

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

* 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

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

Full monitoring for a long-running program introduces large overhead
Conventional one-by-one checking is slow

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

```c
for (int i=0; i<N; ++i) {
    a[i] = i;
    checkAccess(&a[i]);
}
```

- Overhead >100x
- Trace size ~10GB/s

- Static + dynamic hybrid tools
  - 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*

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

Learn memory access pattern from source code

• Only starting address `a` and loop count `N` are unknown
  • Record at runtime

```java
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 $\rightarrow$ 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

int main() {
    ...... 
}

Common static analysis

Tool-specific Analysis & Instrumentation

int main() {
    recordXXX(xx);
    ...... 
}

Memory Access Skeleton (MAS)

Instrumented code

Spindle Runtime Analysis

Monitoring Output

Developer-supplied LLVM plugin

Source code
Spindle common analysis

- Control flow analysis

M-CFG of BubbleSort

Loop 0
(0, N, +1)

Load 2

Flag
False

End Loop 0

Load 1

Loop 1
(0, N, +1)

Call Swap
(A, L0, L1)

True

End Loop 1

M-CFG of Swap

Load 3

Load 4

Store 1

Store 2

void BubbleSort(int *A, int N) {
  for (int i=0; i<N; ++i) {
    for (int j=i+1; j<N; ++j) {
      bool flag = (A[i]>A[j]);
      if (flag) {
        Swap(A, i, j);
      }
    }
  }
}

void Swap(int *S, int i, int j) {
  int tmp = S[i];
  S[i] = S[j];
  S[j] = tmp;
}
Spindle common analysis

Inter-Dependence Analysis

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

Tsinghua University, Qatar Computing Research Institute
Sample Tool 1: S-Tracer

void BubbleSort(int *A, int N) {
    recordAddr(A, variable_id);
    recordLoop(N, loop_id);
    for (int i=0; i<N; ++i) {
        for (int j=i+1; j<N; ++j) {
            bool flag = (A[i]>A[j]);
            recordPath(flag, path_id);
            if (flag) {
                Swap(A, i, j);
            }
        }
    }
    // Omit function Swap
}

Spindle

Common Static Analysis
Tool-specific Static Analysis & Instrumentation
Runtime Analysis

Dynamic Trace
BubbleSort {
    dyn_A: 0x7fffdfc58320;
    dyn_N: 10;
    dyn_flag: {0,0,1,1,0,...,1,1};
}
S-Detector Example

```c
void BubbleSort(int *A, int N){
    checkValid(access_id, N<=sizeof(A));
    for (int i=0; i<N; ++i) {
        for (int j=i+1; j<N; ++j){
            bool flag = (A[i]>A[j]);
            if (flag){
                Swap(A, i, j);
            }
        }
    }
}

void Swap(int *S, int i, int j){
    int tmp = S[i];
    S[i] = S[j];
    S[j] = tmp;
}
```

**Spindle**

- **Common Static Analysis**
- **Tool-specific Static Analysis & Instrumentation**
- **Runtime Analysis**

**S[i] is valid**  
**i < sizeof(S)**

- \( i \leq \text{sizeof}(A) \):
  - 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 -03 -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

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

Overhead comparison (bars over 300% truncated)

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

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
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:

```c
int main() {
    ......
}
```

Spindle Common Analysis

Tool-specific Analysis & Instrumentation

User-supplied Spindle Tool

Instrumented Code

```
int main() {
    recordXXX(xx);
    ......
}
```

Memory Access Skeleton (MAS)

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

Source Code

- Spindle Common Analysis
- Memory Access Skeleton (MAS)

Other Spindle based tools

- Bugs-related Analysis
- Trace-related Analysis
- Instrumented Code 1
- Instrumented Code 2
- Execution

**S-Detector**

- Execution
- Bug Report
- Traces
  - Dynamic Trace
  - Static Trace

**S-Tracer**

- Traces
  - Static Trace

Tsinghua University, Qatar Computing Research Institute
S-Detector Example

```
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;
  // More codes.
  pos = pos/2;
  new[pos-1].tail = tail;
  // More codes
}
```

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

```c
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;
    // More codes.
    pos = pos/2;
    new[pos-1].tail = tail;
    // More codes
}
```

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

&& pos_end/2 - 1 ≥ 0
```

Only have to record: `pos_init`, `pos_end`, `new_size`.

`struct_size` is known at compilation time.

Spindle common analysis

S[i] → %prom = sext i32 %i.0 to i64
 %array = getelementptr i32* %S, i64 %prom
%0 = load i32* %array

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

void BubbleSort(int *A, int N) {
  recordAddr(A, variable_id);
  recordLoop(N, loop_id);
  for (int i=0; i<N; ++i) {
    for (int j=i+1; j<N; ++j) {
      bool flag = (A[i]>A[j]);
      recordPath(flag, path_id);
      if (flag) {
        Swap(A, i, j);
      }
    }
  }
}

void Swap(int *S, int i, int j) {
  int tmp = S[i];
  S[i] = S[j];
  S[j] = tmp;
}
Spindle overview

Source code:
```c
int main() {
    ...... 
}
```

Spindle
Common
Analysis

Memory Access Skeleton (MAS)

Instrumented Code
```c
int main() {
    recordXXX(xx);
    ...... 
}
```

Tool-specific Analysis & Instrumentation

User-supplied LLVM plugin

Spindle Runtime Analysis

Monitoring Output
Spindle overview

Spindle supplied internal analysis

Source code:

```c
int main() {
    ...
}
```

Memory Access Skeleton (MAS)

Spindle supplied internal analysis

User supplied external analysis

Spindle supplied runtime library

User-supplied LLVM plugin

Tool-specific Analysis & Instrumentation

Instrumented Code

```
int main() {
    recordXXX(xx);
    ...
}
```

Spindle Runtime Analysis

Monitoring Output

Runtime Library