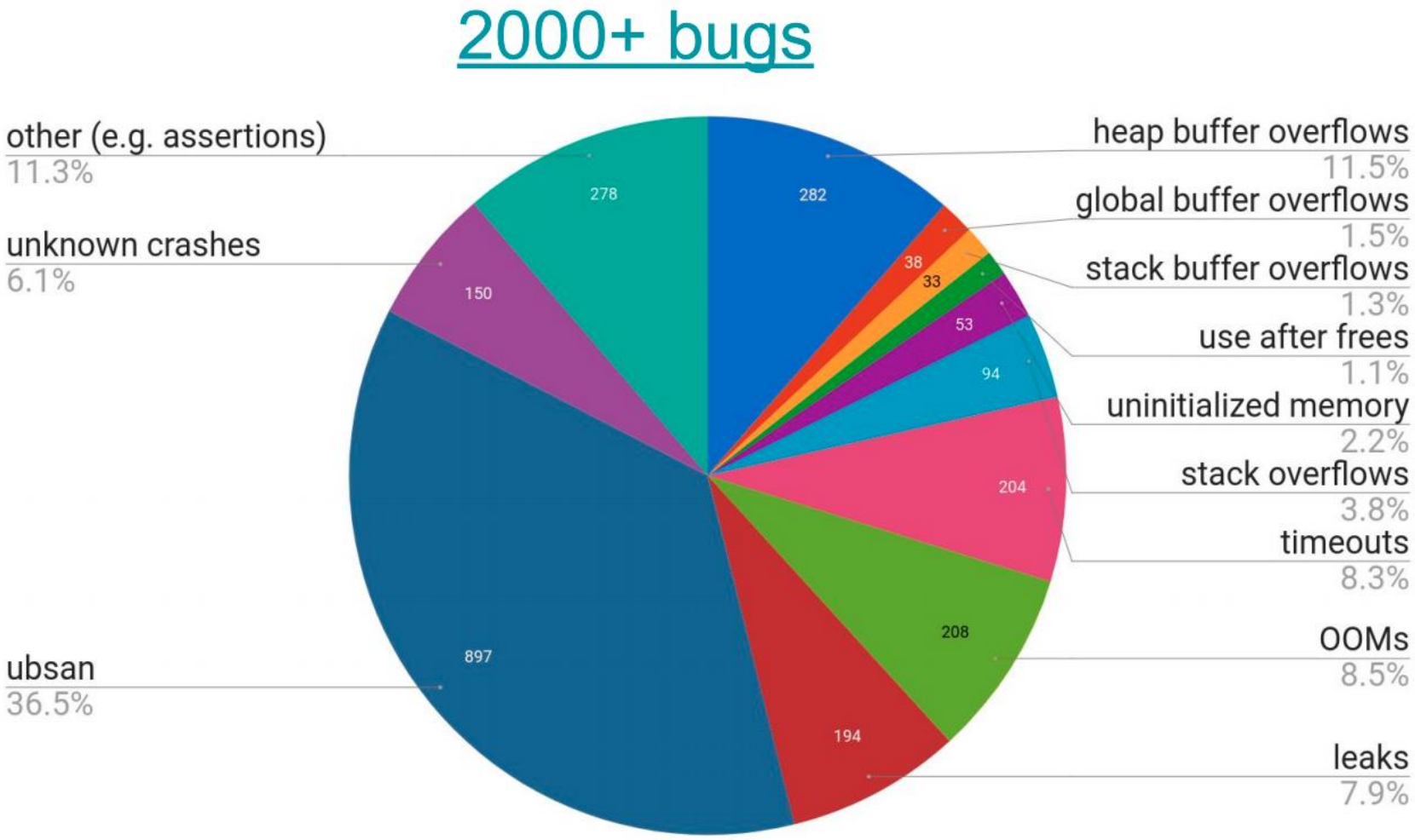
Blacklisting vs Whitelisting for memory safety

Prof. Simha Sethumadhavan

Columbia University

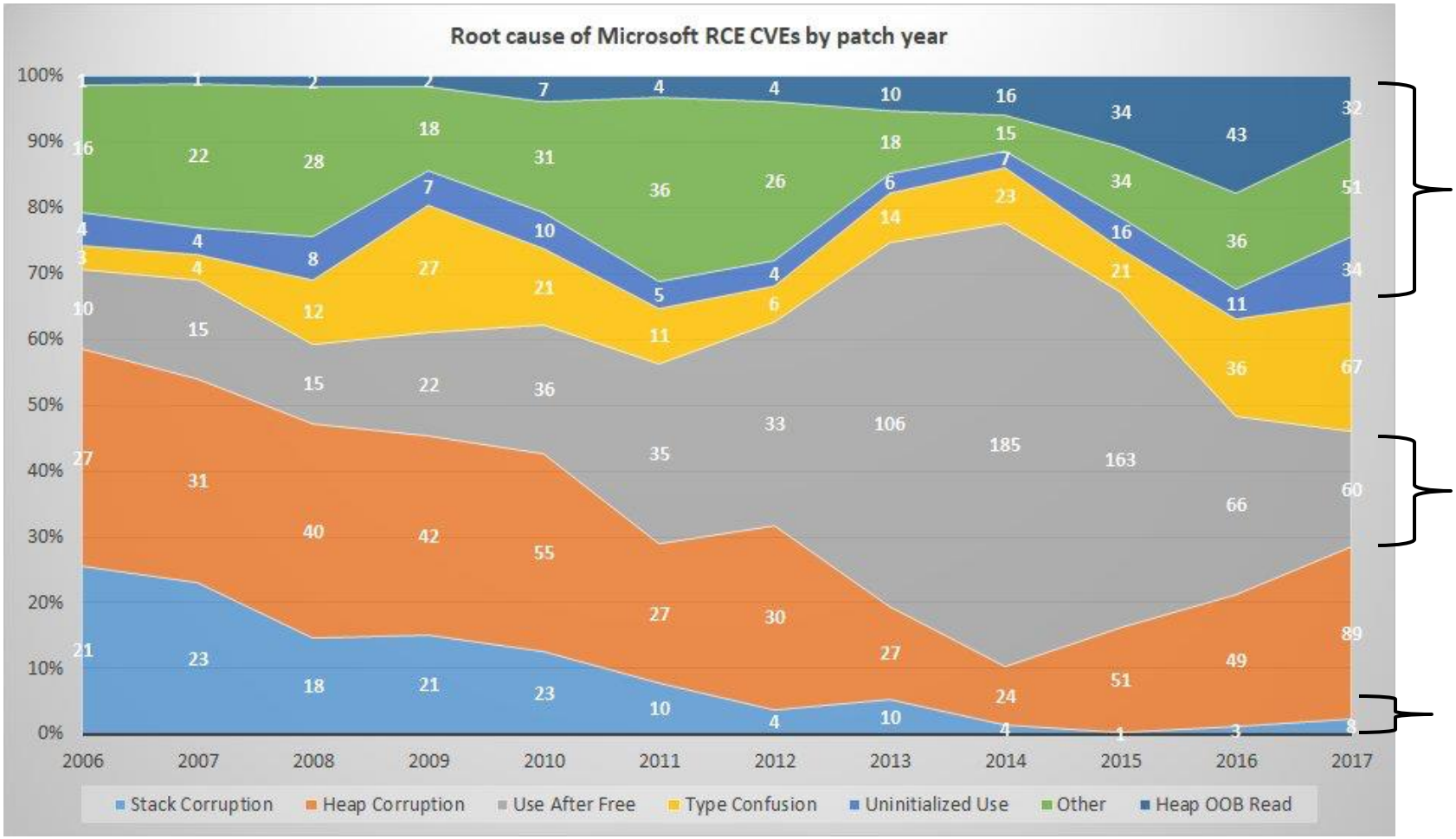
simha@columbia.edu

Motivation: Memory vulns. are pervasive



Source: OSS-Fuzz - Google's continuous fuzzing service for open source software, **Usenix Security 17**

Memory Vulnerabilities -> Remote Exploits



Source: Matt Miller, Microsoft Security Response Center

Not a new problem; many solutions

1988: 1st Internet worm – Morris Worm -- exploited a buffer overflow

- Language-based solutions: Use languages that are strongly-typed w/ automatic memory management.
  - Legacy code problem
  - Performance overhead of automatic memory management (>15%)
  - Programmers appear to prefer weakly-typed languages (e.g., Javascript)

- Instrumentation based solutions
  - Purify -> *Sanitizers
  - High overheads (~200%) but robust for testing
  - Problem: limited test cycles + overhead => many bugs remain

- We will look at hardware solutions: promise low-overhead
  - Recent support from commercial vendors
  - Oracle SPARC ADI, Intel MPX, ARM memory tagging v8.5a
  - Many academic solutions

Taxonomy of Memory Safety

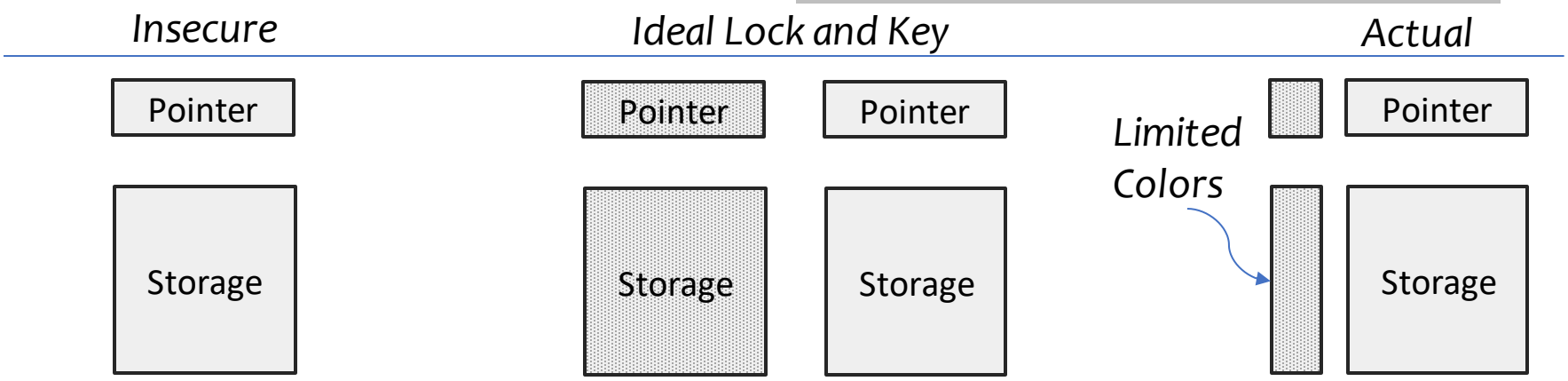
- Type:
  - Whitelisting: Allow access to locations based on authorization
  - Blacklisting: Prevent a location from being accessed
  - Technically, whitelisting is more powerful than blacklisting
- Coverage:
  - Program aspect covered: heap, stack, global etc.
  - Granularity: Allocation safety, Object safety or Field safety
  - Temporal safety
- Commercial State-of-the-art:
  - Allocation granularity spatial safety, and little temporal safety (if any)
  - Whitelisting performance cost: MPX (~80%); SPARC ADI (~10%)
  - Silicon costs vary and usually high; complexity is high.
- We will discuss an alternative solution in this talk.

Whitelisting: Lock and Key (ideal)

```
1: int main() {
2:    char * ptr;
3:    ptr = malloc(sizeof(int) * 10);
4:   *[ptr + 5] = 4;
5:   free(ptr);
6: }
```

→

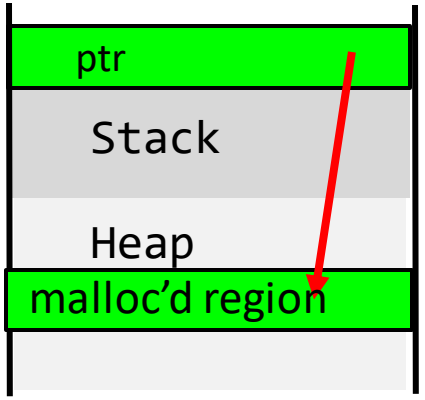
```
1: int main() {
2:    char * ptr;
3:    ptr = malloc(sizeof(int)*10);
3a: ptr_color = malloc (sizeof(color) * 10);
3b:  ptr_color = give_color();
3c:  for (i = 0; I < 10; i++)
        *[ptr_color+i] = ptr_color;
4:   *[ptr + 5] = 4;
4a:  if (ptr_color + 5 != ptr_color) exception
5:   free(ptr);
5a: free(ptr_color);
6: }
```



Basic Base and Bounds Scheme: Inline metadata, allocation granularity - whitelisting

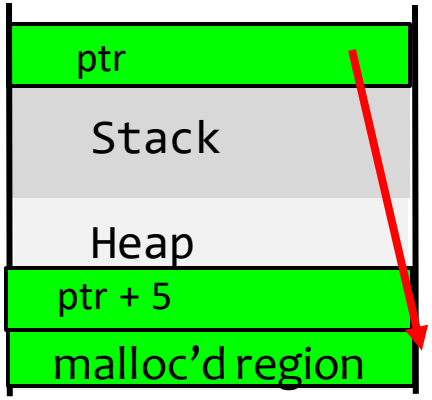
```
int main() {
        char * ptr;
        ptr =
malloc(sizeof(int));

        *[ptr + 5] = 4;

        free(ptr);
}
```

1. Alloc



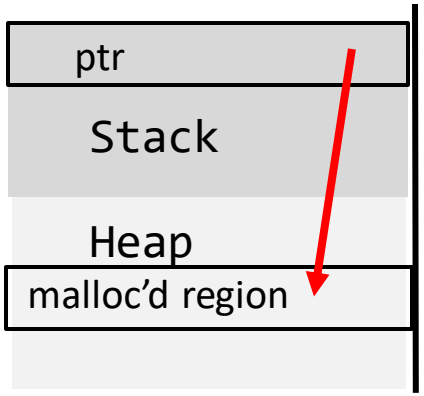
2. Use

All accesses using *ptr* get the same color

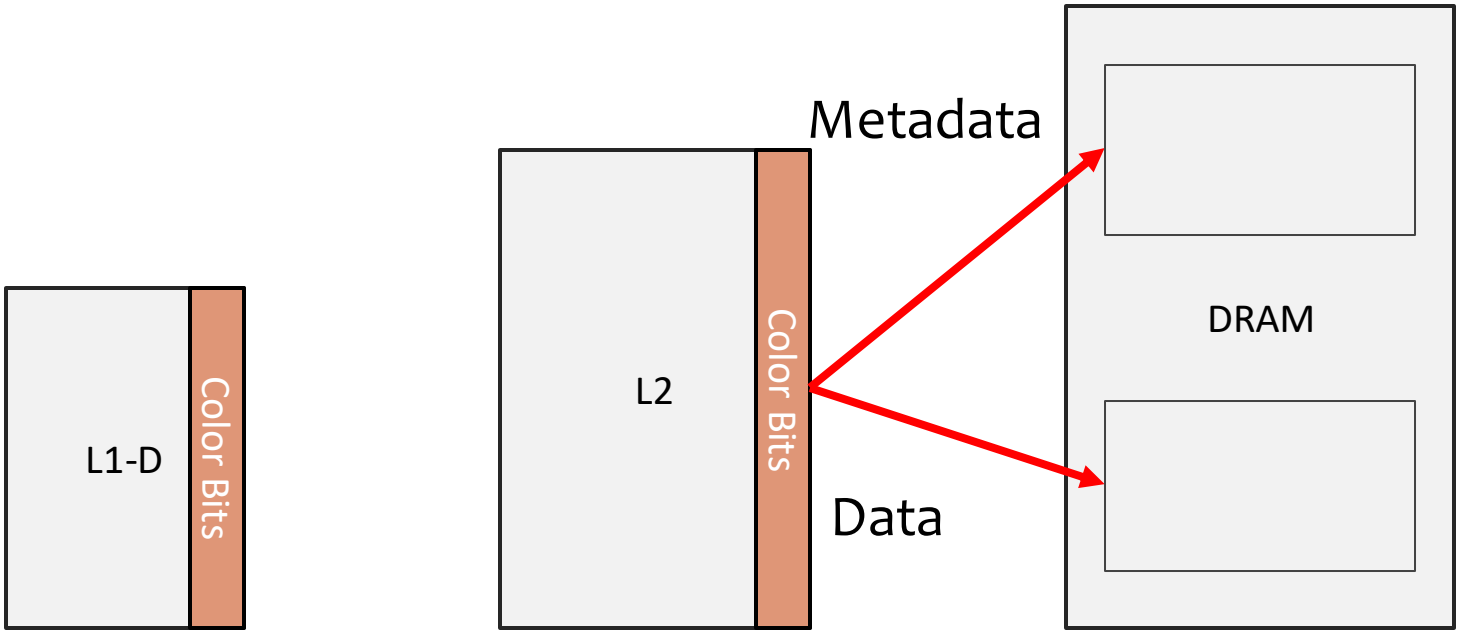


3. Dealloc

Change color



Implementation



Overhead: two memory accesses

Blacklisting solution

Califorms

Practical Byte-Granular Memory Blacklisting using Califorms

Hiroshi Sasaki
Columbia University
sasaki@cs.columbia.edu

Miguel A. Arroyo
Columbia University
miguel@cs.columbia.edu

M. Tarek Ibn Ziad
Columbia University
mtarek@cs.columbia.edu

Koustubha Bhat[†]
Vrije Universiteit Amsterdam
k.bhat@vu.nl

Kanad Sinha
Columbia University
kanad@cs.columbia.edu

Simha Sethumadhavan
Columbia University
simha@cs.columbia.edu

ABSTRACT

Recent rapid strides in memory safety tools and hardware have improved software quality and security. While coarse-grained memory safety has improved, achieving memory safety at the granularity of individual objects remains a challenge due to high performance overheads usually between ~1.7x–2.2x. In this paper, we present a novel idea called *Califorms*, and associated program observations, to obtain a low overhead security solution for practical, byte-granular memory safety.

The idea we build on is called memory blacklisting, which prohibits a program from accessing certain memory regions based on program semantics. State of the art hardware-supported memory blacklisting, while much faster than software blacklisting, creates memory fragmentation (on the order of few bytes) for each use of the blacklisted location. We observe that metadata used for blacklisting can be stored in dead spaces in a program's data memory and that this metadata can be integrated into the microarchitecture by changing the cache line format. Using these observations, a Califorms based system proposed in this paper reduces the performance overheads of memory safety to ~1.02x–1.16x while providing byte-granular protection and maintaining very low hardware overheads. Moreover, the fundamental idea of storing metadata in empty spaces and changing cache line formats can be used for other security and performance applications.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; • **Computer systems organization → Architectures**.

KEYWORDS

memory safety, memory blacklisting, caches

ACM Reference Format:
Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat[†], Kanad Sinha, and Simha Sethumadhavan. 2019. Practical Byte-Granular Memory Blacklisting using Califorms. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358299

[†]Part of this work was carried out while the author was a visiting student at Columbia University.

1 INTRODUCTION

Historically, program memory safety violations have provided a significant opportunity for exploitation by attackers: for instance, Microsoft recently revealed that the root cause of more than half of all exploits are software memory safety violations [30]. To address this threat, software checking tools such as AddressSanitizer [25] and commercial hardware support for memory safety such as Oracle's ADI [21] and Intel's MPX [20] have enabled programmers to detect and fix memory safety violations before deploying software.

Current software and hardware-supported solutions excel at providing coarse-grained, inter-object memory safety which involves detecting memory access beyond arrays and heap allocated regions (`malloc`'d struct and class instances). However, they are not suitable for fine-grained memory safety (i.e., intra-object memory safety—detecting overflows within objects, such as fields within a struct, or members within a class) due to the high performance overheads and/or need for making intrusive changes to the source code [29]. Some real-world scenarios where intra-object memory safety problems manifest are type confusion vulnerabilities (e.g., CVE–2017–5115 [2]) and uninitialized data leaks through padding bytes (e.g., CVE–2014–1444 [1]), both recognized as high-impact security classes [13, 16].

For instance, a recent work that aims to provide intra-object overflow protection functionality incurs a 2.2x performance overhead [10]. These overheads are problematic because they not only reduce the number of pre-deployment tests that can be performed, but also impede post-deployment continuous monitoring, which researchers have pointed out is necessary for detecting benign and malicious memory safety violations [26]. Thus, a low overhead memory safety solution that can enable continuous monitoring and provide complete program safety has been elusive.

The source of overheads stem from how current designs store

Author's Preprint: For Educational Use Only
To Appear at The 45th International Symposium on Computer Architecture - ISCA 2018, LA, USA, June 4 - 6th

Practical Memory Safety with REST

Kanad Sinha Simha Sethumadhavan

Department of Computer Science
Columbia University
New York, NY, USA
{kanad,simha}@cs.columbia.edu

***Abstract*—In this paper, we propose Random Embedded Secret Tokens (*REST*), a simple hardware primitive to provide content-based checks, and show how it can be used to mitigate common types of spatial and temporal memory errors at very low cost. *REST* is simply a very large random value that is embedded into programs. To provide memory safety, *REST* is used to bookend data structures during allocation. If the hardware accesses a *REST* value during execution, due to programming errors or adversarial actions, it reports a privileged memory safety exception.**

Implementing *REST* requires 1 bit of metadata per L1 data cache line and a comparator to check for *REST* tokens during a cache fill. The software infrastructure to provide memory safety with *REST* reuses a production-quality memory error detection tool, AddressSanitizer, by changing less than 1.5K lines of code.

***REST* based memory safety offers several advantages compared to extant methods: (1) it does not require significant redesign of hardware or software, (2) the overhead of heap and stack safety is 2% compared to 40% for AddressSanitizer, (3) the security of the memory safety implementation is improved compared AddressSanitizer, and (4) *REST* based memory safety can mitigate heap safety errors in legacy binaries without recompilation or source code. These advantages provide a significant step towards continuous runtime memory safety monitoring and mitigation for legacy and new binaries.**

***Keywords*-memory safety, hardware support, REST, Random Embedded Secret Tokens, AddressSanitizer, privileged memory safety exception, microarchitecture, load store queue, cache microarchitecture.**

space (for instance, both ends of an array) are marked invalid and any access to them raises a memory violation exception.

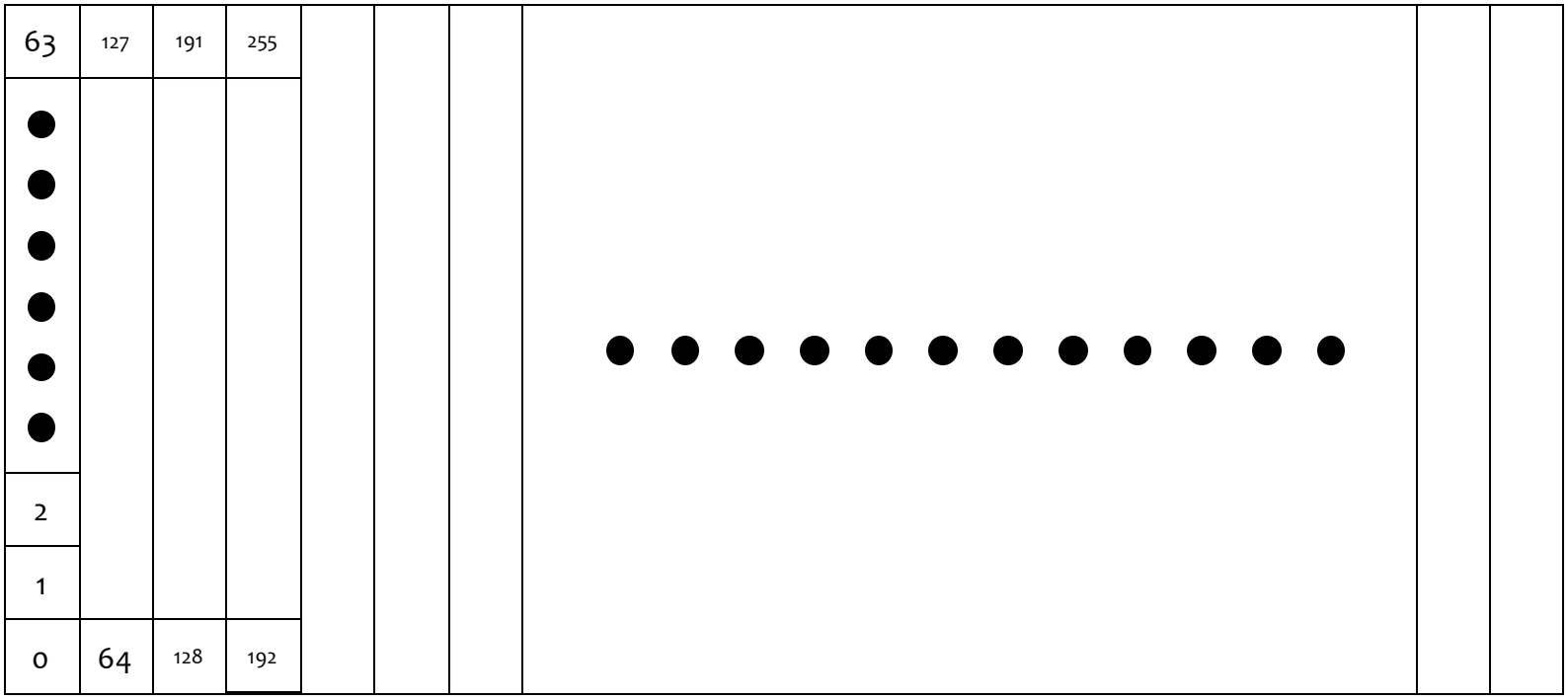
Whitelisting approaches [2], [3], [4], [5], [6], [7] offer stronger security guarantees since they monitor all memory accesses against exact bounds. Another advantage to per-pointer metadata is that some of these mechanisms also maintain liveness/version information about data structures they point to, thus detecting dangling pointers in addition to out-of-bound errors. However, they suffer from one or more of the following problems.

① **Performance Overhead.** Since they monitor every pointer dereference, the performance overhead scales with the number of dynamic pointer references. For each of these references there is at least one additional memory instruction for loading the meta data and one comparison operation for checking the data. Even if some overhead can be mitigated by optimizations such as caching, the energy overheads due to the additional instructions are not easily mitigated.

② **Implementation Overhead.** They usually require significant hardware modifications including modifications to the cache hierarchy [2], [4], execution pipeline [2], [4], [7], or even addition of coprocessors [6].

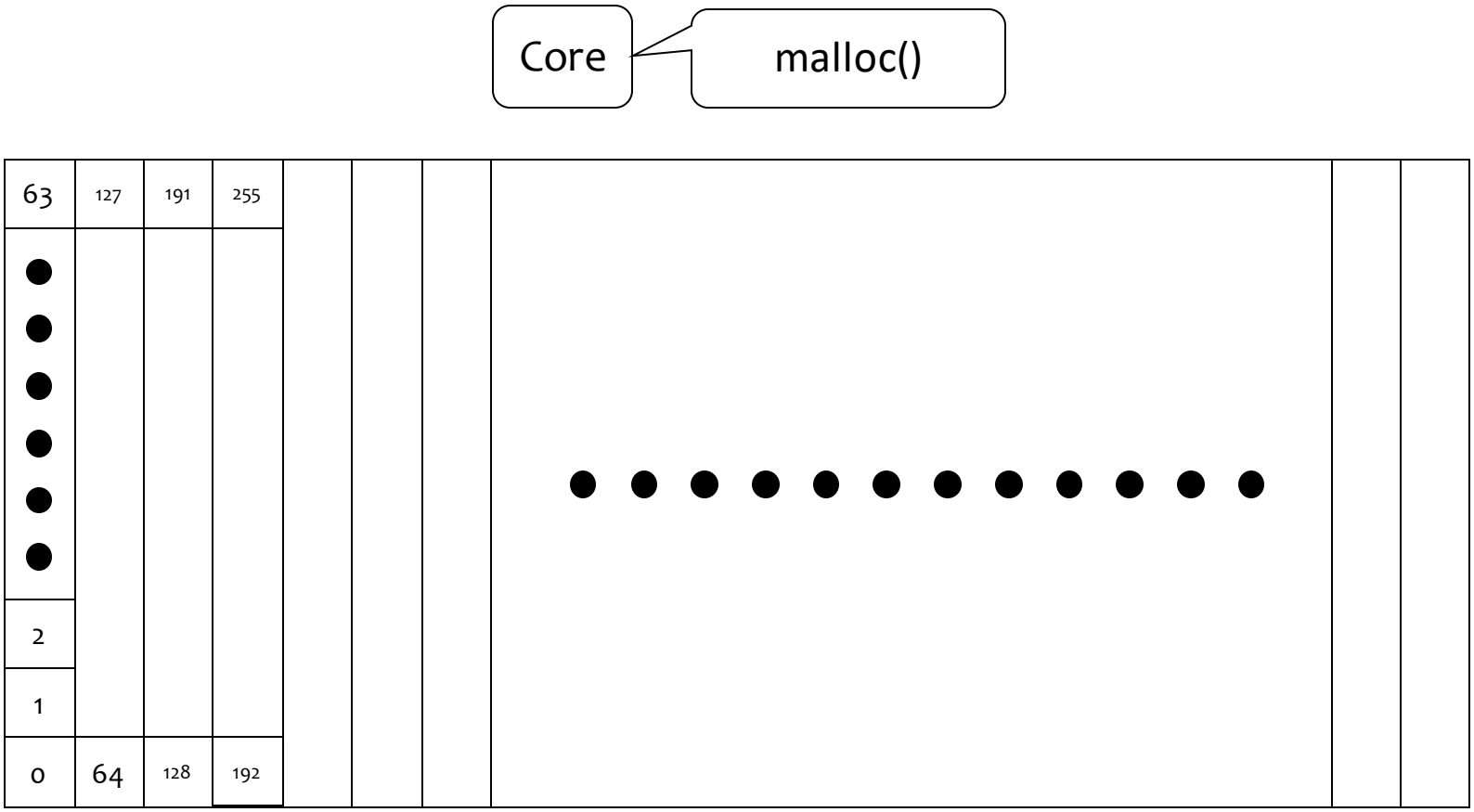
③ **Inaccurate/incomplete Coverage.** Since most of them rely on static pointer analyses for metadata propagation during pointer operations, any inaccuracy in pointer identification leads to incorrect/unstable program behavior. This is espe-

REST and Califorms

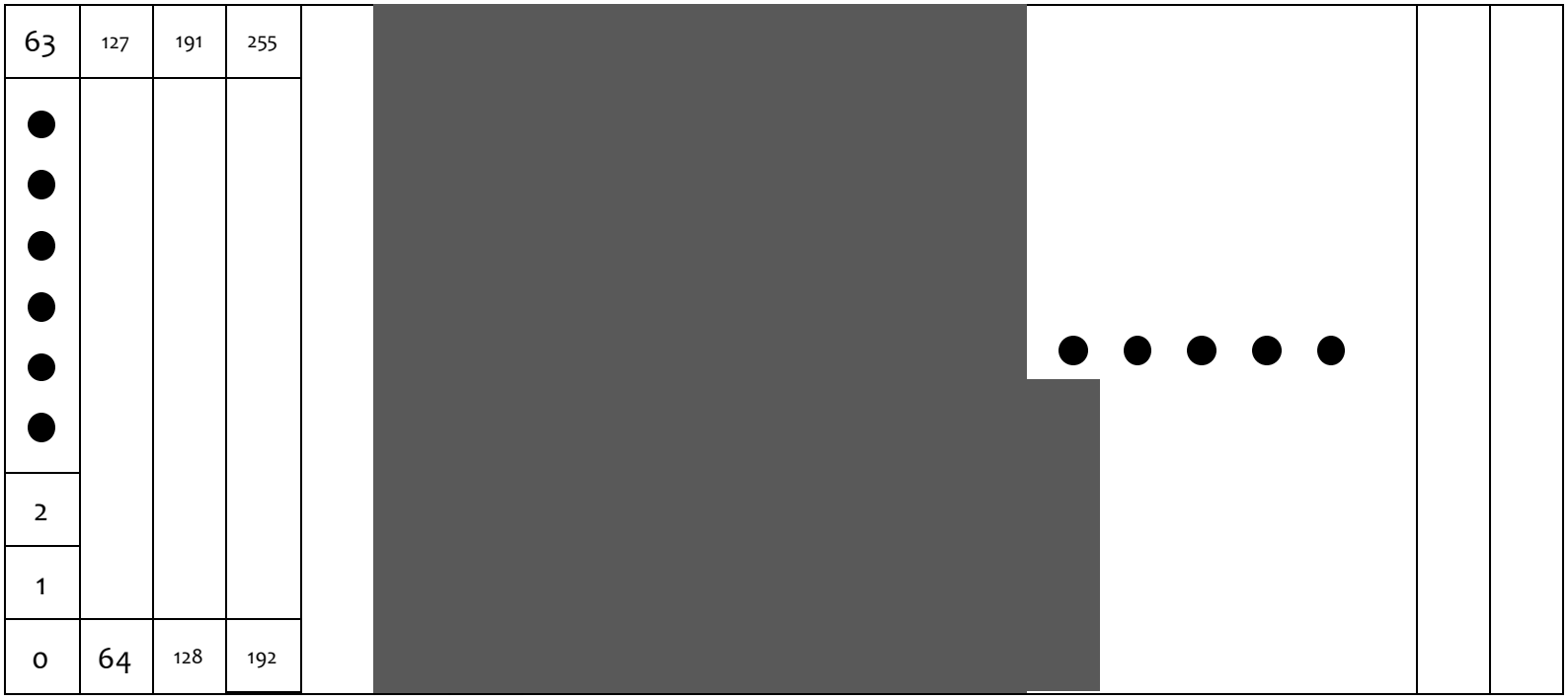


Memory

REST and Califorms



REST and Califorms



REST and Califorms

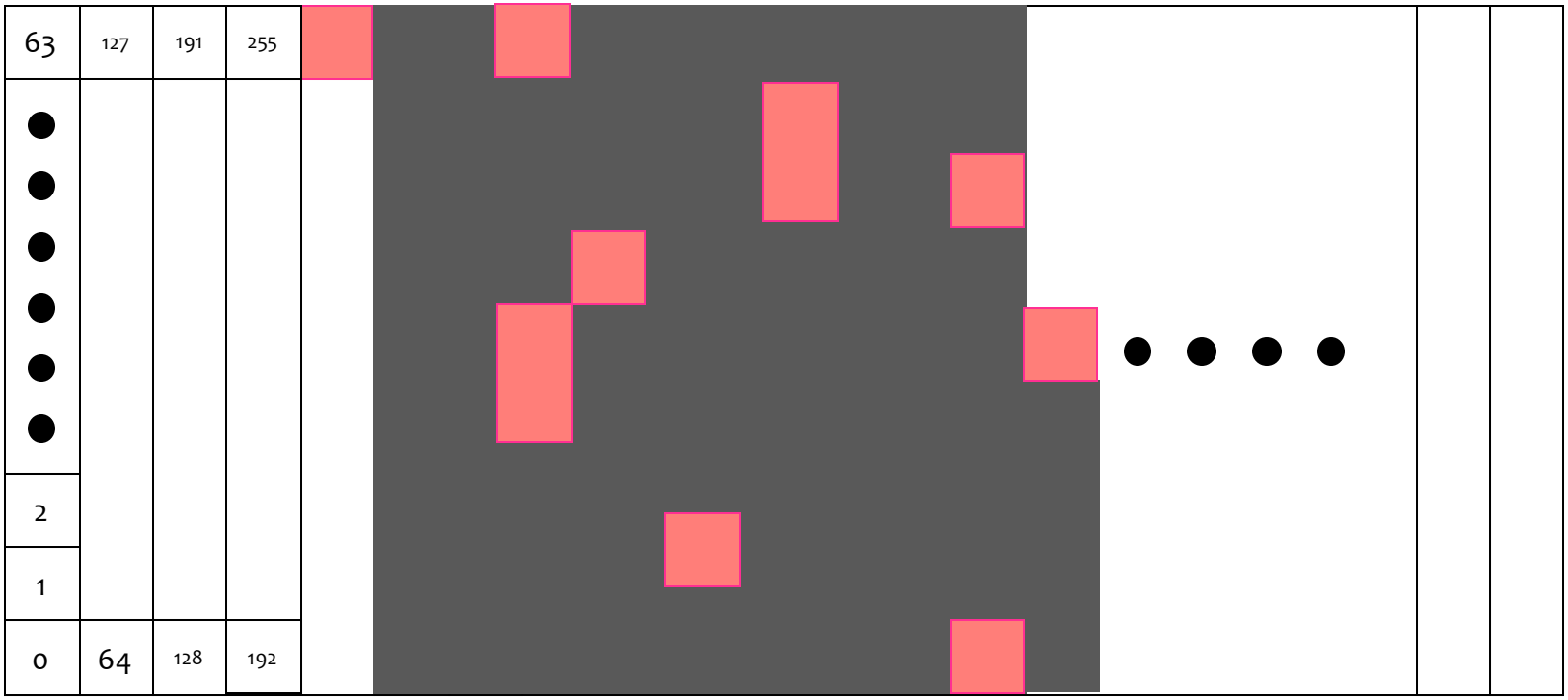
REST inserts 64B tokens at both ends

REST and Califorms

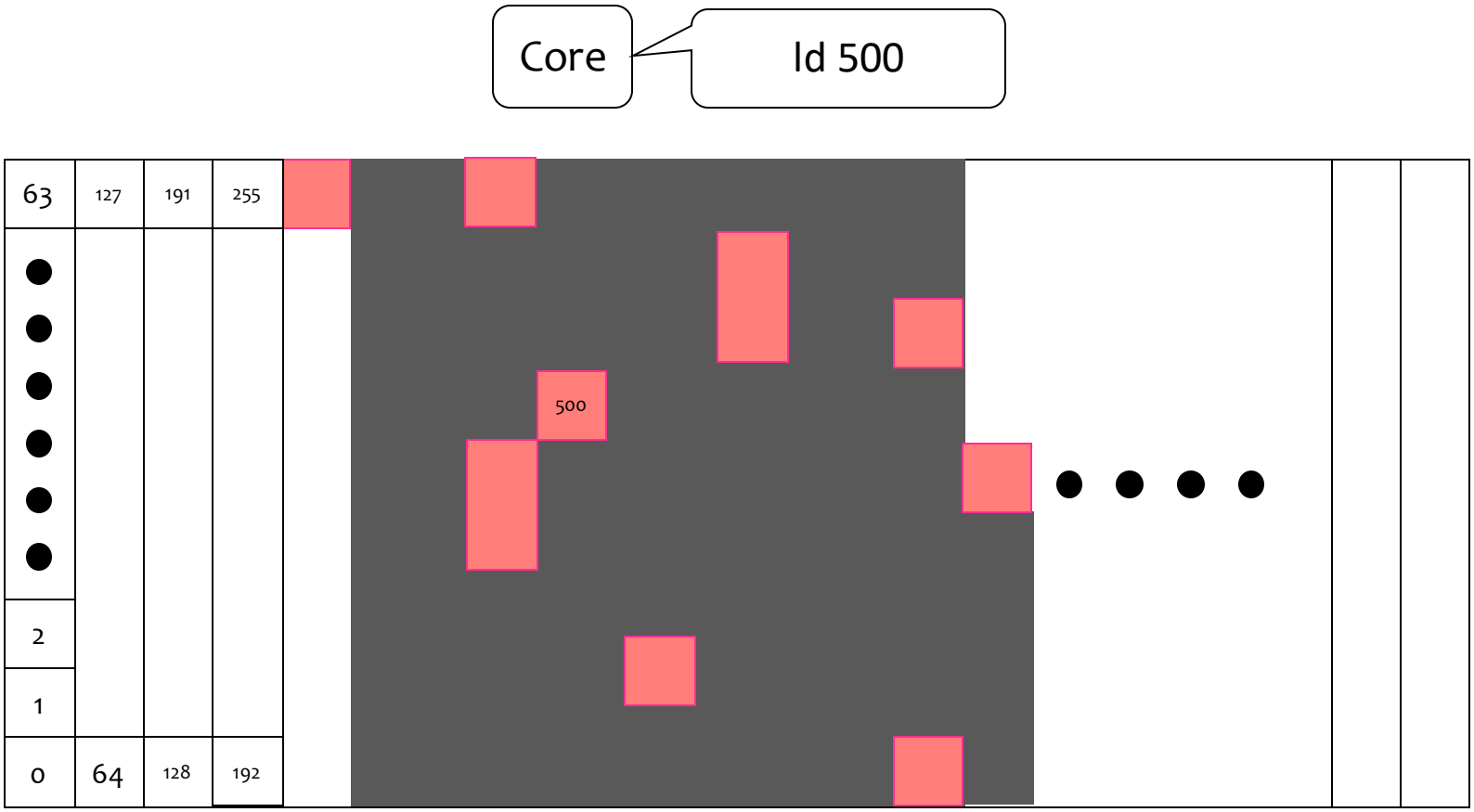
Califorms can do this instead

REST and Califorms

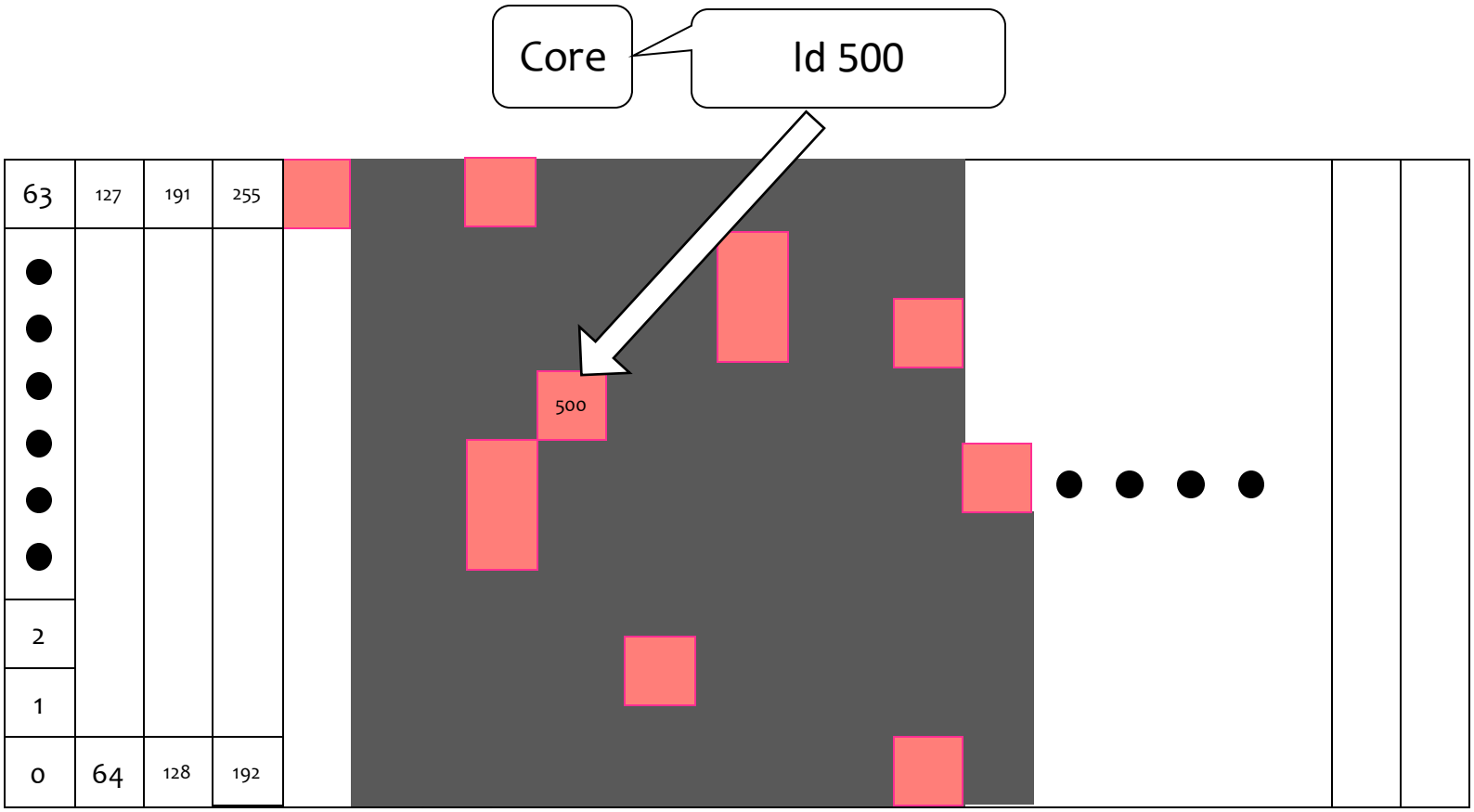
And even more!



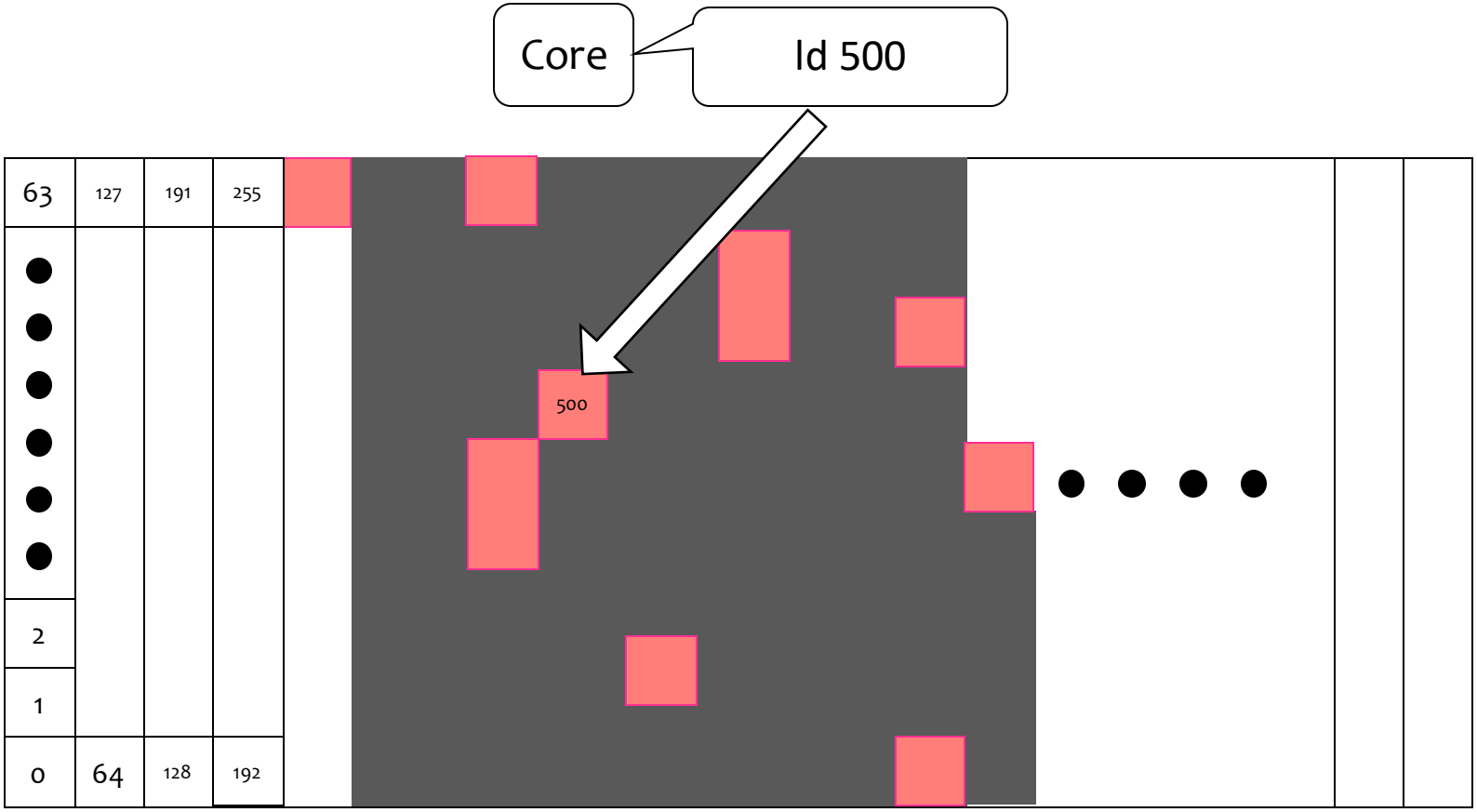
REST and Califorms



REST and Califorms



REST and Califorms



Access triggers a violation (same as REST)

CALIFORMS

Efficient Metadata Representation

We need 1 bit of storage for each byte to represent blacklisted space.

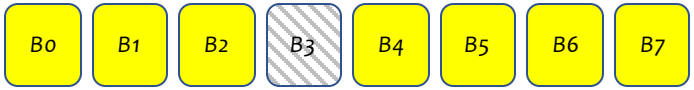
We represent metadata using 1 bit for every 64 bytes using Califorms.

CALIFORMS: A better way to store metadata

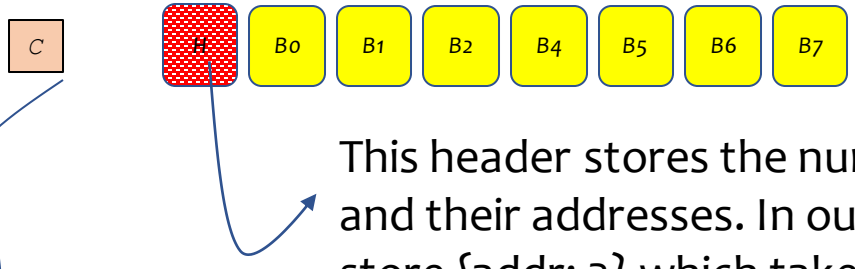
Not just for security!

Example

Consider a cache line of size 8 bytes, say B3 is a security byte.



We can store this information as follows:



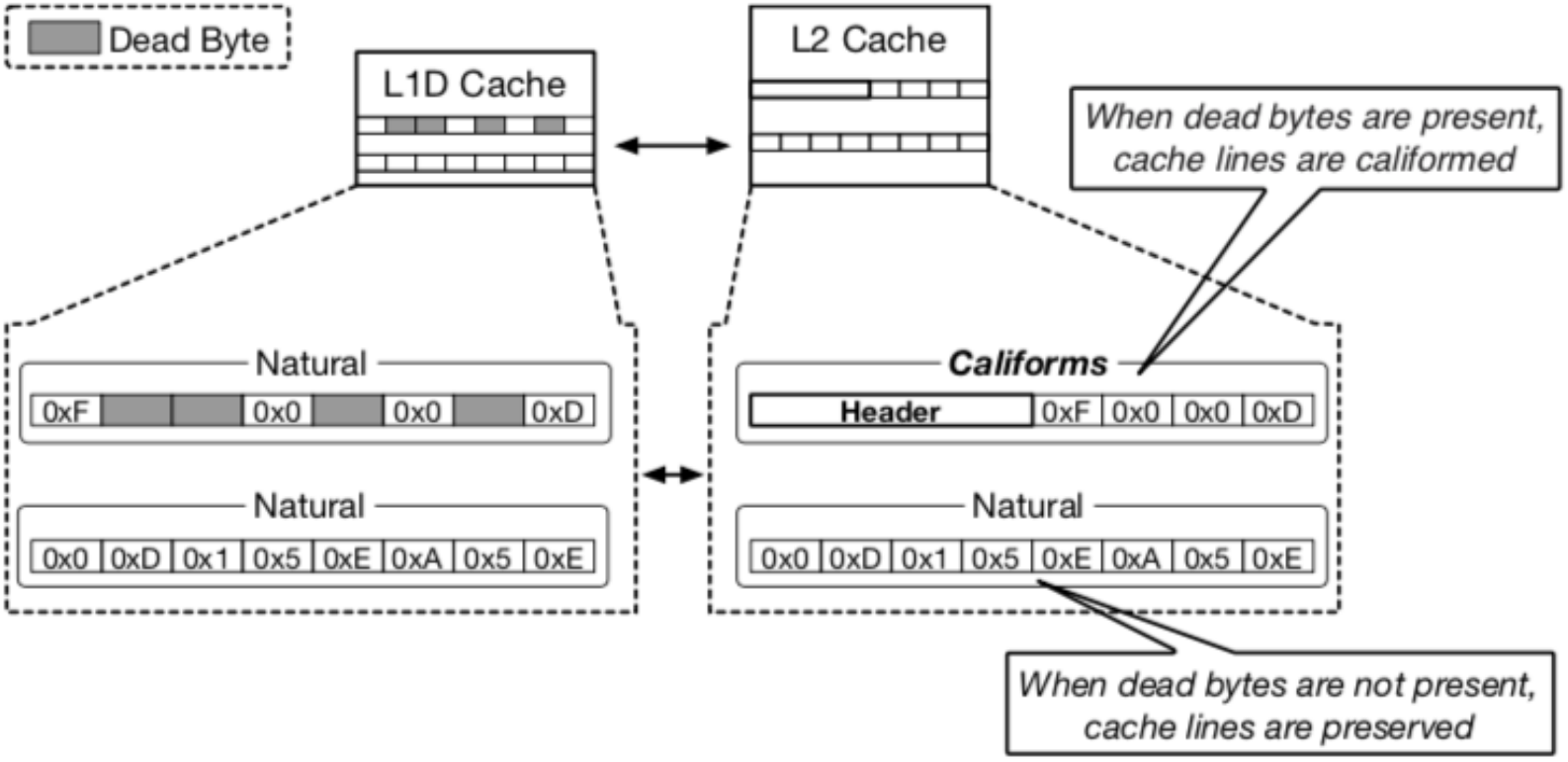
One addl. bit to indicate that format has changed.

This header stores the number of dead bytes and their addresses. In our example, header would store {addr: 3} which takes 3 bits leaving the remaining 5 bits for other uses!

Microarchitecture

Califorms High-Level Idea

CALIFORMS: Cache line formats!



ISA

- Conveying dead bytes happens through the CFORM instruction.

BLOC R1, R2, R3

- R1 points to the starting (or cache) address in the virtual address space, denoting the start of the 64B chunk which fits in a single 64B cache line

- R2 indicates the attributes of said region represented in a bit vector format (1 to set and 0 to unset the security byte).

- The value in register R3 is a mask to the corresponding 64B region, where 1 allows and 0 disallows changing the state of the corresponding byte. The mask is used to perform partial updates of metadata within a cache line

Software Modifications for Califorms

- Compiler
  - Padding insertion in data structures
  - Instrument BLOC instructions
    - Heap (clean-before-use) allocation: remove security bytes from valid objects
    - Stack (dirty-before-use) allocation: add security bytes to paddings
    - Stack deallocation: remove security bytes
  - Emit precise loads/stores
- Memory allocator
  - Issue BLOC instructions
    - Heap deallocation: add security bytes to the entire region
- Operating System
  - Califorms exception handling and whitelisting (e.g., memcpy) support
  - Save/restore Califorms information for 64B chunk of memory when a page is swapped out/in

Evaluation

Software overheads of Califorms

Security bytes insertion policy

```
struct A {
  char c;
  int i;
  char buf[64];
  void (*fp)();
  double d;
}
```

(a) Original

```
struct A_opportunistic {
  char c;
  /* compiler inserts padding
   * bytes for alignment */
  char padding_bytes[3];
  int i;
  char buf[64];
  void (*fp)();
  double d;
}
```

(b) Opportunistic

```
struct A_full {
  /* we protect every field with
   * random security bytes */
  char c;
  char security_bytes[1];
  int i;
  char security_bytes[3];
  char buf[64];
  char security_bytes[2];
  void (*fp)();
  char security_bytes[1];
  double d;
  char security_bytes[2];
}
```

(c) Full

```
struct A_intelligent {
  char c;
  int i;
  /* we protect boundaries
   * of arrays and pointers with
   * random security bytes */
  char security_bytes[3];
  char buf[64];
  char security_bytes[2];
  void (*fp)();
  char security_bytes[3];
  double d;
}
```

(d) Intelligent

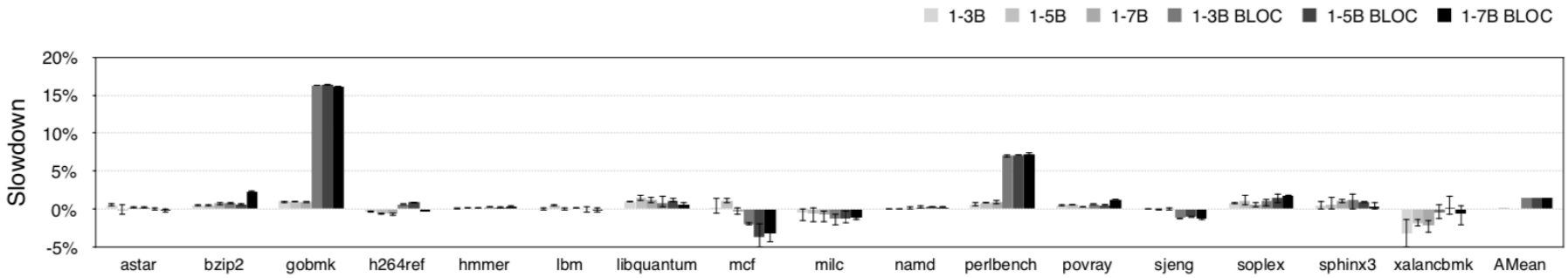


Figure 12: Slowdown of the intelligent insert policy with random sized security bytes (with and without BLOC instructions). The average slowdown is 2.0%.

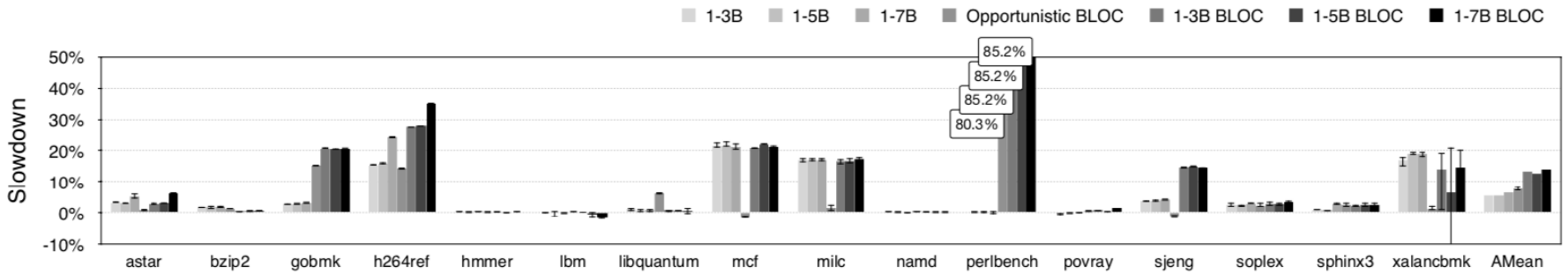
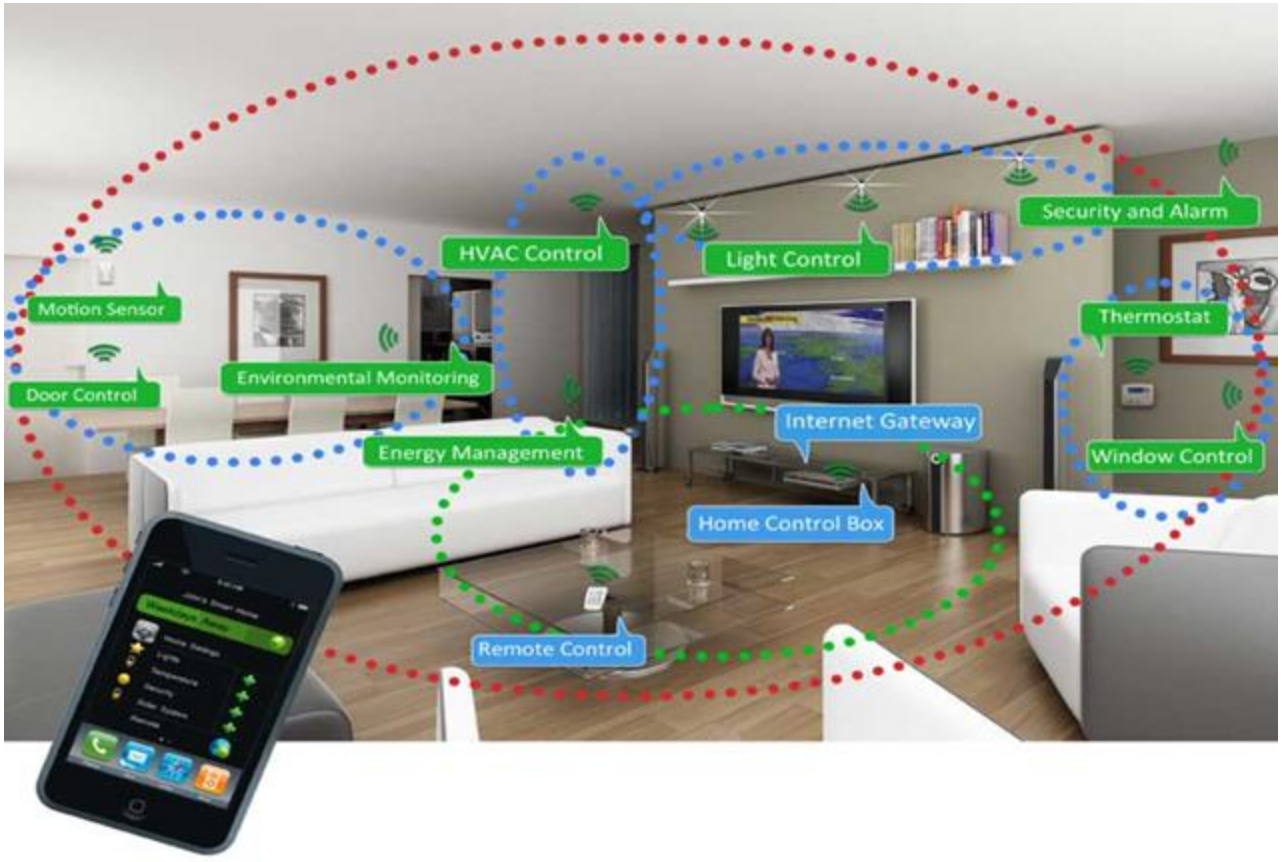


Figure 11: Slowdown of the opportunistic policy, and full insertion policy with random sized security bytes (with and without `BLOC` instructions). The average slowdowns of opportunistic and full insertion policies are 6.2% and 14.2%, respectively.

Post-Moore's-Law Suitability

Context: Things Everywhere



Security reduced to security of the weakest link

Checklist for Post-Moore'-Law

- ✓ Acceptable perf. overhead for continuous in-field deployment
- ✓ Low hardware costs and complexity
- ✓ Must apply to wide range of devices including low-cost devices (non 64-bit systems)
- ✓ Can be easily extended to accelerator based systems

Blacklist vs Whitelist

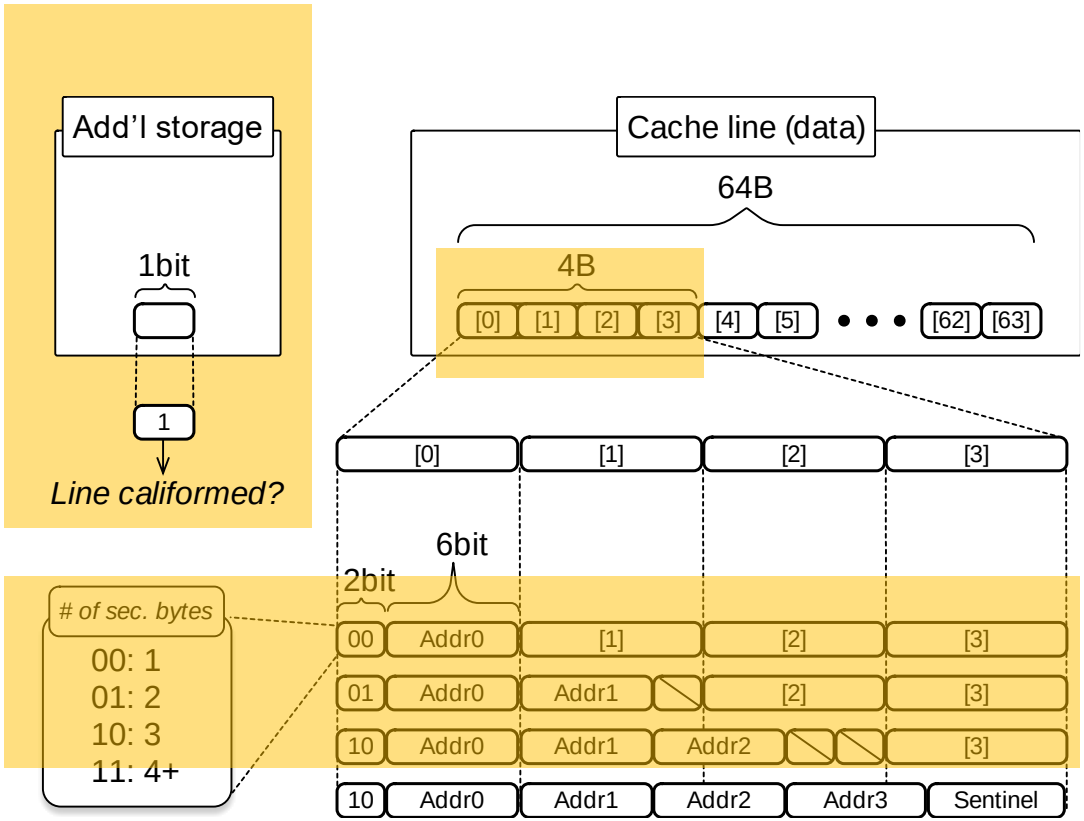
Metric	CALIFORMS	HW-Whitelist (SPARC ADI)
Area	1 bit per DL1: 128 B total for 64KB DL1 (0.0002% for ADI-like config)	4 bits **throughout** the cache hierarchy (0.78% for ADI)
Energy per op	~1 32-bit compare every 50 insts	~1, 4-bit compare every 3 insts
Number of ops	Similar	More
Performance	Similar overhead	Similar overhead
Temporal security	Probabilistic (alias when the heap runs out)	Probabilistic (one is 13 pointers have same color)
Scalability	IoT to Servers; easily extends to accelerator based systems.	Requires 64 bit architecture

- Relies on 64-bit pointers: stores pointer metadata in the top unused bits
- Overheads for the recent ARM memory tagging spec are 4x higher in terms of area (not implemented so far)
- Increasing the tagging capacity will require changes to the DRAM architecture.

Summary

- On paper Whitelists offer better security than Blacklists
- But Blacklists are a much better match for post-Moore's-Law requirements!
  - Low silicon and perf overheads
  - Extends easily to heterogeneous systems
- Research topics:
  - Reduce overheads of whitelists by combining with other techniques?
  - Improve security of blacklists by combining with other techniques?

L2 Caliform



1 bit indicates if the line has a different cache line format.

The header is at most four bytes depending on the number of dead bytes.

The formats for up to 3 dead bytes is straightforward. Note we gain 2 bits for every dead byte.

When we have more than four dead bytes, we store address of the first four, and then store a 6-bit value, a sentinel that denotes other security bytes!