Efficient Dynamic Tainting using Multiple Cores

Yan Huang
University of Virginia
Dec. 5 2007

Dynamic Tainting (DT)
- Keep track of the source for each byte used in the program
- Shadow Memory
- Taint Seed
- Taint Propagation
- Taint Assert

Illustration – Buffer Overflow

Way too slow!
Better be kept from online usage.

- Dynamic Tainting is also applied to:
  - Malware detection
  - Ensuring privacy policies
  - Software testing

So what’s the problem?

Way too slow!
Better be kept from online usage.

- Traditional dynamic tainting systems incurs about 20x ~ 50+x overhead than direct execution.

Why is it the case?
Imagine how we need to instrument this single instruction

```
add %eax, 4(%ebp)
```

Tasks | Costs
--- | ---
Spill a few registers (may include FLAG registers) for taint computation | 2-4
Map `%eax` to its shadow memory location | 1
Map memory (%ebp) to its shadow memory location | 2
Map FLAG registers to its shadow memory (optional) | 1-2
Load the taint status of the two operands | 2
Compute and store the new taint status in the shadow memory | 1-3
Restore the spilled registers (may include status registers) | 2-4

```
add %eax, 4(%ebp)
```

Tatal | 12-19

Our Treatment – Multiple Cores

• Some essential facts
  – the tainting computation and the original computation are highly parallelizable.
  – taint shepparding itself can also be simpler if it is kept separate from the original computation.

The Basic Model

- We have 2 separate processes/threads (main and shadow)
- Main only takes care of original computation
- Shadow only deals with tainting
- They keep similar memory layout
- They communicate via one (or two) dedicated queues

The Basic Model – Quick Recap
**Implementation**

Program Compiling and Execution Diagram

- **source code**
  - compiler front end
  - assembly code
  - static
  - compiler back end
  - binary code
  - dynamic
  - loader
  - process in execution

**Source to Source Static Rewriter (SSSR)**

- **original source code**
- **SSSR**
- **main proc bin code**
- **shadow proc bin code**
- **processes in execution**

**Advantages**
- High level program objects information available;
- Less dependent on ISA;
- No penalty for run-time code generation;
- Easier to debug;

**Disadvantages**
- Requiring the application's source code;
- Hard to deal with low-level (hardware related) control performance dependent on the underlying compiler

**Source to Binary Compiler (SBC)**

- **original source code**
- **SBC**
- **main proc bin code**
- **shadow proc bin code**
- **loader**
- **processes in execution**

**Advantages**
- High level program information available;
- Full control over the binary generation
- Easy to do low level optimizations;
- Able to follow into statically linked libraries.

**Disadvantages**
- Requiring the application's source code;
- ISA dependent implementation;
- Unable to follow through dynamically linked libraries;
- Special care needed to protect the shadow memory;

**Binary to Binary Static Rewriter (BBSR)**

- **original binary code**
- **BBSR**
- **main proc bin code**
- **shadow proc bin code**
- **loader**
- **processes in execution**

**Advantages**
- The rewriting doesn't incur run-time overhead;
- Doesn't require the application's source code;
- Easy to do low level optimizations;
- Able to follow into statically linked libraries;

**Disadvantages**
- Lacking high level program information for optimization;
- Binary static analysis is hard and even infeasible;
- ISA dependent implementation;
- Unable to follow through dynamically linked libraries;
- Special care needed to protect the shadow memory;

**Binary to Binary Dynamic Rewriter**

- **original binary code**
- **loader**
- **BBDR**
- **main proc bin code**
- **shadow proc bin code**
- **process address space**

**Advantages**
- Doesn't require the source code;
- Easy shadow memory protection;
- Able to follow through dynamically linked libraries;
- Dynamic information available for optimization;
- System-wide if BBDR is running underlying the OS;

**Disadvantages**
- Run-time overhead introduced by the dynamic transformer;
- Lacking high level program information to do optimization;
Quick recap

<table>
<thead>
<tr>
<th>Optimization Opportunities</th>
<th>Static Library Tracing</th>
<th>Dynamic Library Tracing</th>
<th>ISA Independence</th>
<th>Shadow Memory Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source to source</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Source to binary</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Static binary rewriter</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Source to source rewriter</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>

Intuitive

-hard

-Source to binary compiler
  - phoenix
  - gcc

-Dynamic binary rewriter
  - Strata
  - Pin

-An assembly to assembly translator could be reused in both approaches

Implementation

Optimizations

- Reducing the number of synchronization points
  - ignore ‘never-tainted’ memory locations
  - ignore checking ‘never-tainted’ return addresses
- Reducing the chance of spinning wait
  - large queue buffers
  - do taint checking only in the shadow process
  - allow the main process to go over less critical points
- Efficient data communication
  - put the queue in L2 cache

Evaluation

- Functional evaluation
  - Does it really work correctly?

- Performance evaluation
  - Is it efficient enough for online deployment?
  - Benchmarks
  - Real programs

Questions