CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science

George Washington University



A3: Exploiting Format String Vulnerabilities

```
char goodPassword() {
  int good='N';
  int *p = &good;
  fgets(Password, sizeof(Password), stdin); // Get input from keyboard

  printf("Password=");
  printf(Password);
  printf("\n");

  return (char)(*p);
}
```

Format String Attacks

- What about the following simple program for echoing user input?

```
int main(int argc, char *argv[]) {
    if (argc > 1) {
        printf(argv[1]);
    }
}
```

- Appears to be normal
- However, what would happen if the input is “`hello%d%d%d%d%d%d%d`”?
  - i.e. `printf(“hello%d%d%d%d%d%d%d”);`
  - It would print numbers from five registers *and* the stack.
    - Allows attackers to peak unintended data **confidentiality vulnerability**
- What if `arg[1]` is “`hello%s`”?
  - Likely a segmentation fault **availability vulnerability**

x86-64/AMD64 Calling Convention



How functions/subroutines pass arguments and return values at the macro-architecture level.

```
void foo() {
    ...
    bar(a, b, c, d, e, f, g, h);
    ...
}

long bar(long a, long b, long c, long d,
         long e, long f, long g, long h) {
   long xx = a * b * c * d * e * f * g * h;
   long yy = a + b + c + d + e + f + g + h;
   long zz = utilfunc(xx, yy, xx % yy);
   return zz + 20;
}
```

- Where to put all the arguments?
- Where to put the return value?

- Arguments are passed
  - in registers: rdi, rsi, rdx, rcx, r8, r9
  - then via stack
- Return value is passed via
  - in registers: rax, rdx
  - then via stack

← Blog

PHP2Shell: AI uncovers critical RCE vulnerabilities affecting 70% of the Internet

Core PHP engine flaws enable remote code execution and Heartbleed-style memory disclosure across the web.

PHP2Shell: What you need to know | PHP2Shell: Technical Threat Briefing w/ Wiz Research

Sagi Tzadik, Shir Tamari, Nir Ohfeld, Tomer Schwartz

October 21, 2026 | 10 minute read

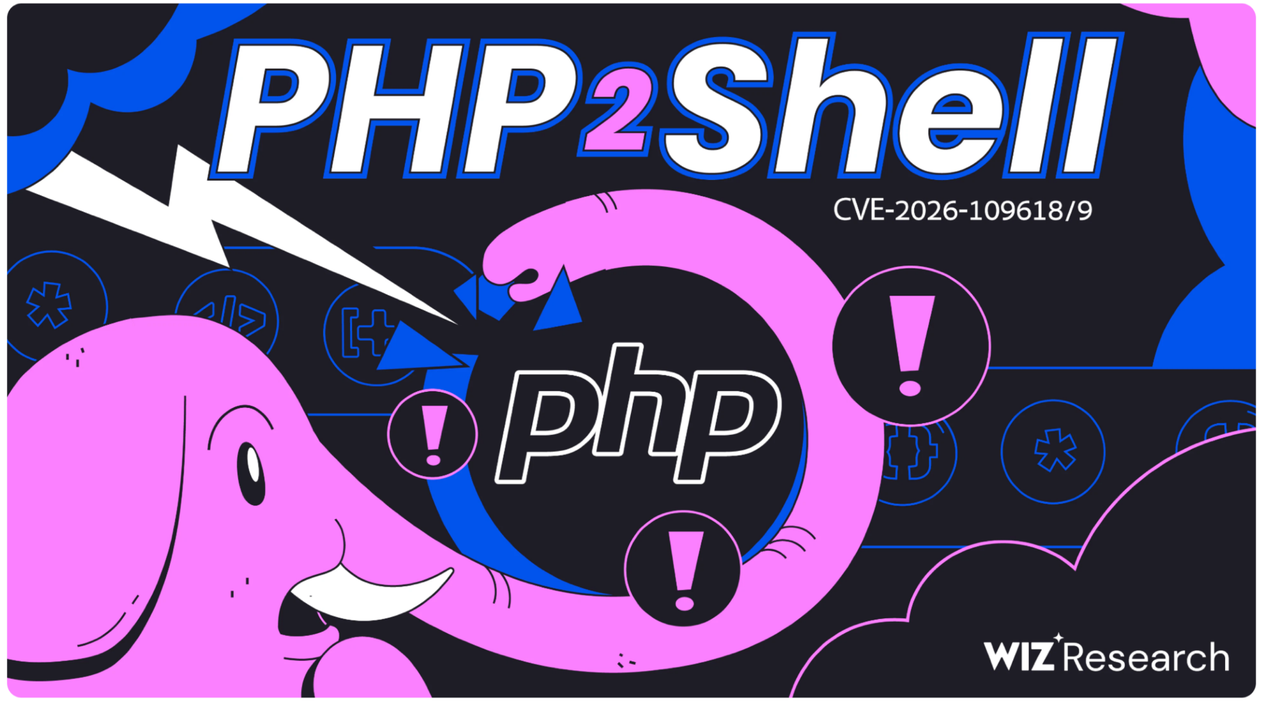


Table of contents

Executive Summary
Overview
The Vulnerabilities
Impact
The AI inflection point
Industry-level collaborations
Technical deep dive
How Can Wiz Help?
Conclusion
Responsible disclosure timeline
Stay in touch

PHP2Shell: Technical Threat Briefing with Wiz Research

Tracking ID	Bug	What an attacker can achieve	What we demonstrated
CVE-2026-109618	Upload-progress unserializer use-after-free	Corrupt process memory and execute code	Pre-authentication RCE on WordPress, Drupal and Joomla
	Remote process-memory disclosure	Unauthenticated disclosure of PHP worker memory	Application-agnostic remote memory disclosure
CVE-2026-109619	Fatal-error backtrace use-after-free	Corrupt process memory and execute code	Pre-authentication RCE on Drupal
	OPcache type confusion	Corrupt shared PHP state and execute code	Pre-authentication RCE on WordPress

Electronics › Computers & Accessories › Computer Components › Internal Components › Graphics Cards



VIPERA NVIDIA GeForce RTX 4090 Founders Edition Graphic Card

Visit the GeForce Store

4.6 ★★★★½ (206) | Search this page

$4,999.95 Price history

Or $416.66 /mo (12 mo) at 0% APR. Select plan

Get a **$50 Gift Card** for each friend who is approved for Prime Visa or Amazon Visa. Get started

Graphics Coprocessor	NVIDIA GeForce RTX 4090
Brand	GeForce
Graphics Ram Size	24 GB
Video Output Interface	HDMI
Graphics Processor Manufacturer	NVIDIA

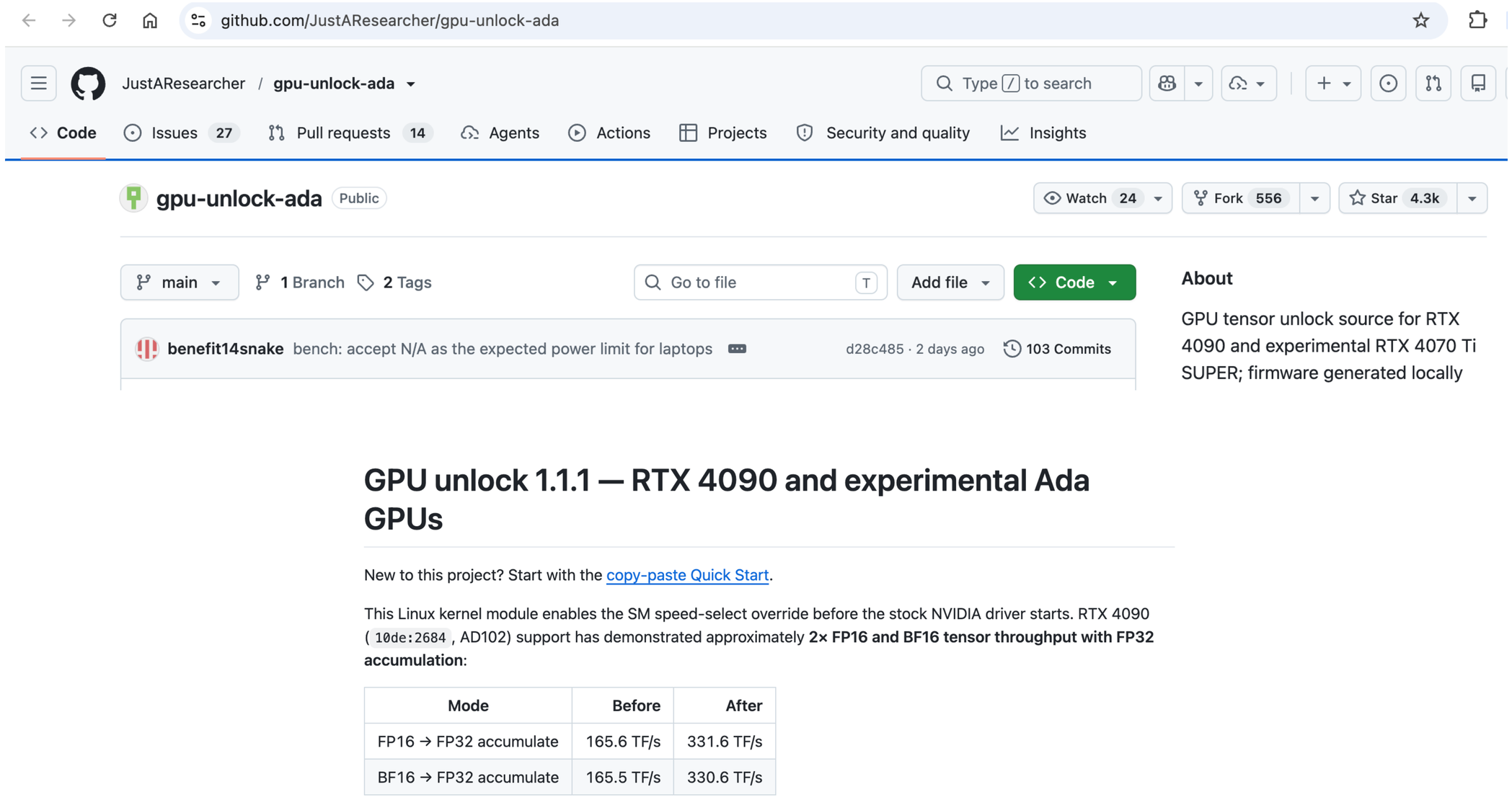
A Canary in the Crypto Mine: Defeating Stack Protection in a GPU Secure Coprocessor

Jon Pry
jonpry@gmail.com

June 2026

Abstract

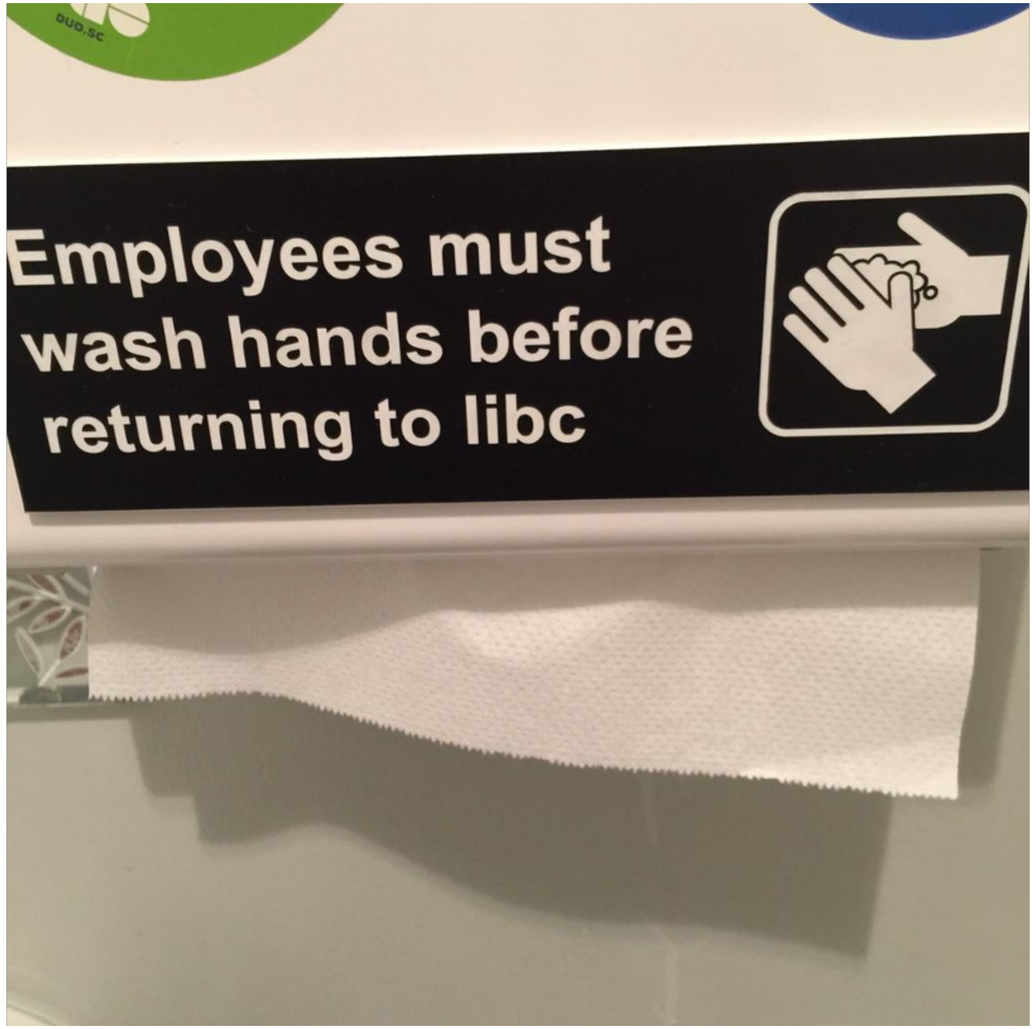
A widely sold datacenter-class accelerator (the NVIDIA "CMP 170HX") is the *same die* as a flagship A100 but is fuse-crippled on three commercial axes: SM math rate (throttled to 1/32), memory capacity (10 GB instead of 80 GB), and PCIe link (Gen1 instead of Gen4). We show that all three caps are software-defeatable from a rooted-but-otherwise-unprivileged host, with no signing key, no debug fuse, and no physical access. The root enabler is arbitrary code execution in the GPU's Heavy Secure (HS) Falcon mode. The HS break does not depend on the value of the stack canary: the stack-protector's *reference word* lives in writable memory reachable by the same data overflow it is meant to detect, so secrecy and entropy are irrelevant. Because the microcode is deterministic, we emulate it offline and author an overflow payload that is a *valid continuation image*, so the victim function passes its own canary check and returns into our chain. The economic consequence is that differentiation the business model treats as hardware-permanent is, for the three axes we defeated, only firmware-gated. Three further axes (PCIe Gen3, on-die ECC, runtime HBM retiming) resisted our attempts; we report this without claiming they are truly unbreakable, since we do not fully understand the relevant fuses or initialization. The HS code-execution bug is a **regression** present in the newer open-kernel-modules-era booter and absent in an older one, so version currency is not a security gradient, and an open host driver did not imply an audited signed booter. Measured gains: roughly 31–62x compute, 8x capacity, 2x link. We distill four transferable findings.


github.com/JustAResearcher/gpu-unlock-ada
JustAResearcher / gpu-unlock-ada
Type / to search
Code
Issues 27
Pull requests 14
Agents
Actions
Projects
Security and quality
Insights
gpu-unlock-ada
Public
Watch 24
Fork 556
Star 4.3k
main
1 Branch
2 Tags
Go to file
T
Add file
Code
benefit14snake bench: accept N/A as the expected power limit for laptops
d28c485 · 2 days ago
103 Commits
About
GPU tensor unlock source for RTX 4090 and experimental RTX 4070 Ti SUPER; firmware generated locally
GPU unlock 1.1.1 — RTX 4090 and experimental Ada GPUs
New to this project? Start with the copy-paste Quick Start.
This Linux kernel module enables the SM speed-select override before the stock NVIDIA driver starts. RTX 4090 (10de:2684 , AD102) support has demonstrated approximately 2× FP16 and BF16 tensor throughput with FP32 accumulation:
Mode
Before
After
FP16 → FP32 accumulate
165.6 TF/s
331.6 TF/s
BF16 → FP32 accumulate
165.5 TF/s
330.6 TF/s

Sanitizers



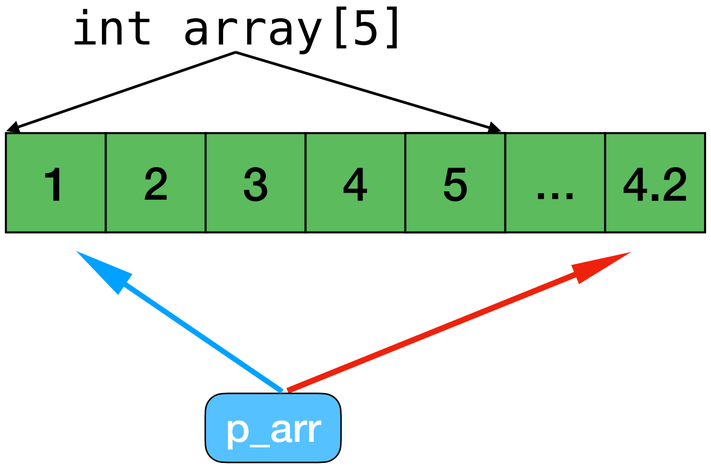
Memory Sanitizers



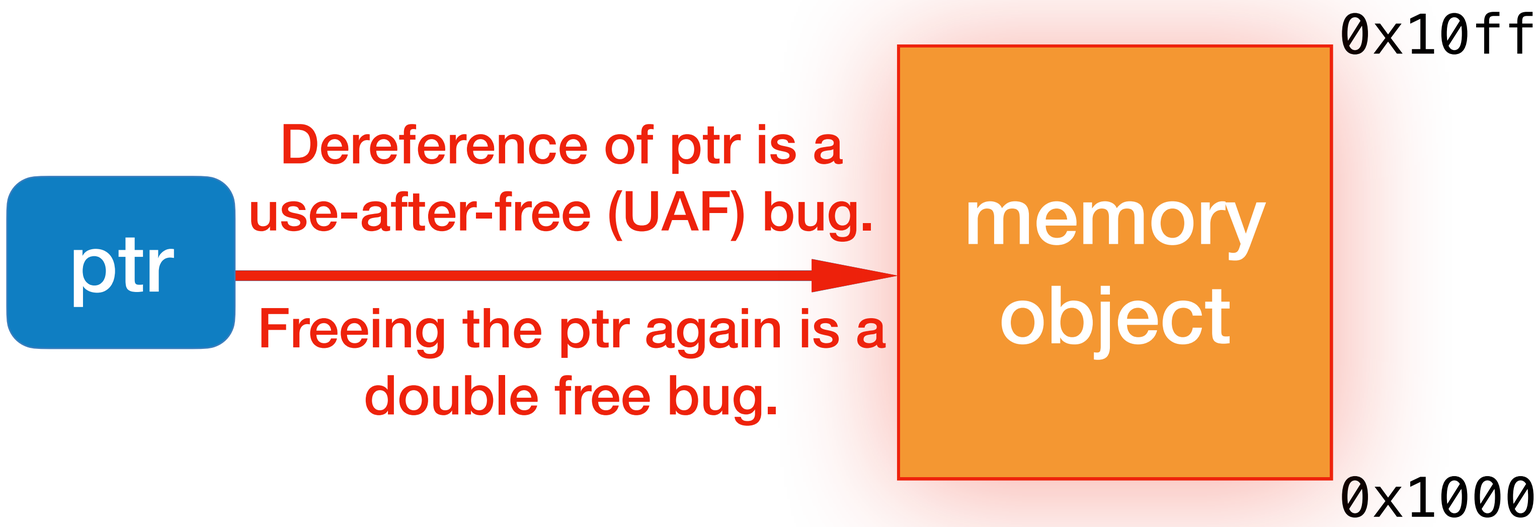
Spatial Memory Safety Bugs: Buffer Overflows



Reading/writing a buffer out of its bounds.



Temporal Memory Safety Bugs



Memory Safety Error: Accessing a semantically illegitimate memory address

Mitigations for Memory Safety Errors

- Address Space Layout Randomization
  - Probabilistic safety; shown to be not very effective
- Control-flow Integrity (CFI)
  - Forward CFI for indirect function calls and jumps
  - Backward CFI for return addresses
  - Often allows bugs to happen and hide for a while; and missing bugs
- Memory Isolation
  - Permit errors inside fault domains.
- Testing & Fuzzing
  - Challenging to have good coverage.
- None of the approaches ensures detecting memory safety bugs!

Stronger memory safety:
catching errors when they occur.

Sanitizer Goal: Crash Early and Quickly!

- Sanitizers instrument programs to check for violations, crash immediately.
- Fuzzers trigger bugs and record crashes, but not every bug crashes.
- Sanitizers enforce a security policy to crash the program upon violation.
- Sanitizers make bugs detectable (at least more likely).

Sanitizers enforce a policy, detect bugs early, and increase testing effectiveness.



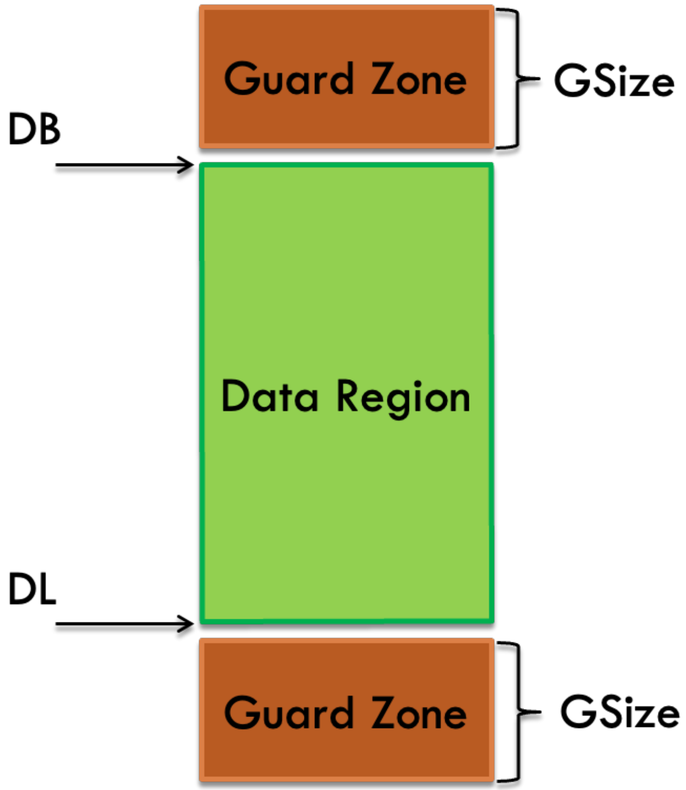
Address Sanitizer (ASan)

- Monitoring (sanitizing) *every* memory access to detect memory safety violations.
- Originally developed, and still maintained, by Google
- Available in LLVM and GCC
- Have found thousands of memory safety bugs.

ASan Algorithm

- Map regular memory to shadow memory.
  - Each byte is mapped.
  - Regular memory includes valid memory objects and redzones.
- Shadow memory indicates the validity of mapped regular memory.
- Before each memory access, check the target memory's validity by querying its mapped shadow memory's status.

Optimization: Guard Zone/Page



- Place a guard zone before/after a data region.
- Guard zones are unmapped or not readable/writable.
  - Access to guard zones are trapped by hardware.
- Assume Guard Zone's size is GSize, a memory read/write is safe if the address is in [DB-GSize, DL+GSize].
- Also called red zone

ASan Algorithm

- Map regular memory to shadow memory.
  - Each byte is mapped.
  - Regular memory includes valid memory objects and redzones.
- Shadow memory indicates the validity of mapped regular memory.
- Before each memory access, check the target memory's validity by querying its mapped shadow memory's status.

Shadow Memory

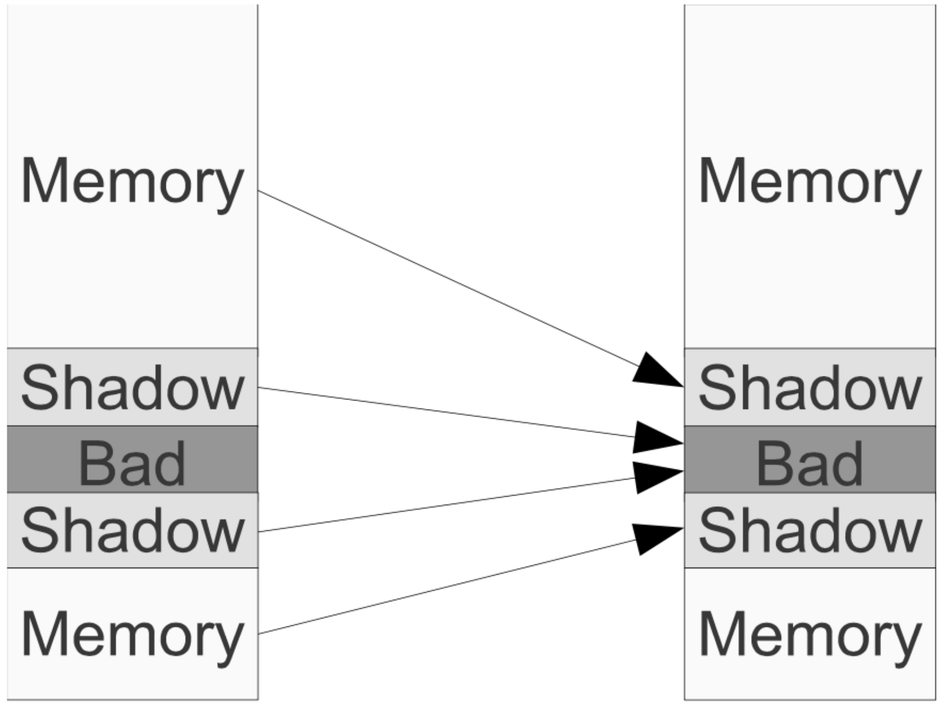
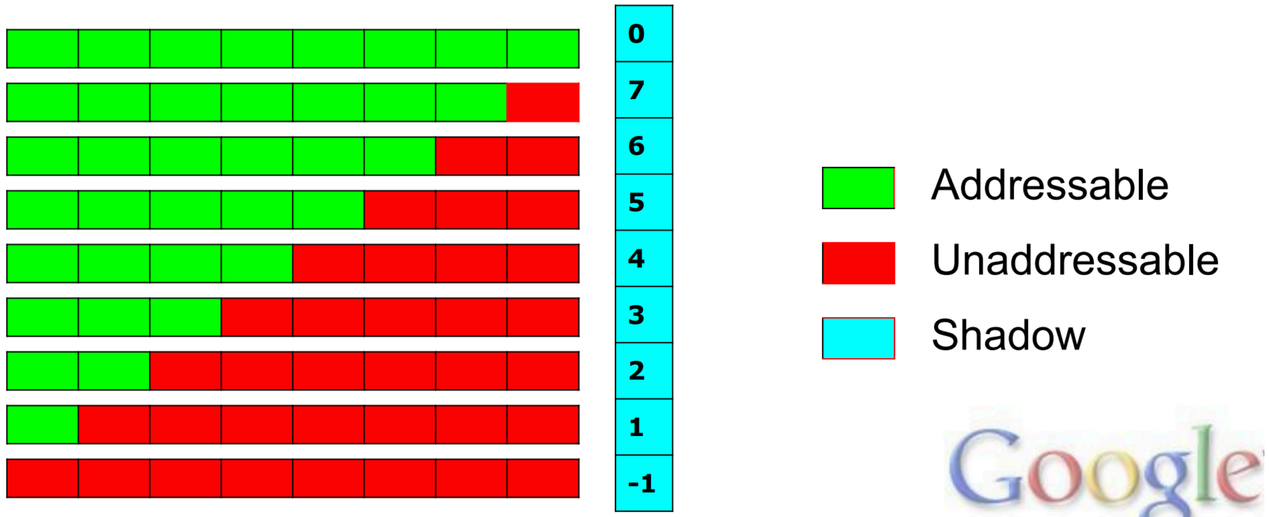


Figure 1: AddressSanitizer memory mapping.

Shadow Memory Mapping

- Newly allocated memory heap objects are typically aligned at a 8-byte boundary.
- Any aligned 8-byte of memory is in one of 9 states:
  - The first *k* (0 <= *k* <= 8) bytes are addressable.
  - The remaining (8 - *k*) bytes are not.
- These 9 states can be encoded into one byte.



Shadow Memory

- Newly allocated memory heap objects are typically aligned at a 8-byte boundary.
- Any aligned 8-byte of memory is in one of 9 states:
  - The first *k* (0 <= *k* <= 8) bytes are addressable.
  - The remaining (8 - *k*) bytes are not.
- These 9 states can be encoded into one byte.
- ASan reserves 1/8 of virtual address space to its shadow memory.
- Given a virtual address `Addr`, its shadow memory is computed by `(Addr >> 3) + Offset`
  - `Offset` is system/implementation-specific.
    - ASLR must be taken into account when choosing `Offset`.
- Shadow memory is mapped to inaccessible memory.

Shadow Memory

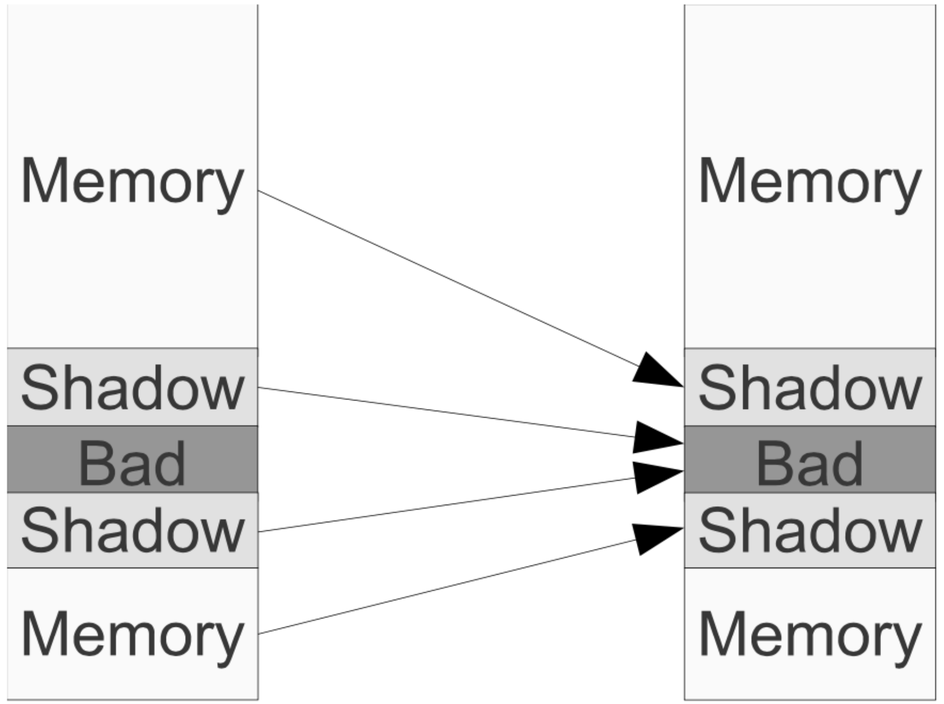
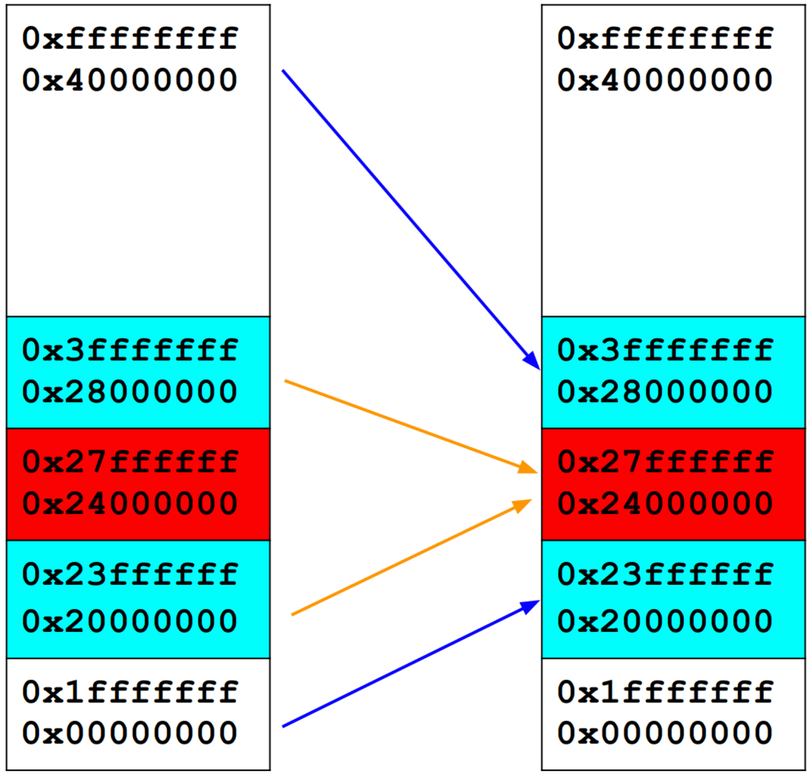


Figure 1: AddressSanitizer memory mapping.

An Example of Shadow Memory for x86-32 Linux



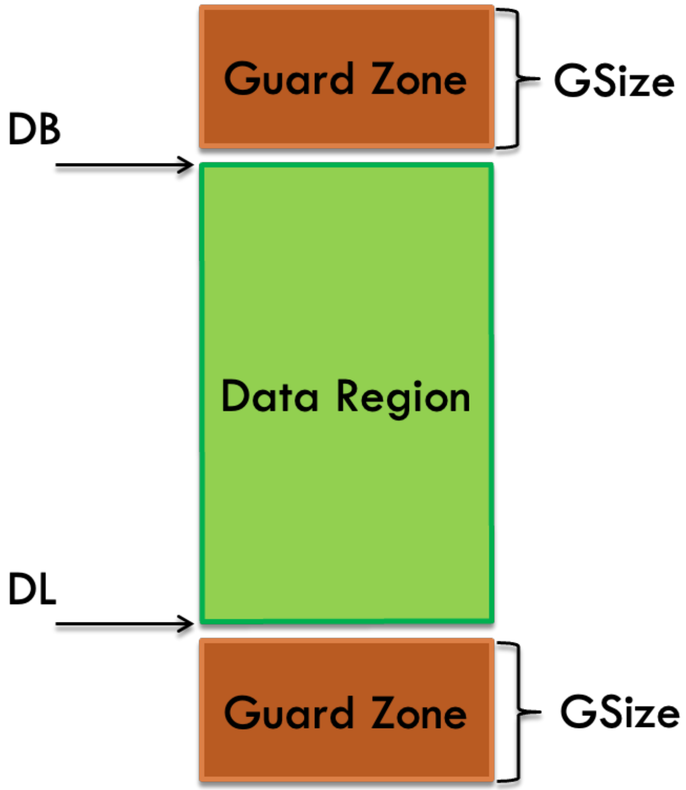




ASan Algorithm

- Map regular memory to shadow memory.
  - Each byte is mapped.
  - Regular memory includes valid memory objects and redzones.
- Shadow memory records the validity of mapped regular memory.
- Before each memory access, check its validity by querying its mapped shadow memory status.

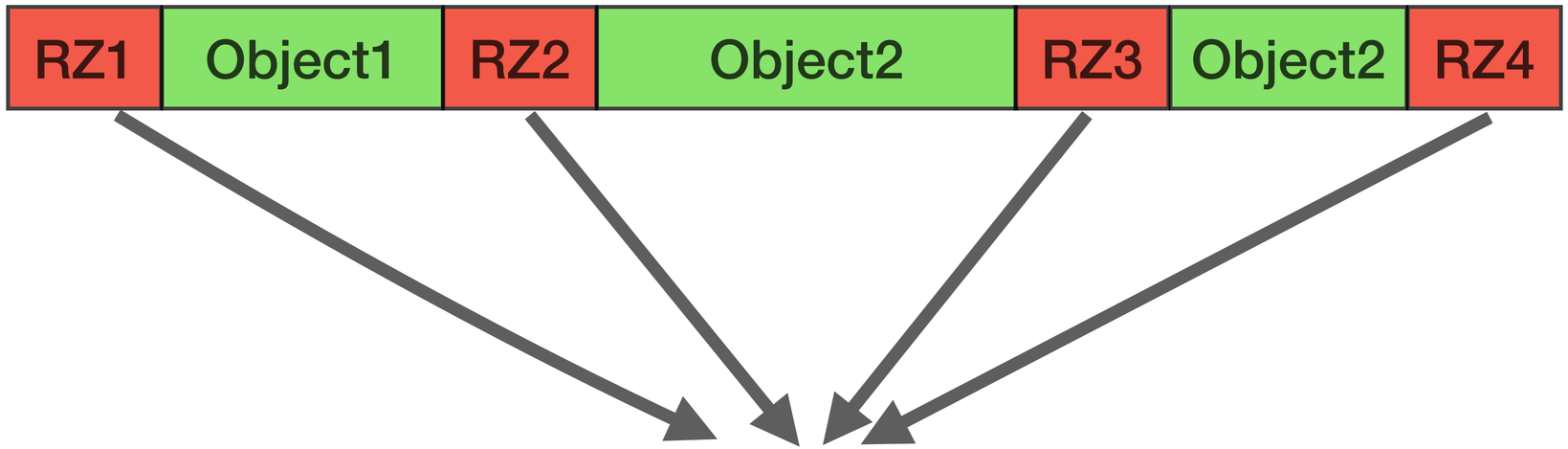
Optimization: Guard Zone/Page



- Place a guard zone before/after a data region.
- Guard zones are unmapped or not readable/writable.
  - Access to guard zones are trapped by hardware.
- Assume Guard Zone's size is GSize, a memory read/write is safe if the address is in [DB-GSize, DL+GSize].
- Also called red zone

Redzones Between Valid Memory Objects

- Page-level redzones are too coarse-grained for memory objects.
- ASan uses small redzones between each valid memory object.
  - Minimum: 32 bytes; default: 128 bytes
  - Larger redzones enable higher probability of detecting buffer overflows.



Mapped to shadow memory indicating
they are not addressable.

Instrumentation: 8-byte Access

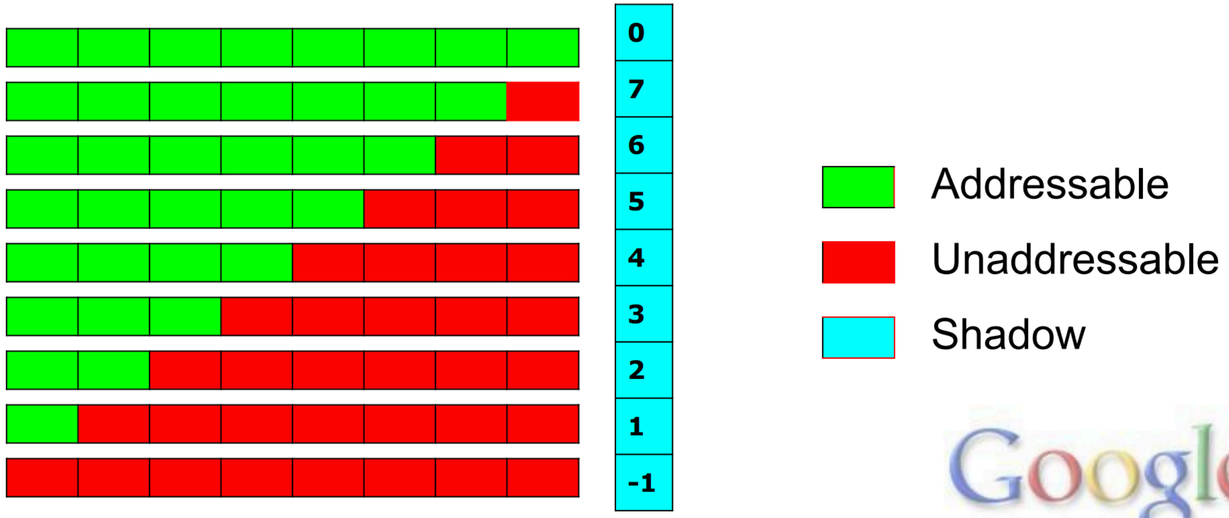
- Accessing an 8-byte value from address p

```
unsigned long val = *p;
```



```
ShadowAddr = (p >> 3) + Offset;
if (*ShadowAddr != 0) {
    ReportAndCrash(Addr);
}

unsigned long val = *p;
```

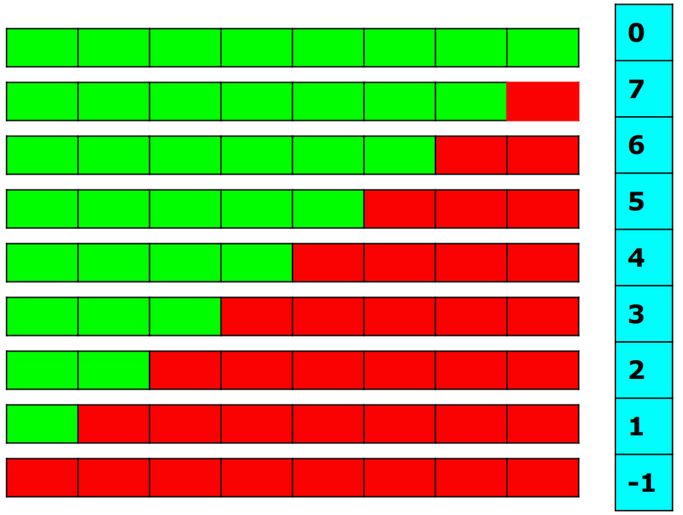


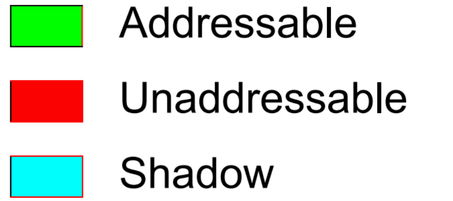
Google

Instrumentation: N-byte Access (N = 1, 2, 4)

- Accessing an k-byte value from address p

```
… *p
```






```
ShadowAddr = (p >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((p & 7) + AccessSize > k)) {
    ReportAndCrash(p);
}
```

Google

Detect Use-After-Free Bugs

- After `free()` is called, ASan “poisons” the object’s shadow memory.
  - Set the corresponding shadow memory as unaddressable
- Delay the free
  - Maintain a quarantine pool for those to-be-free memory objects.
    - Really free them when the pool is full.
- May miss UAF bugs

```
char *a = new char[1 << 20]; // 1MB
delete [] a; // <<< "free"
char *b = new char[1 << 28]; // 256MB
delete [] b; // drains the quarantine queue.
char *c = new char[1 << 20]; // 1MB
a[0] = 0; // "use". May land in ’c’.
```

How to detect buffer overflows on stack/global?

Detecting Buffer Overflows on Stack and Global

- Stack and global objects are not necessarily aligned at 8-bytes boundary.
- ASan needs to poison redzones around such objects.
- For globals, redzones are created at compile time and poisoned at startup.
- For stack objects, redzones are created and poisoned at run-time.

Example of Instrumenting Stack

```
void foo() {
    char arr[10];
    // <function body>
}
```



```
void foo() {
    char rz1[32];
    char arr[10];
    char rz2[32-10+32];
    unsigned *shadow =
        (unsigned*)(((unsigned*)rz1>>3)+Offset);
    // poison the redzones around arr.
    shadow[0] = 0xffffffff; // rz1
    shadow[1] = 0xffff0200; // arr and rz2
    shadow[2] = 0xffffffff; // rz2
    // <function body>
    // un-poison all
    shadow[0] = shadow[1] = shadow[2] = 0;
}
```

Does it guarantee to detect buffer overflows?

*What if a memory over-read/write **strides over** redzones?*

Size of Redzone

- Larger redzones enables higher probability of detecting buffer overflows.
- However, higher performance overhead
  - More memory consumption
  - Slower execution time
    - Larger redzones mean more memory writes to their shadow memory.

Misaligned Issue

- Assume memory accesses are aligned. However,

```
int *a = new int[2]; // 8-aligned
int *u = (int*)((char*)a + 6);
*u = 1; // Access to range [6-9]
```

```
ShadowAddr = (p >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((p & 7) + AccessSize > k)) {
    ReportAndCrash(p);
}
```



K is 0 because 8 bytes of memory starting from `((p >> 3) << 3)` are addressable.

Conflict with Load Widening

- Load widening: A compiler optimization technique that combine multiple smaller memory loads into one bigger (wider) load.

```
struct X { char a, b, c; };
void foo() {
    X x; ...
    ... = x.a + x.c;
```

- `x.a + x.c` may be compiled into one 4-byte load, causing ASan to report an “error” (false positive).
- Solution: Disable this compiler optimization.

High Performance Cost

Reported from the 2012 original paper:

- 73% slowdown on average for SPEC CPU2006
- 3.37X memory consumption
- 2.5x code size bloating

Reported from a new 2025 paper:

- 88% slowdown on average for SPEC CPU2017
- 2X memory consumption

Effective Bug-detection

- Chromium (including WebKit); in first 10 months
  - heap-use-after-free: 201
  - heap-buffer-overflow: 73
  - global-buffer-overflow: 8
  - stack-buffer-overflow: 7
- Mozilla
- FreeType
- FFmepeg
- libjpeg-turbo
- Perl
- Vim
- LLVM
- GCC
- WebRTC
- ...



How to implement ASan?

Compiler, Of Course!

Considerations for Engineering Sanitizers

- What types of memory bugs to detect?
- What kinds of operations to instrument?
- What metadata to maintain?
- What data structures to use to manage metadata?
- What is the performance overhead budget?
- What optimizations can we do?
- How is the compatibility with un-sanitized code?

Summary of ASan

- Using shadow memory to sanitize *every* memory access
- Detecting both spatial and temporal memory safety
- Implementation: Compiler instrumentations + run-time library support
- High performance and memory overhead
- Incomplete bug detection ability

Can we do better than ASan for security?

Of Course!

Memory Safety

- Memory buffers are allocated and deallocated during program execution.
- Each buffer occupies a contiguous range of memory addresses and also has a lifetime.
  - **Bounds**: The lower and upper addresses of the buffer
  - **Lifetime**: When the buffer is valid for use.
    - E.g., a buffer allocated by a function's stack has a lifetime when the function executes; should not be used after the function returns.
    - E.g., a buffer that was created by `malloc` should not be accessed after being freed.

Memory Safety: Expected vs. Abnormal Behaviors

- Expected behavior: A buffer should be accessed within its bounds and only during its lifetime.
  - **Spatial** memory safety: A buffer should be accessed within its bounds.
  - **Temporal** memory safety: A buffer can be accessed only during its lifetime.
- Abnormal behavior:
  - When spatial memory safety is violated, we have buffer overread/overwrite.
  - When temporal memory safety is violated, we have use-after-free & invalid free.

Why are there so many memory safety bugs and vulnerabilities?

Programming *Correctly* in C/C++ is (Extremely) Hard

Simple and primitive language features

- Basic data types (char, integer, boolean, etc.)
- struct
- Pointers
- Basic control flow (conditional branches, loops, etc.)



Pointer: Capability to manipulate memory.

- For C, pointer is usually implemented as a virtual address.



C pointers can do almost arbitrary memory manipulation!

- The correctness is at the discretion of programmers.

ASan is one type of object-based memory safety.

Pointer-based Memory Safety

- For each pointer, maintain information, called *metadata*, about the pointed memory object
- Use the metadata to do validity checking for pointer dereference
- Ideally, we would like a mechanism that is
  - Comprehensive, i.e., catching all memory safety errors
  - Efficient, i.e., low execution time and memory overhead
  - Automatic, i.e., minimal effort from programmers
  - Backward-compatible, i.e., running smoothly with unchanged legacy code

Core Question:
How to manage safety metadata?

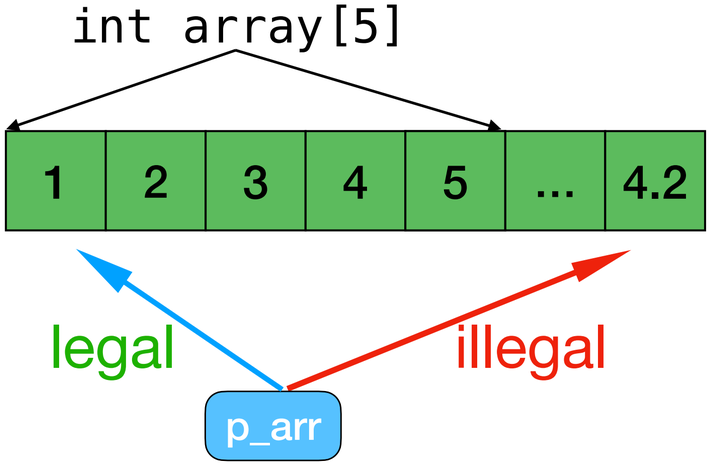
Managing Pointer's Metadata

- What metadata is needed?
- How is a pointer mapped to / associated with its metadata?
- How to propagate metadata during pointer propagation (e.g., assignment)?
- How to update the metadata?
- How to perform memory safety checks using the metadata?

Spatial Memory Safety Bugs: Buffer Overflows



Reading/writing a buffer out of its bounds.



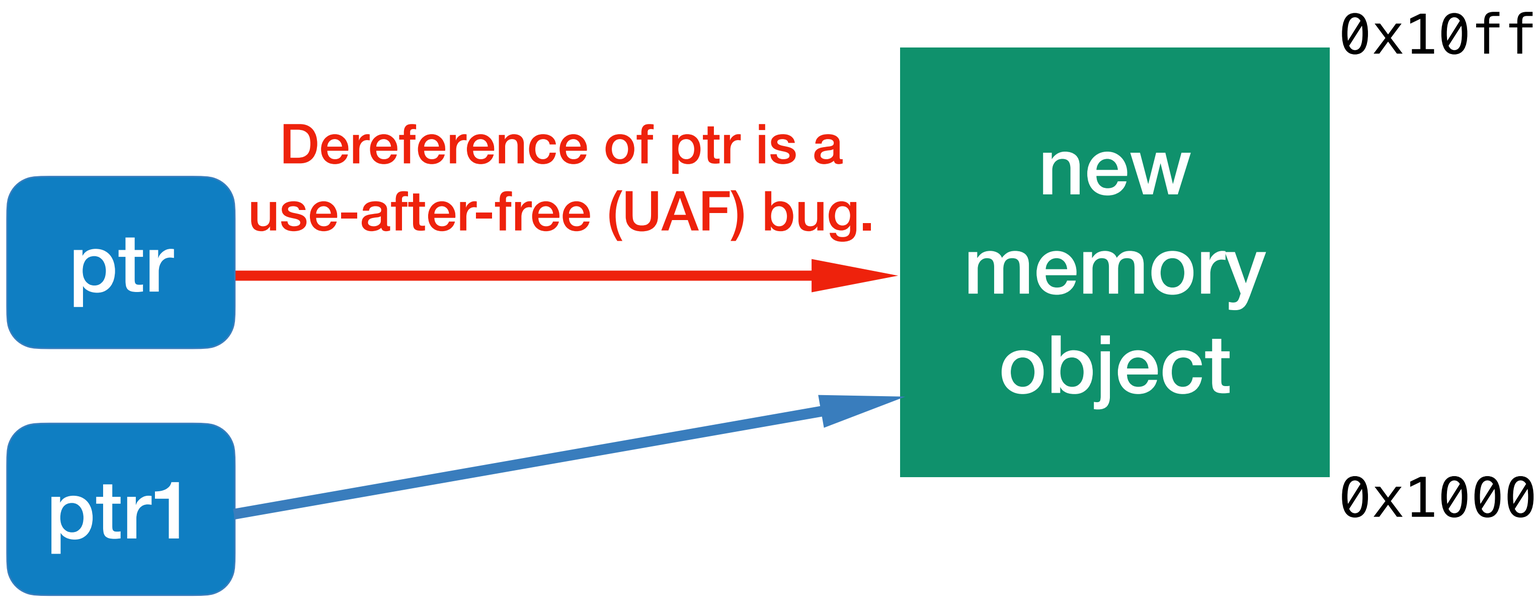
Essential information:

- Starting address (base/lower bound)
- Ending address (upper bound)
- Object size

Metadata for Spatial Memory Safety

- Option 1: (`base, upper_bound`)
- Pointer dereference: Check if `base <= p < upper_bound`
- Option 2: (`base, size`)
- Pointer dereference: Check if `base <= p < base + size`
- More accurate check: base <= p && p + sizeof(referent) <= upper_bound

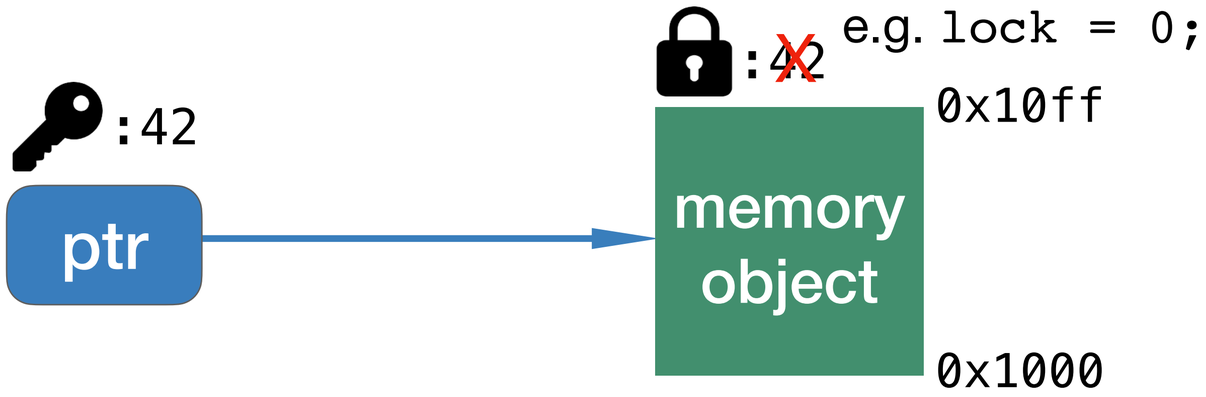
Temporal Memory Safety Bugs



Security risks

- ⚠ Information leaking
- ⚠ Data corruption
- ⚠ Denial of service

Key-lock Checking for Temporal Memory Safety



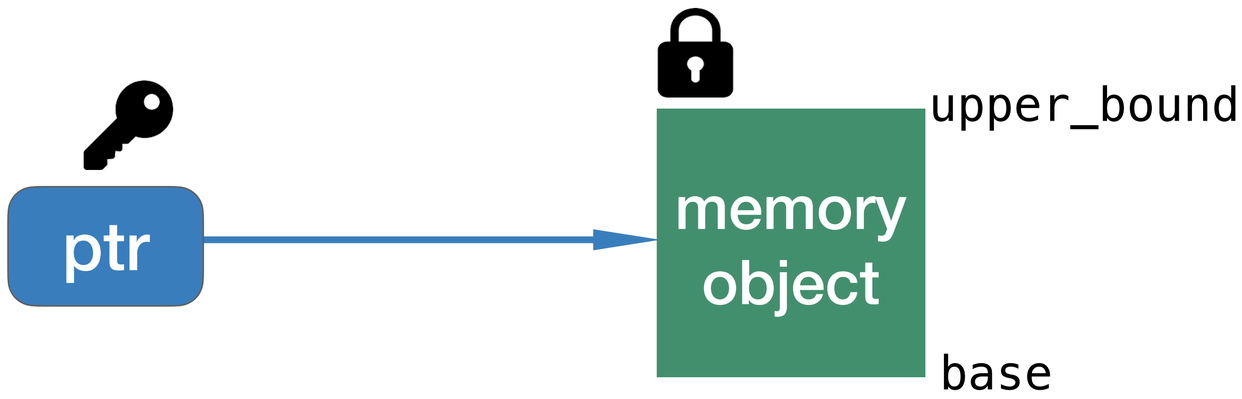

```
check_if_key_matches_lock(ptr);
ptr->num = 30;
```

- Assign memory object a *lock* and pointer a *key*
- Initialize key and lock to the same value
- Invalidate lock upon memory deallocation
- Dynamically check if key matches lock

Charles Fischer and Richard LeBlanc. *The Implementation of Run-Time Diagnostics in Pascal*.
1980 IEEE Transactions on Software Engineering

Metadata for Pointers

- Spatial memory safety: `base, upper_bound/size`
- Temporal memory safety: `key, lock`



Where to put the metadata?

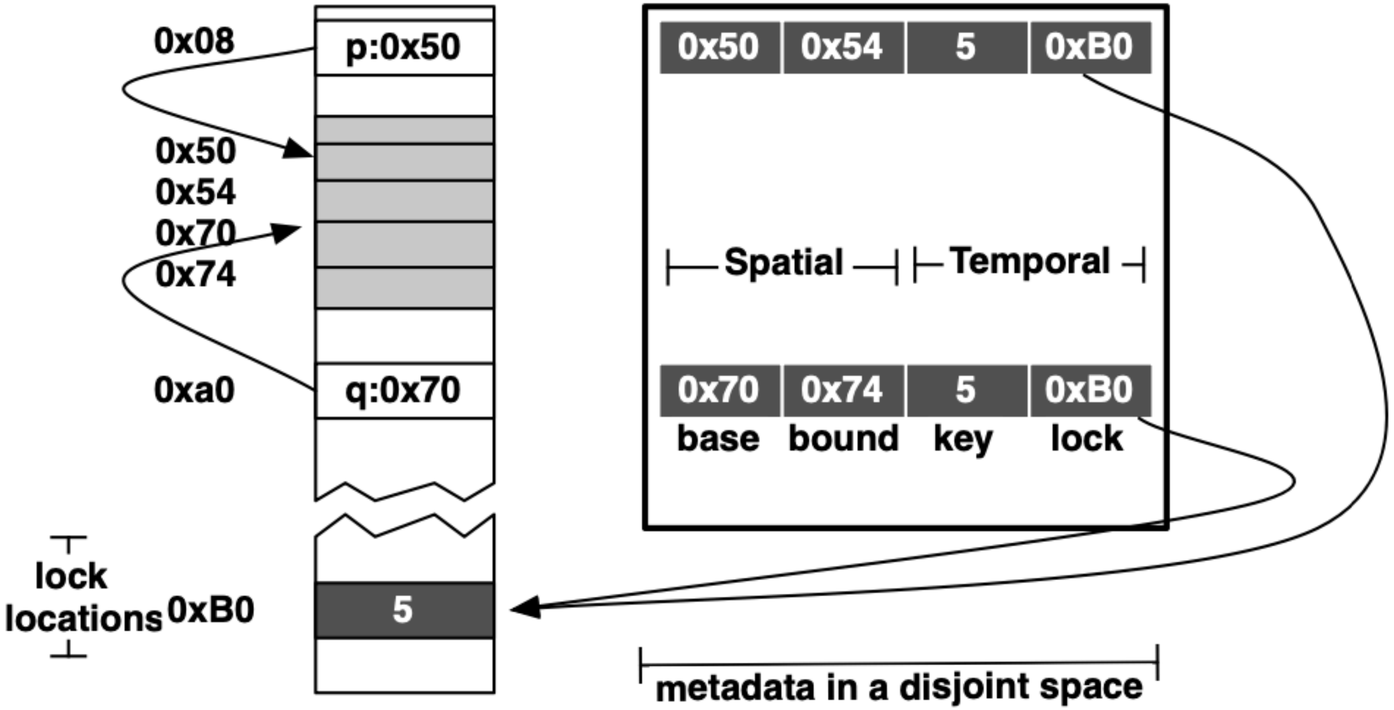
Managing Pointer's Metadata

- What metadata is needed?
- How is a pointer mapped to / associated with its metadata?
- How to propagate metadata during pointer propagation (e.g., assignment)?
- How to update the metadata?
- How to perform memory safety checks using the metadata?

Disjoint Metadata Management: e.g., SoftBoundCETS

SoftBoundCETS' Metadata Management

- Record metadata in a disjoint memory region
- E.g., p and q points to different sub-fields in the same memory object, so they maintain different base-upper bounds but the same key-lock.



SoftBoundCETS' Metadata Management

- Record metadata in a disjoint memory region
- Implementation options: hash table, shadow memory
- More efficient option: A two-level lookup trie to locate the metadata
  - Use a pointer's address as the initial lookup index

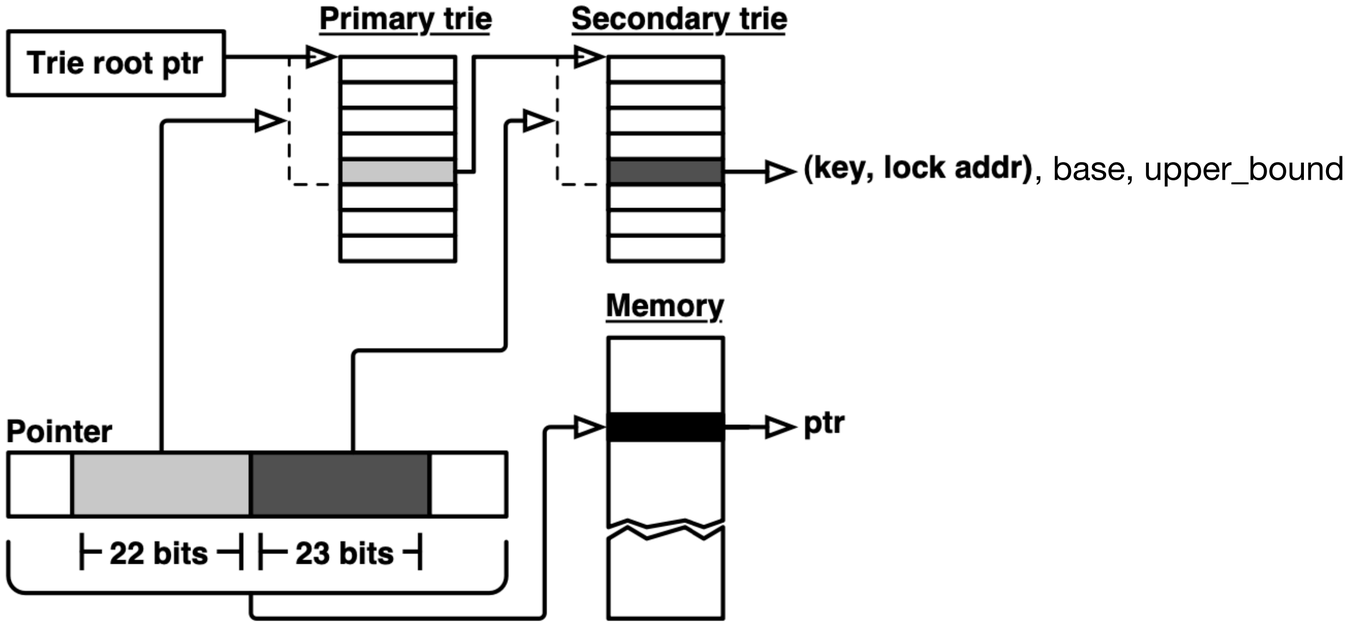


Figure 3. Organization of the trie for pointer metadata

SoftBoundCETS: Creating Pointers

- Metadata are created when their pointers are created.
  - E.g., “`p = malloc(size);`” becomes

```
p = malloc(size);
```

```
p_key = next_key++;
p_lock = allocate_lock();
*(p_lock) = p_key;
p_ base = p;
p_bound = p != 0 ? p+size: 0;
```

  - E.g., “`free(p);`” becomes

```
free(p);
```

```
*(p_lock) = INVALID;
deallocate_lock(p_lock);
```

SoftBoundCETS: Pointer Arithmetic

- When an expression contains pointer arithmetic (e.g. `ptr+index`), array indexing (e.g. `&(ptr[index])`), or pointer assignment (e.g., `newptr = ptr;`), the resulting pointer inherits the metadata of the original pointer.
  - E.g., "`q = p + index;`" becomes

```
q = p + index;
// or &p[index]
```

```
q_base = p_base;
q_bound = p_bound;
q_key =  p_key;
q_lock = p_lock;
```

SoftBoundCETS: Pointer Dereference

- Retrieve metadata and perform memory safety checks

(c) Temporal Check

```
tcheck(p_key, p_lock) {
  if (p_key != *(p_lock))
     raise exception();
}
```

(d) Spatial Check

```
scheck(p, p_base, p_bound, size) {
  if (p < p_base ||
       p + size >= p_bound)
     raise exception();
}
```

SoftBoundCETS: Pointer Load & Store

(b) Pointer Load

```
int **p, *q;
...
scheck(p, p_base, p_bound);
tcheck(p_key, p_lock);
q = *p;
q_base = lookup(p)->base;
q_bound = lookup(p)->bound;
q_key = lookup(p)->key;
q_lock = lookup(p)->lock;
```

(c) Pointer Store

```
int **p, *q;
...
scheck(p, p_base, p_bound);
tcheck(p_key, p_lock);
*p = q;
lookup(p)->base = q_base;
lookup(p)->bound = q_bound;
lookup(p)->key = q_key;
lookup(p)->lock = q_lock;
```

Comprehensive Memory Safety

- Metadata is manipulated/accessed only through the additional instrumentation added.
- Metadata is not corrupted and accurately depicts the region of memory that a pointer can legally access.
- All memory accesses are conceptually checked before a dereference.

SoftBoundCETS

- Achieves full memory safety with good backward-compatibility
- However, very high performance overhead.
  - ~75% on SPEC CPU2006, reported in 2015, based on LLVM-3.4.
  - 167% on 8 C SPEC CPU2017 benchmarks, reported in 2024, based on LLVM-12

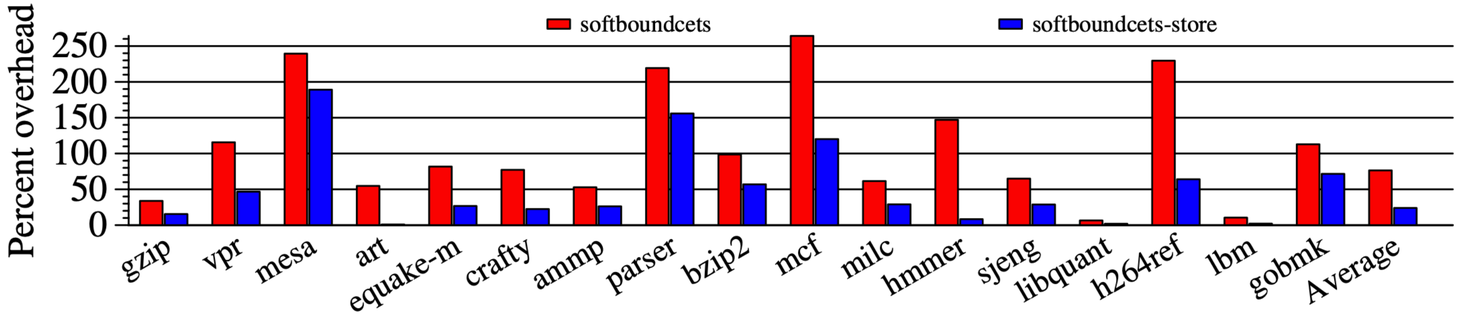


Figure 6 Runtime execution time overheads of the SoftBoundCETS compiler prototype with comprehensive memory safety checking (left bar of each stack) and while checking only stores (right bar of each stack) on a Intel Haswell machine. Smaller bars are better as they represent lower runtime overheads.

Why is this approach slow?

Reasons for SoftBoundCETS' Slow Performance

- Inefficient metadata management
- Uniform treatment of all pointers

SoftBoundCETS' Metadata Management

- Use a two-level lookup trie to locate the metadata
  - Use a pointer's address as the initial lookup index

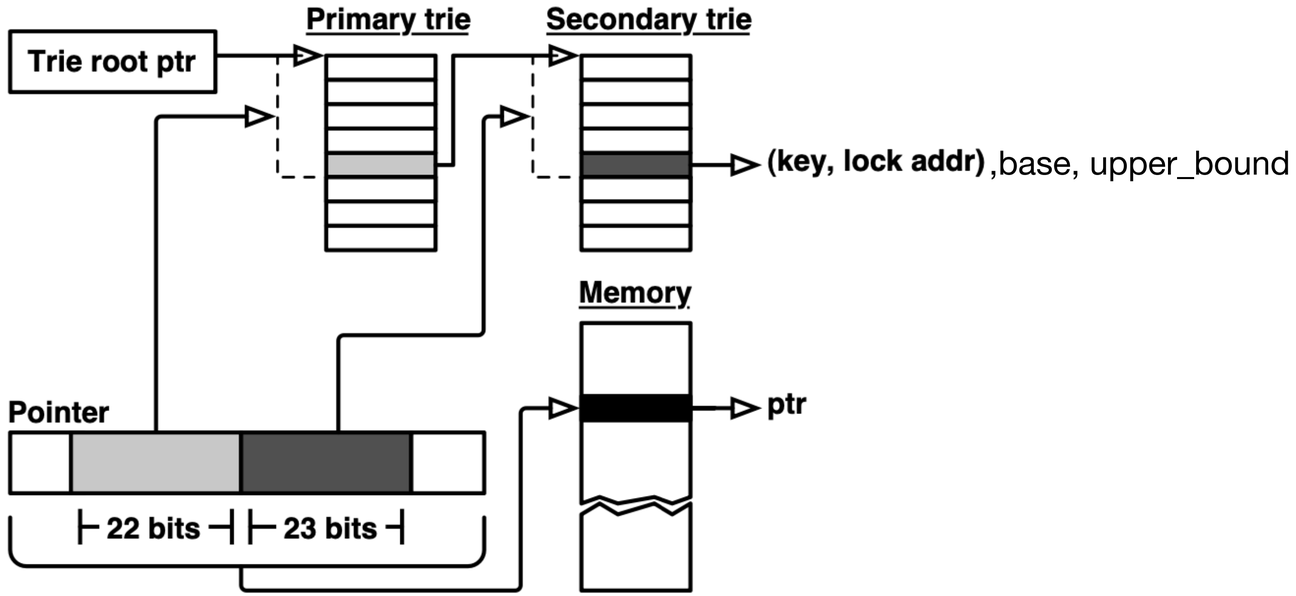
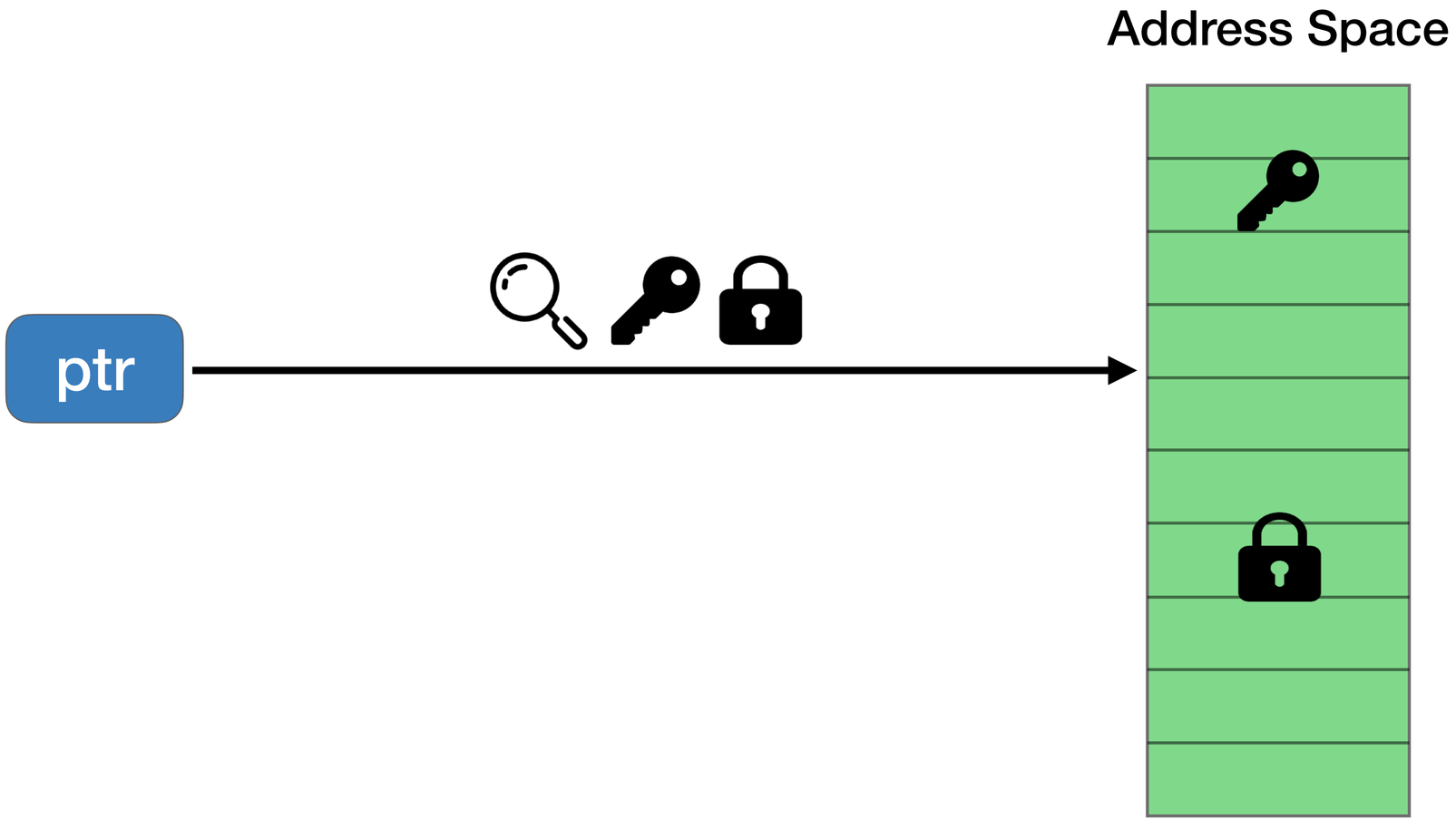


Figure 3. Organization of the trie for pointer metadata

- Loading metadata:14 x86-64 instructions
- Storing metadata: 16 x86-64 instructions

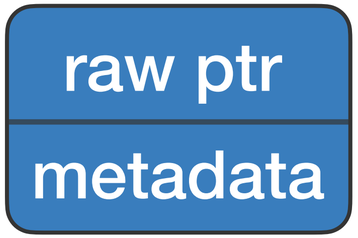
Disjoint Metadata Management is Slow



Fat Pointers

- Pointer representation carries metadata besides the raw pointer (address).

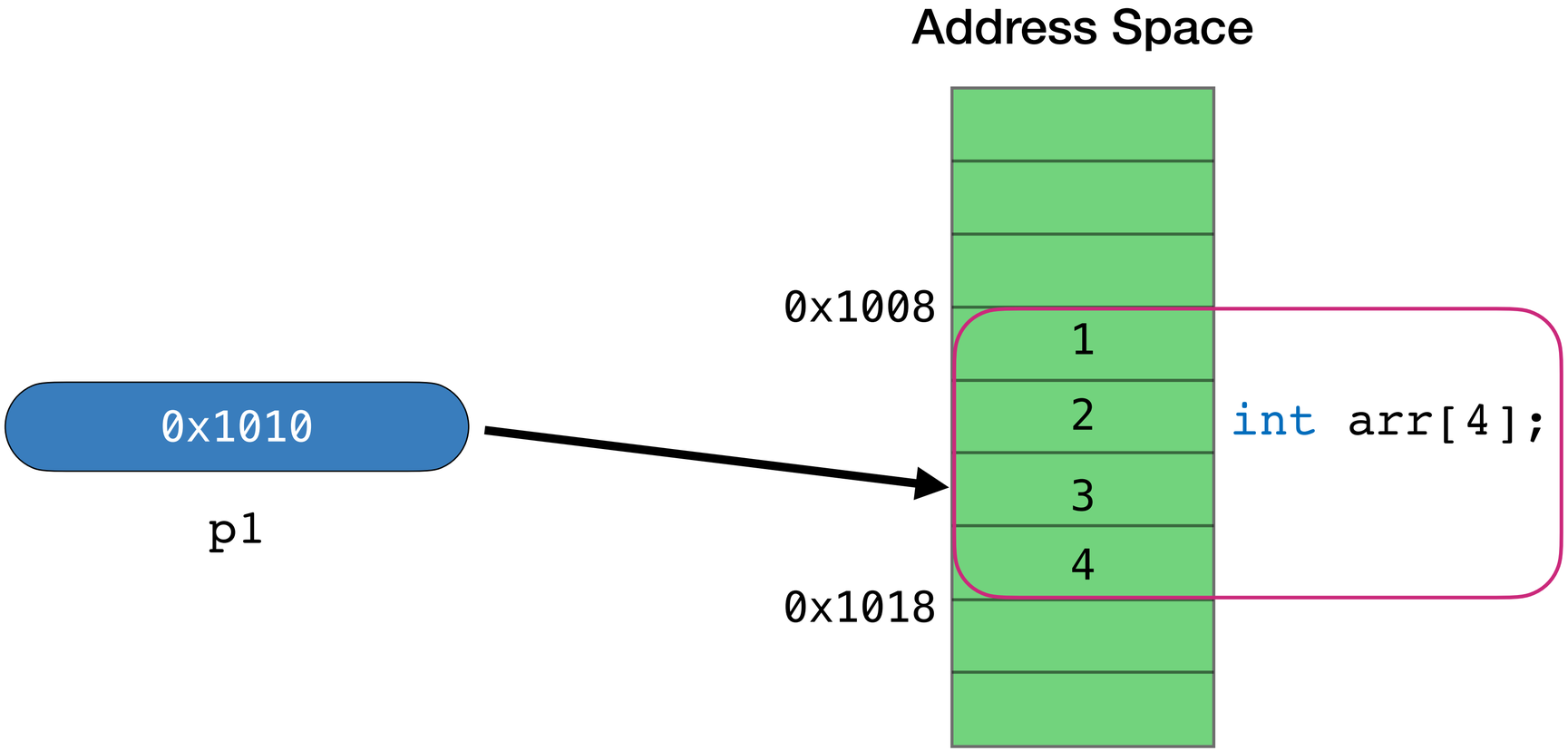
Fat pointer:



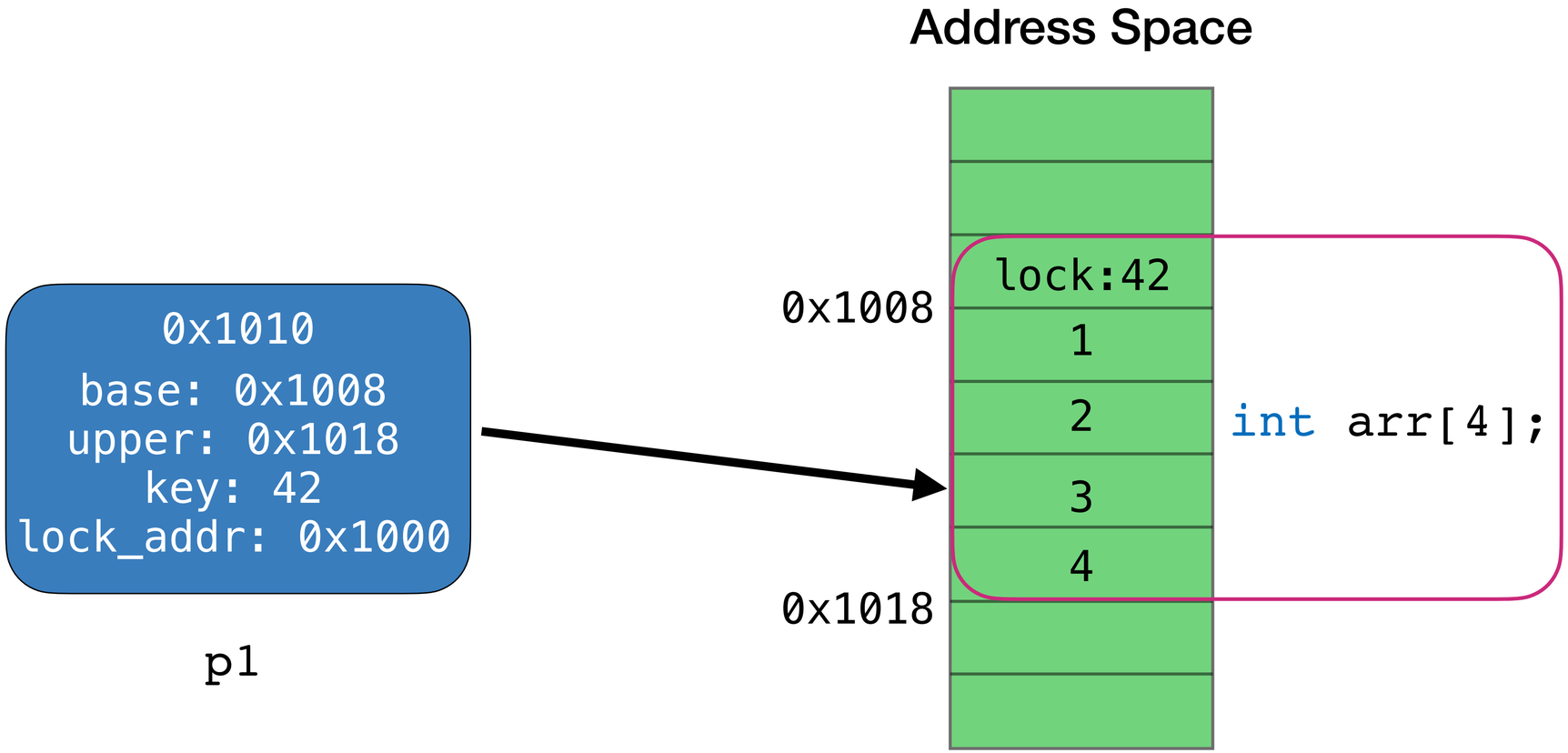
- One implementation

```
struct _safe_ptr {
    void *raw_ptr;
    void *base_addr;
    void *upper_bound;
    uint64_t key;
    void *lock_addr;
};
```

Example of Using Fat Pointers



Example of Using Fat Pointers



How do fat pointers interact with legacy library code?

Legacy libraries are unaware of the new fat pointers.

Type Compatibility with Unchanged Code

- One implementation

```
struct safe_ptr {
    char *raw_ptr;
    char *base_addr;
    char *upper_bound;
    uint64_t key;
    void *lock_addr;
};
```

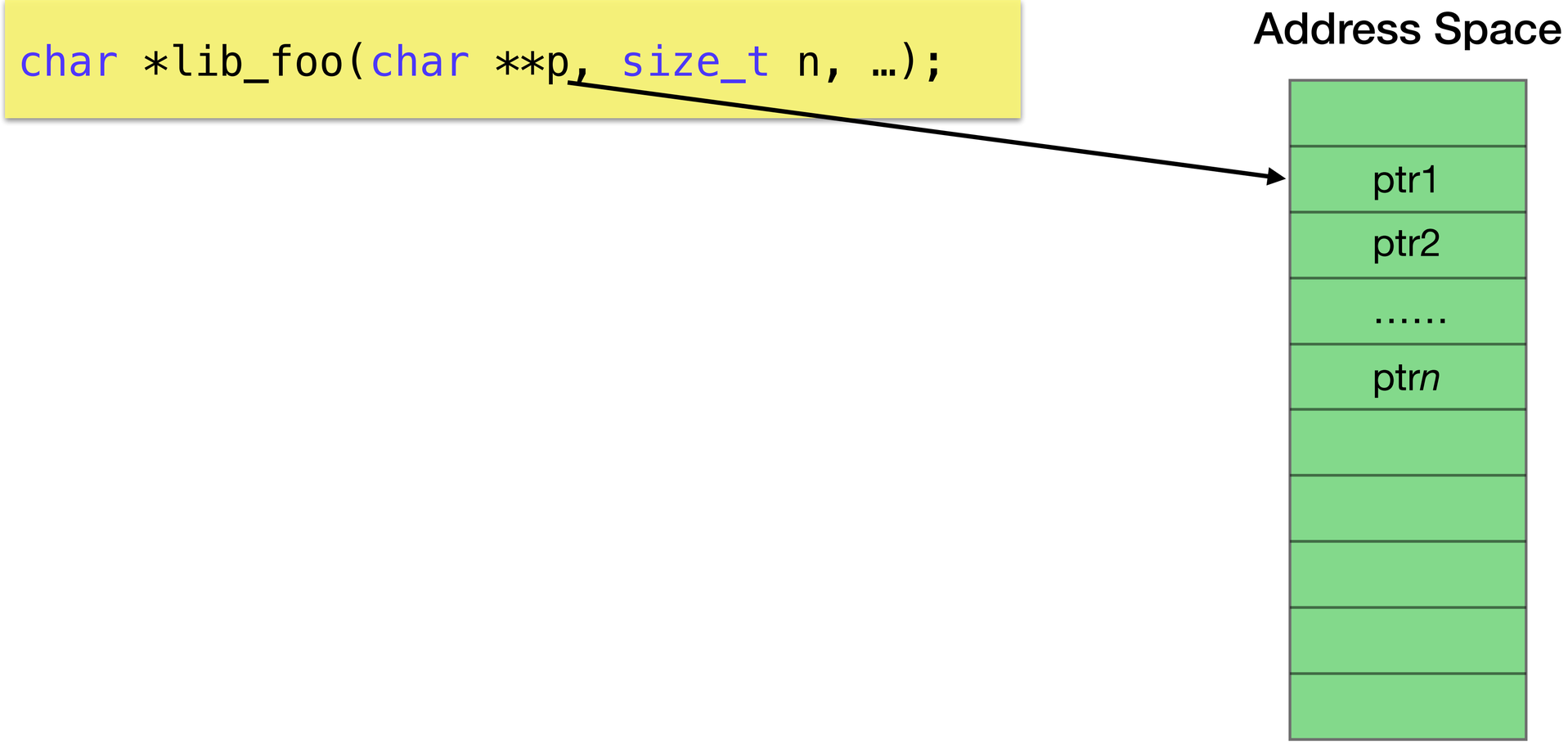
```
char *strchr(const char *s, int c);
```

How to pass a `safe_ptr` to `strchr()`?

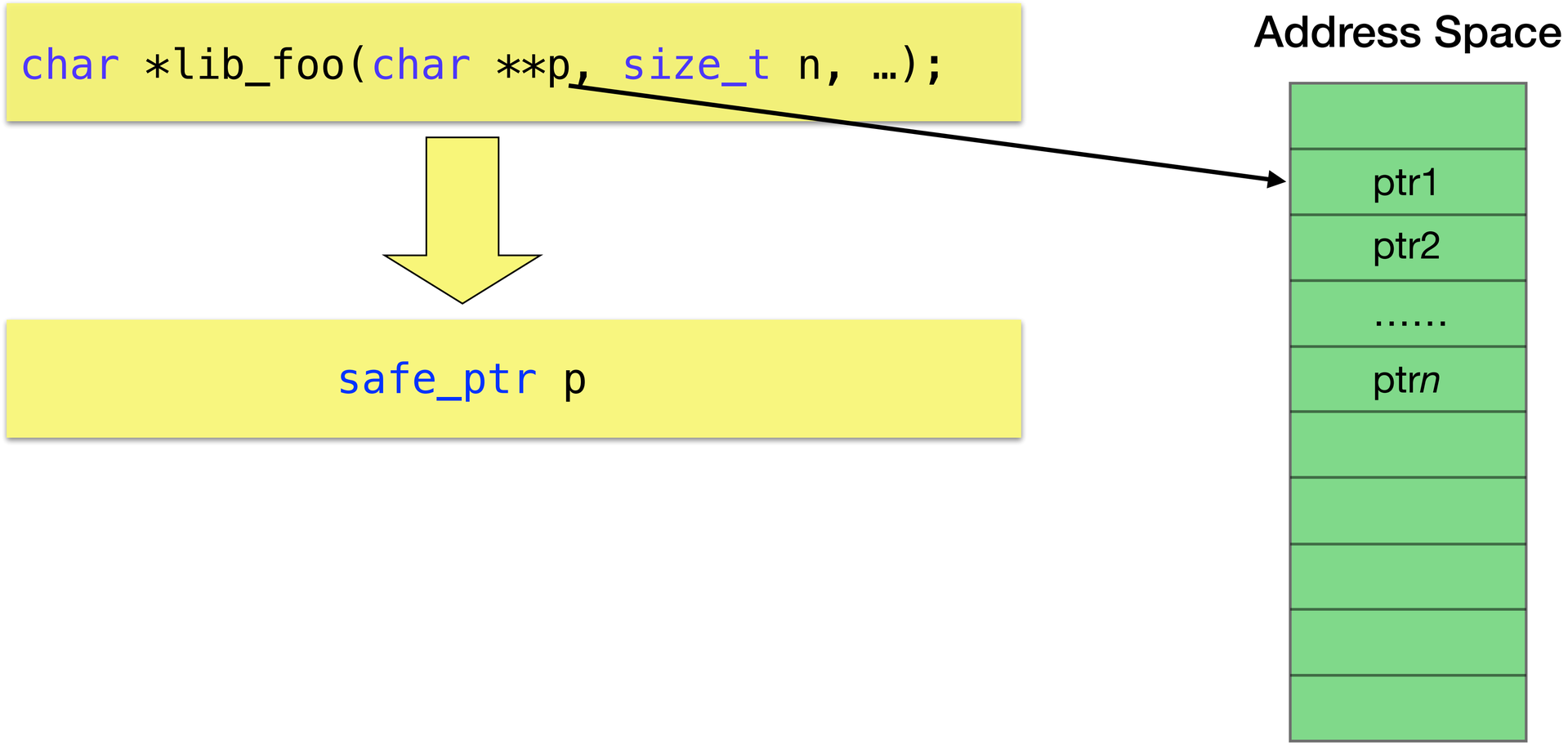
`strchr()` returns a raw pointers.
How to make it a safe_ptr?

Memory Layout Compatibility with Unchanged Code

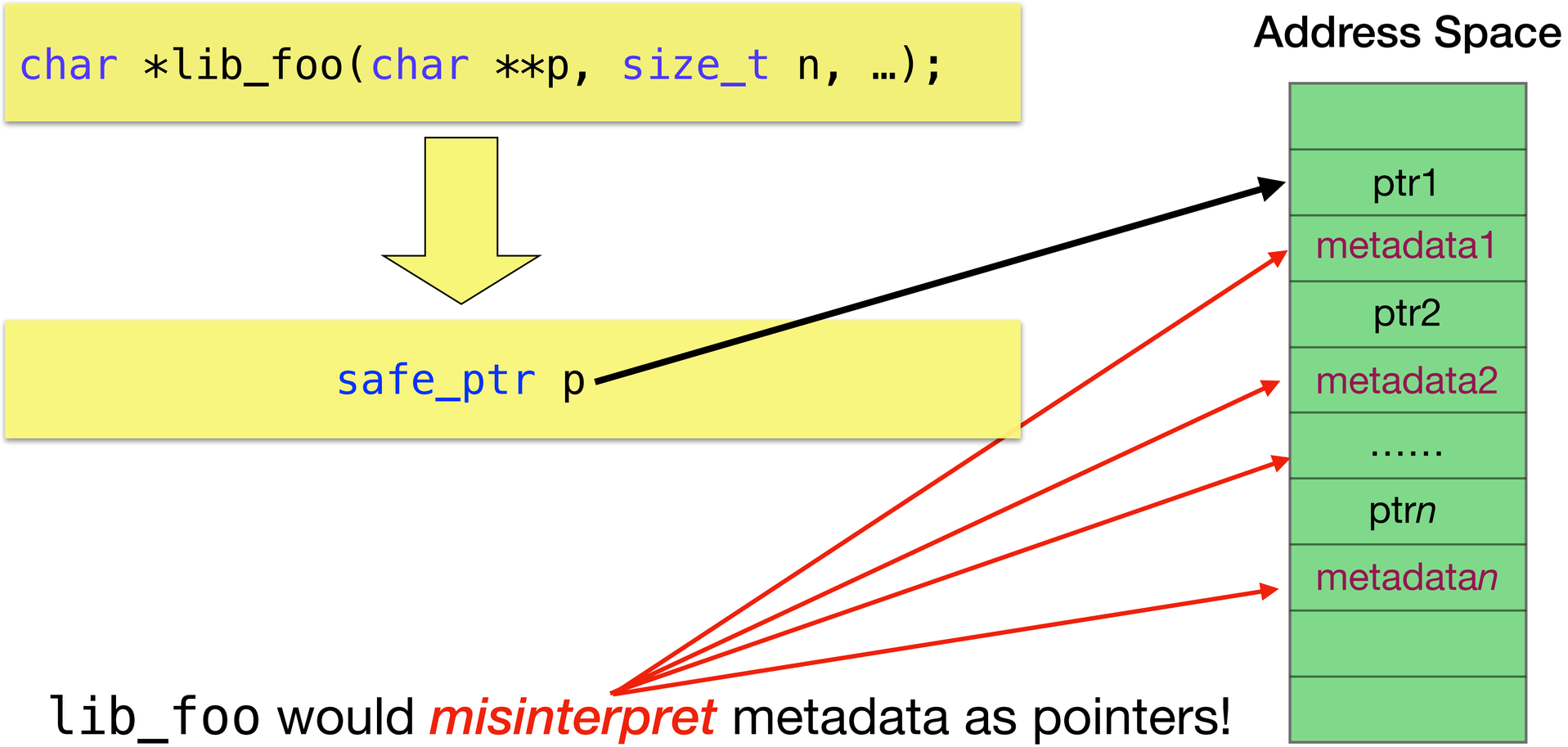
```
char *lib_foo(char **p, size_t n, …);
```



Memory Layout Compatibility with Unchanged Code



Memory Layout Compatibility with Unchanged Code



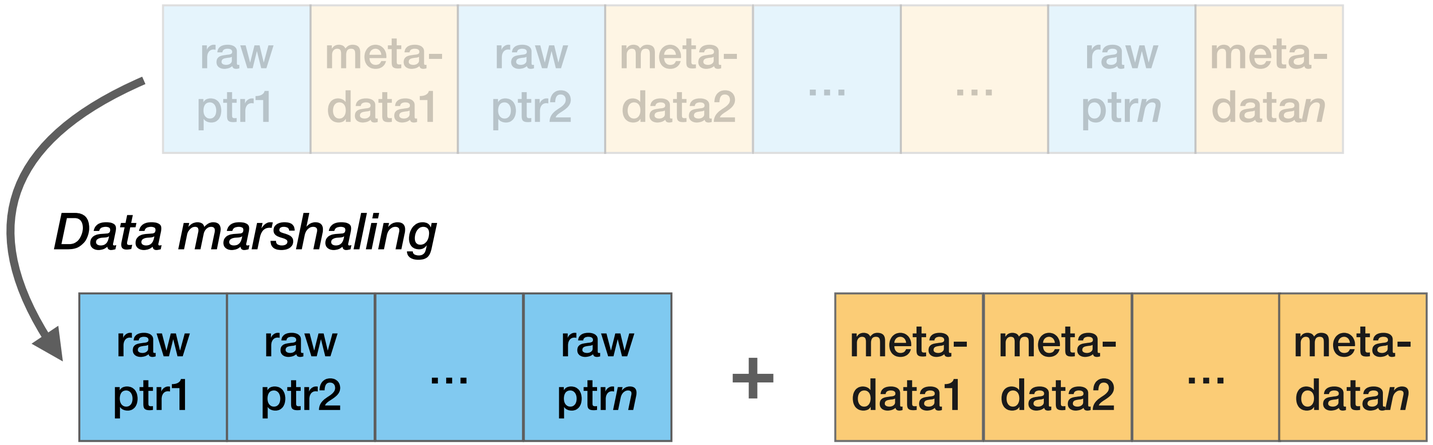
Solutions to Backward Compatibility Issues

- Porting library functions using fat pointers
  - Pros: complete and secure
  - Cons: high programmer effort; not always viable
- Mixing using raw pointers and fat pointers
  - Pros: easy to implement
  - Cons: security hazards
- Hybrid of disjoint and in-place metadata
  - Pros: secure
  - Cons: complex and slow

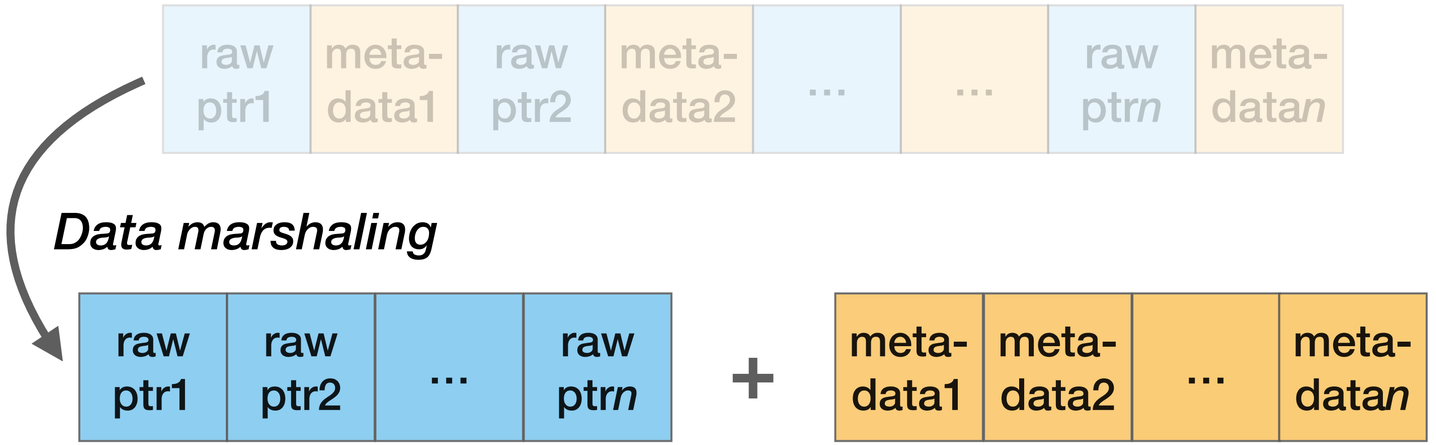
Data Marshaling



Data Marshaling



Two Important Questions on Data Marshaling



Q1. How much *programmer effort* is required?

Q2. What is the *performance penalty*?

SoftBoundCETS' Metadata Management

- Use a two-level lookup trie to locate the metadata
  - Use a pointer's address as the initial lookup index

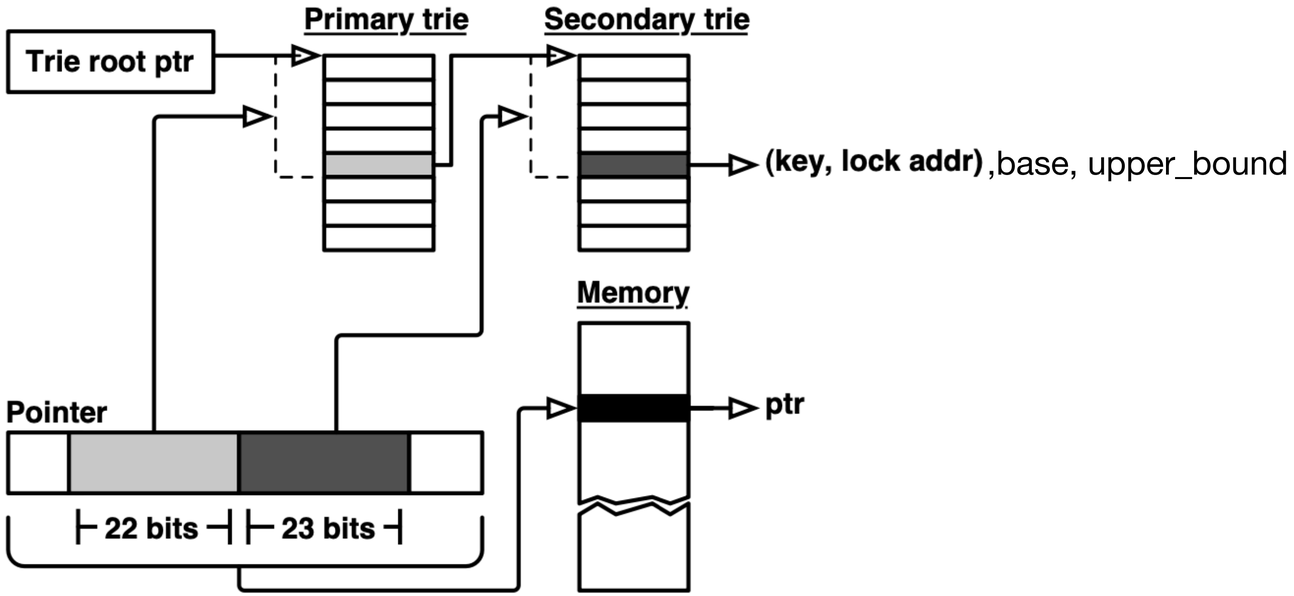


Figure 3. Organization of the trie for pointer metadata

Is the disjoint metadata scheme completely free of backward-compatibility issues?

Compatibility Issues with Disjoint Metadata Scheme

```
void qsort(void *base, size_t n, size_t width, int (*compar)(const void *, const void *));
```

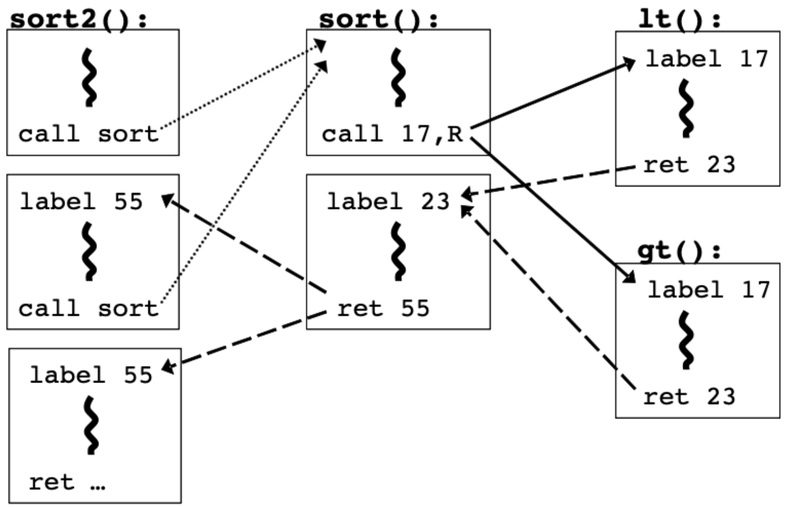
- `qsort`: Sort an array of `n` item, each of size `width`, using function `compar`.
  - E.g., Use a `greater_than()` function to sort an array of `n` integers.

Label-based CFI

- Assign and insert a label (ID) before each indirect transfer destination
- Before executing an indirect transfer, check the destination's label
  - Similar to using stack canaries / shadow stacks

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Direct forward transfer

Indirect forward transfer

Backward transfer

Compatibility Issues with Disjoint Metadata Scheme

```
void qsort(void *base, size_t n, size_t width, int (*compar)(const void *, const void *));
```

- `qsort`: Sort an array of `n` item, each of size `width`, using function `compar`.
  - E.g., Use a `greater_than()` function to sort an array of `n` integers.
- What about sorting an array of pointers?
  - SoftBoundCETS' metadata lookup procedure will not work!
    - It uses a pointer's address to index the metadata.
  - SoftBound's solution: Rewrite `qsort()` considering the associated metadata

Strengths and Weaknesses of Fat Pointers

- Good performance
  - Run-time can (very) quickly finds metadata to use.
- Difficult to interoperate with libraries that do not use fat pointers
  - Need wrappers to convert fat pointers to raw pointers and vice versa
  - Need to change memory layout of data structures

Uniform Treatment of Pointers

- Not all pointers are created equal.
  - e.g., most pointers to a singleton object do not need bounds check
- Many pointers to array do not requires bounds checks on both ends.
- Maintaining different metadata for pointers can be beneficial.