Efficient Online Tracing and Analysis

References:

V. Nagarajan, D. Jeffrey, R. Gupta, and N. Gupta
ONTRAC: A System for Efficient Online TRACing for Debugging, ICSM 2007

V. Nagarajan, H-S. Kim, Y. Wu, and R. Gupta
Dynamic Information Flow Tracking on Multicores, INTERACT 2008

ONTRAC

- Online Tracing System
  - Tracing, while the program executes
  - Tracing in a real machine
- Targeted towards Debugging.
  - Scenario: Dynamic Slicing based Debugger
  - Dynamic Dependences Traced
  - Trace those dependences that can be potentially useful to capture the bug

Online Dynamic Analysis

- Applications
  - Debugging
    - Tracing for Debugging (ONTRAC)
    - Eraser
  - Security
    - DIFT (Multicore DIFT)
  - Performance
    - Speculation

Debugging - Dynamic Slicing

- Dynamic Slicing based debugging
  - Korel and Laski proposed the idea; Korel and Rilling applied debugging
  - Agrawal and Horgan PLDI '90, Wang and Roychoudary ICSM '04, Zhang & Gupta PLDI '04, FSE '04
- Principle
  - An erroneous state is a result of statements that influenced this state
  - Perform a backward dynamic slice on the erroneous state
  - Transitive closure of statements that influenced the state
- Computation of Dynamic Slice
  - Requires dynamic data and control dependences exercised - DDG
- Debugging with Dynamic Slicing
  - Generate DDG for the execution
  - Slice backwards from erroneous state
  - Examine Statements in the slice

Cost of Dynamic Slicing

- Space and Time Costs
  - Size of DDG around 1.5 GB for 100 million instructions (< 1 sec)
  - Time to perform 1 dynamic Slice around 10 minutes
- Zhang and Gupta – PLDI 2004
  - Compact representation for DDG
    - 1.5 GB to 100 MB
  - Reduced Slicing time to a few seconds
  - Steps Involved
    - 1. Online: Collect address and control flow trace (15x)
    - 2. Post-processing: Compute compacted DDG (500x)
    - 3. Perform Slicing on Compacted DDG (few seconds)

Problem!

- Debugging an iterative process
  - Execute
    - 1. Collect address, control flow trace
  - Post-processing (500x)
  - Fix
    - 3. Perform several Slices

Post-processing in critical path!
**ONTRAC Outline**

- Debugging Using Dynamic Slicing
- Our Approach
- ONTRAC System
- Optimizations for limiting trace
- Experimental Evaluation

**Our Approach**

- Compute DDG online as program executes.
- Store the DDG in a circular Buffer.
- Perform Slicing using DDG in buffer.
- Use Optimizations to limit the DDG size
  - Online Compression
  - More Execution history per byte of buffer
  - Reduces the Execution time overhead.

**Online Computation of DDG**

- DDG Representation
  - 2 stmts: def and use
    - (inst id, use) (inst id, use)
  - If(predicate) { stmt…}
    - (inst id, stmt) (inst id, predicate)

- Shadow Memory
  - Separate memory space maintained
  - Contains the (inst id, instance) pair of most recent instruction
  - Enables lookup when the above value is used subsequently

- Data Dependences are computed dynamically.
  - Control dependencies are implicitly captured.
  - Recovered using a fast one-pass post-processing step.

**Online Computation DDG**

- Data Dependency
  - 1. a = const
    - \(a_{\text{shadow}} = \text{stmt 1, inst } x\)
  - 2. \(b = a\)
    - \(b_{\text{shadow}} = \text{stmt 2, inst } y\)

- Control Dependency
  - Static control dependences available
  - Unstructured code \(\rightarrow\) potential multiple control ancestors
  - Dynamic control dependence = ancestor that was defined latest

**Post-Processing**

- After the program terminates/crashes
  - One pass post processing step

- Post Processing
  - Classify the collected dependences instruction-wise in increasing order of instances
  - Compute the dynamic control dependence

- Time taken
  - Dominated by time for reading the dependence info.
  - Does not exceed 10 secs for 16MB trace buffer.
Outline

• Debugging Using Dynamic Slicing
• Our Approach
• ONTRAC System
• Optimizations for limiting trace size
• Experimental Evaluation
• Conclusions and Future Work

ONTRAC System

• Built using DBT
  – Dynamic Binary Translator
  – Support for instrumentation
  – DynamoRIO, Intel PIN, Valgrind

• Shadow Memory Support
  – We used half-and-half scheme
  – Robust shadow memory support [Nethercote VEE ’07]

• Design Decision: Storing DDG in a circular Buffer
  – Otherwise, too slow [Tallam and Gupta TACO’07]
  – Bugnet Observation [Narayanasamy ASPLOS ’06]
    • Root cause of the bug found within a window of 18 million instructions.

ONTRAC System

Optimization

• Optimizations
  – Limits the size of dependence trace
  – Effects
    • Slicing time comparable with prior work
    • More execution history per byte of trace buffer
    • Limits execution time overhead

  – Generic Optimizations
    • Limits the trace, by exploiting program properties
    • The optimizations are applicable, irrespective of the use of trace information

  – Targeted Optimizations
    • Optimizations that are exclusively targeted toward Debugging

Generic Optimizations

• Basic Block Optimization
  – Dependencies inferred by static examination of code at Basic block level

• Trace Optimization
  – Sequence of frequently executed Basic blocks that do not cross loop boundaries
  – Traces can be identified at runtime.
  – Several dependencies can hence be inferred at instrumentation time.
Generic Optimizations

```c
int fun(char *input) {
    1. len = strlen(input)
    2. array = malloc(len + 2);
    3. while (j < len) {
        4.     j++;
        5.     i = 2*j;
        6.     array[i] = input[j];
        7.     if (isupper(array[j]))
        8.         array[i+1] = input[j];
        9.     else
        10.         array[i+1] = toupper(input[j]);
    }  //end while
    11. }  //end fun
```

Generic Optimization

- **Trace Optimization**
  - Memory Dependences can also be inferred statically
  - An aliasing check is added

- **Redundant Load Optimization**
  - A significant percentage of dynamic loads are redundant (~ 20%)
  - Redundant loads are detected and not traced

Targeted Optimizations

- **Selective Tracing**
  - Incorporates the knowledge of programmer
  - Selectively trace functions which are likely to contain error
    - Eg. In most cases the bug is not present in library functions
  - We can't ignore other functions completely

Selective Tracing

- Propagation performed in functions that are not expected to contain error
  - Dependencies are propagated, but not output

Forward Slice Optimization

- **Observation:** Root-cause of the failure contained in forward slice of input
  - Transitive closure of the instructions that depend on the input.
  - Intuitively, harder-to-find errors are revealed on particular inputs.
  - Also shown in prior work, failure inducing chops [Gupta et al. ASE ’05]
  - Failure inducing chop is intersection of backward slice from error and forward slice of input
  - Shown to be much smaller than backward slice

Forward Slice Implementation

- Selectively trace those instructions in the forward slice of the input
  - Propagate extra forward slicing bit, which is also stored in shadow memory
  - Indicates whether the memory location is dependent on input.
  - Tracing performed, when the forward slicing bit is set.
Outline

- Debugging Using Dynamic Slicing
- Our Approach
- ONTRAC System
- Optimizations for limiting trace
- Experimental Evaluation
- Conclusions and Future Work

Experimental Evaluation

- Goals of evaluation
  - Execution time overhead
  - Effects of optimization
  - Rate at which trace buffer is filled
    - Trace rate
  - Effects of Optimization on trace rate
    - Make sure that targeted optimizations don’t affect bug detection
- We used a trace buffer size of 16MB
  - Intel Pentium 4 3GHz machine with 2GB memory
  - DynamoRIO and Intel PIN infrastructure for dynamic instrumentation

Efficacy of Targeted OPT

- Test if the reduced trace info enough to capture bug
- We considered 6 real world memory bugs
- For Selective tracing OPT
  - Tracing performed only in functions containing bug
  - Propagation performed in other functions
- For Forward Slicing OPT
  - Instructions in forward data slice traced.

<table>
<thead>
<tr>
<th>Bug</th>
<th>Type</th>
<th>S.T</th>
<th>F.S</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc-1.06</td>
<td>Heap/Overflow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mc-4.5.55</td>
<td>Stack/Overflow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mutt-1.4.2</td>
<td>Heap/Overflow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>phe-4.4.44</td>
<td>Heap/stack/Overflow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>squid-2.3</td>
<td>Heap/Overflow</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Overhead of ONTRAC

- Spec Integer programs considered
  - Training input provided.
  - Overhead of Baseline : around 19x
    - Marked improvement from 540x slowdown
  - After basic block OPT: around 15x
  - After Trace OPT: around 12x
  - After Selective tracing OPT : around 9x
    - Performed tracing within 5 largest static functions
    - After Redundancy OPT: around 15x
      - Extra work done to eliminate redundancies
      - Finally after forward slicing Optimization: 19x
        - Extra work done to maintain forward slicing information

Trace Rate

- Trace rate of Baseline : around 16 bytes/instruction
  - After basic block OPT: around 8 bytes/instruction
    - 50 % reduction
  - After Trace OPT: around 5 bytes/instruction
  - After Redundancy OPT: around 4 bytes/instruction
    - 20 % reduction
  - After Selective tracing OPT : around 2.2 bytes/instr.
    - Additional 40% reduction
  - After forward slicing Optimization: 0.8 bytes/instr.
    - Additional 4 fold reduction
    - Only 25% of dependences in forward data slice of ip
  - 16 bytes/instruction to 0.8 bytes/instruction
    - Online Compressions comparable to prior work.

Execution Histories stored

- Execution history stored in 16 MB buffer
  - After Generic Optimizations: 3.4 million instructions
  - After Selective Tracing Optimization: 7 million instructions
  - After Forward Slicing Optimization: 20 million instructions

- Greater than 18 million instructions in a 16 MB buffer
ONTRAC Review

- ONTRAC, software Tracing System
  - Traces Dynamic dependences exercised.
  - Main ideas
    - Computes dependencies online
    - Stores them in a buffer
    - Online Compression
      - Generic Optimizations
    - Trace dependences useful to capture the bug.
  - Slows down the program only by a factor of 19
  - Able to capture a history of 20 million instructions in a 16MB buffer.

Dynamic Analysis: Costs

- Dynamic Analysis
  - Need to execute instrumentation code
  - Need to save and restore registers
  - Cache Pollution: Shadow Memory
  - Can we use multicores to speed up?
    - Extra processing power
    - Extra memory
      - Registers
      - Caches
    - Challenge
      - Communication and synchronization

DIFT: Background

- DIFT: Dynamic Information flow Tracking
  - Promising Technique for detecting a range of attacks
    - Buffer overflows
    - Format String Attacks
    - SQL Injection attacks
  - No Recompilation required
    - Provides protection for legacy binaries

DIFT

- DIFT Principle
  - Data from untrusted input channels tainted. E.g. Network input
  - Usage of tainted data in “malicious” fashion detected
    - Policy determines the malicious usage
  - Eg. Tainted data should not alter the PC

- DIFT Implementation
  - Each register/word of memory associated with a taint-bit (or tag)
    - 0: untainted 1: tainted

<table>
<thead>
<tr>
<th>Original</th>
<th>Tracking Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Mem)+Input</td>
<td>Tag(Mem)+=1</td>
</tr>
<tr>
<td>R3=R1+R2</td>
<td>Tag(R3)=Tag(R1)+Tag(R2)</td>
</tr>
<tr>
<td>Jmp R3</td>
<td># Tag(R3)==1 exception()</td>
</tr>
</tbody>
</table>

Prior DIFT Implementations

- Hardware based
  - [Suh ASPLOS ’04], [Raikia ISCA ’07]
    - Tracking performed concurrently.
    - Specialized hardware changes.
      - Processor pipeline and Caches
      - DRAM and Memory bus

- Software based
  - [Newsome NDSS ’05], [LIFT MICRO ’06]
    - Uses a DBT to generate code for tracking
    - High overhead (around 4 times slowdown)
    - Sources of overhead
      - Almost all instructions require tracking
      - Need registers for storing taint bits
      - Cache Pollution due to taint bits
      - Need to prevent flags from getting clobbered by DIFT instructions.

Multicore DIFT

- Efficient DIFT without specialized HW?
  - Spawn helper thread which uses additional core for DIFT
    - Perform DIFT concurrently
  - Take advantage of extra registers and local cache in additional core
Helper thread performs only DIFT

- Information needs to be communicated for the following:
  - Branches (flags need to be communicated)
  - Indirect memory references (address register(s))
  - Indirect jumps

```
1. cmp eax, 03h
jge <exit>
// receive branch outcome
2. push eax
// spin until safe
// receive flag
3. jmp eax
// receive esp value
```

```
1. jge <exit>
// receive esp value
2. mov eax, (esp+off.)
3. cmp eax, $0
jz L1
raise exception
L1:
jmp eax
```

Enabling Communication

- SW based communication
- HW Support for Communication

SW based Queue

- Circular Queue using shared memory
  - [Ottoni MICRO '05, Wang CGO '07]
- Around 5-10 instructions for each enqueue / dequeue
  - Known as intra-thread latency
- Each enqueue/dequeue needs to at least go to L2
- Since L2 hit latency > 1 cycle, each dequeue causes lag for trailing thread.
  - Known as inter-thread latency

HW Support for Queue

- Dedicated HW queue b/w the cores. [Taylor ITPDS '05]
  - ISA support for enqueue and dequeue instructions
  - Intra-thread latency = 1 cycle
  - No separate synchronization required.
  - Dequeue blocks on empty
  - Inter-thread latency relatively lower
- Light weight memory enhancement [Rangan MICRO '06]
  - ISA support for enqueue/dequeue
  - Intra-thread latency = 1 cycle
  - Actually uses memory subsystem for communication
- HW Queue is general purpose hardware mechanism
  - Debugging – HeapMon [Shethy et al. IBM Journal '06]
  - Optimization [Ottoni MICRO '05]
  - Reliability [Wang CGO '07]

UCRiverside
Multicore DIFT

```
[Mem]← R1
[Mem]← R2
R1← [Mem]
R2← [Mem]
If R2 exception())
Jmp [P3]
```

```
[Mem]← R1
[Mem]← R2
R1← [Mem]
R2← [Mem]
If R2 exception())
Jmp [P3]
```

```
4x
```

```
4x
```

```
4x
```

Helper thread performs only DIFT!!

Still 4x, Since Main thread needs to wait before crucial instructions for fail safety!!

4x

Helper thread performs only DIFT!!

Enabling Communication

- SW based communication
- HW Support for Communication

SW based Queue

- Circular Queue using shared memory
  - [Ottoni MICRO '05, Wang CGO '07]
- Around 5-10 instructions for each enqueue / dequeue
  - Known as intra-thread latency
- Each enqueue/dequeue needs to at least go to L2
- Since L2 hit latency > 1 cycle, each dequeue causes lag for trailing thread.
  - Known as inter-thread latency
Communication Optimizations

- Indirect memory addressing causes sizable communication overhead (1.55 bytes/instr)
  - Significant indirect addresses based on esp, ebp
  - Maintain values of esp, ebp in helper thread.
  - Reduces overhead to 0.4 bytes/instr.
- eflags optimization
  - Instead of communicating all the bits of eflags, communicate just the status flags
  - Use lahf and sahf instructions; Communicate overflow flag separately. (Bruening’s Thesis 2004)
- Relaxed Fail-Safety
  - Malicious code typically executes system call to compromise the system
  - Fail Safety Synchronization only before system calls
    - Number of System calls < Indirect jumps
    - Reduces the overhead of synchronization

Relaxed Fail Safety

- Fail Safety
  - Main thread synchronizes with helper thread before indirect jumps
  - Otherwise, attack can take place before it is detected in the helper thread.
- Relaxed Fail-Safety
  - Malicious code typically executes system call to compromise the system
  - Fail Safety Synchronization only before system calls

Experiments

- DIFT Code Generation
  - StarDBT Infrastructure: DBT
    - Similar to Valgrind, PIN or DynamoRIO
  - Implemented HW support for Queue in Simics
    - Simulated Application under StarDBT
    - Dual core machine
    - Shared L2 cache
      - 4 way, 512 KB, 10 cycle latency
    - Local L1 cache
      - 2 way, 16 K, 1 cycle

Communication Bandwidth

- Communication Optimizations
  - OPT1: Indirect Addressing optimization: 3 bytes/cycle to 1.4 bytes/cycle
  - OPT2: EFLAGS Optimization 1.4 bytes/cycle to 1.2 bytes/cycle

Overhead using SWQ

- Overhead around 4x, which is similar to LIFT overhead
- High bandwidth requirement: Instructions for performing enqueue / dequeue the bottleneck

Overhead with HW Queue

- Size of Queue = 512 bytes, Inter-thread latency = 10 cycles
- Relaxed Fail Safety Implementation
- 48% Overhead for DIFT in Multicores using HWQ
Break-up of Overhead

- 48% = 25% (DBT) + 5% (L2 cache Pollution) + 8% (Main thread Stalling) + 10% (extra instructions for communication)

Sensitivity Analysis

- Sensitivity with Inter-thread Queue Latency
  - Varied latency from 10 cycles to 30 cycles
  - No significant increase in execution times.
  - Light Weight memory enhancement instead of fully HW queue
- Sensitivity with Queue Size
  - Marginal Increases in execution time with decreasing queue size
- Sensitivity with L2 Cache Size
  - Marginal Increases in execution time with decrease in L2 cache size.

Attack Detection

<table>
<thead>
<tr>
<th>Program</th>
<th>Vulnerabilities</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wilander’s attack Benchmarks</td>
<td>Return address, Function Pointer, Long Jump</td>
<td>Yes</td>
</tr>
<tr>
<td>ncompress4.2.4</td>
<td>Return Address</td>
<td>Yes</td>
</tr>
<tr>
<td>Polymorph0.4</td>
<td>Return address</td>
<td>Yes</td>
</tr>
<tr>
<td>SPEC</td>
<td>None</td>
<td>No</td>
</tr>
</tbody>
</table>

Multicore DIFT Review

- Designed and implemented DIFT on multicores.
  - Utilizes resources of additional core to perform DIFT
- The helper thread keeps up with the main thread
  - 48% Overhead utilizing HW Queue support
  - Overhead is tolerant to Queue latency
  - ISA support for enqueue / dequeue crucial

Conclusions

- Online Dynamic Analysis
  - Is reasonably efficient
  - Need to decide
    - What to track/store
    - How much to track/store
    - When we can not perform instrumentation
- Multicores can be used to speed up
  - Utilize processing power and memory
  - Need to reduce communication
  - Adapting to number of free cores
Example

```c
void fun (int choice) {
  int buf[10];
  // bounds checking
  if (choice > 3) exit(0);
  ...  
  // read into global
  global = fgets();
  // vulnerability
  strcpy(buf, global);
  ... 
  switch(choice):
  case 1:
    f1(buf);
  case 2:
    f2(buf);
  case 3:
    f3(buf);
}
```

(Example Source Code)

1. Assume eax is not tainted.
2. receive branch outcome
   1. jge <exit>
3. receive esp value
4. call <get>
5. *(i/p[] + offset) = 1
6. global+off[], i/p+off[]
7. call <strcpy>
8. buf+off[i] = global+off[i]
5. receive esp value
6. mov (esp+off), %eax
7. or %eax, %ebp
8. cmp %eax, $01h
9. jnz L1
10. raise exception
11. jmp %eax

(Example Main thread)

1. Assume choice in eax
2. bounds checking
   1. cmp eax, 03h
3. jge <exit>
4. spill code
   1. push eax
   2. call <get>
      1. call sys_read
      2. global[] = i/p[]
6. call <strcpy>
7. buf[i] = global[i]
8. spill code
   1. pop eax
   2. add %eax, %ebp
3. jmp %eax

(Example Helper thread)

Bug Location in ncompress

- Flow instruction addresses instead of just taint bits.
- 0: untainted
- Non-zero instruction address: tainted
  - Identity of instruction that wrote to it most recently.

DIFT for Bug Location

<table>
<thead>
<tr>
<th>Original</th>
<th>Tracking Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Mem</td>
<td></td>
</tr>
<tr>
<td>%Mem</td>
<td>%Reg</td>
</tr>
</tbody>
</table>

Figure 10: Bug location for ncompress program

SWQ Queue Implementation

We use the SWQ implementation in [Wang et al. CGO 2007]

```c
enqueue(data) {
  buf[head_DB] = data;
  tail_DB = (tail_DB + 1) % QUEUE_SIZE;
  while(head_DB != tail_DB) {
    while(tail_DB == head_DB) {
      data = buf[head_DB];
      head_DB = (head_DB + 1) % QUEUE_SIZE;
      return data;
    }
  }
}
depqueue(
  if (TagReg != 0) {
    If TagReg = 0 | TagMem = 0 | Inst Mem
    Jmp R3
  }
```

We use the SWQ implementation in [Wang et al. CGO 2007]