Lock and Unlock: A Data Management Algorithm for A Security-Aware Cache

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

Architectural Support for

Security

SCache

(improved data management)
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
  - Overview
  - Security Issue
- Lock&Unlock Algorithm
- Evaluation
  - Security Strength
  - Performance Overhead
- 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
- Highjack the program execution control

CERT Advisories relating to buffer-overflow (%)


0 10 20 30 40 50 60

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

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

Program code

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

```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];
    ...
    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()
Stack Smashing

Program code

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( )

Insert the attack code!
Corrupt the return address!
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()

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

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
  - Overview
  - Security Issue
- Lock&Unlock Algorithm
- Evaluation
  - Experimental Set-Up
  - Security Strength
  - Performance Overhead
- Conclusions
Secure Cache Architecture

- Protect return-address (RA) values in the cache!
  - Generate one or more “Replicas” on each RA store
  - Compare the original RA with a replica on the RA load
  - If they are not the same, we know that the popped RA has been corrupted!

**Store (push)**

**Load (pop)**
Security Issue

- The replica lines are also evicted from the cache
- Miss the opportunity to check the RA value if no replica lines reside in the cache
- So...
  - Good for many applications (w/ high cache-hit rates)
  - Bad for memory intensive applications (w/ high cache-miss rates)
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
  - Overview
  - Security Issue
- Lock&Unlock Algorithm
- Evaluation
  - Security Strength
  - Performance Overhead
- Conclusions
Lock and Invalidate Approach

- Prohibit the eviction of replica lines until they are loaded later! (Lock)
- Invalidate the loaded replicas to release the cache resource (Invalidate)

Pros.
- Effective use of cache resource

Cons.
- Squashed return-address loads prematurely invalidate the replica lines
Lock & Unlock Approach

- Prohibit the eviction of replica lines until they are loaded later! (Lock)
- Keep the replicas in the cache until they are evicted due to the replacement policy! (Unlock)

Pros.
- Avoid premature replica-line invalidation

Cons.
- Waste cache resource
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
  - Overview
  - Security Issue
- Lock&Unlock Algorithm
- Evaluation
  - Security Strength
  - Performance Overhead
- Conclusions
Experimental Set-Up

- Processor Simulator
  - SimpleScalar3.0
    - 4-way OOO Superscalar
    - 4-way 16KB L1 D-Cache

- SCache Model
  - LRU1-L&I: Lock & Invalidate (w/ LRU replica placement)
  - LRU1-L&U: Lock & Unlock (w/ LRU replica placement)
  - MRU1-L&U: Lock & Unlock (w/ MRU replica placement)

- Benchmark Programs
  - SPEC2000
    - 7 integer programs, 4 fp programs
    - Small input
Vulnerability

\[ \frac{(N_v - rald)}{N_{rald}} \times 100 \]

Insecure issued RA load

Total #of issued RA load

Benchmark Programs

LRU1-L&I

LRU1-L&U

MRU1-L&U
Performance Overhead

Benchmark Programs

Performance Overhead

- LRU1-L&I
- LRU1-L&U
- MRU1-L&U
Outline

- Introduction
- Buffer-Overflow Attack
- Secure Cache Architecture
  - Overview
  - Security Issue
- Lock&Unlock Algorithm
- Evaluation
  - Security Strength
  - Performance Overhead
- Conclusions
Conclusions

**Summary**
- Architectural support for run-time buffer-overflow detection
- New data management algorithms
  - Lock and Invalidation
    - Inefficient for some benchmark programs
    - Max. : 23% vulnerability
    - Performance overhead is less than 0.9%
  - Lock and Unlock
    - Very efficient for all benchmark programs
    - Max. : 2.5% vulnerability (average < 0.8%)
    - Performance overhead is less than 0.3%

**Future Work**
- Integrate to a secure microprocessor platform