Hardware Support for Low-Cost Memory Safety

Rick Boivie[1], Gururaj Saileshwar[1]*, Tong Chen,
Benjamin Segal, Alper Buyuktosunoglu
IBM Thomas J. Watson Research Center
Yorktown Heights, New York, USA

DSN 2021
The 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks
Taipei, Taiwan, June 21-24, 2021

[1] Equal Contribution
*Gururaj Saileshwar is currently affiliated with Georgia Tech

Executive Summary

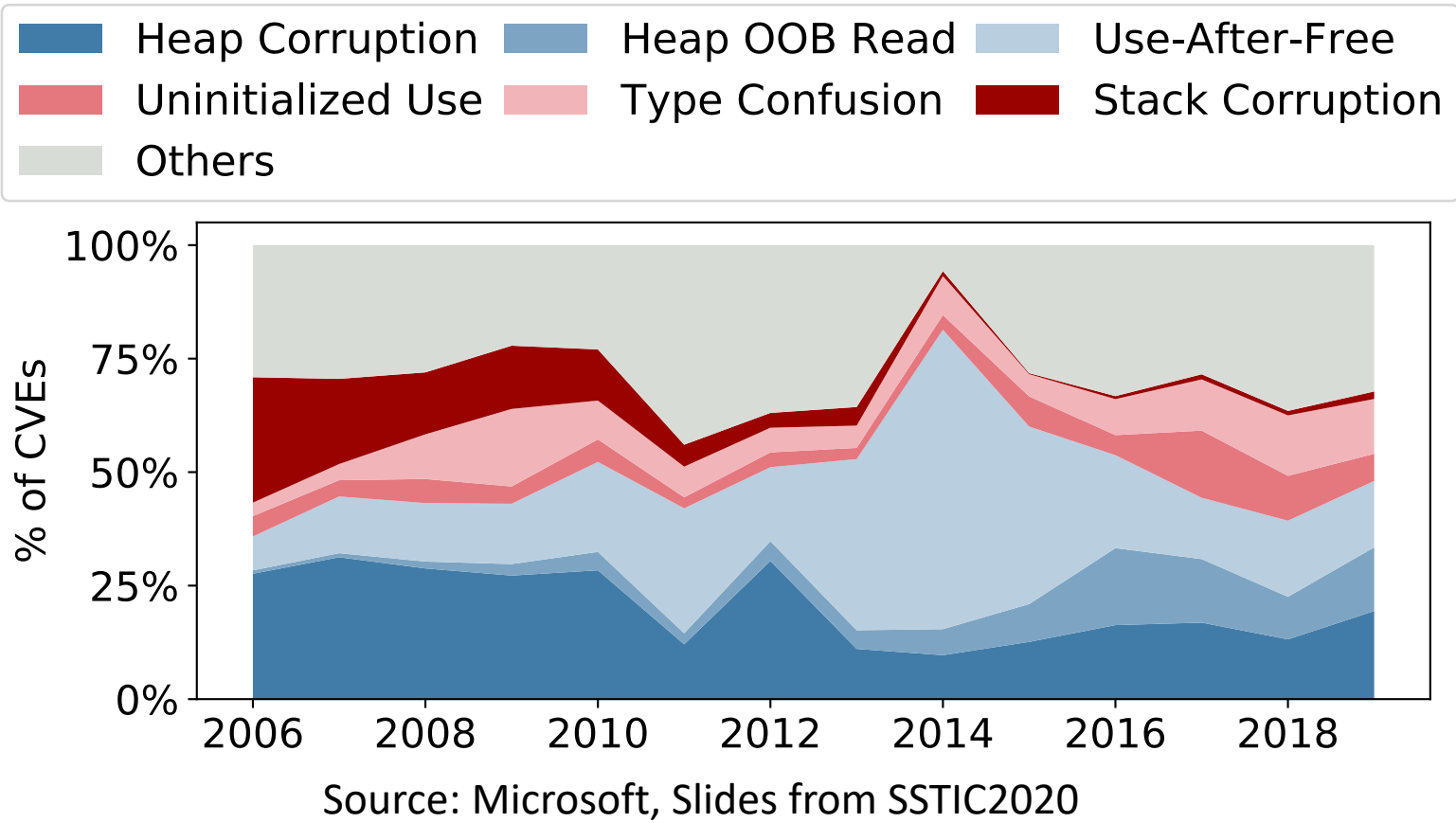
- **Problem**: Memory-Safety bugs like buffer-overflow & use-after-free are a serious problem and have been at the root of many security problems for more than 3 decades
  - E.g. The Morris Worm (1988), Buffer Overflow Based Code Injection, HeartBleed, Return-Oriented Programming (ROP), Spectre … -- and are at the root of ~70% of CVEs
- **Existing solutions** for enforcing memory safety have poor coverage, or high performance-overhead, or require disruptive changes
- **Our Solution:** A HW/SW co-design for precise bounds-checking
  - that is **minimally-invasive**
    - requires **no changes to source-code**
    - and **no changes to binary-layout** (i.e., retains compatibility with existing libraries)
  - with **minimal performance impact (<2% overhead)**
- **Some Results:**
  - Our design detected all the vulnerabilities and prevented all the attacks in the 'How2Heap' Exploit Suite
  - Detected 87 memory-safety bugs in glibc and in the SPEC-CPU 2017 benchmark programs that, to our knowledge, had not been previously detected
    - Our design can detect & prevent bugs in unmodified/un-instrumented shared library code

Problem: Memory-Safety Bugs in C/C++ Programs

Memory Safety Bugs in C/C++

```
arr = (int*) malloc(10 *
                    sizeof(int));

int func(int offset) {

  //offset=11 (Out of bounds)
  y = arr[offset];

  free(arr);
  //offset=9 (Use-After-Free)
  y = arr[offset];

}
```

Root cause of ~70% CVEs in Production-Software



Source: Microsoft, Slides from SSTIC2020

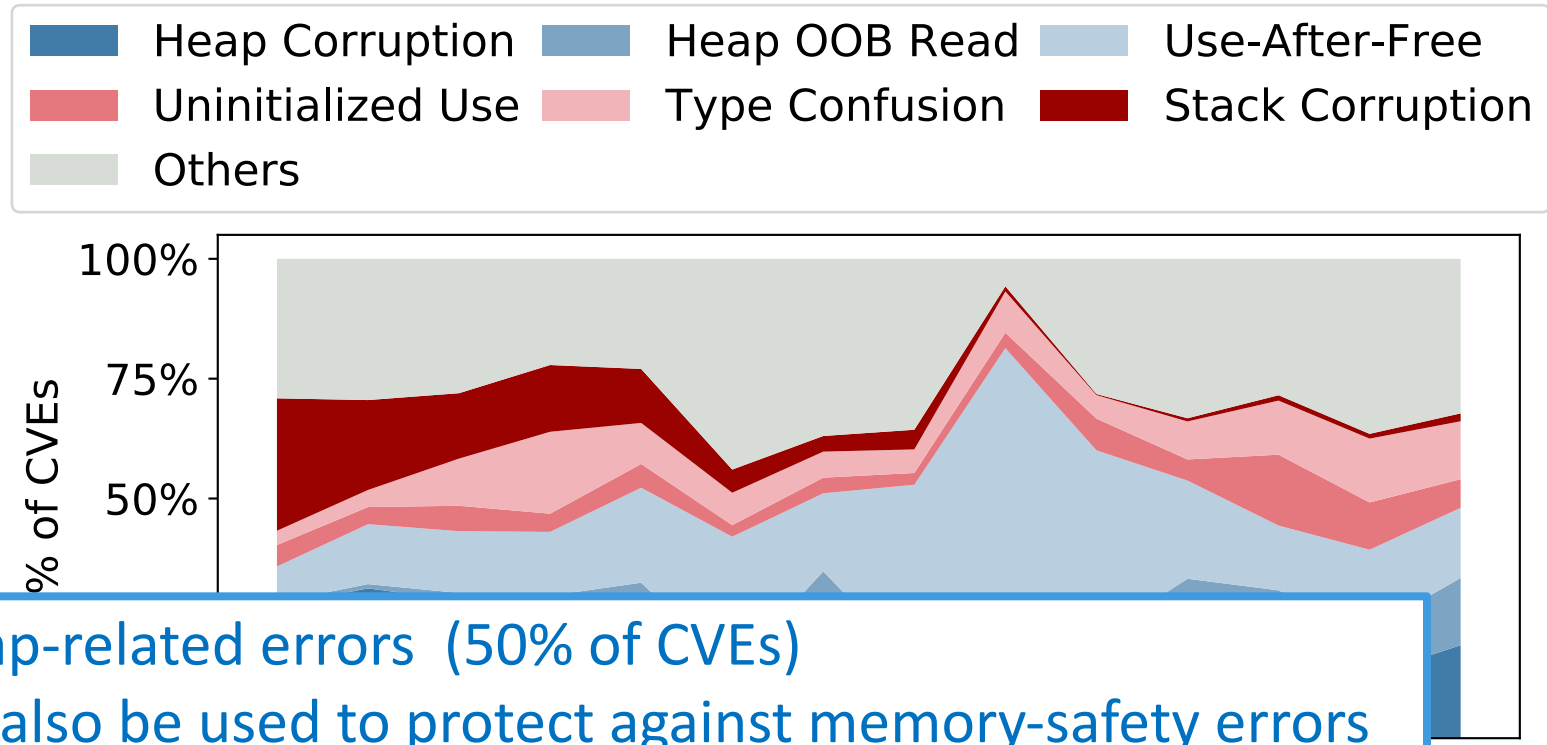
- Memory-safety bugs are the root cause of ~70% of the CVEs in Microsoft's Production S/W
- 50% of the CVEs are related to Heap Objects
- A Google study showed that 60% of memory-safety errors in Google S/W were Heap related

Problem: Memory-Safety Bugs in C/C++ Programs

Root cause of ~70% CVEs in Production-Software

Memory Safety Bugs in C/C++

```
arr = (int*) malloc(10 *
                    sizeof(int));

int func(int offset) {

  //offset=11 (Out of bounds)
  y = arr[offset];

  free(arr);
  //of
  y =

}
```



- Our initial focus is Heap-related errors (50% of CVEs)
- But our approach can also be used to protect against memory-safety errors for Stack and Global Objects as well

- Memory-safety bugs are the root cause of ~70% of the CVEs in Microsoft's Production S/W
- 50% of the CVEs are related to Heap Objects
- A Google study showed that 60% of memory-safety errors in Google S/W were Heap related

Protecting Against Buffer-Overflow Based Attacks

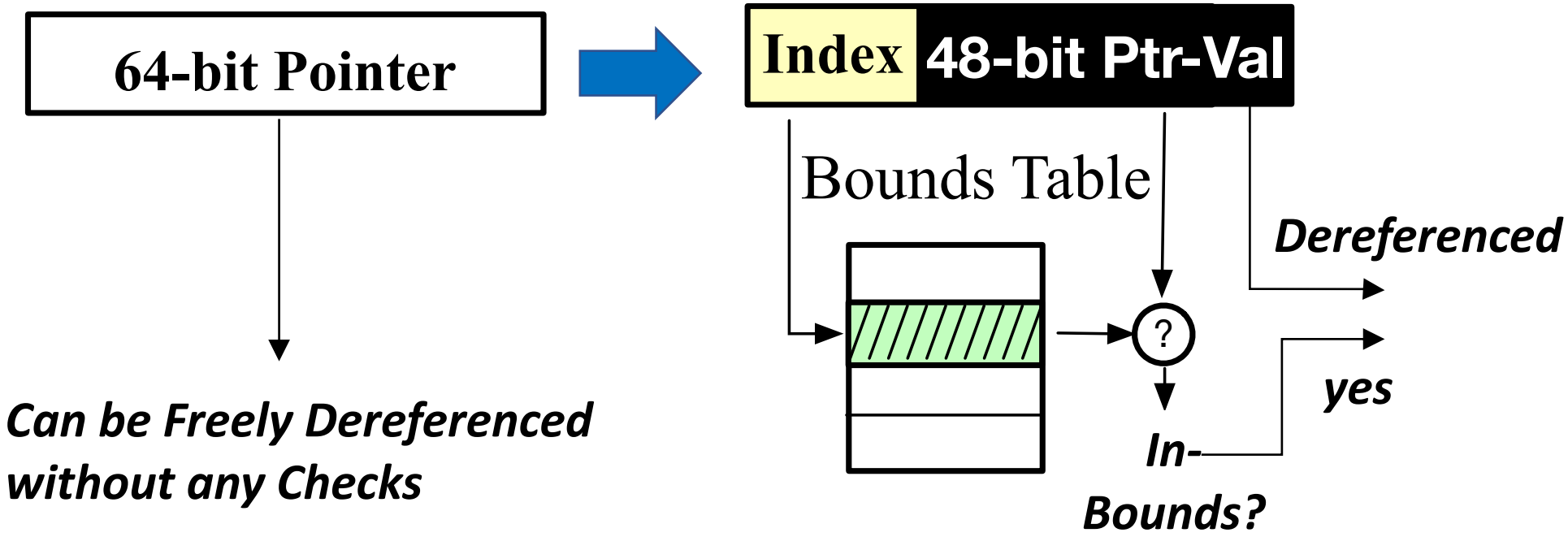
- Protecting against *some* buffer-overflow based attacks
  - Stack Canaries
  - Data Execution Protection (W XOR X memory pages)
  - Address Space Layout Randomization (ASLR)
  - Shadow Stacks & other mechanisms for protecting “control flow integrity”
- These address some specific attacks but don’t *prevent* out-of-bounds references

- Eliminating out-of-bounds references
  - Intel MPX. Uses extra instructions
    - to keep a mapping from each pointer to its bound information
    - to check bounds information on loads and stores
  - Oracle Silicon Secured Memory
    - “Colors” data in memory and prevents access by pointers of ”the wrong color”
      - Requires coloring and re-coloring of memory
    - Only supports a limited number of colors
    - Granularity of protection is a cache-line

Other Mechanisms to Detect/Prevent Memory Safety Bugs

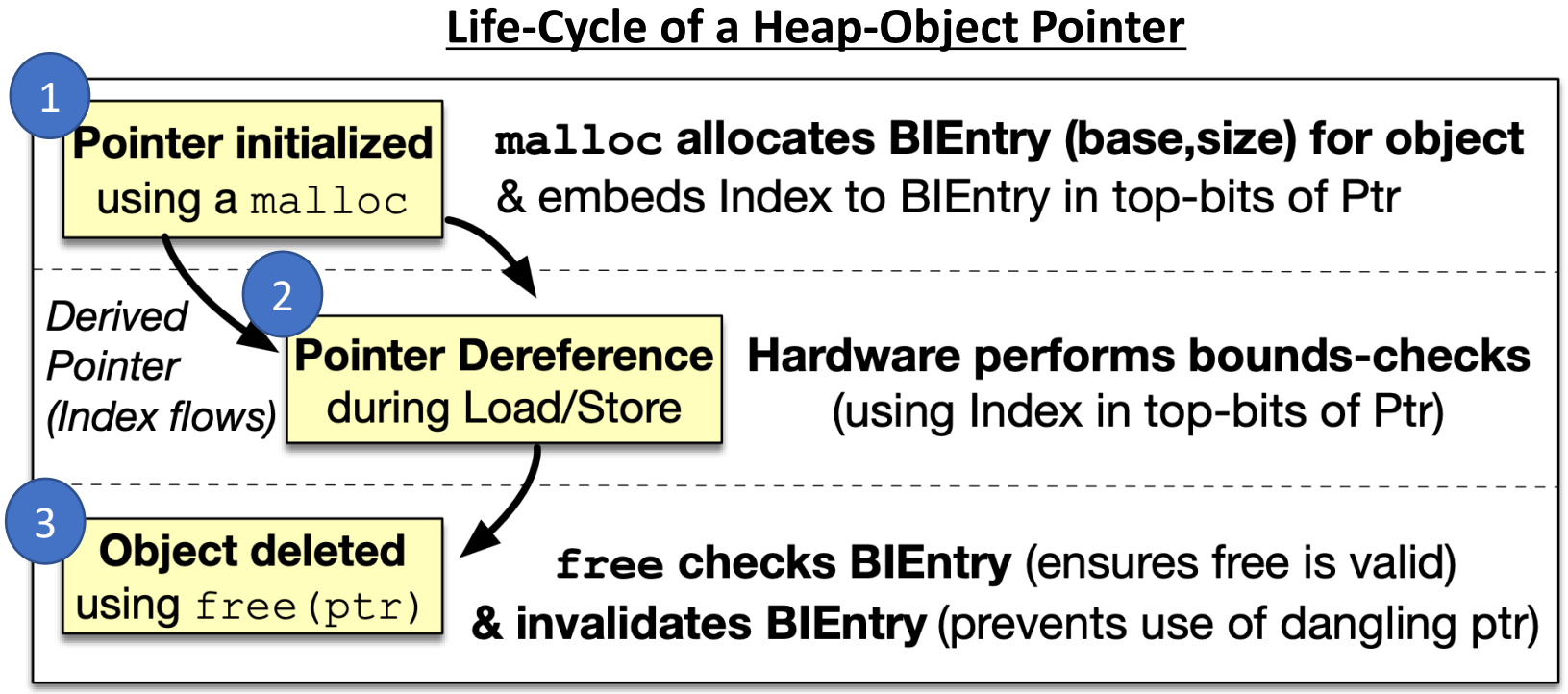
Prior Solutions	Coverage	Performance Overheads	Level of Invasiveness
Google's Address Sanitizer (industry standard SW tool)	**Only Detects small errors** 	**High** (70%) 	**No Changes to Source-Code / Binary Layout** 
CHERI [ISCA-2014, S&P-2015] (HW/SW framework from academia + SRI)	**Precise Bounds Protection** 	**Medium** (20-30%) (leverages HW-caching) 	**Changes to Binary Layout** 
Our H/W Support for Low-Cost Memory Safety	**Precise Bounds Protection** 	**Low** 	**Low** 

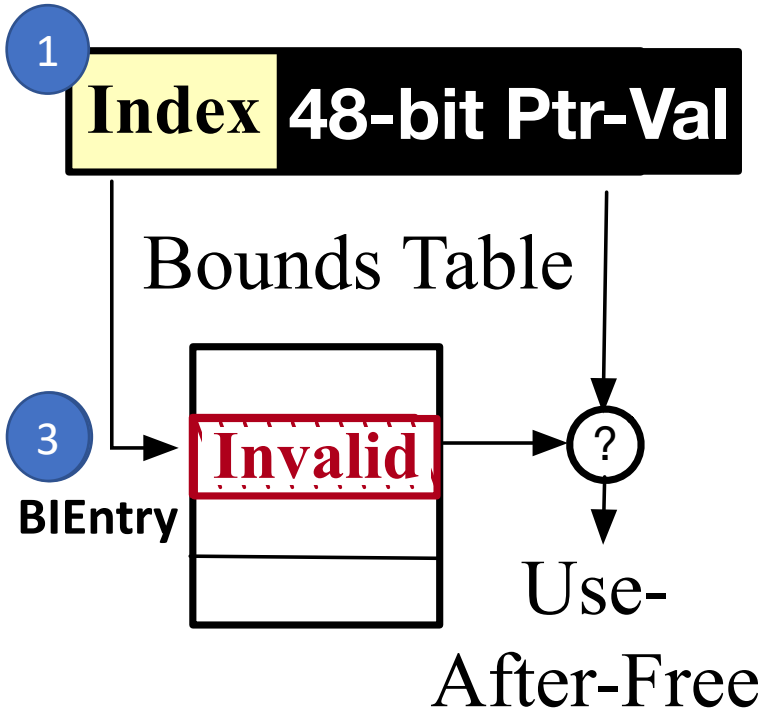
High-Level Idea: HW-assisted Bounds-Checking on Loads and Stores using Pointers



Re-Use the Top-Bits of the Pointer to Store Metadata;
Enforce Bounds-Checks Transparently in Hardware

Formalizing the Bounds-Checking for Heap-Objects

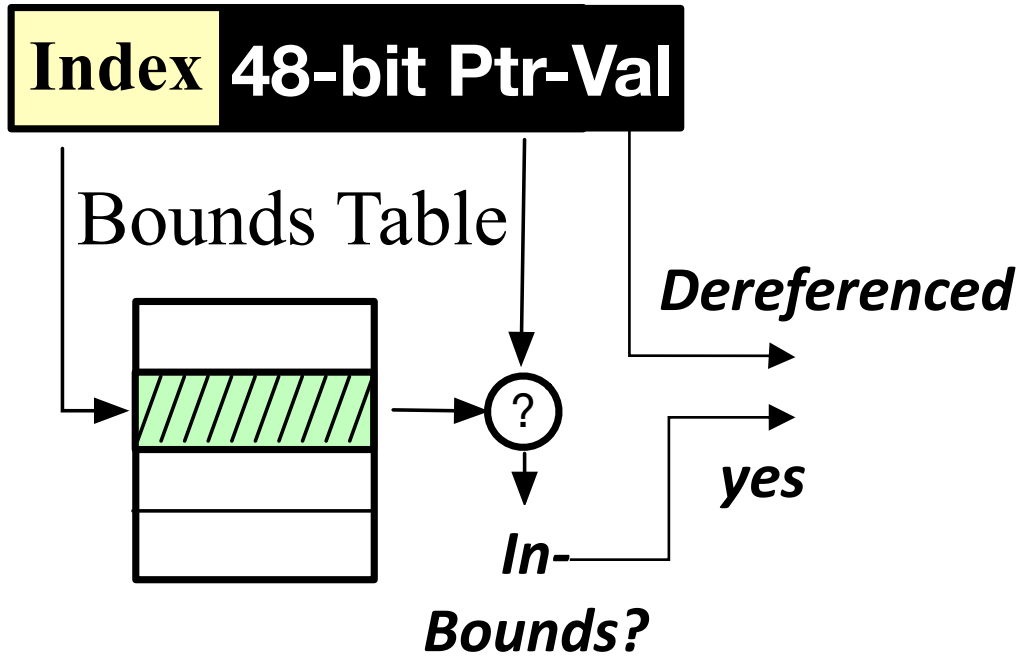




High-Level Idea: HW-assisted Bounds-Checking on Loads and Stores using Pointers

Minimal performance impact since:

- Bounds information flows "automatically" without extra instructions when a pointer is assigned to another, passed in a function call or used to compute another address in array indexing or pointer arithmetic, and
- Since all the addresses associated with a given buffer have the same "index", the bounds information for an address is often available in an on-chip "Bounds Information" cache

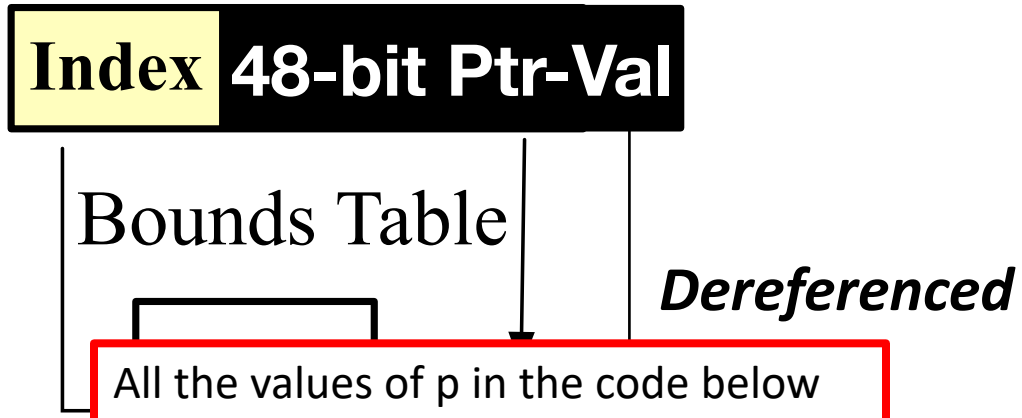


Re-Use the Top-Bits of the Pointer to Store Metadata;
Enforce Bounds-Checks Transparently in Hardware

High-Level Idea: HW-assisted Bounds-Checking on Loads and Stores using Pointers

Minimal performance impact since:

- Bounds information flows "automatically" without extra instructions when a pointer is assigned to another, passed in a function call or used to compute another address in array indexing or pointer arithmetic, and
- Since all the addresses associated with a given buffer have the same "index", the bounds information for an address is often available in an on-chip "Bounds Information" cache



All the values of p in the code below "automatically" have the right bounds information -- which is usually available in the on-chip BI cache.

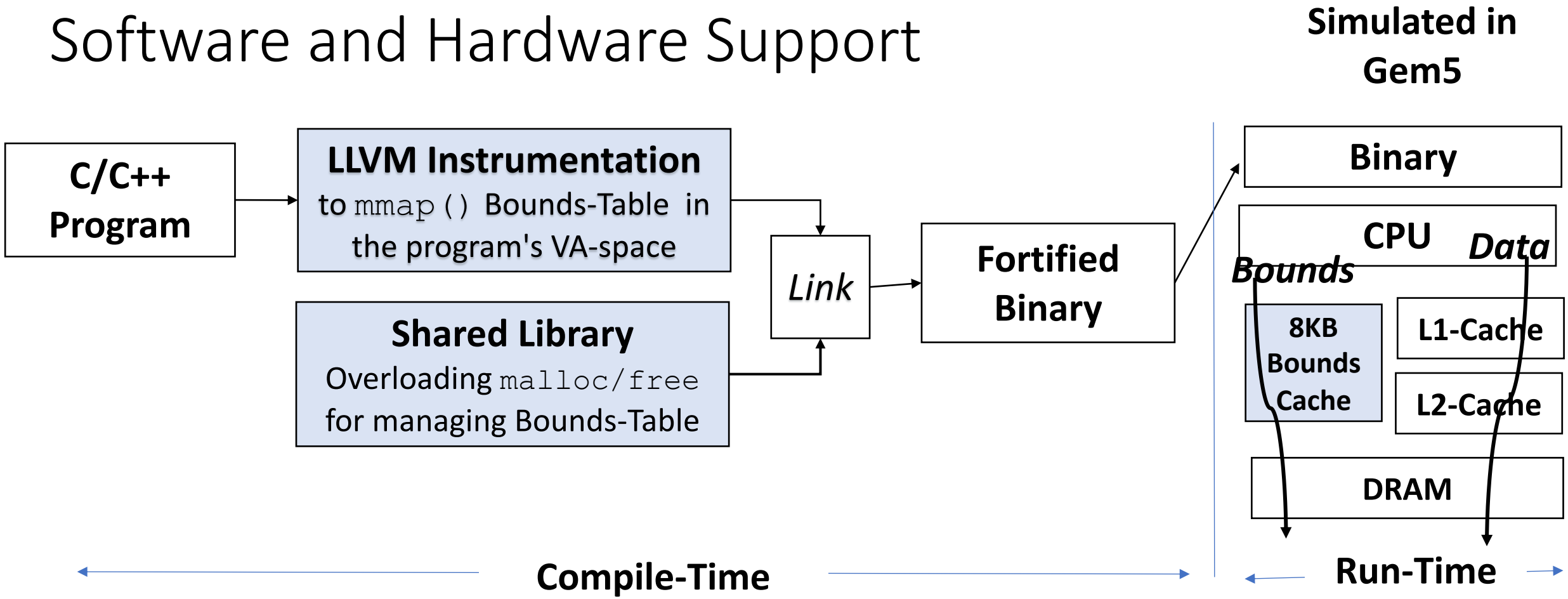
Similarly for all the addresses a[i]

```
while (p++){
        …
}
```

```
For (i=0;  i<…;  i++){
        a[i] = …
}
```

Re-Use the Top-Bits of the Poin… ada…
Enforce Bounds-Checks Trans… wa…

Software and Hardware Support



- **Minimally invasive** – Shared-library that needs to be linked
- **Precise Enforcement of Exact Object Bounds** (wherever pointer is passed, even shared-libraries)
- **Low-Overhead Checks** (If Bounds Hits in Bounds-Cache, No Slowdown)

Security Results (1): Detection of Heap-Based Memory Safety Bugs

- How2Heap Exploit Suite Consists of 25 Heap Exploits
  - Some exploits developed for CTF (Capture-the-Flag) competitions

- Our design was able to detect the Bugs in all 25 programs and raise an exception to terminate the program before the objective of the exploit could be achieved
  - **Out-Of-Bounds Accesses** (load/store access falls outside BIEntry bounds) (8 exploits)
  - **Use-After-Free** (invalid bounds read from BIEntry) (10 exploits)
  - **Invalid-Free** (free called with ptr-value not matching BIEntry ptr-value)
  - **Double-Free** (free called for pointer whose BIEntry has invalid bounds)

  } 7 exploits

Security Results (2) – Detected Several (Previously Undetected) Weaknesses in SPEC CPU-2017 + GLIBC-v2.27

- 87 lines of code with out-of-bounds accesses
  - In Glibc-v2.27 functions and in some SPEC-CPU2017 functions compiled with O3 flag
  - SPEC-CPU2017 out-of-bounds go away when compiled with O0 compilation flag
- All of these memory safety errors were due to aggressive use of SIMD instructions (e.g., 16B load near the boundary of an object in strlen)

Unlike some other schemes, e.g., Address Sanitizer, our design can detect and prevent bugs in unmodified / un-instrumented libraries like the string functions in glibc

Security Results (2) – Example Out-Of-Bounds access in strlen with `pcmpeqb`

Assembly in libc

(branch after each load)

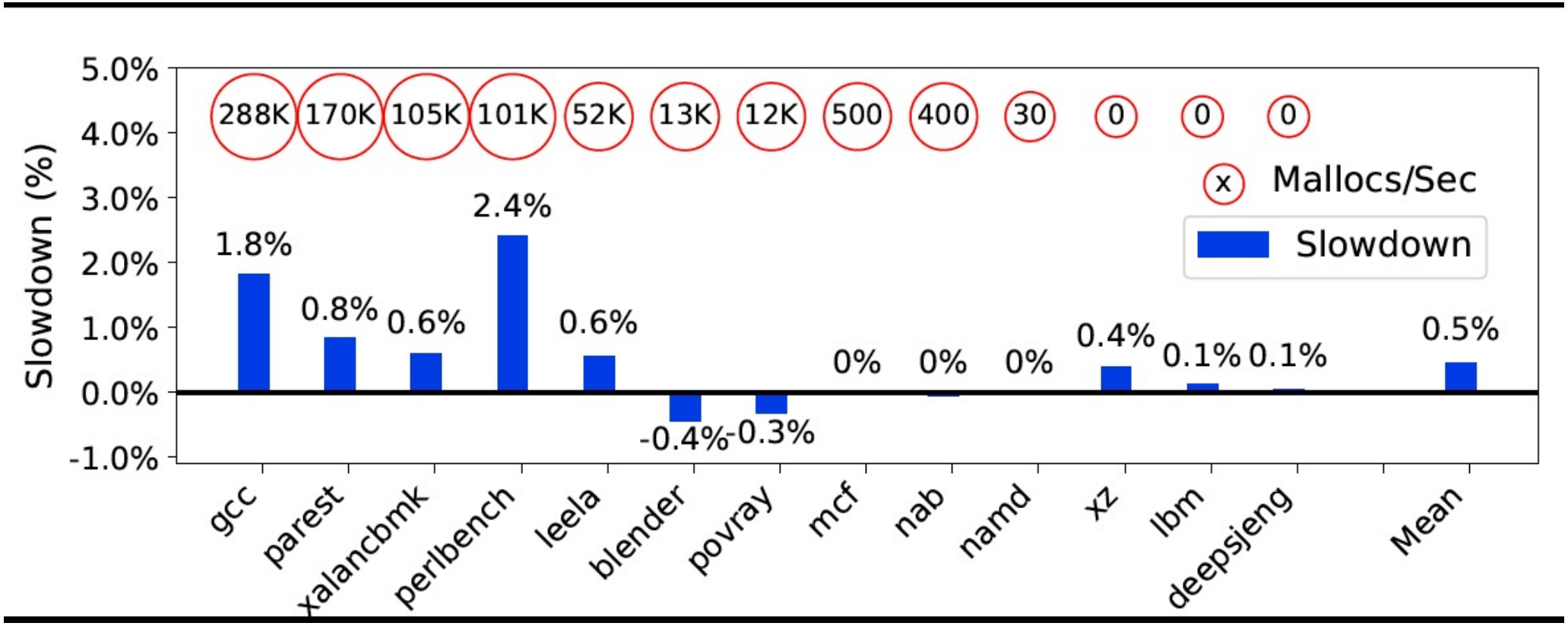
```
pcmpeqb (%eax), %xmm1
pmovmskb %xmm1, %edx
pxor    %xmm2, %xmm2
test    %edx, %edx
lea     16(%eax), %eax
jnz     L(exit)

pcmpeqb (%eax), %xmm2
pmovmskb %xmm2, %edx
pxor    %xmm3, %xmm3
test    %edx, %edx
lea     16(%eax), %eax
jnz     L(exit)

pcmpeqb (%eax), %xmm3
pmovmskb %xmm3, %edx
test    %edx, %edx
lea     16(%eax), %eax
jnz     L(exit)
```

Object Dump of libc.a for strlen function

Several 16B loads done with `pcmpeqb` (to check `'\0'`) before actual conditional check executed with `test`

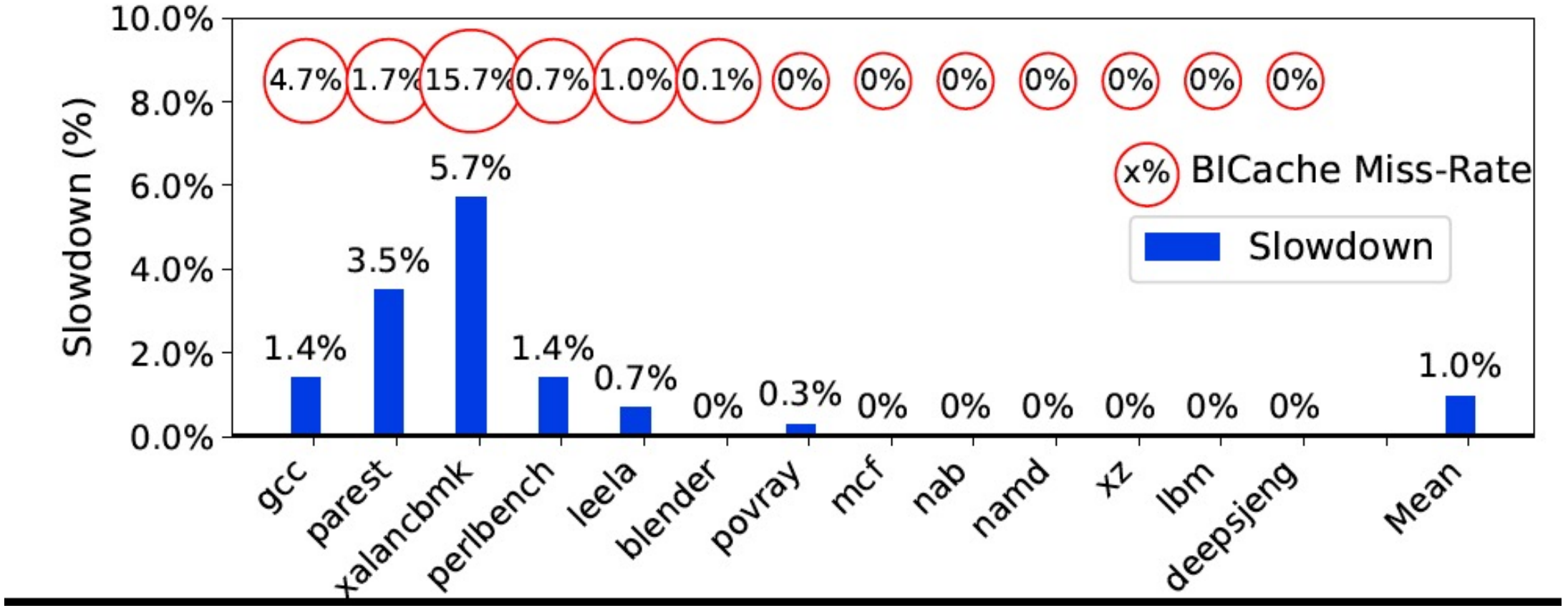
```
3e:   66 0f 74 48 10          pcmpeqb 0x10(%rax),%xmm1
43:   66 0f 74 50 20          pcmpeqb 0x20(%rax),%xmm2
48:   66 0f 74 58 30          pcmpeqb 0x30(%rax),%xmm3
4d:   66 0f d7 d1             pmovmskb %xmm1,%edx
51:   66 44 0f d7 c2          pmovmskb %xmm2,%r8d
56:   66 0f d7 cb             pmovmskb %xmm3,%ecx
5a:   48 c1 e2 10             shl    $0x10,%rdx
5e:   48 c1 e1 10             shl    $0x10,%rcx
62:   4c 09 c1                or     %r8,%rcx
65:   48 c1 e1 20             shl    $0x20,%rcx
69:   48 09 ca                or     %rcx,%rdx
6c:   48 89 f9                mov    %rdi,%rcx
6f:   48 31 c1                xor    %rax,%rcx
72:   48 83 e0 c0             and    $0xffffffffffffffc0,%rax
76:   48 d3 fa                sar    %cl,%rdx
79:   48 85 d2                test   %rdx,%rdx
7c:   0f 84 7e 00 00 00       je     100 <__strlen_sse2+0x100>
```

Performance Impact of Software (for Bounds Table Management)



On average, performance impact of Bounds Table Management is 0.5%

Performance Impact of Hardware (for Bounds Checks on load & store instructions)



On average, performance impact of Bounds Checking on loads and stores is 1.0%

Summary

- Memory-Safety bugs like buffer-overflow & use-after-free are a serious problem and have been at the root of many security problems for more than 3 decades
- Existing solutions for enforcing memory safety have poor coverage, or high performance-overhead, or require disruptive changes
- Our HW/SW co-design for precise bounds-checking
  - is **minimally-invasive**
    - requires **no changes to source-code**
    - and **no changes to binary-layout** (i.e., retains compatibility with existing libraries)
  - with **minimal performance impact (<2% overhead)**
- Our design is effective
  - Our design detected all the vulnerabilities and prevented all the attacks in the 'How2Heap' Exploit Suite
  - Detected 87 memory-safety bugs in glibc and in the SPEC-CPU 2017 benchmark programs that, to our knowledge, had not been previously detected
    - Our design can detect & prevent bugs in unmodified/un-instrumented libraries

Thank you