Watchdog:
Hardware for Safe and Secure Manual Memory Management and Full Memory Safety

Santosh Nagarakatte, Milo M. K. Martin, Steve Zdancewic
University of Pennsylvania
This work licensed under the Creative Commons Attribution-Share Alike 3.0 United States License

• You are free:
  • to Share — to copy, distribute, display, and perform the work
  • to Remix — to make derivative works

• Under the following conditions:
  • Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  • Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

• For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to:

  http://creativecommons.org/licenses/by-sa/3.0/us/

• Any of the above conditions can be waived if you get permission from the copyright holder.

• Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.
Project goal:
Make C/C++ as safe and secure as Java

Why?
Lack of *memory safety* is the root cause of serious *bugs* and *security vulnerabilities*
Lack of Memory Safety is the Root Cause of Many Security Vulnerabilities

Firefox – use-after-free vulnerability
CVE-2012-1940 - Severity: 9.3 (High)       June 5, 2012

Google Chrome – use-after-free vulnerability
CVE-2012-1940 - Severity: 10.0 (High)       May 24, 2012

Apple Quicktime – use-after-free vulnerability
CVE-2012-0661 - Severity: 6.8 (Medium)       May 11, 2012
“allows remote attackers to execute arbitrary code... via a crafted movie file with JPEG2000 encoding”

DHS/NIST National Vulnerability Database:
• Last three months: 49 use-after-free disclosures
• Last three years: 292

Buffer overflows still a problem, too
Project Overview & Progression

Memory safety has two components:
Bounds safety       Use-after-free safety
Memory safety has two components:

**Bounds safety**

**HardBound**
- [ASPLOS 2008]
- Pointer-based
- Disjoint metadata
- ~10% overhead

**SoftBound**
- [PLDI 2009]
- Pointer-based
- Disjoint metadata
- ~75% overhead

**Use-after-free safety**
Project Overview & Progression

Memory safety has two components:

Bounds safety

**HardBound**
- [ASPLOS 2008]
- Pointer-based
- Disjoint metadata
- ~10% overhead

**SoftBound**
- [PLDI 2009]
- Pointer-based
- Disjoint metadata
- ~75% overhead

Use-after-free safety

**Watchdog**
- [this work]
- Pointer-based, disjoint
- Unique identifier check
- ~15% overhead

**CETS**
- [ISMM 2010]
- Pointer-based, disjoint
- Unique identifier check
- ~50% overhead
Memory safety has two components: Bounds safety and Use-after-free safety.

**Watchdog**
- [also this work]
- Pointer-based, disjoint
- Unique identifier check
- ~25% overhead

**SoftBound**
- [PLDI 2009]
- Pointer-based
- Disjoint metadata
- ~75% overhead

**CETS**
- [ISMM 2010]
- Pointer-based, disjoint
- **Unique identifier check**
- ~50% overhead
What exactly is a use-after-free error? & How can they be detected?
Use-After-Free Errors

- Use-after-free error
  - Use of pointer to a deallocated object
  - a.k.a. “dangling pointer”
  - Consequence of C/C++’s manual memory management

- Example:

  Initial state:

  Step 1: free(A);

  Step 2: C = malloc(…);

  Step 3: B[0] = …;

- Can occur to objects on heap or stack
Location-Based Checking: Incomplete
[Valgrind, J&K, LBA, SAFECODE, SafeProc, MemTracker, ...]

• Track valid/invalid state of memory
  • On deallocation, mark the memory as invalid
  • On reallocation, mark memory as valid again!
  • Does not detect all still-dangling pointer accesses

• Delayed/random reuse can help mask some errors...

```
Step 1: free(A);
Step 2: C = malloc(...);
Step 3: B[0] = ...;
```

Initial state: Invalid

A

B

Step 1: Invalid

Valid

Invalid

Step 2: Invalid

Step 3: Invalid

Valid
Can we get comprehensive detection?
Identifier-Based Checking: Comprehensive
[SafeC, P&F, MSCC, Chuang et al, CETS]

• Allocate **unique identifier (UID)** for each allocation
  • Record the set of valid identifiers
  • Track this UID with each pointer
  • Invalidate identifiers on memory deallocation
  • Check for identifier validity on memory accesses

**Metadata**

**Initial state**

Valid IDs: #1

(A, #1)
(B, #1)

Step 1: free(A);
Valid IDs: #1

Step 2: C = malloc(…);
Valid IDs: #1, #2
(C, #2)
(B, #1)

Step 3: B[0] = …;
Valid IDs: #1, #2
(C, #2)
(B, #1)

Milo Martin – Watchdog – ISCA 2012
Where are the per-pointer metadata stored?

How is the validity check performed efficiently?
Shadow Memory for Per-Pointer Metadata

- A *shadow memory* for per-pointer metadata
  - Leaves memory layout unchanged
  - Protects metadata from corruption
Shadow Memory for Per-Pointer Metadata

- A *shadow memory* for per-pointer metadata
  - Leaves memory layout unchanged
  - Protects metadata from corruption
- Mapped into virt. address space
  - Simple mapping
  - Allocated on demand by system
Shadow Memory for Per-Pointer Metadata

- A **shadow memory** for per-pointer metadata
  - Leaves memory layout unchanged
  - Protects metadata from corruption
- Mapped into virt. address space
  - Simple mapping
  - Allocated on demand by system
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)

- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

• Split unique ID (UID) into:
  • Location of “lock” (reusable)
  • “key” (unique)

• Allocation:
  • Assign a key & lock location
  • Set memory[lock] = key
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)

- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

- Invariant:
  - valid if memory[lock] == key
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)

- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

- Invariant:
  - valid if memory[lock] == key

- Propagate as pointer copied
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)

- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

- Invariant:
  - valid if memory[lock] == key

- Propagate as pointer copied

- Deallocation
  - memory[lock] = invalid
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key
- Invariant:
  - valid if memory[lock] == key
- Propagate as pointer copied
- Deallocation
  - memory[lock] = invalid
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

- Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)

- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

- Invariant:
  - valid if memory[lock] == key

- Propagate as pointer copied

- Deallocation
  - memory[lock] = invalid

- Check:
  - Does memory[lock] == key?
Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]

• Split unique ID (UID) into:
  - Location of “lock” (reusable)
  - “key” (unique)

• Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

• Invariant:
  - valid if memory[lock] == key

• Propagate as pointer copied

• Deallocation
  - memory[lock] = invalid

• Check:
  - Does memory[lock] == key?
  - **Just a “Load and compare”**
So, what is “Watchdog”?

Hardware enforced use-after-free checking...

...using:

• Shadow memory for metadata
• Lock-and-key checking
Watchdog Hardware Implementation

• **Goal:** reduce intrusiveness of hardware changes
  - Avoid massive modifications to cores
  - Memory system beyond cores unchanged

• **Approach:** insert additional operations as micro-ops
  - At decode, insert new custom micro-ops
    - “check”, “load metadata”, “store metadata”
  - Extend registers with metadata

• **Software responsible for UID manipulations**
  - Augmented allocation library
Identifying Loads/Stores of Pointers

• When does Watchdog access the shadow space?
  • When loading/storing a pointer value from/to memory

• How does the hardware identify such memory ops?
  • Option #1: conservatively
    • Any pointer-sized (64-bit) memory operation
    • Con: unnecessary shadow accesses in some cases
    • Pro: requires almost no software support
  • Option #2: compiler-identified
    • Compiler marks instructions that load/store pointers
    • Pro: more precise (less overhead)
    • Con: requires more help from compiler, new opcodes

Milo Martin – Watchdog – ISCA 2012
Micro-Op Insertion & Rename Example

Original Code

- `ld.word Memory[r1] → r2`
- `add r2, 8 → r3`
- `ld.byte Memory[r3] → r4`

Insert Micro-Ops

- `check r1.meta`
- `ld.word Memory[r1] → r2`
- `ld.total Shadow[r1] → r2.meta`
- `add r2, 8 → r3`
- `copy r2.meta → r3.meta`
- `check r3.meta`
- `ld.byte Memory[r3] → r4`
- `ld.total Shadow[r3] → r4.meta`

Example of unneeded shadow access (e.g., byte load is not a pointer)
Micro-Op Insertion & Rename Example

Original Code

```
ld.word Memory[r1] → r2
add r2, 8 → r3
ld.byte Memory[r3] → r4
```

Insert Micro-Ops

```
check r1.meta
ld.word Memory[r1] → r2
ld Shadow[r1] → r2.meta
add r2, 8 → r3
copy r2.meta → r3.meta
check r3.meta
ld.byte Memory[r3] → r4
ld Shadow[r3] → r4.meta
```

Rename

Decoupled metadata:
Two mappings for each register

<table>
<thead>
<tr>
<th>Reg</th>
<th>Value</th>
<th>Meta</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Micro-Op Insertion & Rename Example

Original Code

- `ld.word Memory[r1] → r2`
- `add r2, 8 → r3`
- `ld.byte Memory[r3] → r4`

Insert Micro-Ops

- `check r1.meta`
- `ld.word Memory[r1] → r2`
- `ld Shadow[r1] → r2.meta`
- `add r2, 8 → r3`
- `copy r2.meta → r3.meta`
- `check r3.meta`
- `ld.byte Memory[r3] → r4`
- `ld Shadow[r3] → r4.meta`

Rename

Decoupled metadata:
Two mappings for each register

<table>
<thead>
<tr>
<th>Reg</th>
<th>Value</th>
<th>Meta</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>p1</td>
<td>p10</td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Micro-Op Insertion & Rename Example

Original Code

```
ld.word Memory[r1] → r2
add r2, 8 → r3
ld.byte Memory[r3] → r4
```

Insert Micro-Ops

```
check r1.meta
ld.word Memory[r1] → r2
ld Shadow[r1] → r2.meta
add r2, 8 → r3
copy r2.meta → r3.meta
check r3.meta
ld.byte Memory[r3] → r4
ld Shadow[r3] → r4.meta
```

Rename

```
check p10
ld.word Memory[p1] → p2
ld Shadow[p1] → p11
```

Decoupled metadata:
Two mappings for each register

<table>
<thead>
<tr>
<th>Reg</th>
<th>Value</th>
<th>Meta</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>p1</td>
<td>p10</td>
</tr>
<tr>
<td>r2</td>
<td>p2</td>
<td>p11</td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Micro-Op Insertion & Rename Example

Original Code

\[ \text{ld.word Memory}[r1] \rightarrow r2 \]
\[ \text{add } r2, 8 \rightarrow r3 \]
\[ \text{ld.byte Memory}[r3] \rightarrow r4 \]

Insert Micro-Ops

\[ \text{check } r1.\text{meta} \]
\[ \text{ld.word Memory}[r1] \rightarrow r2 \]
\[ \text{ld Shadow}[r1] \rightarrow r2.\text{meta} \]
\[ \text{copy } r2.\text{meta} \rightarrow r3.\text{meta} \]
\[ \text{check } r3.\text{meta} \]
\[ \text{ld.byte Memory}[r3] \rightarrow r4 \]
\[ \text{ld Shadow}[r3] \rightarrow r4.\text{meta} \]

Rename

\[ \text{check } p10 \]
\[ \text{ld.word Memory}[p1] \rightarrow p2 \]
\[ \text{ld Shadow}[p1] \rightarrow p11 \]
\[ \text{add } p2, 8 \rightarrow p3 \]

Copy elimination via register remapping

[Jourdan et al, Petric et al]
Micro-Op Insertion & Rename Example

Original Code

```
ld.word Memory[r1] → r2
add r2, 8 → r3
ld.byte Memory[r3] → r4
```

Insert Micro-Ops

```
check r1.meta
ld.word Memory[r1] → r2
ld Shadow[r1] → r2.meta
add r2, 8 → r3
check r3.meta
ld.byte Memory[r3] → r4
ld Shadow[r3] → r4.meta
```

Rename

```
check p10
ld.word Memory[p1] → p2
Id Shadow[p1] → p11
add p2, 8 → p3
```

Copy elimination via register remapping

[Jourdan et al, Petric et al]

Reg | Value | Meta
---|------|-----
| r1 | p1  | p10 |
| r2 | p2  | p11 |
| r3 | p3  | p11 |
| r4 |     |     |
Micro-Op Insertion & Rename Example

Original Code

\[
\begin{align*}
\text{ld.word Memory[r1]} & \rightarrow r2 \\
ad r2, 8 & \rightarrow r3 \\
\text{ld.byte Memory[r3]} & \rightarrow r4
\end{align*}
\]

Insert Micro-Ops

\[
\begin{align*}
\text{check r1.meta} \\
\text{ld.word Memory[r1]} & \rightarrow r2 \\
\text{ld Shadow[r1]} & \rightarrow r2.meta \\
\text{add r2, 8} & \rightarrow r3 \\
\text{copy r2.meta} & \rightarrow r3.meta \\
\text{check r3.meta} \\
\text{ld.byte Memory[r3]} & \rightarrow r4 \\
\text{ld Shadow[r3]} & \rightarrow r4.meta
\end{align*}
\]

Rename

\[
\begin{align*}
\text{check p10} \\
\text{ld.word Memory[p1]} & \rightarrow p2 \\
\text{ld Shadow[p1]} & \rightarrow p11 \\
\text{add p2, 8} & \rightarrow p3 \\
\text{copy p11} & \rightarrow p3 \\
\text{check p11} \\
\text{ld.byte Memory[p3]} & \rightarrow p5
\end{align*}
\]

Copy elimination via register remapping

[Jourdan et al, Petric et al]
Micro-Op Insertion & Rename Example

**Original Code**

- ld.word Memory[r1] → r2
- add r2, 8 → r3
- ld.byte Memory[r3] → r4

**Insert Micro-Ops**

- check r1.meta
- ld.word Memory[r1] → r2
- ld Shadow[r1] → r2.meta
- add r2, 8 → r3
- copy r2.meta → r3.meta
- check r3.meta
- ld.byte Memory[r3] → r4
- ld Shadow[r3] → r4.meta

**Rename**

- check p10
- ld.word Memory[p1] → p2
- ld Shadow[p1] → p11
- add p2, 8 → p3
- check p11
- ld.byte Memory[p3] → p5

**Copy elimination via register remapping**

[Jourdan et al, Petric et al]

Decoupled metadata operations off critical path, becomes “extra” ILP

<table>
<thead>
<tr>
<th>Reg</th>
<th>Value</th>
<th>Meta</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>p1</td>
<td>p10</td>
</tr>
<tr>
<td>r2</td>
<td>p2</td>
<td>p11</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
<td>p11</td>
</tr>
<tr>
<td>r4</td>
<td>p5</td>
<td></td>
</tr>
</tbody>
</table>
Micro-Op Insertion & Rename Example

Original Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld.word Memory[r1]</td>
<td></td>
<td>r2</td>
</tr>
<tr>
<td>add r2, 8 → r3</td>
<td></td>
<td>r3</td>
</tr>
</tbody>
</table>

Insert Micro-Ops

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>check r1.meta</td>
<td></td>
<td>Id.word Memory[p1] → p2 meta</td>
</tr>
<tr>
<td>add r2, 8 → r3</td>
<td></td>
<td>r3</td>
</tr>
<tr>
<td>check r3.meta</td>
<td></td>
<td>Id Shadow[p1] → p11 meta</td>
</tr>
<tr>
<td>ld.byte Memory[p3]</td>
<td></td>
<td>p5</td>
</tr>
</tbody>
</table>

Rename

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>check p10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p2, 8 → p3</td>
<td></td>
<td>p3</td>
</tr>
<tr>
<td>check p11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Copy elimination via register remapping**

[Jourdan et al, Petric et al]

**Decoupled metadata operations**

off critical path, becomes “extra” ILP

#### Register-to-register copies eliminated

- Only for load/store of pointers
- ~25% of memory operations

#### Checks occur on each memory operation...

- And each requires a cache access...

#### Decoupled metadata operations

- off critical path, becomes “extra” ILP
Reducing Cost of “Check”: Lock Location Cache

• A “check” occurs before each memory operation
  • Memory[lock] == key
  • Load ports are a precious resource

• So, Watchdog adds a lock location cache for “checks”
  • Like a smaller data cache
  • Peer to I$ and D$
  • Normal misses handling
  • Dedicated TLB

Just to amplify cache access bandwidth

• Overhead: 24% → 15%
Experiments
Efficacy

• Does Watchdog actually detect use-after-free errors?
  • 291 use-after-free test cases from NIST Juliet Test Suite
  • Detected and thwarted the attack in all 291 cases

• Does Watchdog incur false violations?
  • No spurious violations in any tests or benchmarks tested
Watchdog Performance Overhead

- Timing simulations of wide-issue out-of-order x86 core
- Average performance overhead: 15%
  - Average increase in micro-ops is 44%, but off critical path

Average overhead of 15%
What About Buffer Overflows?
What About Buffer Overflows?

Memory safety has two components:
- **Bounds safety**
- **Use-after-free safety**

**Watchdog +**
- [also this work]
- Pointer-based, disjoint
- Unique identifier check
- ~25% overhead

- Implemented via micro-op injection
- Identification of pointer operations
- Efficient metadata propagation via copy elimination
- Uses lock location cache to increase check throughput
Extend Metadata for Bounds Enforcement

- Extend metadata
- Base & bounds
Extend Metadata for Bounds Enforcement

- Extend metadata
  - Base & bounds
- Check each memory op
  - Bounds check micro-op
- Watchdog optimizations apply directly
  - Pointer identification
  - Copy elimination
Watchdog + Bound Checking = Memory Safety

- Bounds checking increases overhead
  - 3% for due to cache pressure from larger metadata
  - 6% for extra “bounds check” micro-ops (just an ALU op)
- Overall, just 24% for full memory safety
See Paper For...

• Results for more conservative pointer identification
• Detect dangling pointers to stack
  • Inject micro-ops for call and return instructions
• Handling
  • Custom memory allocators
  • memcpy()
• More thorough discussion of related work
• Discussion of multithreading
Conclusion

• **Use-after-free errors**
  - Cause memory corruption bugs & security vulnerabilities
  - Prior software techniques identified checking approaches
    - Per-pointer tracking of unique identifiers
    - Disjoint shadow space & lock-and-key checking

• **Watchdog: in-hardware use-after-free checking**
  - Goal: low runtime overhead & implementable hardware
  - Micro-op injection, copy elimination, lock location cache
  - 15% average performance overhead

• **Integrates well with bounds checking, too**
  - Watchdog + bounds checking = full memory safety
  - 24% overhead