Watchdog:

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

Santosh Nagarakatte, **Milo M. K. Martin**, Steve Zdancewic
University of Pennsylvania

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

Project Overview & Progression

Memory safety has two components:

Bounds safety | Use-after-free safety

Hardware

HardBound

[ASPLOS 2008]

- Pointer-based
- Disjoint metadata
- ~10% overhead

Software

SoftBound

[PLDI 2009]

- Pointer-based
- Disjoint metadata
- ~75% overhead

Project Overview & Progression

Memory safety has two components:

	Bounds safety	Use-after-free safety
Hardware	**HardBound** [ASPLOS 2008] • Pointer-based • Disjoint metadata • ~10% overhead	**Watchdog** **[this work]** • Pointer-based, disjoint • Unique identifier check • ~15% overhead
Software	**SoftBound** [PLDI 2009] • Pointer-based • Disjoint metadata • ~75% overhead	**CETS** [ISMM 2010] • Pointer-based, disjoint • **Unique identifier check** • ~50% overhead

Project Overview & Progression

Memory safety has two components:

Bounds safety . **Use-after-free safety**

Watchdog +

[also this work]

- Pointer-based, disjoint
- Unique identifier check
- ~25% overhead

Hardware

Software

SoftBound

[PLDI 2009]

- Pointer-based
- Disjoint metadata
- ~75% overhead

CETS

[ISMM 2010]

- Pointer-based, disjoint
- **Unique identifier check**
- ~50% overhead

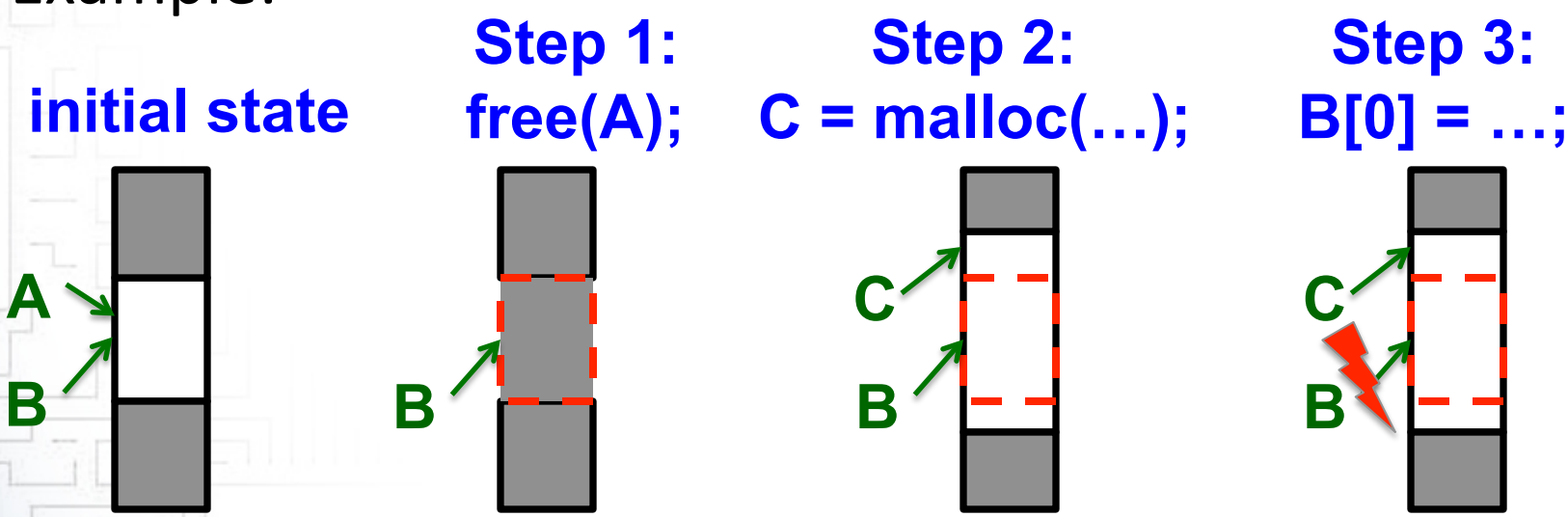
What exactly is a use-after-free error?

&

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:

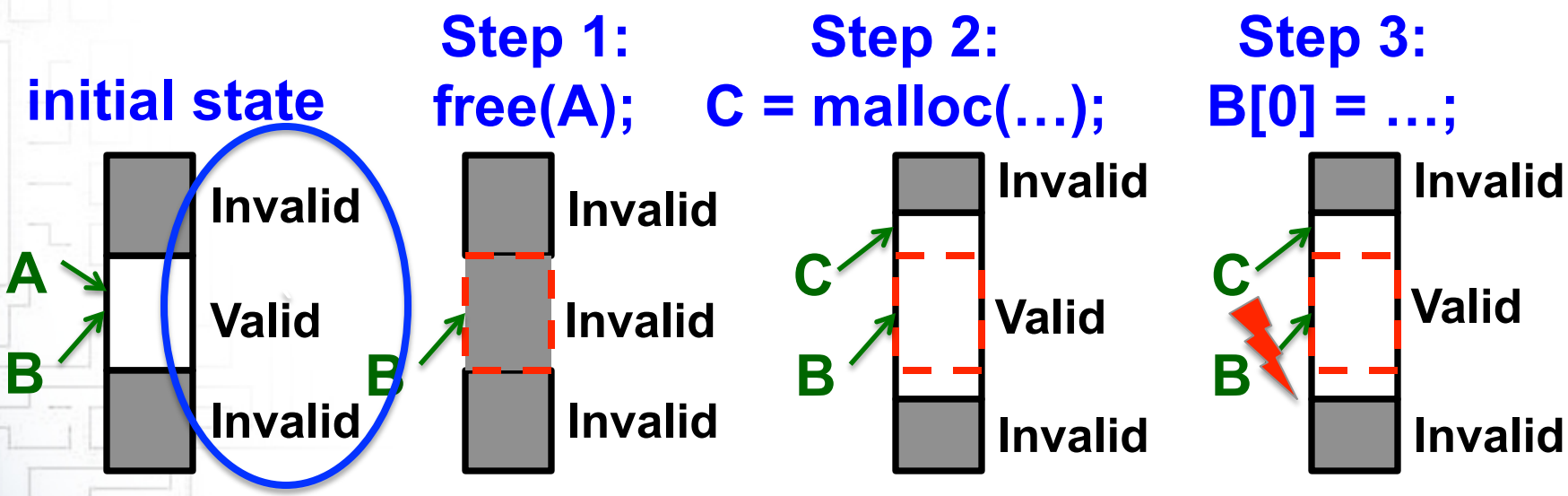


- Can occur to objects on heap or stack

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**



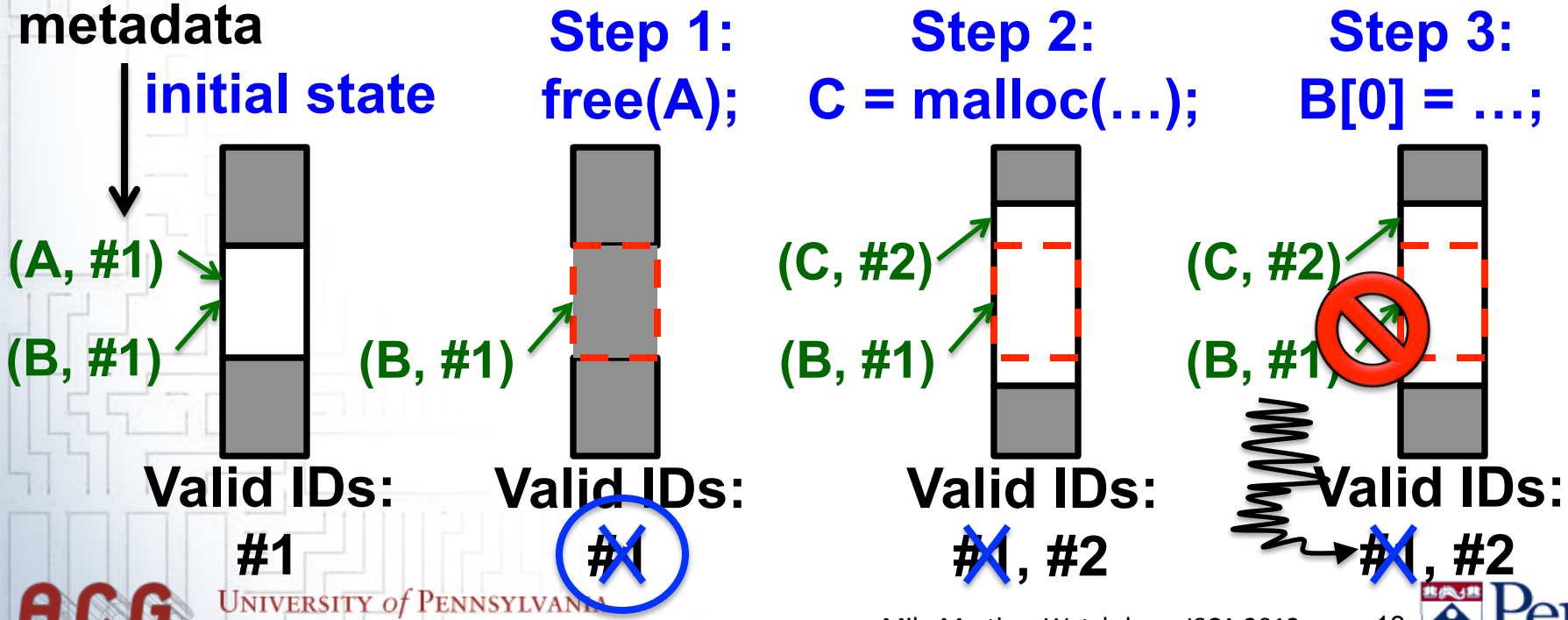
- Delayed/random reuse can help mask some errors…

Can we get comprehensive detection?

Identifier-Based Checking: Comprehensive

[SafeC, P&F, MSCC, Chuang et al, CETS]

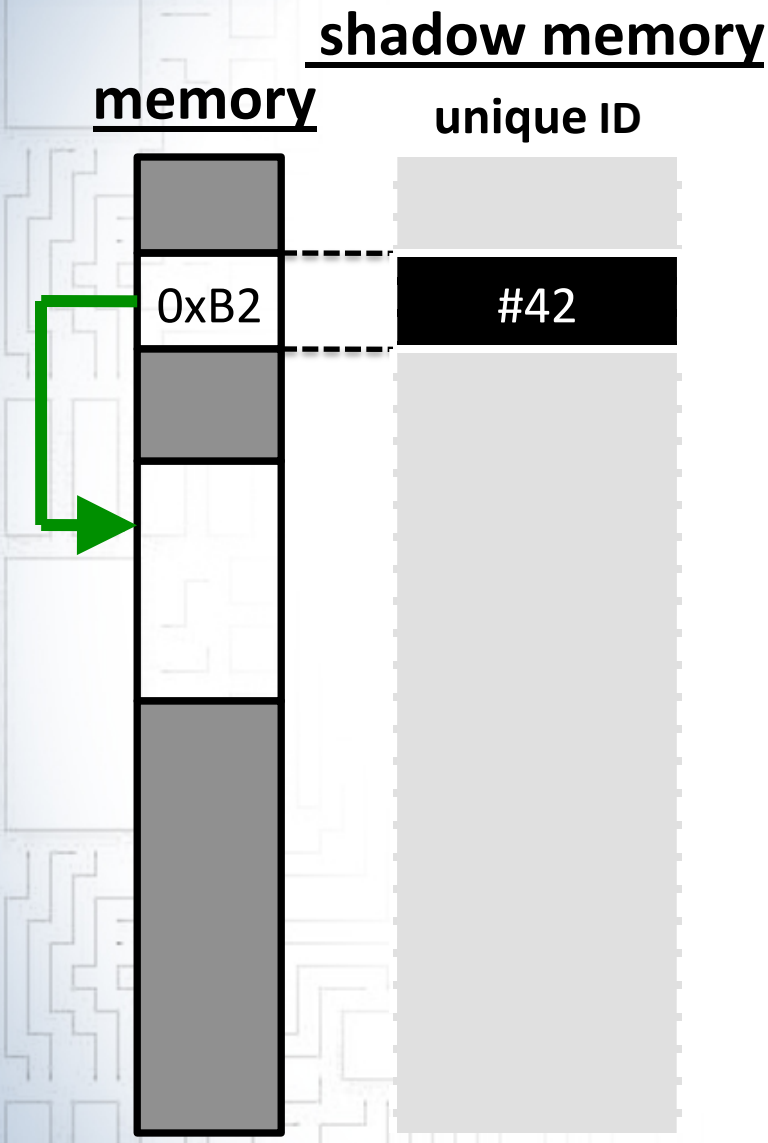
- Allocate ***unique identifier (UID)*** for each allocation
  - Record the set of valid identifiers
  - Track this UID with each pointer
  - Invalidate identifiers on memory deallocation
  - Check for identifier validity on memory accesses



Where are the
per-pointer metadata stored?

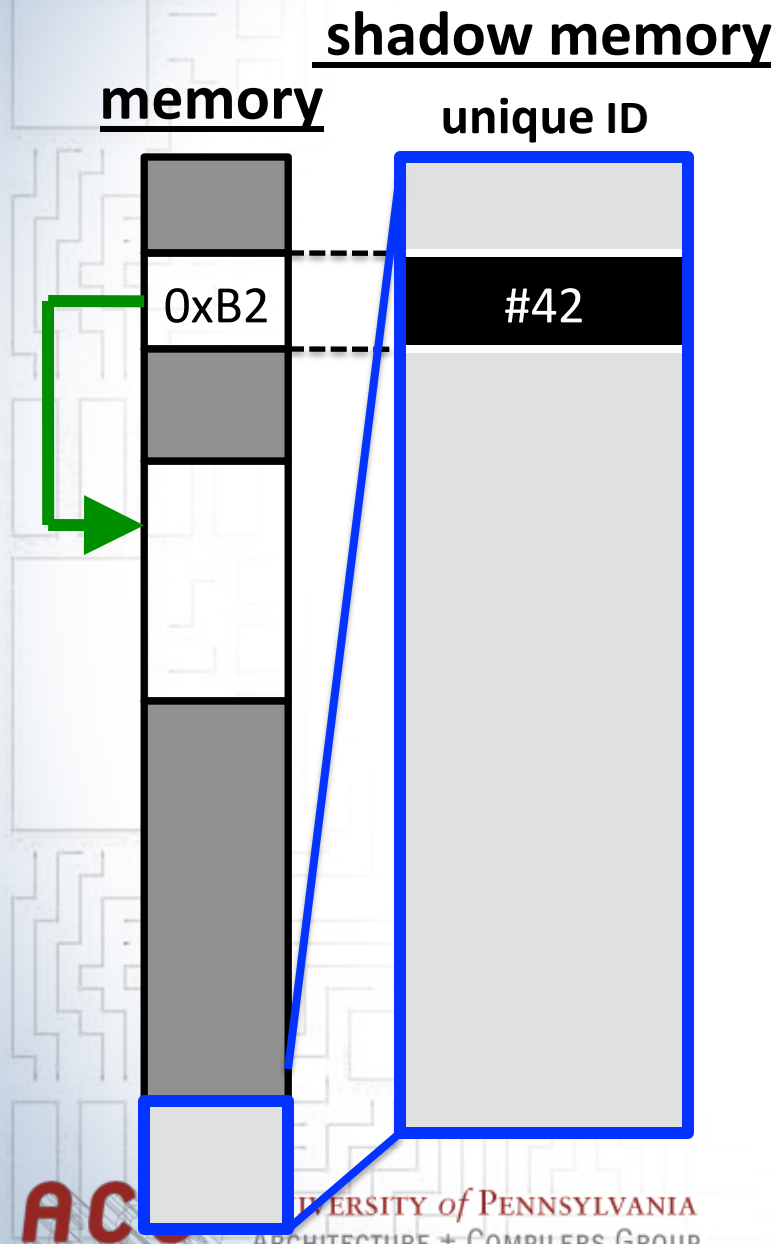
How is the validity
check performed efficiently?

Shadow Memory for Per-Pointer Metadata



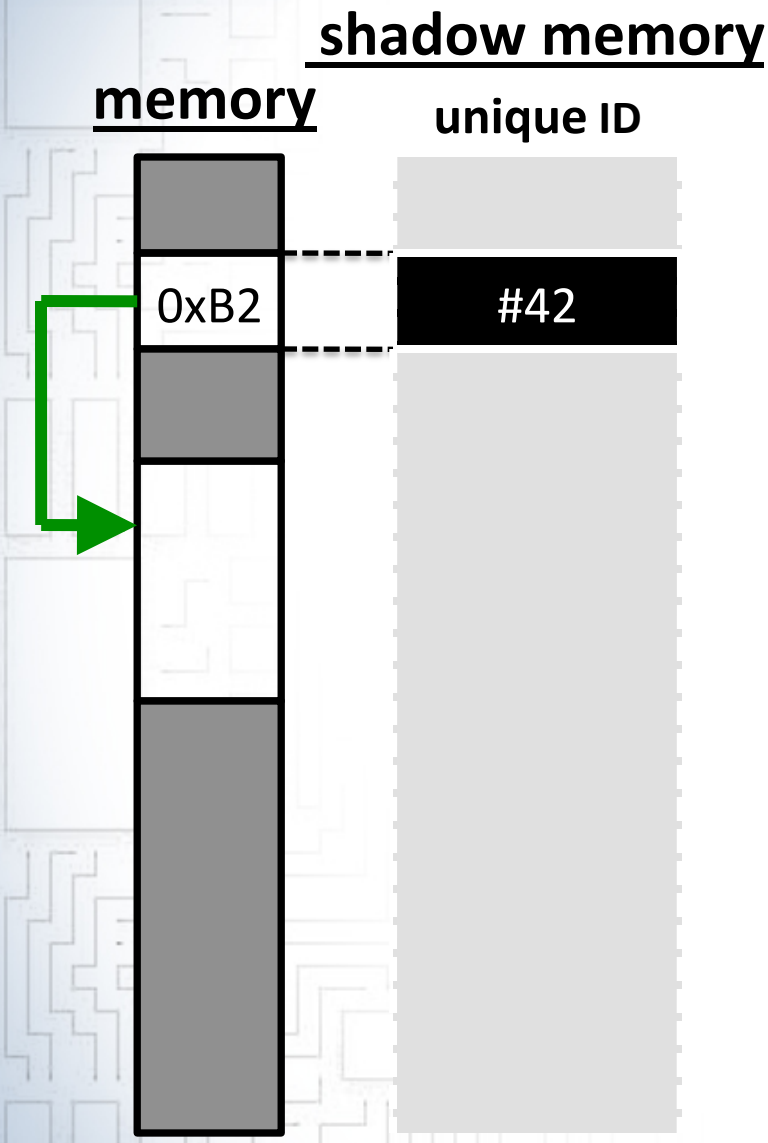
- A ***shadow memory*** for per-pointer metadata
  - **Leaves memory layout unchanged**
  - Protects metadata from corruption

Shadow Memory for Per-Pointer Metadata



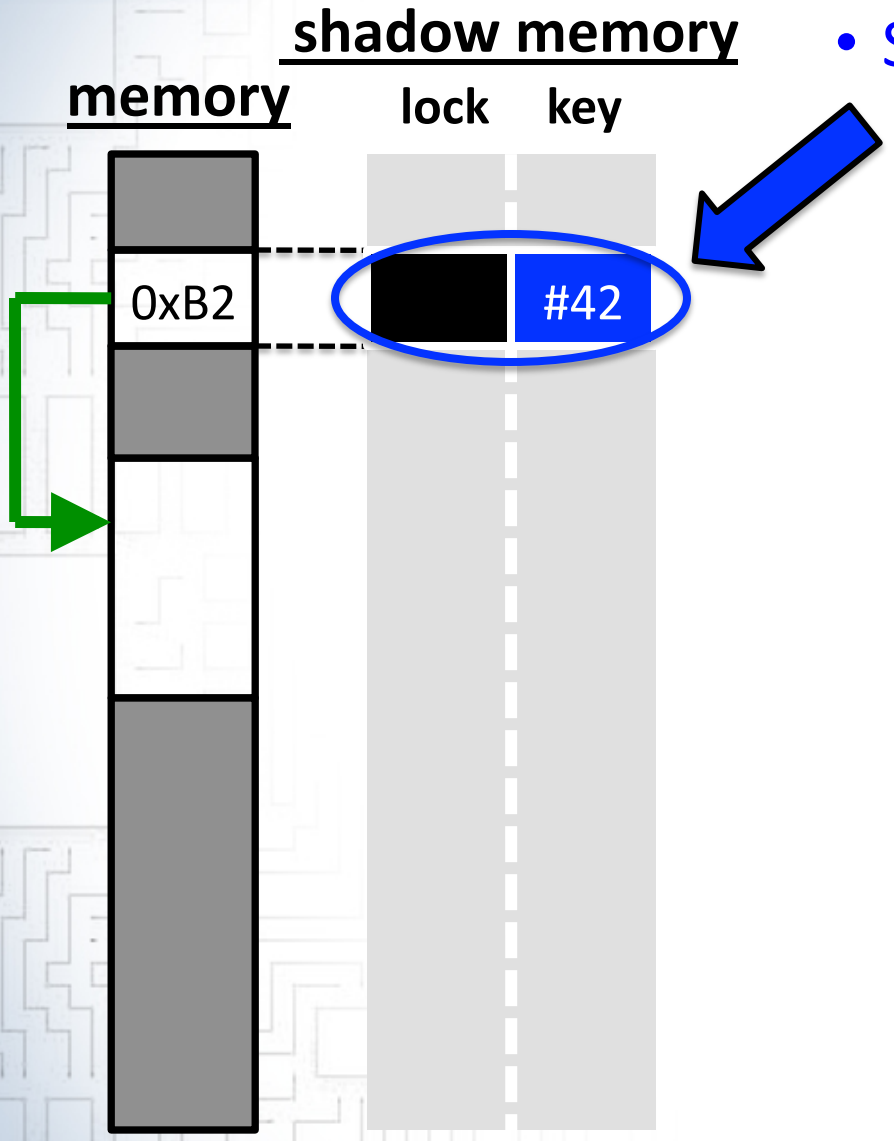
- A ***shadow memory*** for per-pointer metadata
  - **Leaves memory layout unchanged**
  - Protects metadata from corruption
- Mapped into virt. address space
  - Simple mapping
  - Allocated on demand by system

Shadow Memory for Per-Pointer Metadata



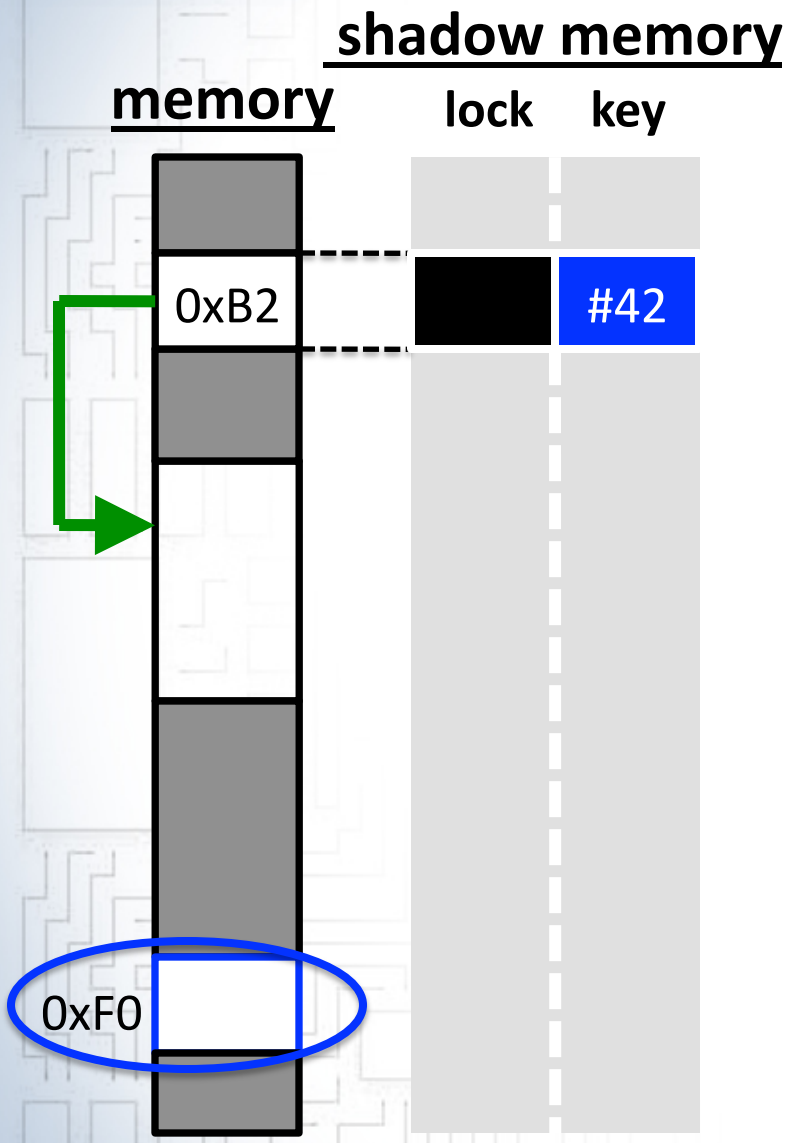
- A ***shadow memory*** for per-pointer metadata
  - **Leaves memory layout unchanged**
  - Protects metadata from corruption
- Mapped into virt. address space
  - Simple mapping
  - Allocated on demand by system

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



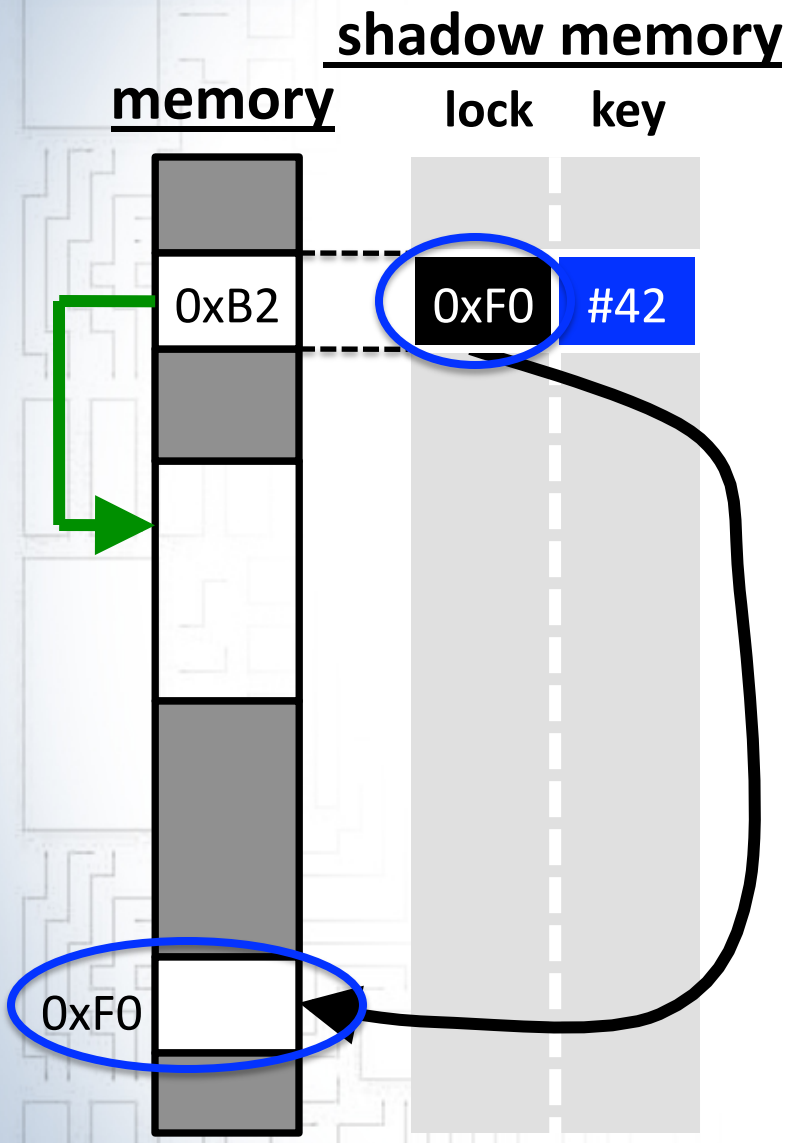
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



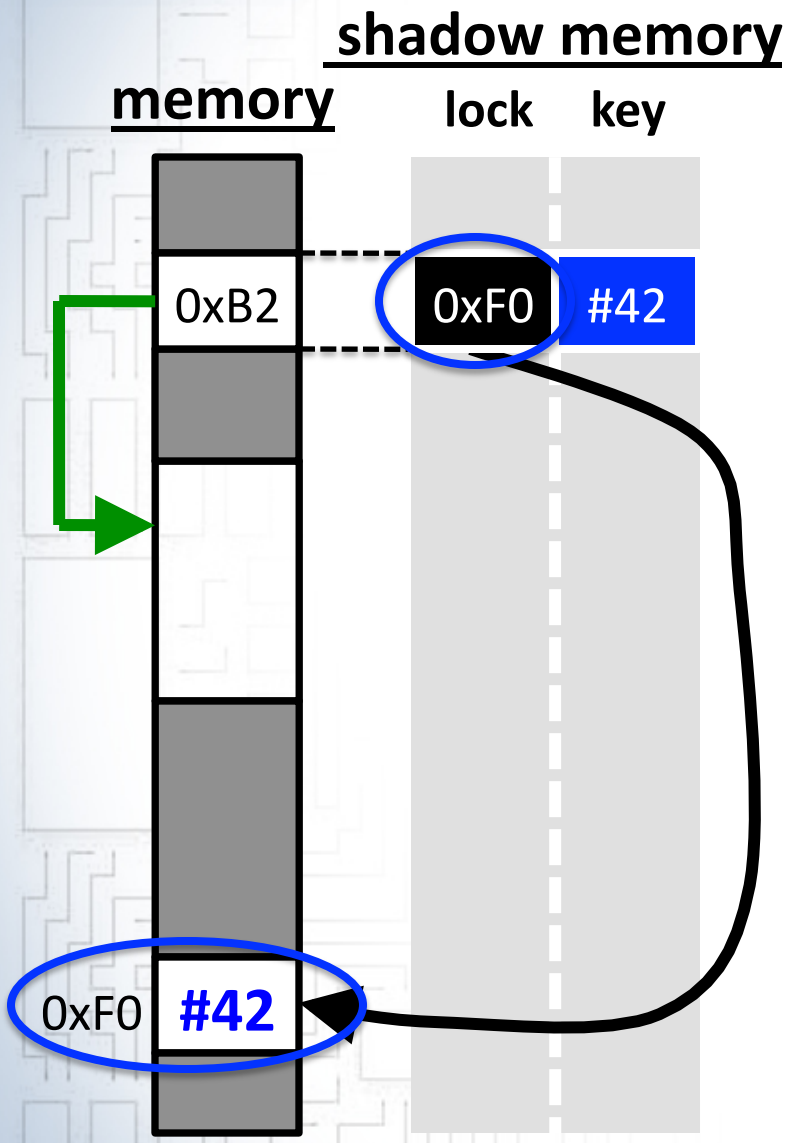
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



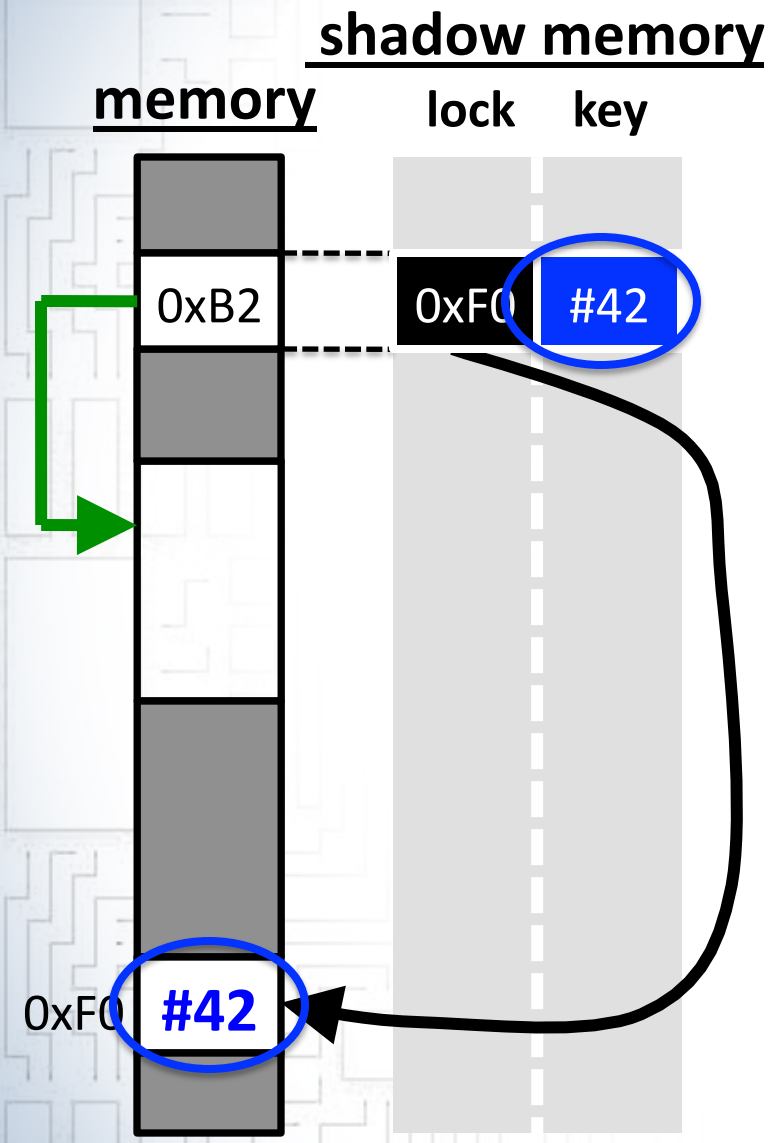
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



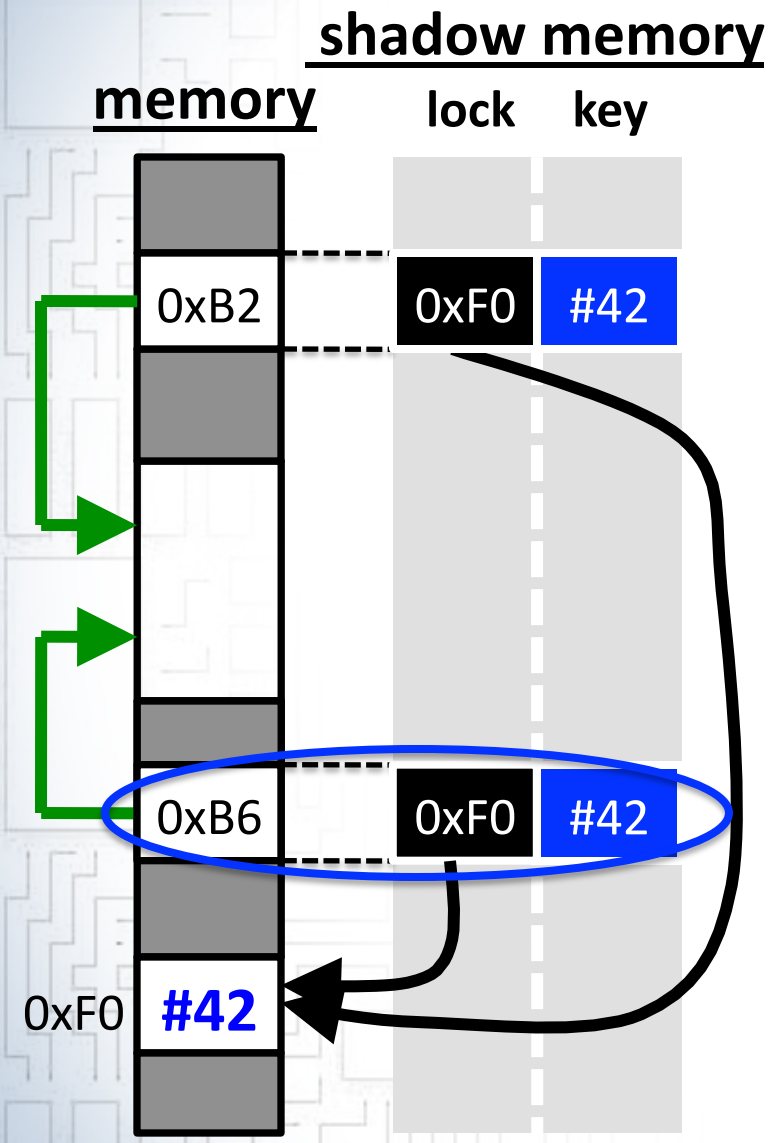
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



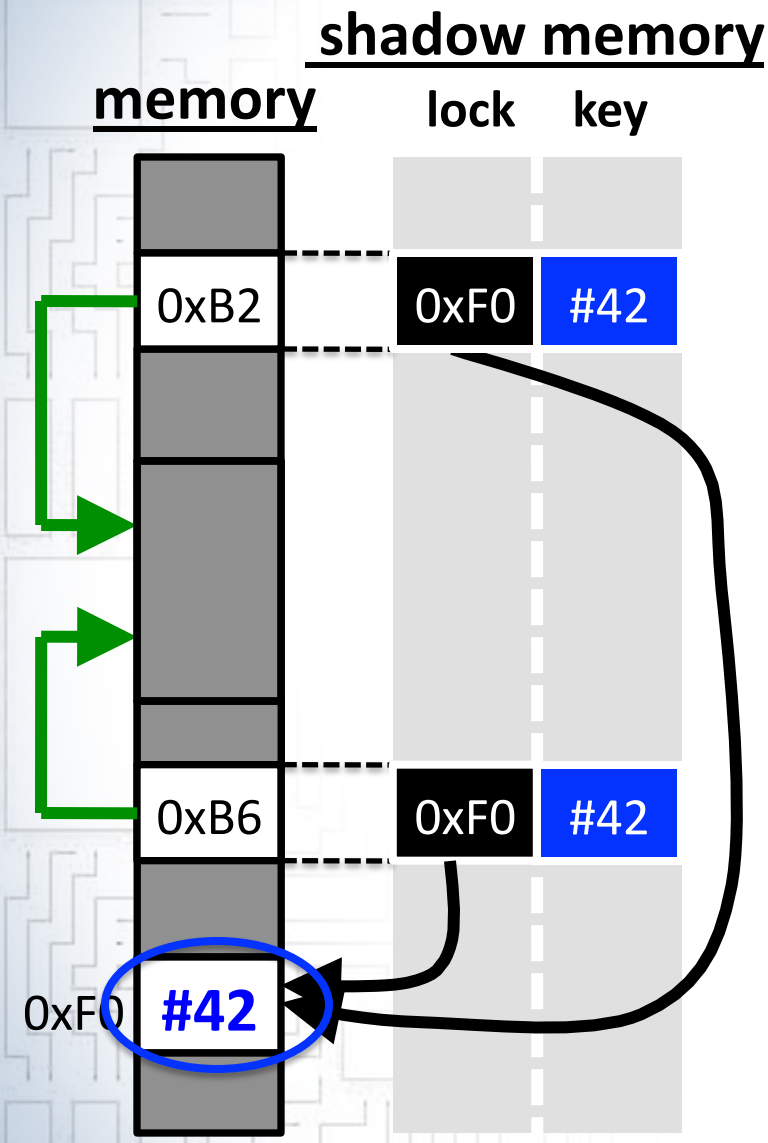
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key
- Invariant:
  - valid if memory[lock] == key

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



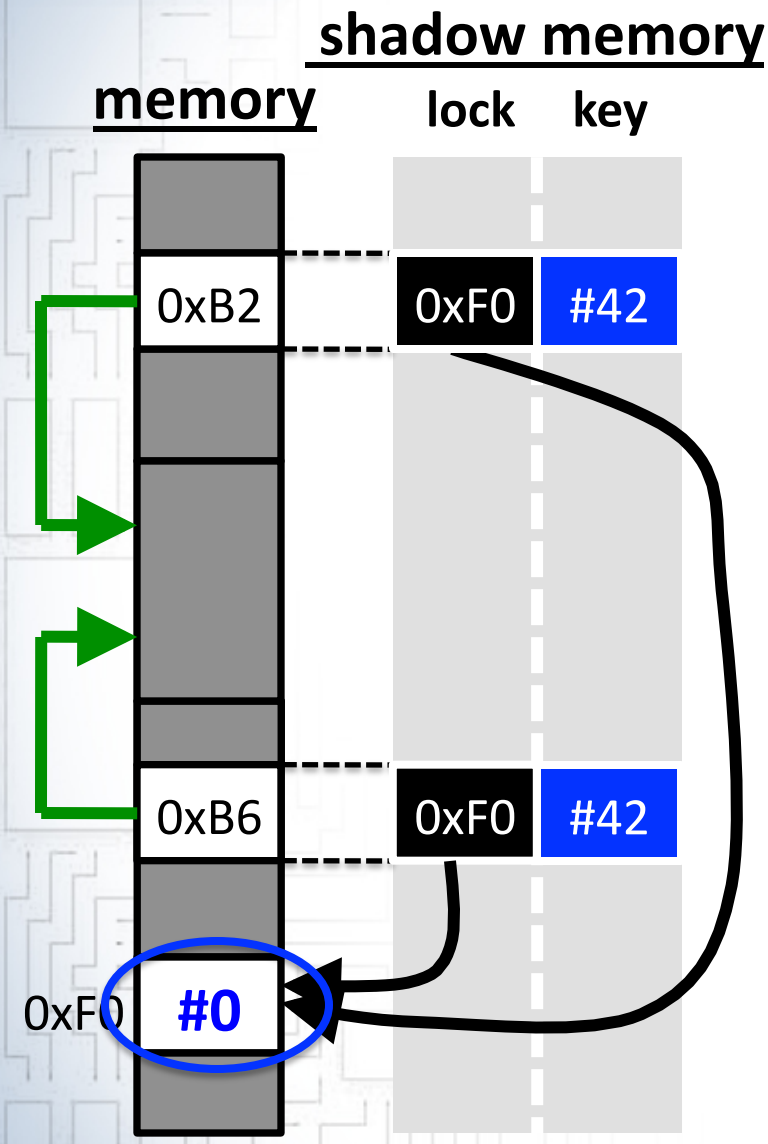
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key
- Invariant:
  - valid if memory[lock] == key
- Propagate as pointer copied

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



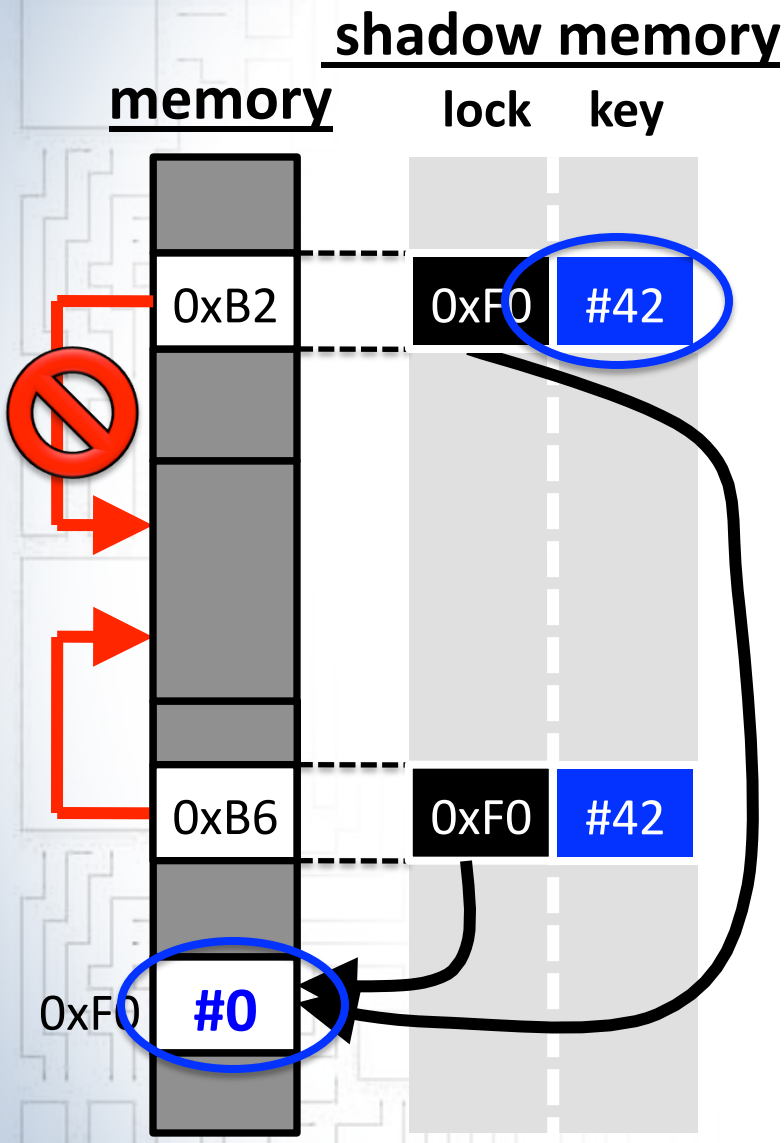
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key
- Invariant:
  - valid if memory[lock] == key
- Propagate as pointer copied
- Deallocation
  - memory[lock] = invalid

Lock and Key Checking [P&F, MSCC, Chuang et al, CETS]



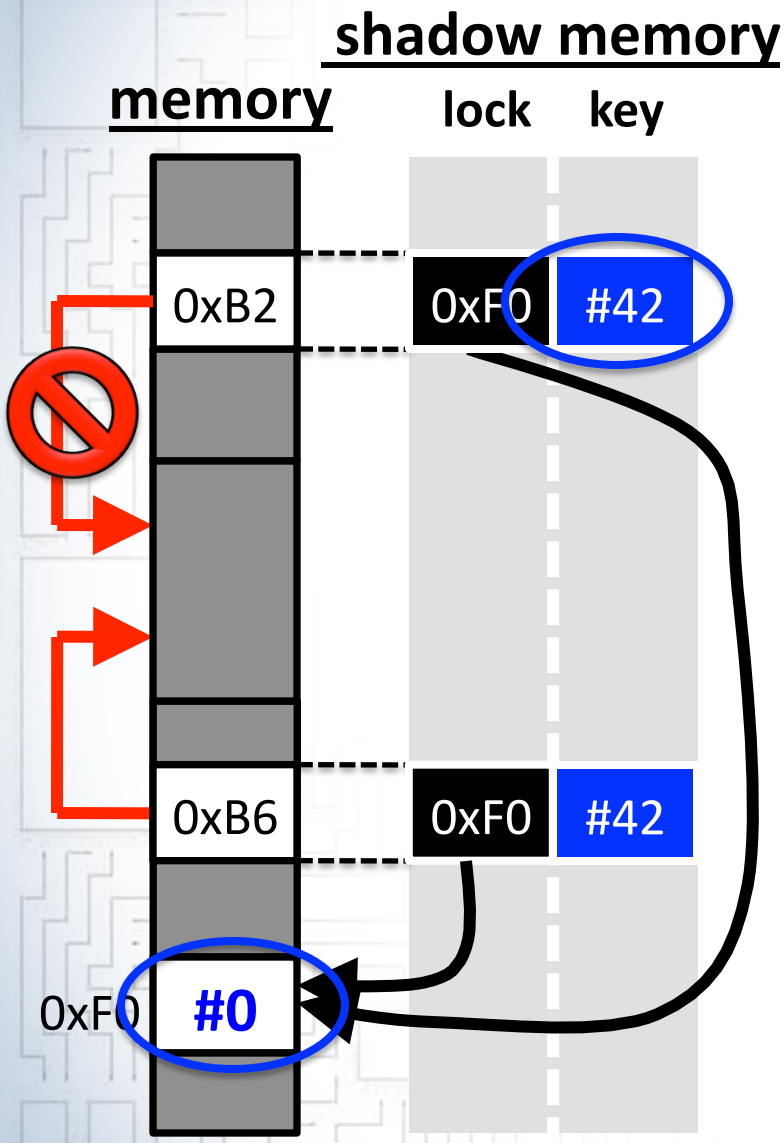
- Split unique ID (UID) into:
  - Location of "lock" (reusable)
  - "key" (unique)
- Allocation:
  - Assign a key & lock location
  - Set memory[lock] = key
- Invariant:
  - valid if memory[lock] == key
- Propagate as pointer copied
- Deallocation
  - memory[lock] = invalid

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

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops
ld.word Memory[r1] ➜ r2	check r1.meta ld.word Memory[r1] ➜ r2 ld Shadow[r1] ➜ r2.meta
add r2, 8 ➜ r3	add r2, 8 ➜ r3 copy r2.meta ➜ r3.meta
ld.byte Memory[r3] ➜ r4	check r3.meta ld.byte Memory[r3] ➜ r4 ~~ld Shadow[r3] ➜ r4.meta~~

Example of unneeded shadow access
(e.g., byte load is not a pointer)

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops	Rename
ld.word Memory[r1] ➜ r2	check r1.meta ld.word Memory[r1] ➜ r2 ld Shadow[r1] ➜ r2.meta	
add r2, 8 ➜ r3	add r2, 8 ➜ r3 copy r2.meta ➜ r3.meta	
ld.byte Memory[r3] ➜ r4	check r3.meta ld.byte Memory[r3] ➜ r4 ~~ld Shadow[r3] ➜ r4.meta~~	

Decoupled metadata:
Two mappings for each register

Reg	Value	Meta
r1		
r2		
r3		
r4		

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops	Rename
ld.word Memory[r1] ➜ r2	check r1.meta ld.word Memory[r1] ➜ r2 ld Shadow[r1] ➜ r2.meta	
add r2, 8 ➜ r3	add r2, 8 ➜ r3 copy r2.meta ➜ r3.meta	
ld.byte Memory[r3] ➜ r4	check r3.meta ld.byte Memory[r3] ➜ r4 ~~ld Shadow[r3] ➜ r4.meta~~	

Decoupled metadata:
Two mappings for each register

Reg	Value	Meta
r1	**p1**	**p10**
r2		
r3		
r4		

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops	Rename
ld.word Memory[r1] ➜ r2	check r1.meta	check p10
	ld.word Memory[r1] ➜ r2	ld.word Memory[p1] ➜ **p2**
	ld Shadow[r1] ➜ r2.meta	ld Shadow[p1] ➜ **p11**
add r2, 8 ➜ r3	add r2, 8 ➜ r3	
	copy r2.meta ➜ r3.meta	
ld.byte Memory[r3] ➜ r4	check r3.meta	
	ld.byte Memory[r3] ➜ r4	
	~~ld Shadow[r3] ➜ r4.meta~~	

Decoupled metadata:
Two mappings for each register

Reg	Value	Meta
r1	p1	p10
r2	**p2**	**p11**
r3		
r4		

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops	Rename
ld.word Memory[r1] ➜ r2	check r1.meta	check p10
	ld.word Memory[r1] ➜ r2	ld.word Memory[p1] ➜ p2
	ld Shadow[r1] ➜ r2.meta	ld Shadow[p1] ➜ p11
add r2, 8 ➜ r3	add r2, 8 ➜ r3	add p2, 8 ➜ **p3**
	copy r2.meta ➜ r3.meta	
ld.byte Memory[r3] ➜ r4	check r3.meta	
	ld.byte Memory[r3] ➜ r4	
	~~ld Shadow[r3] ➜ r4.meta~~	

Copy elimination via register remapping

[Jourdan et al, Petric et al]

Reg	Value	Meta
r1	p1	p10
r2	p2	p11
r3	**p3**	
r4		

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops	Rename
ld.word Memory[r1] ➜ r2	check r1.meta	check p10
	ld.word Memory[r1] ➜ r2	ld.word Memory[p1] ➜ p2
	ld Shadow[r1] ➜ r2.meta	ld Shadow[p1] ➜ p11
add r2, 8 ➜ r3	add r2, 8 ➜ r3	add p2, 8 ➜ **p3**
	copy r2.meta ➜ r3.meta	**<<copy eliminated>>**
ld.byte Memory[r3] ➜ r4	check r3.meta	
	ld.byte Memory[r3] ➜ r4	
	~~ld Shadow[r3] ➜ r4.meta~~	

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	Insert Micro-Ops	Rename
ld.word Memory[r1] ➜ r2	check r1.meta	check p10
	ld.word Memory[r1] ➜ r2	ld.word Memory[p1] ➜ p2
	ld Shadow[r1] ➜ r2.meta	ld Shadow[p1] ➜ p11
add r2, 8 ➜ r3	add r2, 8 ➜ r3	add p2, 8 ➜ p3
	copy r2.meta ➜ r3.meta	**<<copy eliminated>>**
ld.byte Memory[r3] ➜ r4	check r3.meta	check p11
	ld.byte Memory[r3] ➜ r4	ld.byte Memory[p3] ➜ **p5**
	~~ld Shadow[r3] ➜ r4.meta~~	

Reg	Value	Meta
r1	p1	p10
r2	p2	p11
r3	p3	p11
r4	**p5**	--

Copy elimination via register remapping

[Jourdan et al, Petric et al]

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops	Rename
ld.word Memory[r1] ➜ r2	check r1.meta	check p10
	ld.word Memory[r1] ➜ r2	ld.word Memory[p1] ➜ p2
	ld Shadow[r1] ➜ r2.meta	ld Shadow[p1] ➜ p11
add r2, 8 ➜ r3	add r2, 8 ➜ r3	add p2, 8 ➜ p3
	copy r2.meta ➜ r3.meta	**<<copy eliminated>>**
ld.byte Memory[r3] ➜ r4	check r3.meta	check p11
	ld.byte Memory[r3] ➜ r4	ld.byte Memory[p3] ➜ **p5**
	~~ld Shadow[r3] ➜ r4.meta~~	

Reg	Value	Meta
r1	p1	p10
r2	p2	p11
r3	p3	p11
r4	**p5**	--

Copy elimination via register remapping

[Jourdan et al, Petric et al]

Decoupled metadata operations off critical path, becomes "extra" ILP

Micro-Op Insertion & Rename Example

Original Code	Insert Micro-Ops	Rename
	check r1.meta	check p10
ld.word Me	] ➜ r2	ld.word Memory[p1] ➜ p2
	meta	ld Shadow[p1] ➜ p11
add r2, 8 ➜ r3	add r2, 8 ➜ r3	add p2, 8 ➜ p3
	eta	**<<copy eliminated>>**
	check r3.meta	check p11
ld.	4	ld.byte Memory[p3] ➜ **p5**
	~~ta~~	

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

Register-to-register copies eliminated

Checks occur on each memory operation…
And each requires a cache access…

Reg	Value	Meta
r1	p1	p10
r2	p2	p11
r3	p3	p11
r4	**p5**	--

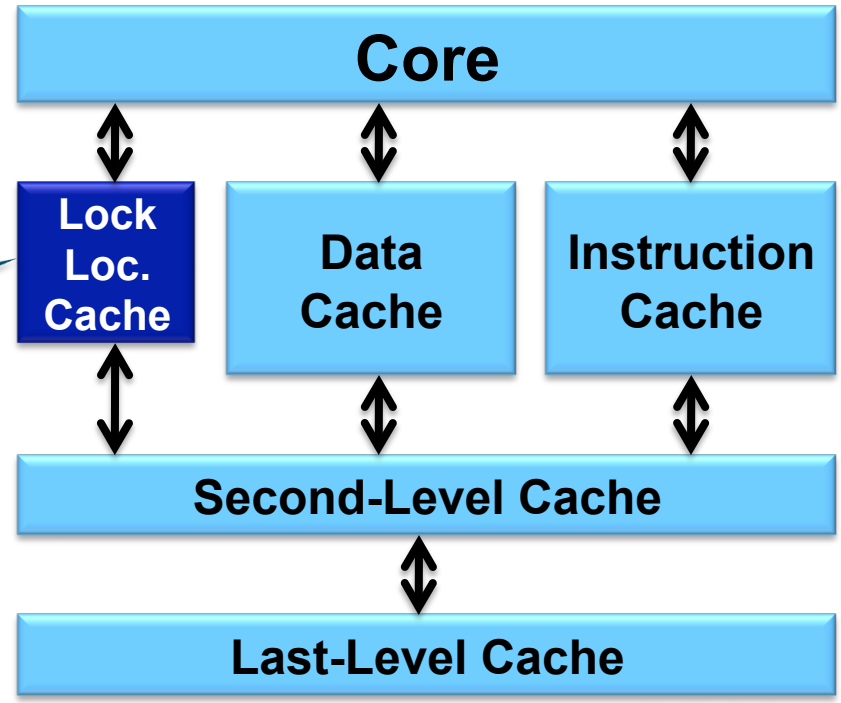
Copy elimination via register remapping
[Jourdan et al, Petric et al]

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



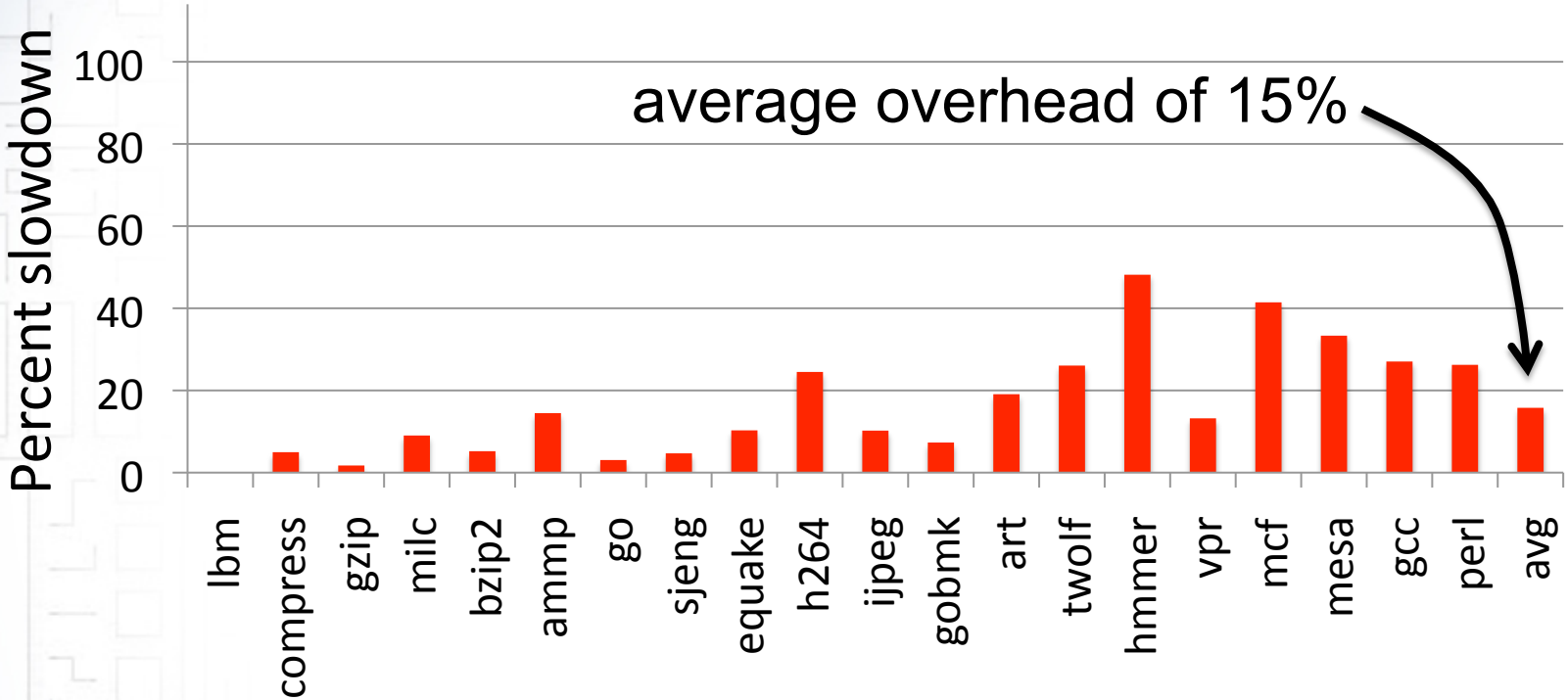
- Overhead: 24% ➜ 15%

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

What About Buffer Overflows?

What About Buffer Overflows?

Memory safety has two components:

Bounds safety . **Use-after-free safety**

Watchdog +

[also this work]

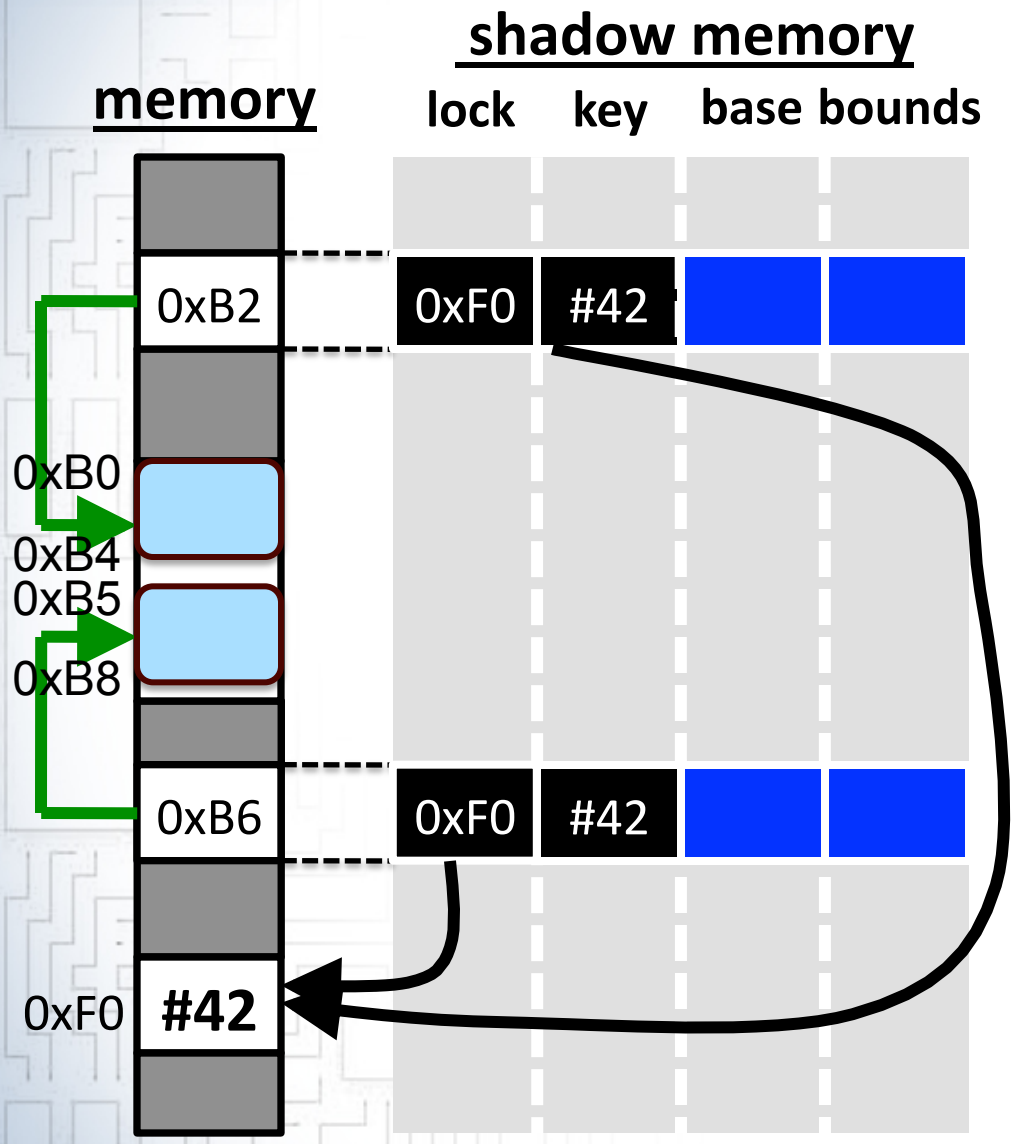
- Pointer-based, disjoint
- Unique identifier check
- ~25% overhead

Hardware

Software

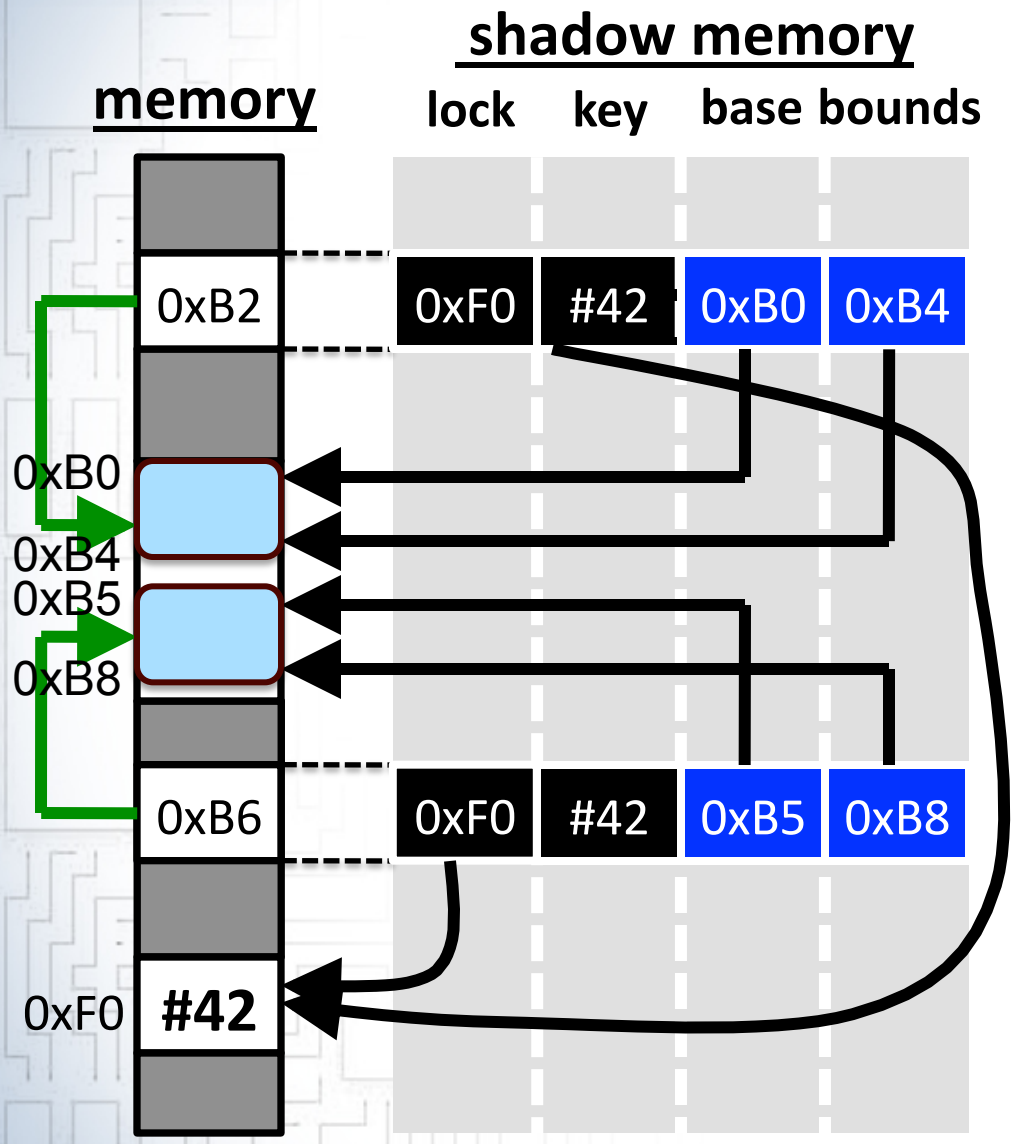
- Implemented via micro-op injection
- Identification of pointer operations
- Efficient metadata propagation via copy elimination
- Uses lock location cache to increase check throughput

Extend Metadata for Bounds Enforcement



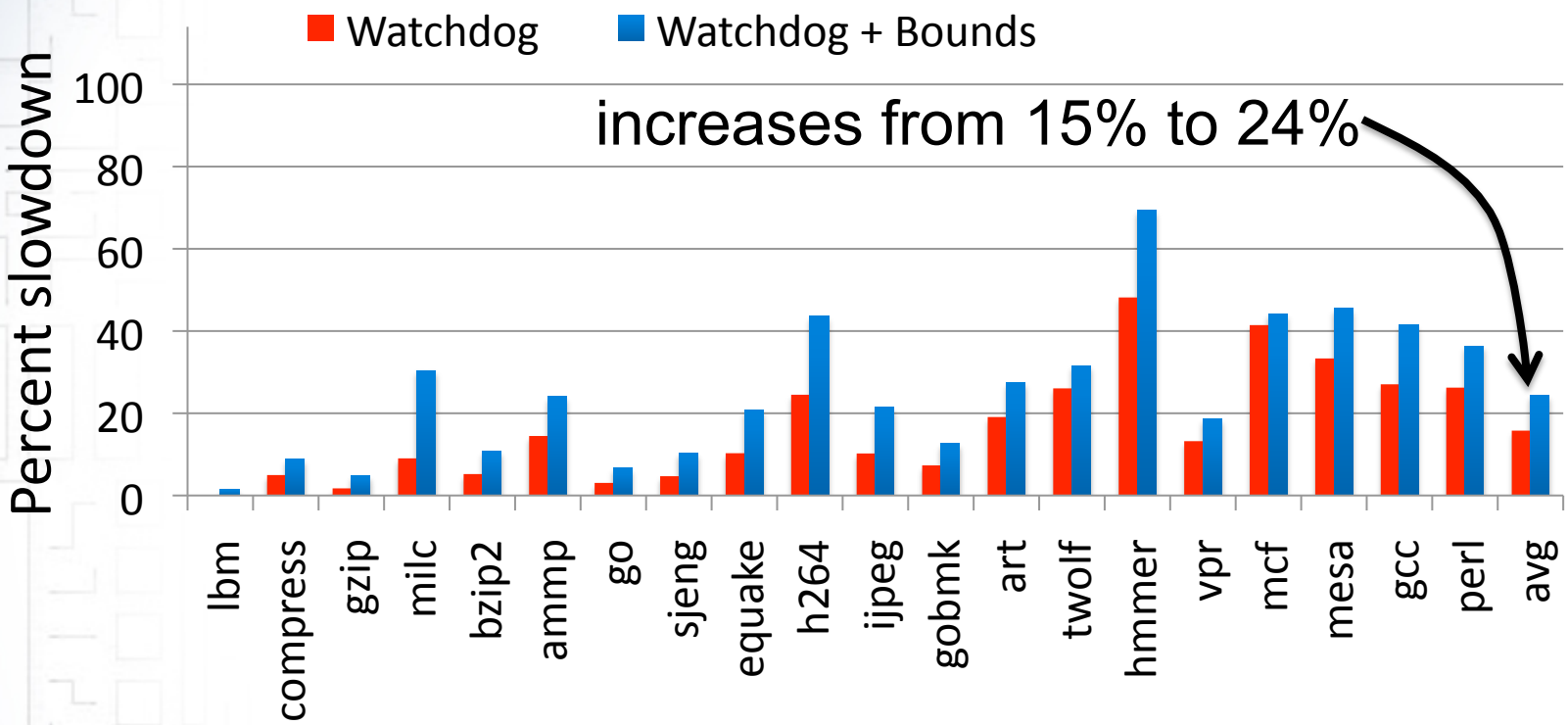
- Extend metadata
  - Base & bounds

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

ACG

University of Pennsylvania

Architecture + Compilers Group

Penn

University of Pennsylvania