Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks

Koji Inoue
Department of Informatics,
Kyushu University
Japan Science and Technology Agency
Background (1/2)

Security

Trusted Program

Malicious Program

Branch Prediction

Selective Activation

SuperScalar

Signal Gating

Pipelining

Clock Gating

Value Prediction

Resizing

On-chip Cache

Drowsy Operation

MLP

TLP

ILP

OOO Exe.

DVS

High Performance

Low Power/Energy
Background (2/2)

Don't you have such an experience?

Did I Lock the Door?

Did I Lock The Door?

Don’t Care!

Very Tired

Let's Work!

You locked me! Go to your office!
The Goal of This Research

Architectural Support for

Security

SCache

Branch Prediction
SuperScalar
ILP
OOO Exe.
On-chip Cache

Pipelining
Value Prediction
MLP

High Performance

Selecteive Activation
Signal Gating
Clock Gating

Low Power/Energy

Resizing
Drowsy Operation

DVS

Clock Gating

Resizing
Drowsy Operation

DVS

Resizing
Drowsy Operation

DVS

Resizing
Drowsy Operation

DVS
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
- Evaluation
  - Experimental Set-Up
  - Security and Energy Consumption
  - Tradeoff
  - Performance Overhead
- Related Work
- Conclusions
Buffer-Overflow Attack

Buffer Overflow

- Well-Known vulnerability
- Exploited by Blaster@2003
- Caused by unexpected operations
  - writing an inordinately large amount of data into a buffer
  - This vulnerability exists in the C standard library (e.g. strcpy)
- Lead to a stack smashing
  - An attack code is inserted
  - The return address is corrupted
- Used to highjack the program execution control

CERT Advisories relating to buffer-overflow (%)


Function Call/Return

Program code

```c
int f ( ) {
    ...
    g (s1);
    ...
}

int g ( char *s1) {
    char buf [10];
    ...
    strcpy(buf, s1);
    ...
}
```

1. Start f( )
2. Call g( )
3. Execute strcpy( )
4. Return to f( )
Function Call/Return

```
int f ( ) {
    ...
    g (s1);
    ...
}

int g ( char *s1) {
    char buf [10];
    ...
    strncpy(buf, s1);
    ...
}
```

1. Start f( )
2. Call g( )
3. Execute strcpy( )
4. Return to f( )
Function Call/Return

Program code

```c
int f() {
    ... 
    g(s1);
    ...
}

int g(char *s1) {
    char buf[10];
    ...
    strcpy(buf, s1);
    ...
}
```

1. Start f()
2. Call g()
3. Execute strcpy()
4. Return to f()
Function Call/Return

Program code

```c
int f() {
    ...
    g(s1);
    ...
}

int g(char *s1) {
    char buf[10];
    ...
    strcpy(buf, s1);
    ...
}
```

1. Start f()
2. Call g()
3. Execute strcpy()
4. Return to f()
Function Call/Return

Program code

```c
int f ( ) {
    ...
    g(s1);
    ...
}

int g ( char *s1) {
    char buf [10];
    ...
    strcpy(buf, s1);
    ...
}
```

1. Start f( )
2. Call g( )
3. Execute strcpy( )
4. Return to f( )
Stack Smashing

Program code:

```c
int f() {
    ...
    g(s1);
    ...
}
int g(char *s1) {
    char buf[10];
    ...
    strcpy(buf, s1);
    ...
}
```

1. Start `f()`
2. Call `g()`
3. Execute `strcpy()`
4. Return to `f()`

Diagram:

- `s1` is pushed onto the stack.
- `strcpy()` copies the string from `s1` to `buf`.
- The Next PC of Call `g()` is saved.
- `buf` is a local variable.
- The string is allocated on the stack.

Higher Addr. Stack Growth

Lower Addr. Stack Growth
Stack Smashing

Program code

```c
int f() {
    ... 
g(s1);
    ...
}

int g(char*s1) {
    char buf[10];
    ...
    strcpy(buf, s1);
    ...
}
```

1. Start f()
2. Call g()
3. Execute strcpy()
4. Return to f()

```
\[\text{Attack Code}\]
\[\text{String}\]
```

Insert the attack code!
Corrupt the return address!
Stack Smashing

Program code

1. Start f()
2. Call g()
3. Execute strcpy()
4. Return to f()

- Insert the attack code!
- Corrupt the return address!
- Hijack the program execution!
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
- Evaluation
  - Experimental Set-Up
  - Security and Energy Consumption
  - Tradeoff
  - Performance Overhead
- Related Work
- Conclusions
Concept

- **Problem**
  - The return address (RA) in the memory stack can be corrupted

- **Solution**
  - RA is stored via On-Chip Caches!
  - Protect RA in the cache!

- **Implementation**
  - Generate one or more “Replicas” on each RA store
  - Compare the original with a replica on the corresponding RA load
  - If they are not the same, we know that the popped RA has been corrupted!
Organization

- Replica Flag (1b)
- Data (Ret. Addr.)
- MUX for replica lines
- Hit Condition
- Comparator (32b)
- Safe?
- Ref. Addr.
- Way 0
- Way 1
Operation: Return-Address Store

Store

- Store the Return Addr. into ML
- Store the Return Addr. into existing RLs (if it has the same tag)
- Generate RLs until \#RL == Nrep

Load

- Cache Access (for Return Addr.)
- RL Hit?
  - yes
    - Return Addr. Check
    - match
      - Safe Complete
    - un-match
      - Error!
  - no

Original

- Data (Ret. Addr.)

Replica

- Data
- Safe?

ML: Master Line
RL: Replica Line

# of Replica lines (Nrep) = 2
Operation: Return-Address Load

# of Replica lines (Nrep) = 2

Store
- Store the Return Addr. into ML
- Store the Return Addr. into existing RLs (if it has the same tag)
- Generate RLs until #RL == Nrep

Load
- Cache Access (for Return Addr.)
- RL Hit?
  - yes: Return Addr. Check
    - match: Safe Complete
    - un-match: Error!
  - no: Read lines from all ways (provide the data to CPU)

---

Original

Data

Replica

Data (Ret. Addr.)

Safe?
Summary of SCache

Pros

- Run-time detection of return-address corruption
  - If at least a replica line exists
  - Does not affect processor complexity
  - Small impact on cache area and access time

- Controllable # of replica lines
  - Tradeoff between energy and security

Cons

- Incomplete protection
  - Replica lines may be evicted

- Degraded cache-hit rates
  - Increase in the average memory access time
  - Increase in the memory access energy

- Increased cache energy
  - Generating replica lines
  - Reading replica lines
    (compared to a low-power cache)
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
- Evaluation
  - Experimental Set-Up
  - Security and Energy Consumption
  - Tradeoff
  - Performance Overhead
- Related Work
- Conclusions
Experimental Set-Up

Security/Energy/Performance

- SimpleScalar3.0
  - 16KB 4-way D-cache
  - Line size :32B
  - OOO execution
- SPEC2000
  - 7 integer programs
  - 4 fp programs
  - Small input

Energy

- 4KB SRAM design
  - 0.18 µm CMOS technology
  - One way of the 16KB cache
- Hspice simulation
  - w/ extracted load capacitances
  - Measure the energy consumed for 1-bit accesses
## SCache Models

### Evaluated SCaches

<table>
<thead>
<tr>
<th>name</th>
<th>Replica Lines</th>
<th>Placement</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU1L</td>
<td>LRU</td>
<td>Locked</td>
<td>1</td>
</tr>
<tr>
<td>LRU1</td>
<td>LRU</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LRU2</td>
<td>LRU</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>MRU1</td>
<td>MRU</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>MRU2</td>
<td>MRU</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>ALL</td>
<td>-----</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>CONV</td>
<td>MRU way prediction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Security

- **Vulnerability**
  \[ \text{Vulnerability} = \left( \frac{Nv \text{- rald}}{Nrald} \right) \times 100 \]
  - Insecure issued RA load
  - Total # of issued RA load

### Energy Consumption

- **ETotal**
  \[ E_{\text{total}} = E_{\text{read}} + E_{\text{write}} + E_{\text{writeback}} + E_{\text{replacement}} \]

- Read
- Write
- Writeback to place replica lines
- Replacement (on misses)

*) Only load/store operations issued to the cache are considered
Results (Vulnerability)

ALL: more than 99.3% of RA load

MRU1: more than 88.5% of RA load
Results (Energy Consumption)

ALL: 23% of energy overhead

MRU1: 9.9% of energy overhead
Results (Energy Breakdown)
Results (EVP, E\textsuperscript{2}VP, EV\textsuperscript{2}P)

Normalized to LRU1

- **EVP**
  - 164.gzip
  - 176.gcc
  - 188.ammp
  - 175.vpr
  - 197.parser

- **E\textsuperscript{2}VP**
  - 164.gzip
  - 176.gcc
  - 188.ammp
  - 175.vpr
  - 197.parser

- **EV\textsuperscript{2}P**
  - 164.gzip
  - 176.gcc
  - 188.ammp

**MRU1**: good for Energy-Oriented Applications

**MRU2**: good for Security-Oriented Applications
Results (Performance)

MRU1: 0.1% of overhead

ALL: 1.1% of overhead
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
- Evaluation
  - Experimental Set-Up
  - Security and Energy Consumption
  - Tradeoff
  - Performance Overhead
- Related Work
- Conclusions
Related Work

- **Static**:
  - SASI [WNSP99]
  - StakcGuard [USENIX98]
  - Source Code Analysis
  - Re-Compilation

- **Dynamic**:
  - SW: LibSafe/Verify [USENIX00]
    - Library Update
    - Performance Overhead
  - SW: StackGhost [USENIX01]
    - Only for SPARC architecture
  - HW: SRAS [SPC03]
    - Inside of the processor core
    - HW support for LI FO operations

---

**SCache: Dynamic + HW**

**Cache-Level Protection**
- De-coupled implementation
- Random Access
- Large Capacity
- Reduced Cost Overhead

**Energy-Security Tradeoff**
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
- Evaluation
  - Experimental Set-Up
  - Security and Energy Consumption
  - Tradeoff
  - Performance Overhead
- Related Work
- Conclusions
Conclusions

**Summary**
- Architectural support for run-time buffer-overflow detection
- Evaluation of Energy and Security
  - Security-Oriented 'ALL (or MRU2) model
    - More than 99.3% of RA load can be protected (9/11 programs)
    - 23% of energy overhead
  - Energy-Oriented 'MRU1 model
    - More than 98.5% of RA load can be protected (9/11 programs)
    - 9.9% of energy overhead

**Future Work**
- Evaluate with vulnerable benchmarks
- Consider a good measurement for security
- Complete design of the SCache
- Develop an optimization technique to adapt to user requirements for security and energy consumption
Back-Up Slides ...
## Cache Miss Rates

**IRA: Issued Return Address**  
**CA: Cache Access**

<table>
<thead>
<tr>
<th>Model Bench</th>
<th>#IRA Load (Nrald)</th>
<th>CONV</th>
<th>LRU1-L</th>
<th>LRU1</th>
<th>LRU2</th>
<th>MRU1</th>
<th>MRU2</th>
<th>ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>4,930,467</td>
<td>5.22%</td>
<td>5.23%</td>
<td>5.22%</td>
<td>5.22%</td>
<td>5.22%</td>
<td>5.23%</td>
<td>5.25%</td>
</tr>
<tr>
<td>175.vpr</td>
<td>5,627,709</td>
<td>3.53%</td>
<td>3.59%</td>
<td>3.56%</td>
<td>3.63%</td>
<td>3.59%</td>
<td>3.66%</td>
<td>3.74%</td>
</tr>
<tr>
<td>176.gcc</td>
<td>37,519,156</td>
<td>4.26%</td>
<td>6.06%</td>
<td>4.29%</td>
<td>4.37%</td>
<td>4.33%</td>
<td>4.43%</td>
<td>4.64%</td>
</tr>
<tr>
<td>181.mcf</td>
<td>992,419</td>
<td>20.02%</td>
<td>20.05%</td>
<td>20.02%</td>
<td>20.03%</td>
<td>20.05%</td>
<td>20.06%</td>
<td>20.10%</td>
</tr>
<tr>
<td>197.parser</td>
<td>45,466,527</td>
<td>4.13%</td>
<td>4.25%</td>
<td>4.18%</td>
<td>4.44%</td>
<td>4.23%</td>
<td>4.55%</td>
<td>5.07%</td>
</tr>
<tr>
<td>255.vortex</td>
<td>22,101,265</td>
<td>1.75%</td>
<td>1.83%</td>
<td>1.79%</td>
<td>1.91%</td>
<td>1.82%</td>
<td>1.94%</td>
<td>2.32%</td>
</tr>
<tr>
<td>256.bzip</td>
<td>18,147,017</td>
<td>2.31%</td>
<td>2.31%</td>
<td>2.31%</td>
<td>2.32%</td>
<td>2.31%</td>
<td>2.32%</td>
<td>2.45%</td>
</tr>
<tr>
<td>177.mesa</td>
<td>4,727,396</td>
<td>0.14%</td>
<td>0.15%</td>
<td>0.15%</td>
<td>0.16%</td>
<td>0.15%</td>
<td>0.16%</td>
<td>1.08%</td>
</tr>
<tr>
<td>179.art</td>
<td>32,466</td>
<td>42.93%</td>
<td>42.93%</td>
<td>42.93%</td>
<td>42.93%</td>
<td>42.93%</td>
<td>42.93%</td>
<td>42.93%</td>
</tr>
<tr>
<td>183.equake</td>
<td>3,580,827</td>
<td>2.44%</td>
<td>2.45%</td>
<td>2.44%</td>
<td>2.46%</td>
<td>2.45%</td>
<td>2.47%</td>
<td>2.52%</td>
</tr>
<tr>
<td>188.ammp</td>
<td>6,307,839</td>
<td>36.27%</td>
<td>36.29%</td>
<td>36.28%</td>
<td>36.31%</td>
<td>36.28%</td>
<td>36.30%</td>
<td>36.38%</td>
</tr>
</tbody>
</table>
WP Cache v.s SCache

WP

1cycle
Correct prediction

2cycle
Incorrect prediction

SCache + WP

1cycle
Return-Address Load