quad (2)$$

Spindle common analysis



Inter-Dependence Analysis

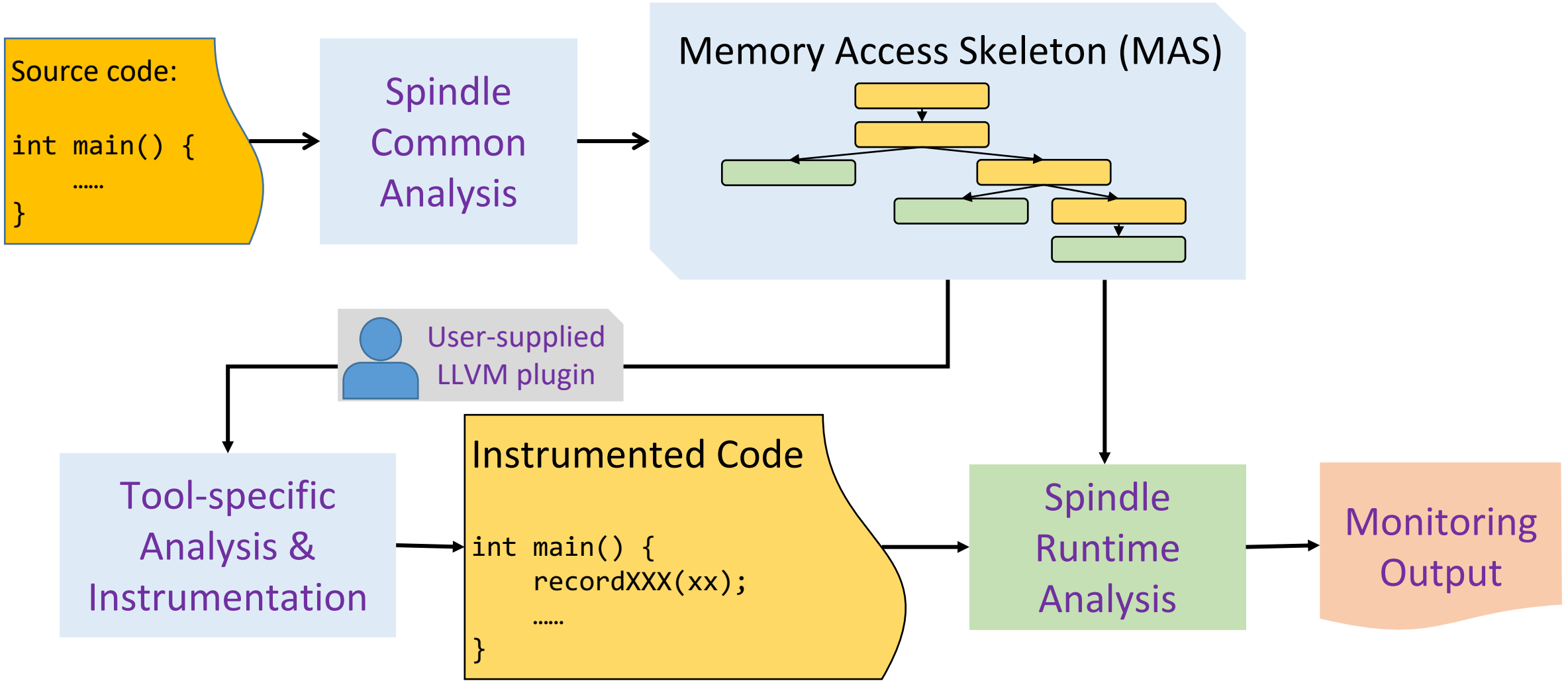
Instrumentation Analysis



The initial value and increment of L_0 is constant, no need to instrument.

```
1  void BubbleSort(int *A, int N) {
2       recordAddr(A, variable_id);
3       recordLoop(N, loop_id);
4       for (int i=0; i<N; ++i) {
5           for (int j=i+1; j<N; ++j) {
6               bool flag = (A[i]>A[j]);
7               recordPath(flag, path_id);
8               if (flag) {
9                   Swap(A, i, j);
10 }}}}
11
12 void Swap(int *S, int i, int j) {
13      int tmp = S[i];
14      S[i] = S[j];
15      S[j] = tmp;
16 }
```

Spindle overview



Spindle overview

