CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science
George Washington University

A2: Is Less Really More?

- Suggested future research directions:
  - Improve how we measure/evaluate the security of binaries
  - Improve debloating tools to generate more secure code in the first place

A3: Exploiting Format String Vulnerabilities

```c
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.

```c
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

Outline

- Review: Least Privilege Principle & Software Compartmentalization
- Address Sanitizer
- Pointer-based Memory Safety

Principle of Least Privilege

- What are privileges?
- What problems do current systems have with privileges?
- What can we do to more safely use privileges?

Privileges

priv·i·lege | ˈpriv(ə)lij |

noun

a special right, advantage, or immunity granted or available only to a particular person or group: *education is a right, not a privilege* | *[mass noun] : he has been accustomed all his life to wealth and privilege.*

- Override (i.e., make exceptions to) access control rules
- Usually a thread or process attribute

Why Do We Need Privileges?

- Real systems need “exceptions” to access control rules.
  - Installing new software
  - Change of policy
  - Change of ownership
  - Fix incorrect configurations
  - Help users solve problems

Privilege Granularity

Coarse-grained Privileges

- All or nothing
- Effective UID of 0 (root) overrides all access controls.

Medium-grained Privileges (Linux)

- Named “capabilities” in the Linux documentation
- Kernel checks for privileges in process’s effective privilege set
- Kernel provides “hack” to mimic Unix access control
  - Turns all privileges on when effective UID is root
  - Turns all privileges off when effective UID is not root
  - A process can disable this behavior using `prctl()`
- Linux kernel 6.17 uses 41 capabilities.
  - “cat /proc/sys/kernel/cap_last_cap” prints the last capability ID (start from 0)

Medium-grained Privileges (Linux)

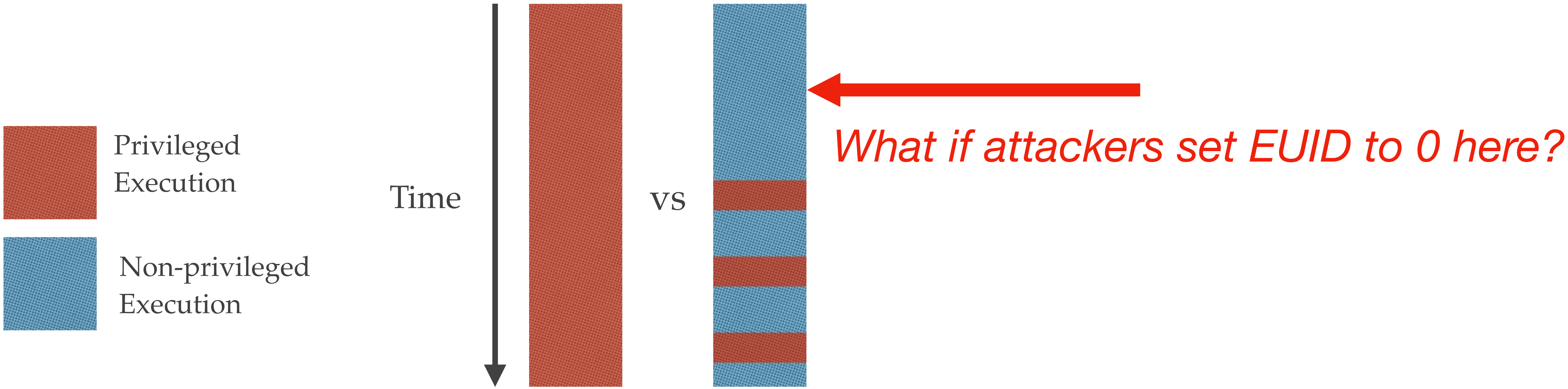
Privilege	Description
CAP_DAC_READSEARCH	Override read permissions on files Override search permissions on directories
CAP_DAC_OVERRIDE	Override read, search, and write access on files and directories
CAP_CHOWN	Change owner of files
CAP_SETUID	Change real, effective, and saved UIDs to any value

Turning Privileges On and Off

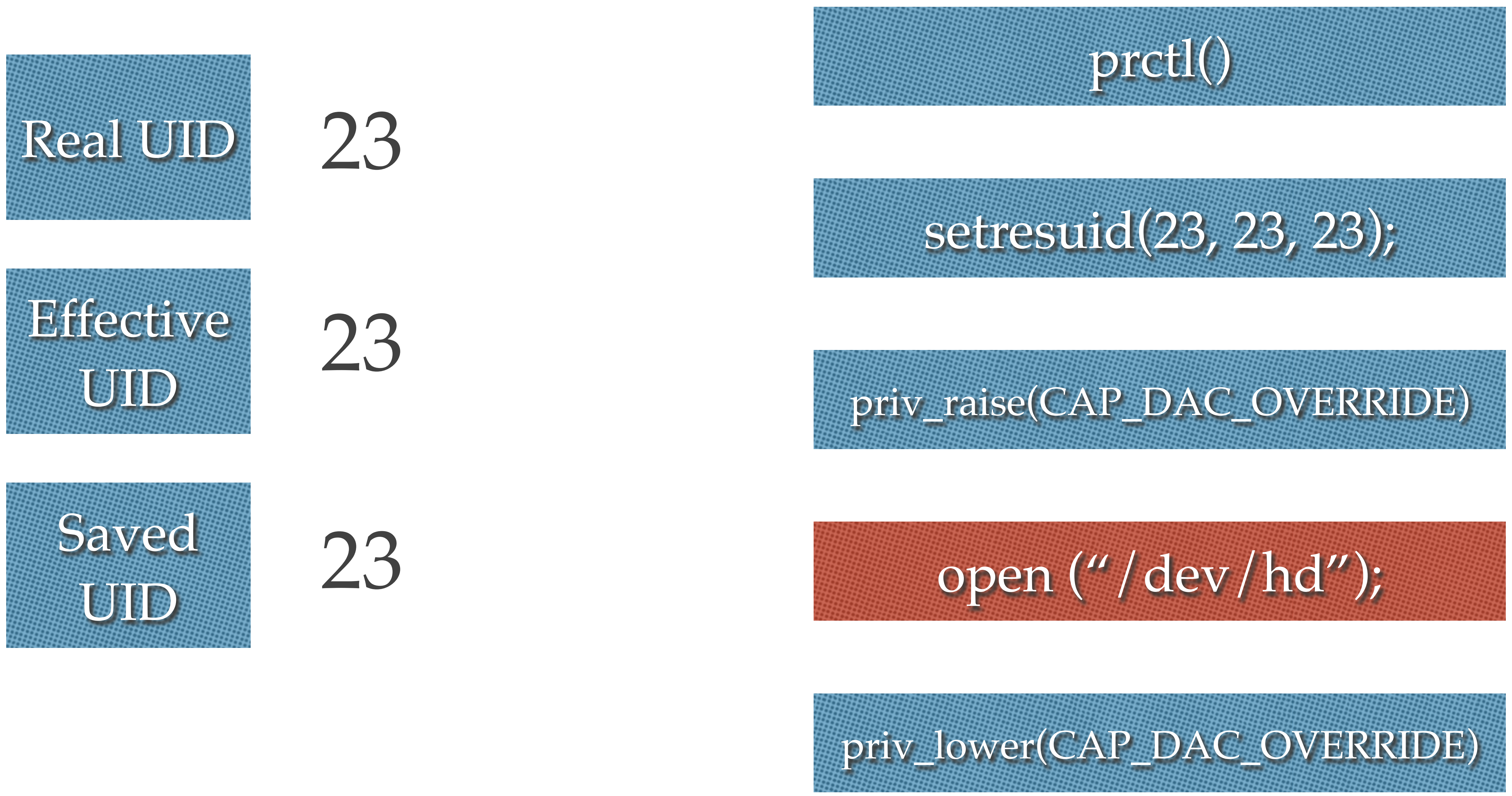
- Programs do not need all operations to be privileged.
  - The program's functionality may not need privileges.
    - Use privilege to open password file
    - Don't use privilege to open user preferences file
- Follow Saltzer and Schroeder Principle of Least Privilege
  - Programs using fewer privileges tend to have fewer vulnerabilities.

Privilege Bracketing

- Enable privileges before an privileged operation
- Disable privileges after the operation



Linux Privilege Bracketing



What if `priv_raise()` is exploited by attackers?

Process Privilege Sets

- Maximum privilege set: All allowed privileges
  - E.g., CAP_DAC_READSEARCH, CAP_DAC_OVERRIDE
- Effective privilege set: Currently active privileges
  - E.g., CAP_DAC_READSEARCH
- Process can add effective privilege if it is in maximum set.
- Process can remove privileges in maximum and effective.
- **Challenge: Removing unneeded privileges at the earliest point.**

Using Compiler to Enforce Least Privilege

- Programmers priv-bracket operations needing privileges
- Compiler analyzes and inserts calls to `priv_remove()`
- Model checker determines if system is capable of entering unsafe state.
  - Assume attacker can exploit memory safety errors, e.g. buffer overflows
  - Assume attacker can use system calls in program in any order
  - Measured amount of execution spent with each privilege set

Security Analysis Results

Program	Unsafe State			
	Read /dev/mem	Write /dev/mem	Privileged Port	Kill Process
passwd	100%	100%	0%	63%
ping	0%	0%	0%	0%
sshd	100%	100%	~0%	100%
su	88%	88%	0%	88%
thttpd	10%	~0%	10%	~0%

Why Do Programs Use Privileges Ineffectively?

- Programs originally designed for root user
  - No reason to design program with least privilege
- System files are owned by root.
  - Privileged processes access subset of system files.
  - Root processes can access all system files even with no privileges!
- Privileges needed late in execution

Summary of Refactoring Strategies

- Switch UID early in execution to owner of needed files
  - Use privilege to place file owner UID in saved UID
  - Remove `CAP_SETUID` from maximum privilege set
  - Can switch to file owner using `setresuid()` with no privilege
- Change owner of files to a unique unused UID
  - Unprivileged process can now open only needed files
  - Other system files inaccessible

Refactored Security Analysis Results

Program	Unsafe State			
	Read /dev/mem	Write /dev/mem	Privileged Port	Kill Process
passwd	100%	100%	0%	63%
Refactored passwd	**4%**	**4%**	**0%**	**4%**
su	88%	88%	0%	88%
Refactored su	**1%**	**1%**	**0%**	**1%**

Principle: Fault Compartmentalization

Separate individual components into smallest functional entity possible.

- These units contain faults to individual components.
- Allows abstraction and permission checks at boundaries.
- In practice, smallest functional entity can be too expensive.

A chatting app's image processing module and audio processing module are compartmentalized so that bugs in one module will not affect another.

Isolation vs. Compartmentalization

- Isolation is a fundamental *mechanism*.
  - Emphasizes barriers/walls between components; often no intended interactions
  - E.g., between processes, VMs, untrusted libs
- Compartmentalization is a design strategy/policy that often uses isolation.

Compartmentalization builds on least privilege and isolation. Both properties are most effective in combination: many small components that are isolated and running and interacting with least privileges.

Policy Definition Method (PDM)

PDMs identify subjects, objects, and permissions to enforce.

- Automation
- Policy languages
- Separation granularities
- Analysis techniques
- Subject selections
- Generality

Compartmentalization Abstractions

Key factors to consider

- A model of actions
- Trust models
- Target properties to enforce
- Composing with other abstractions

Enforcing Compartmentalization

- Software-based
  - E.g., SFI, AutoPriv
- Hardware-based
  - E.g., page-based, MPK
- Hybrid
- Other considerations
  - Granularity: byte-level to entire physical memory
  - Supported number of domains
  - Performance

Evaluating Compartmentalization

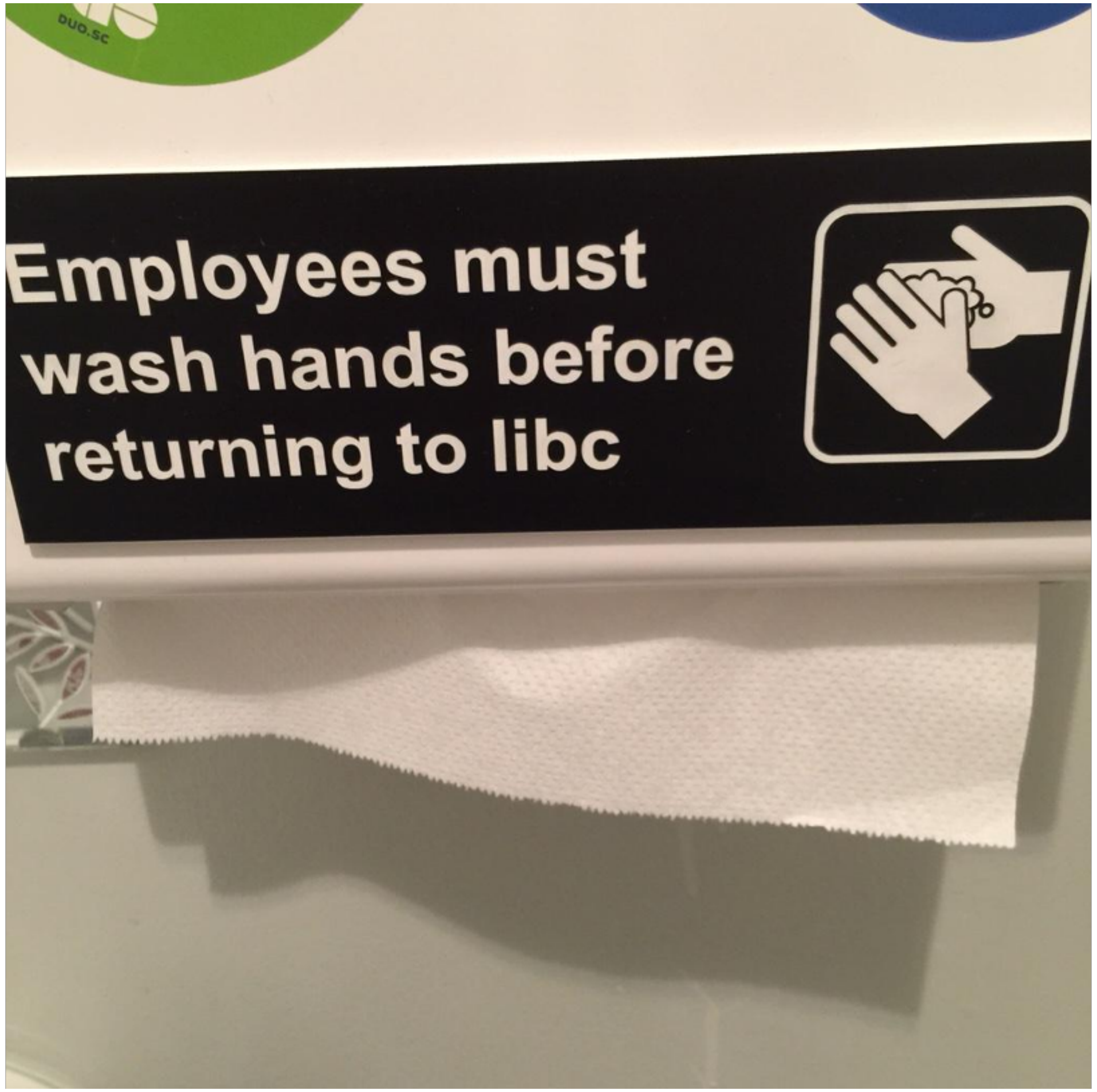
- Security benefits
- Performance compared to a monolithic design
- Compatibility with existing software and programming idioms
- Usability of separated software

A6: Group Project: Paper Presentation and Discussion

- Four groups; each group picks one paper from a provided list.
  - Other papers are allowed, as long as it is approved by the instructor.
- Deadline 1: Pick a paper by 11/19, Wednesday.
- Presentation (10 points)
  - Students and the instructor will ask questions during the presentation.
  - Everyone should contribute *equally*.
  - Everyone in the same group receives the same score for slides and presentation.
  - Teamwork evaluation is worth 2 points.
- Deadline 2: 12/1 before class.
- One presentation on 12/1 and three on 12/8.

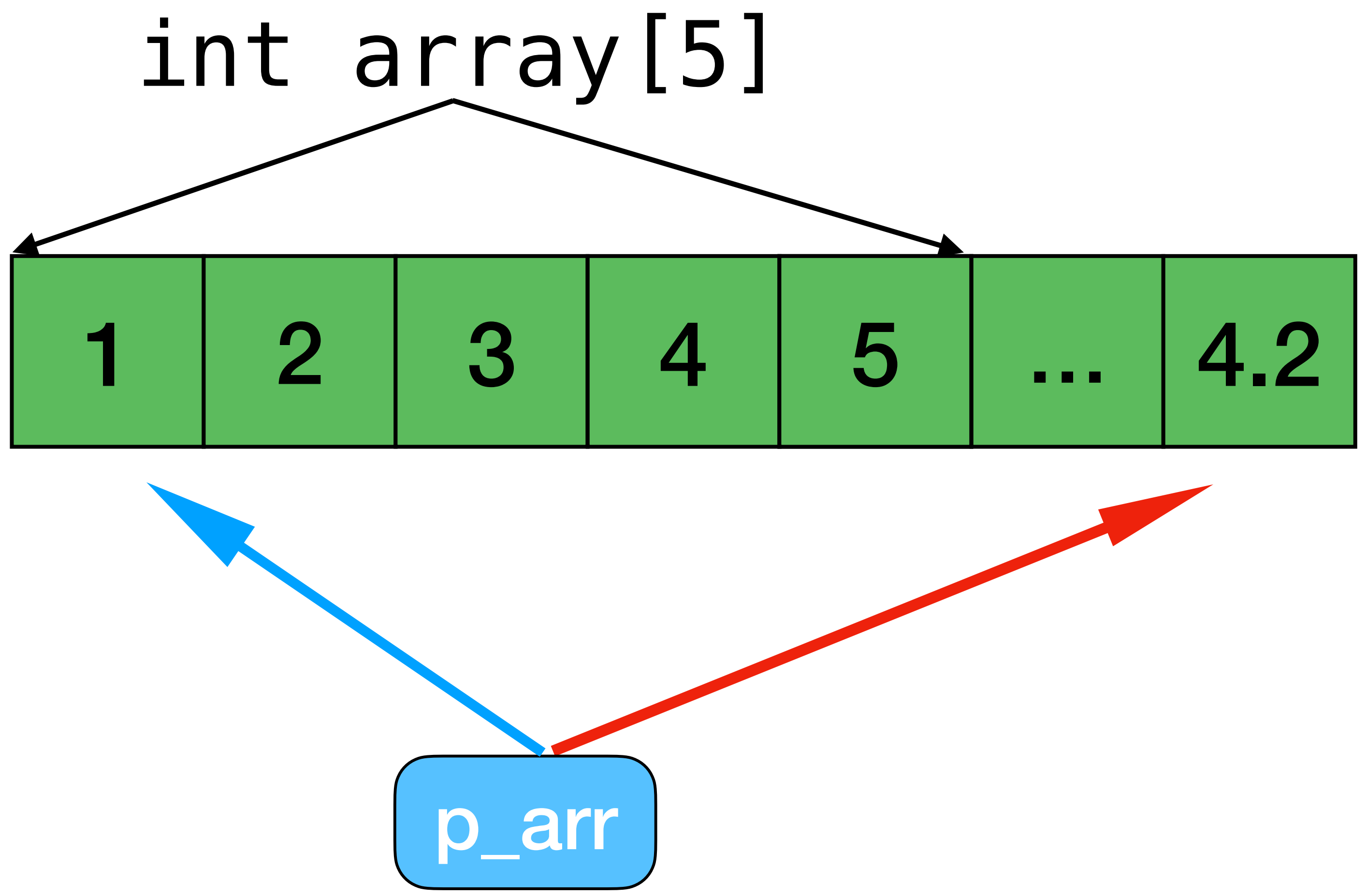
Sanitizers

Memory Sanitizers

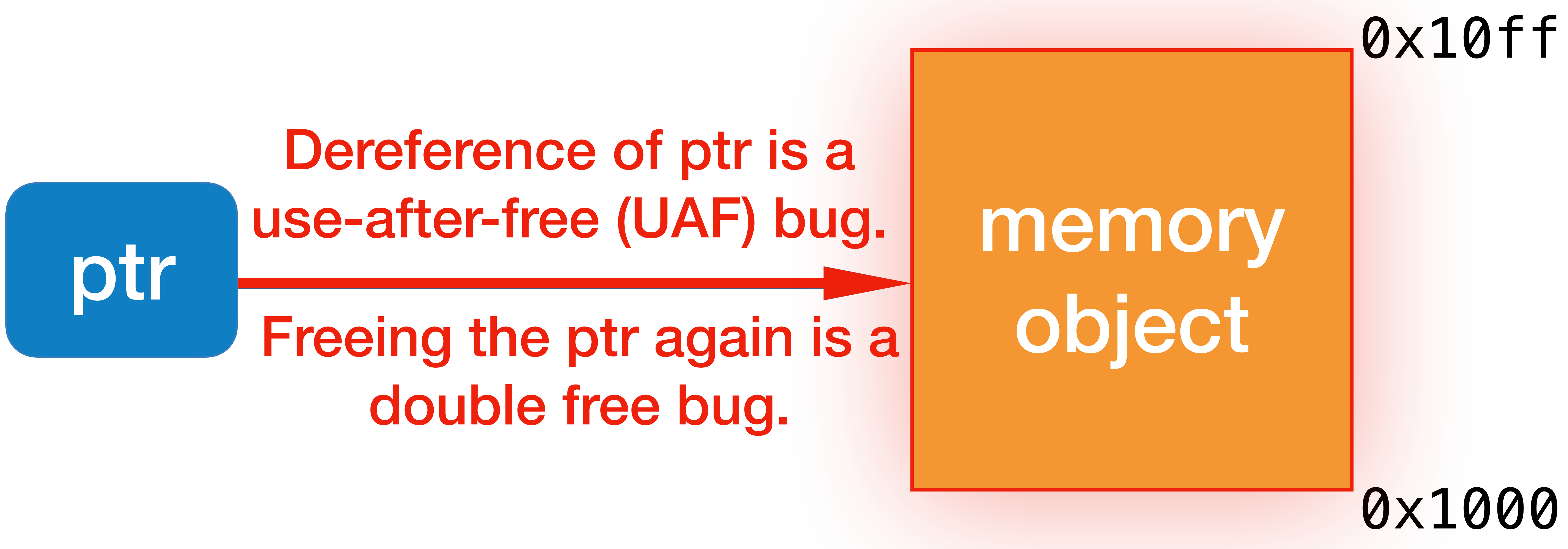


Spatial Memory Safety Bugs: Buffer Overflows

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:

Catch 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.

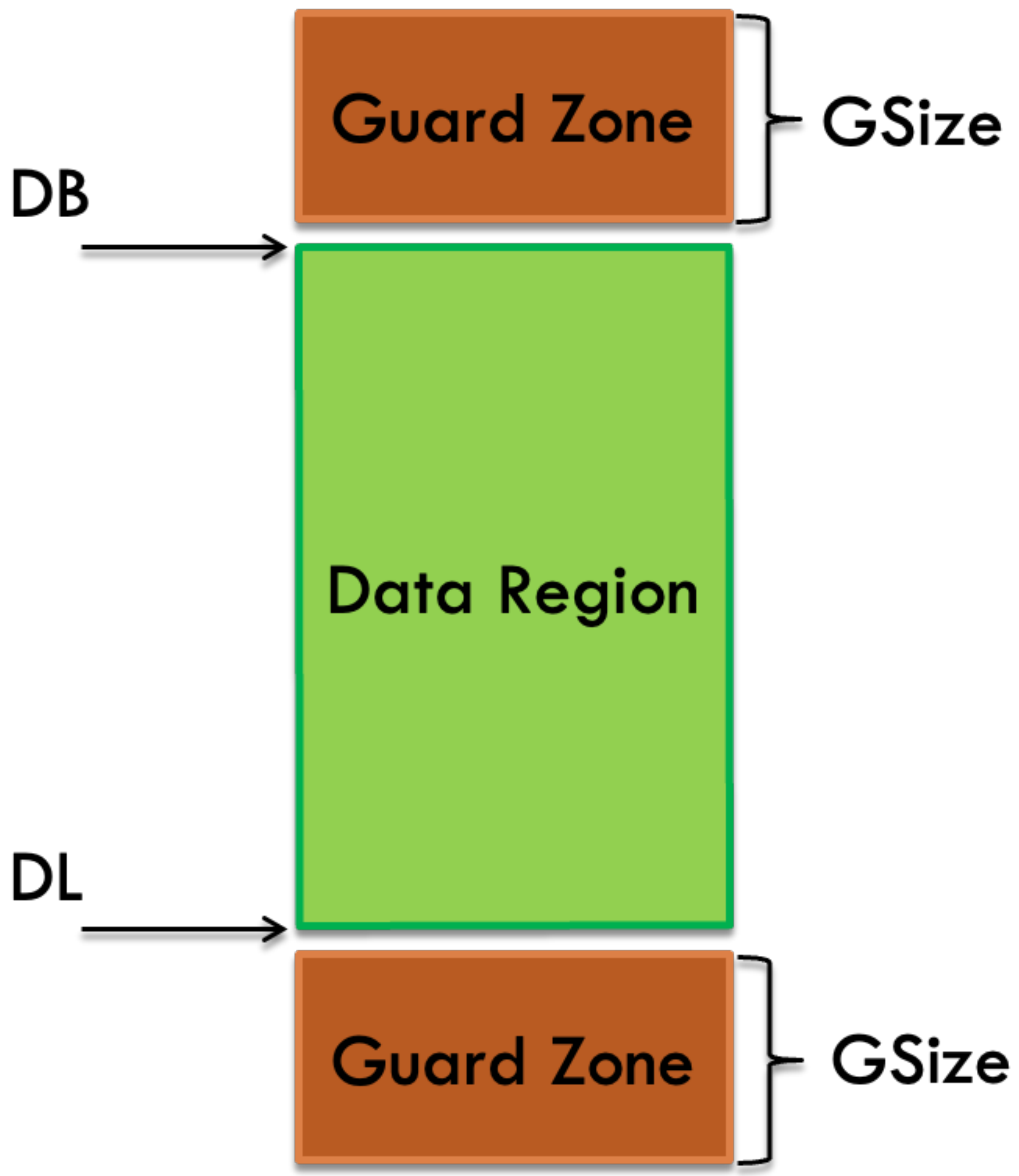
Address Sanitizer (ASan)

- 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

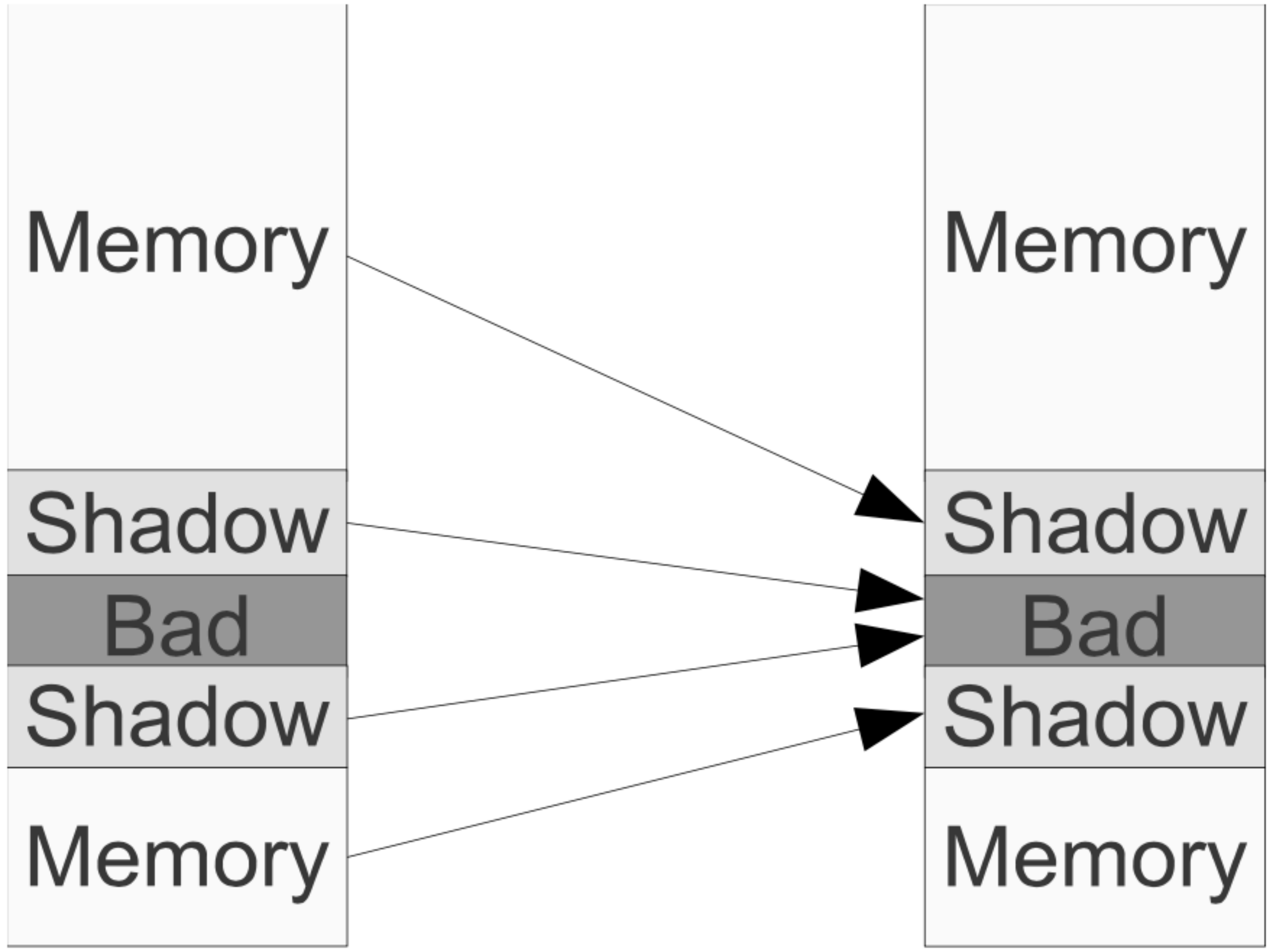
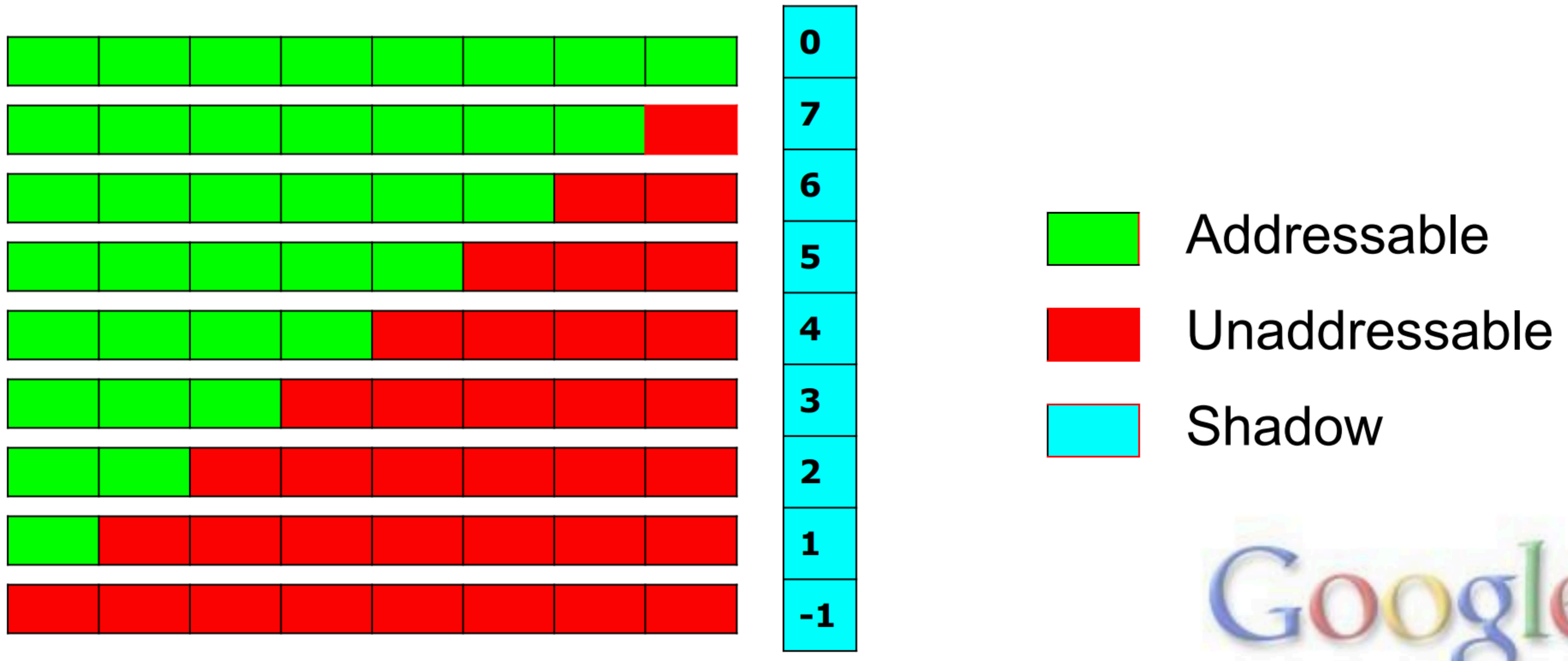


Figure 1: AddressSanitizer memory mapping.

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

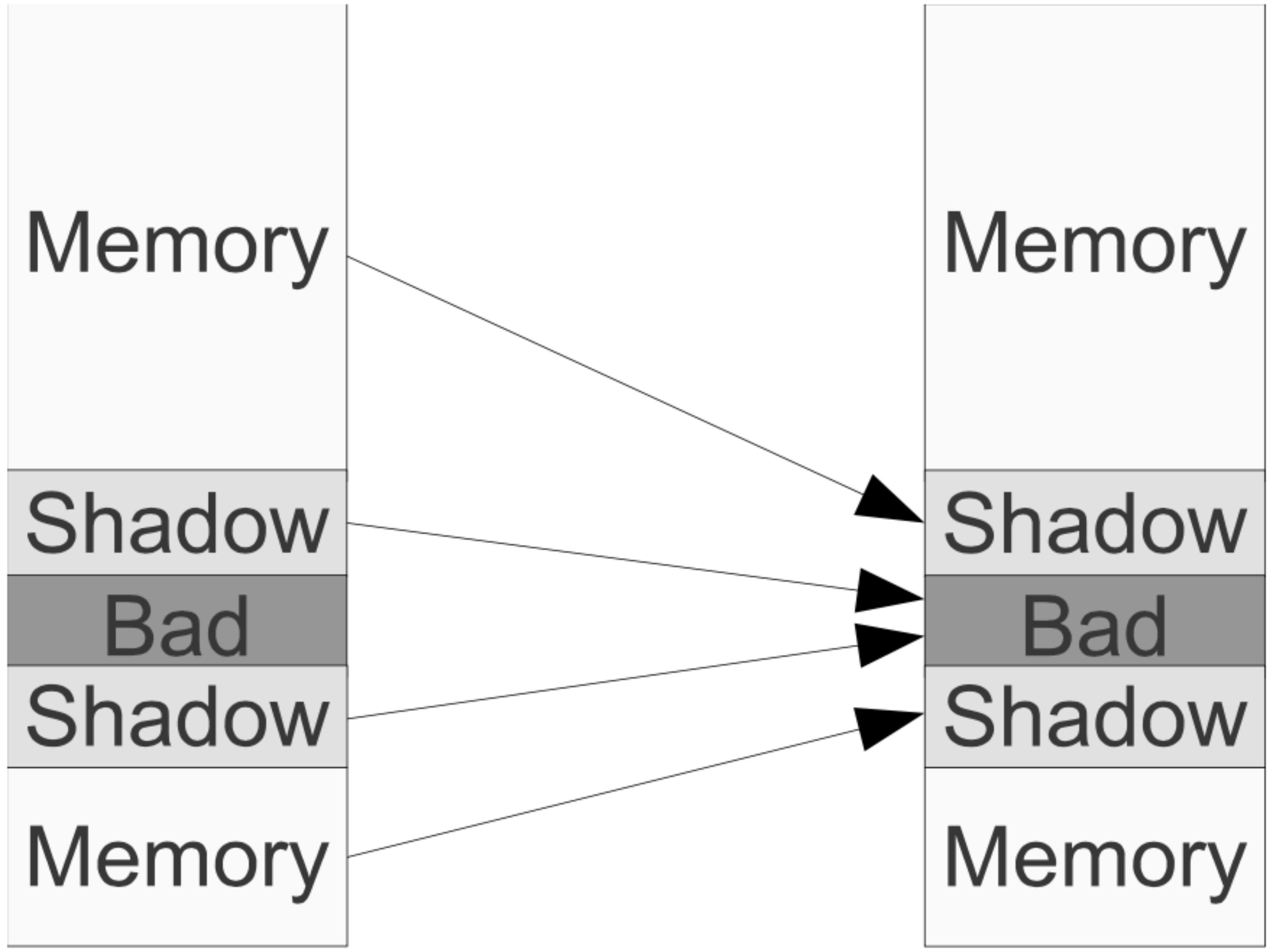
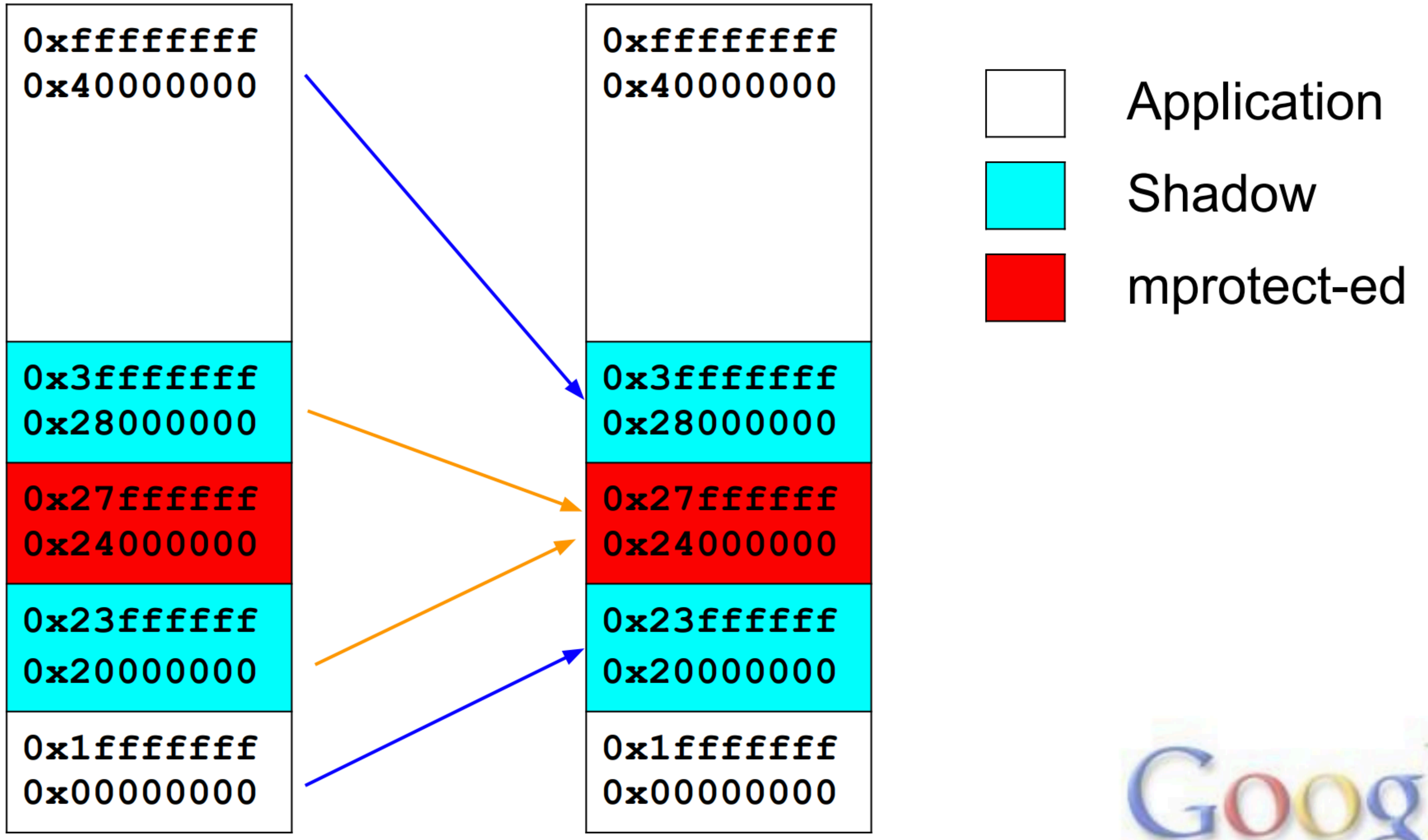


Figure 1: AddressSanitizer memory mapping.

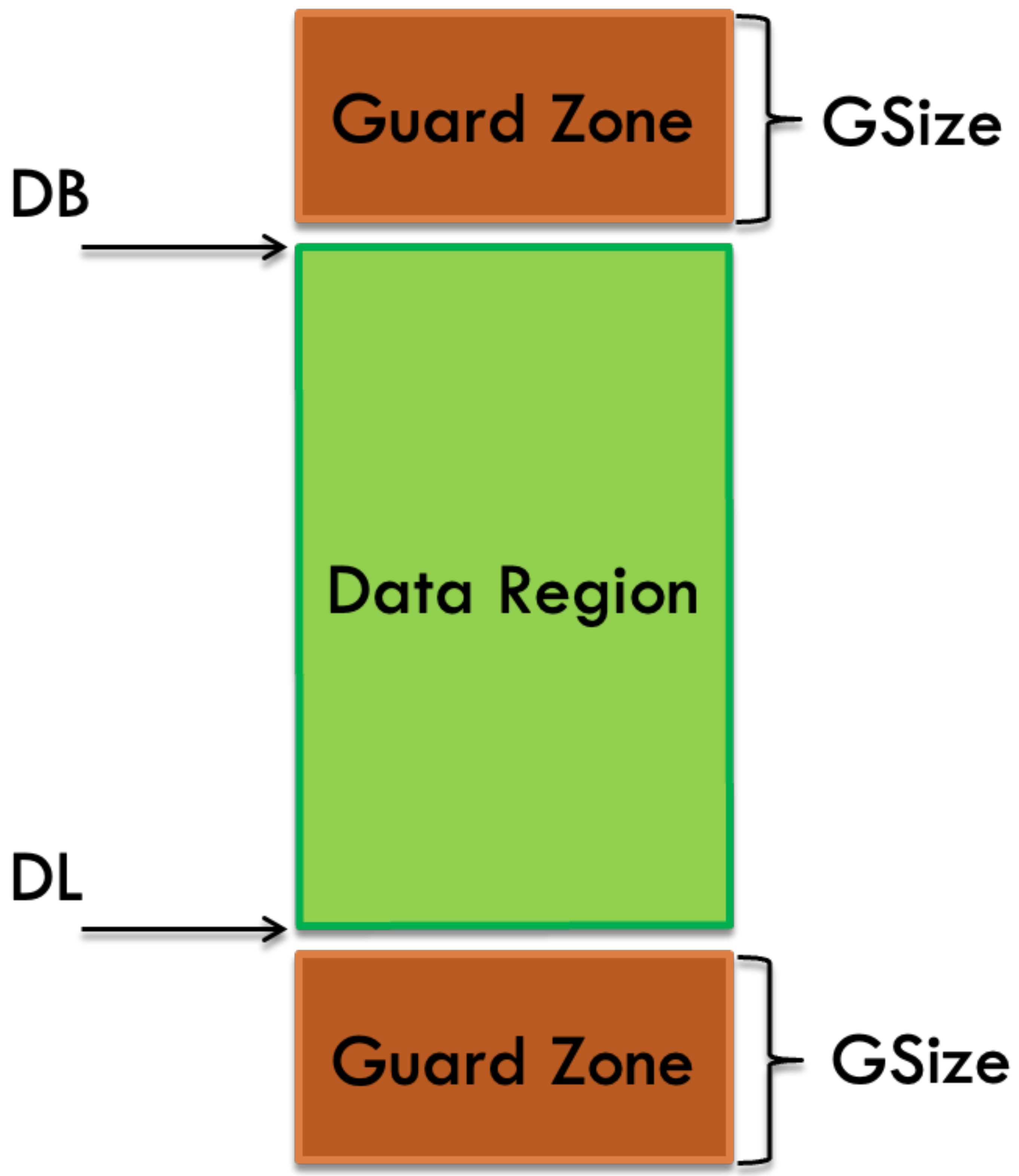
An Example of Shadow Memory for x86-32 Linux



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.

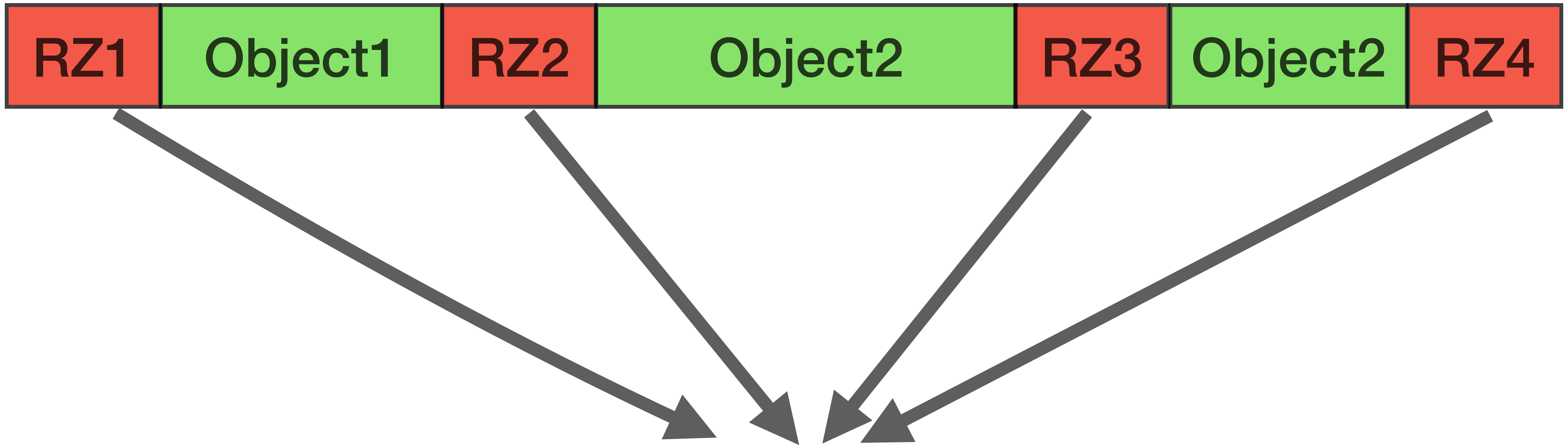
Optimization: Guard Zone/Page



- 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.

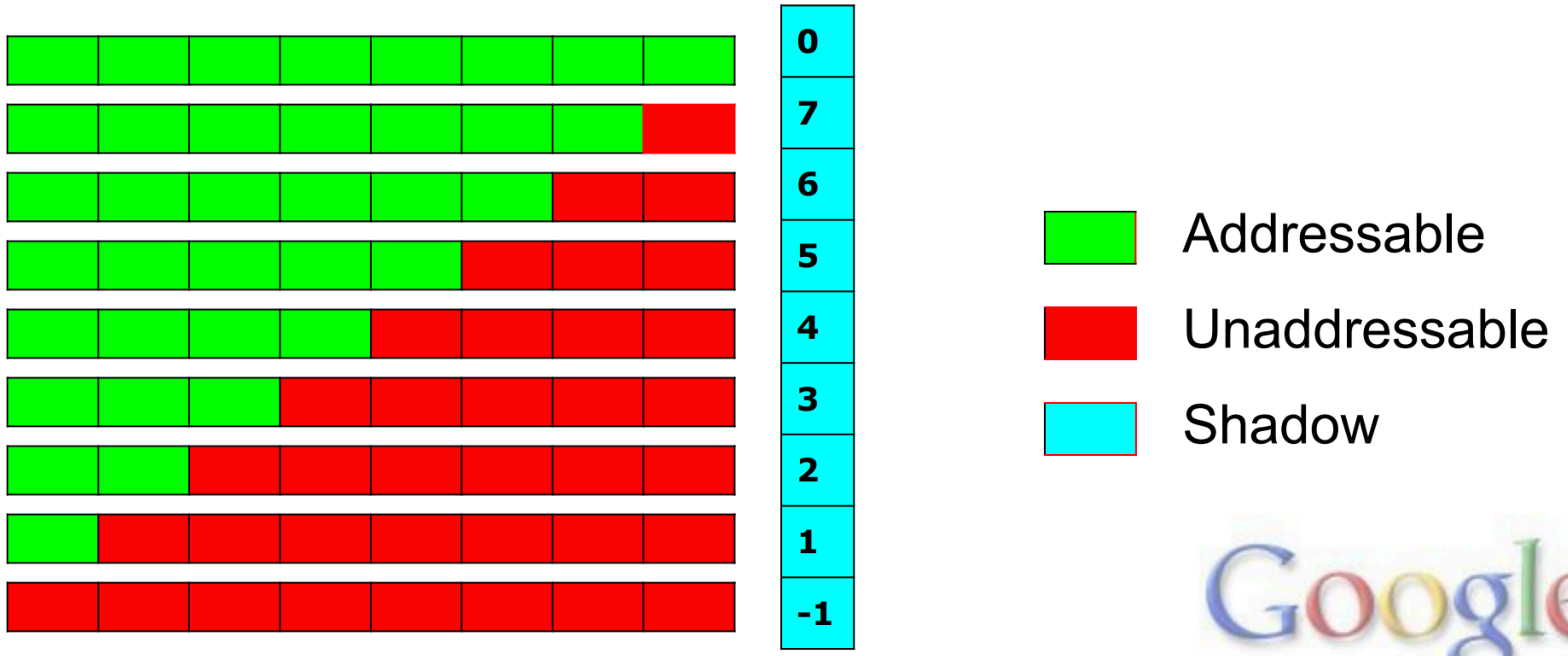
Instrumentation: 8-byte Access

- 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;
```

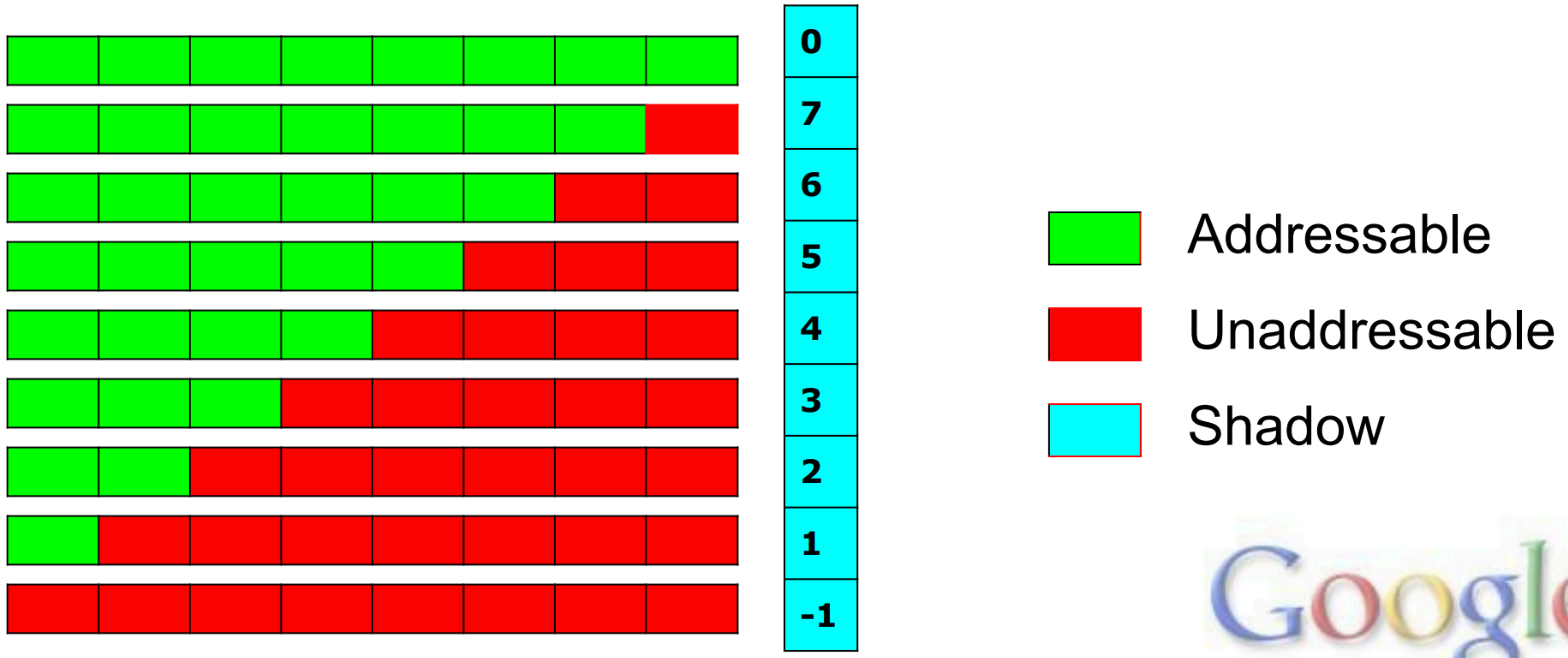


Instrumentation: N-byte Access (N = 1, 2, 4)

- Accessing an k-byte value from address p

```
… *p
```

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



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 by The 2012 Paper)

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

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:

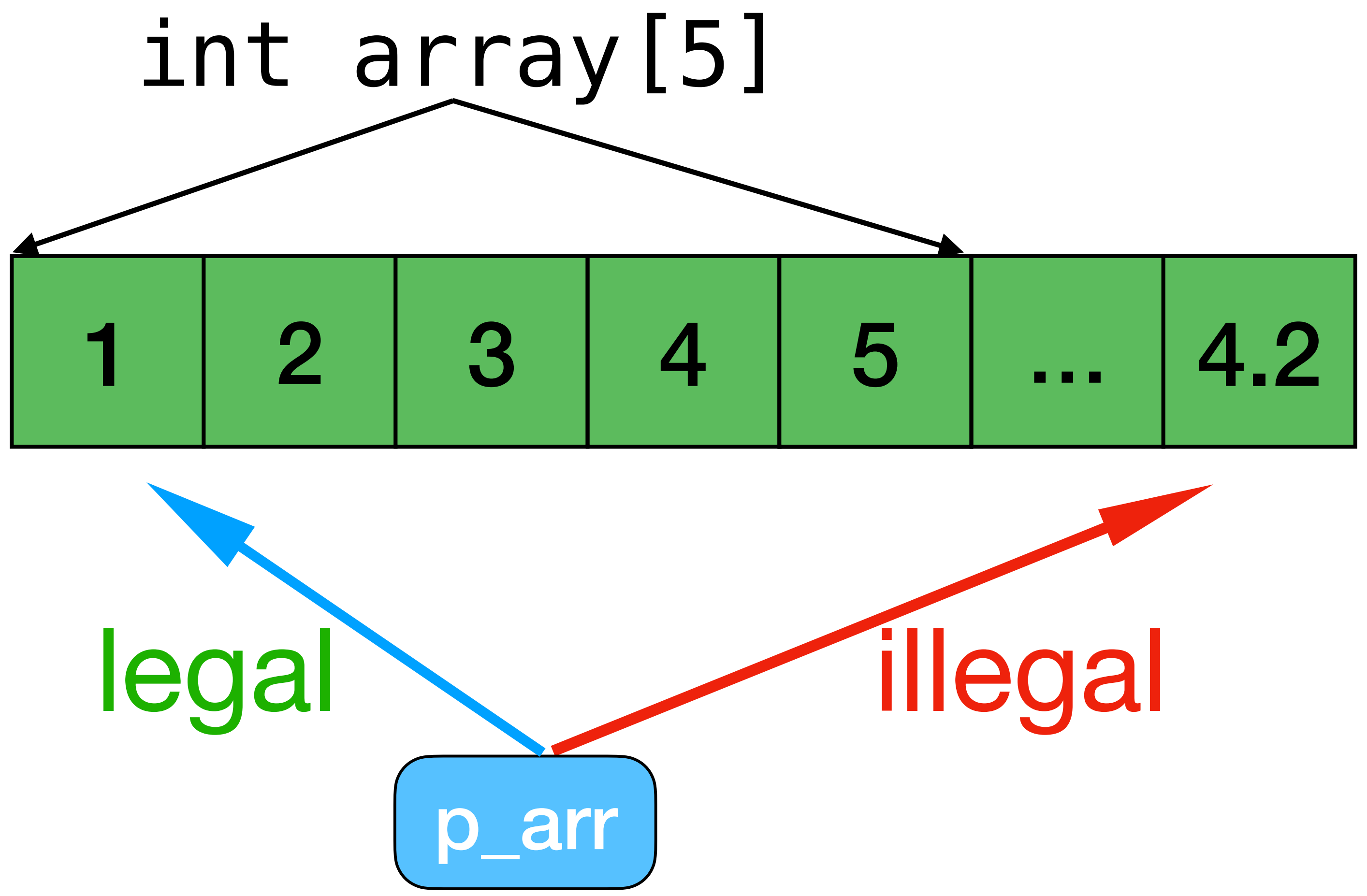
How to manage safety metadata?

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.



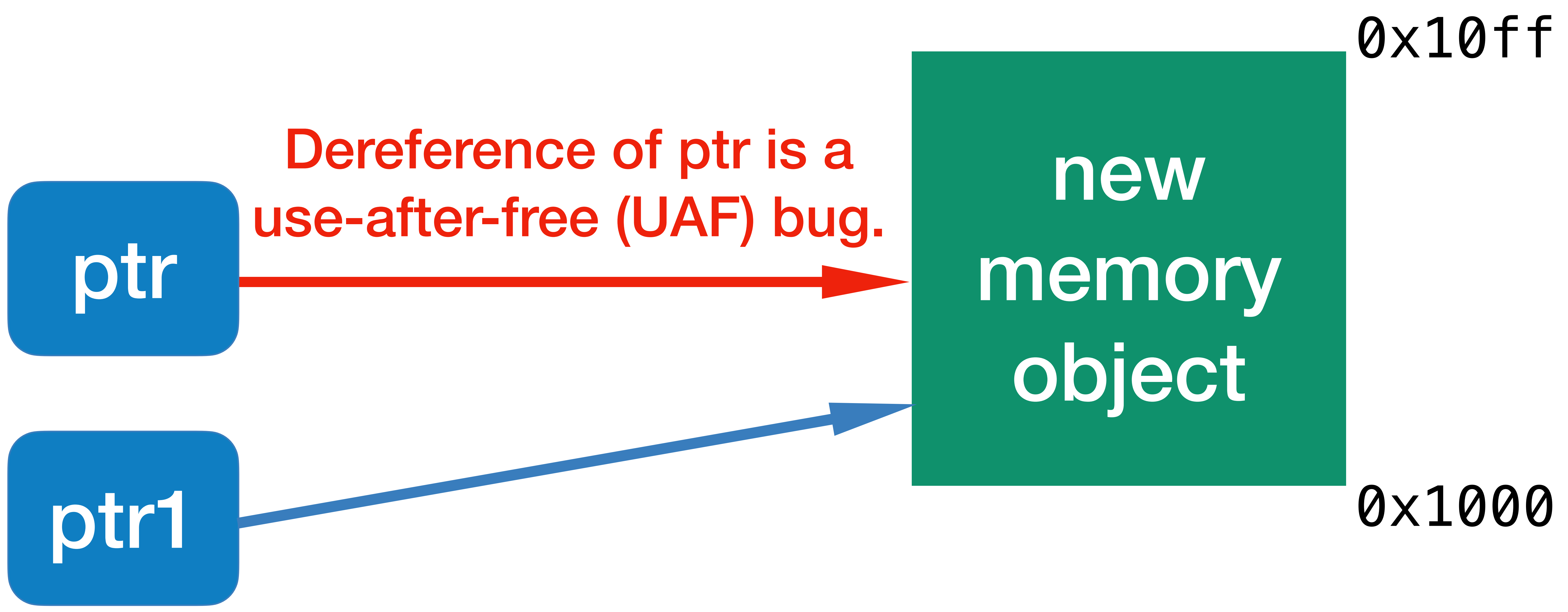
Essential information:

- 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

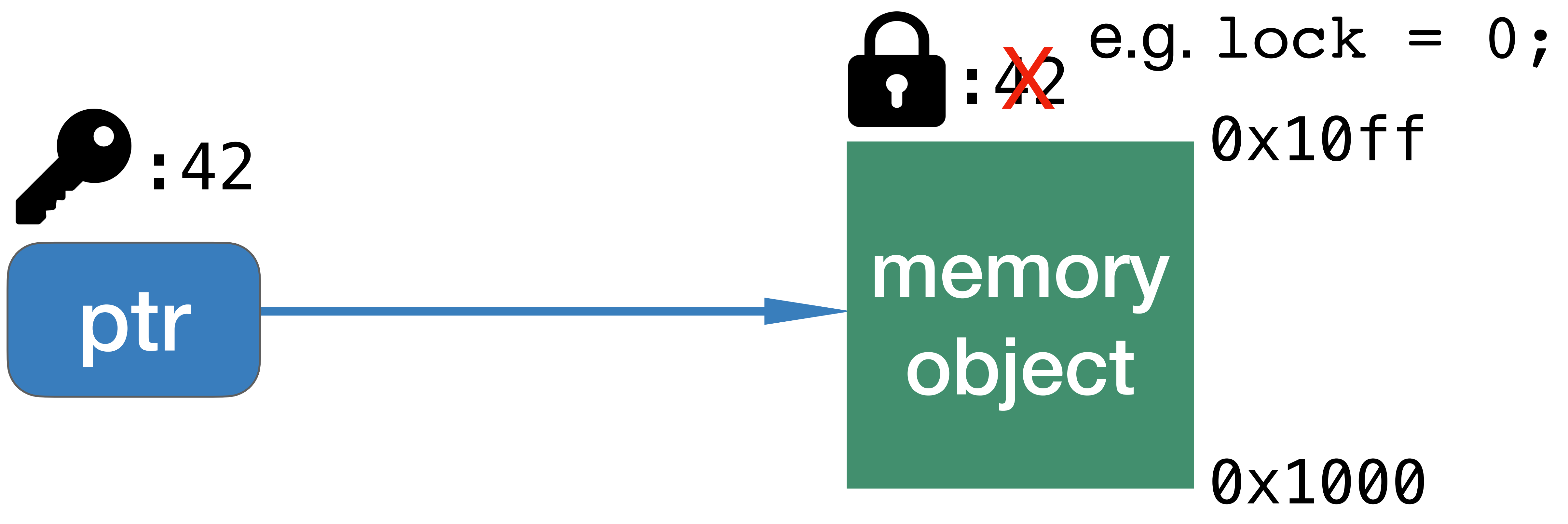
Temporal Memory Safety Bugs



Security risks

⚠ Information leaking
⚠ Data corruption
⚠ Denial of service

Key-lock Checking for Temporal Memory Safety



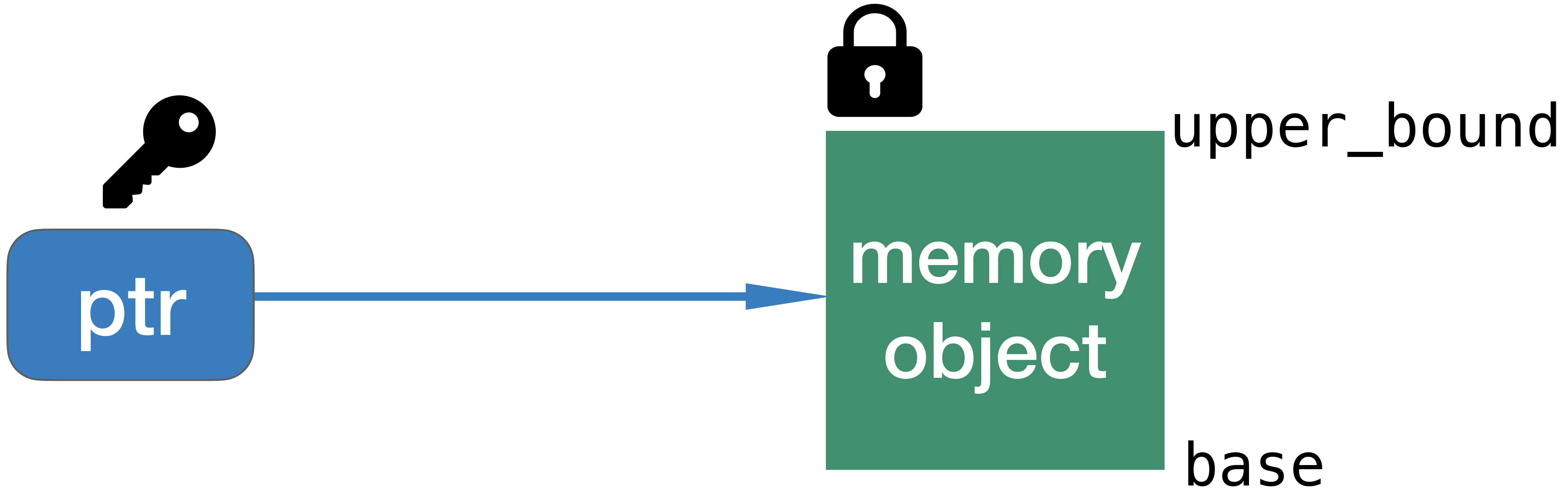

```
check_if_key_matches_lock(ptr);
ptr->num = 30;
```

- 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 these metadata?

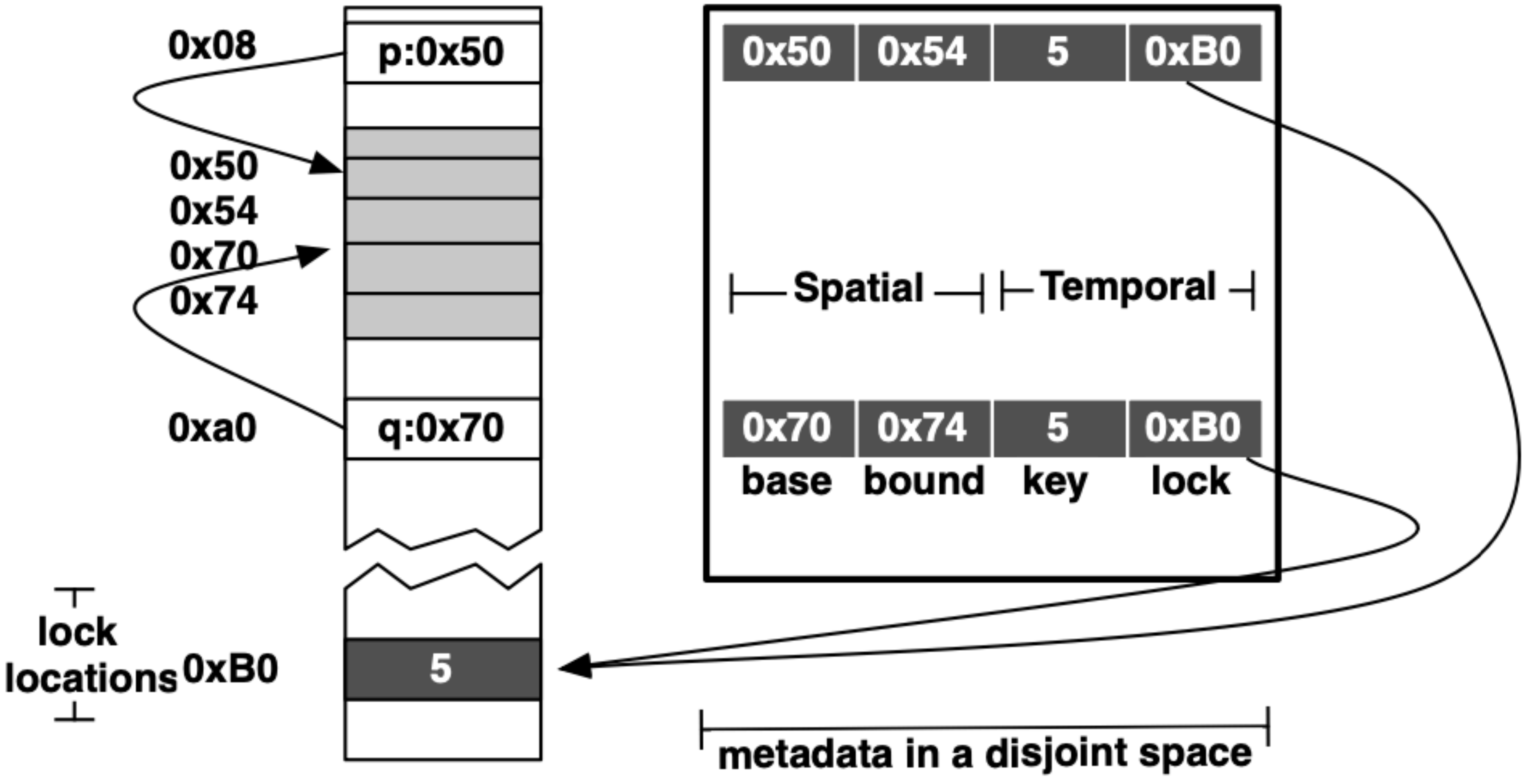
Managing Pointer’s Metadata

- 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: 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

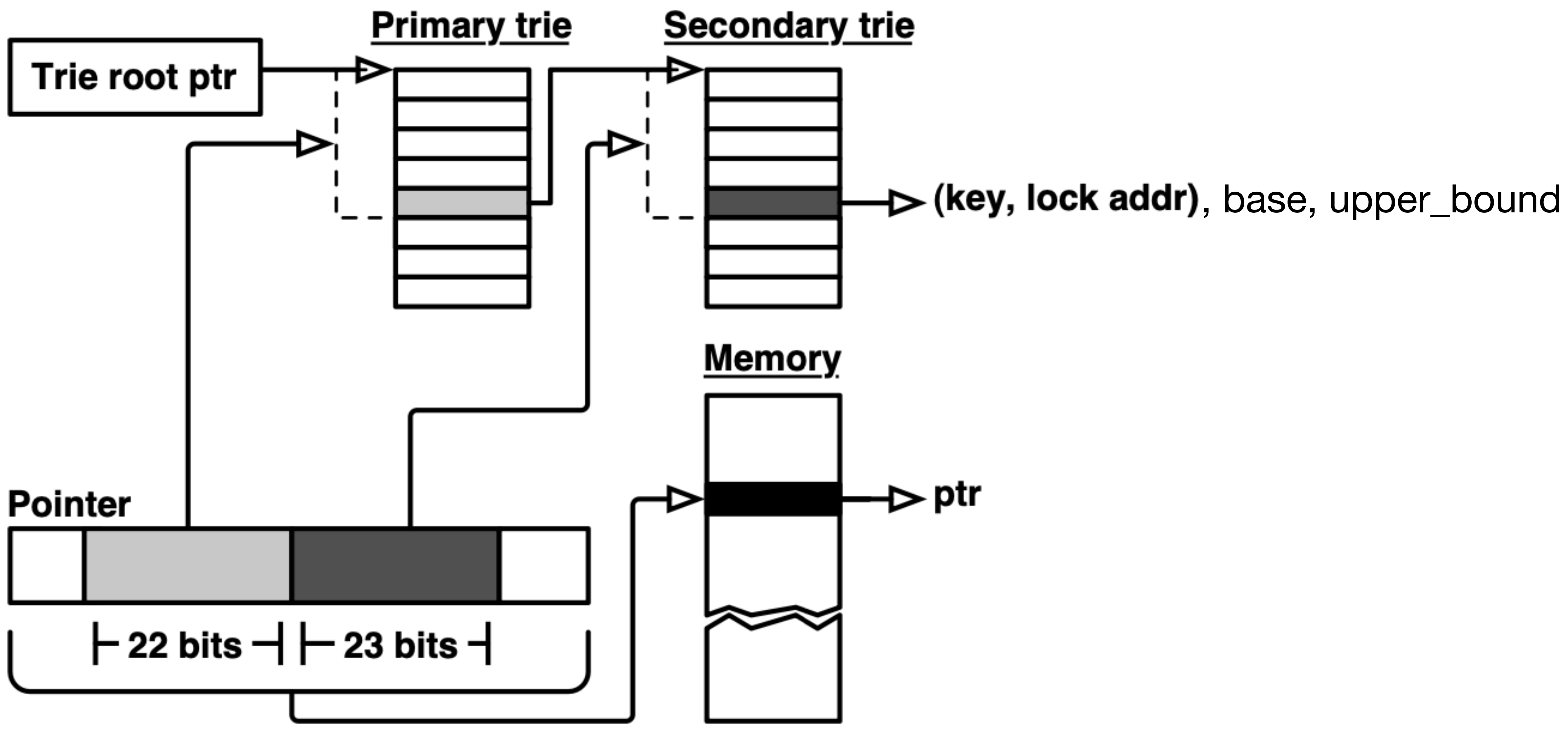


Figure 3. Organization of the trie for pointer metadata

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) <u>Temporal Check</u>

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

(d) <u>Spatial Check</u>

```
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.
  - 140% on 8 C SPEC CPU2017 benchmarks, reported in 2024, based on LLVM-12

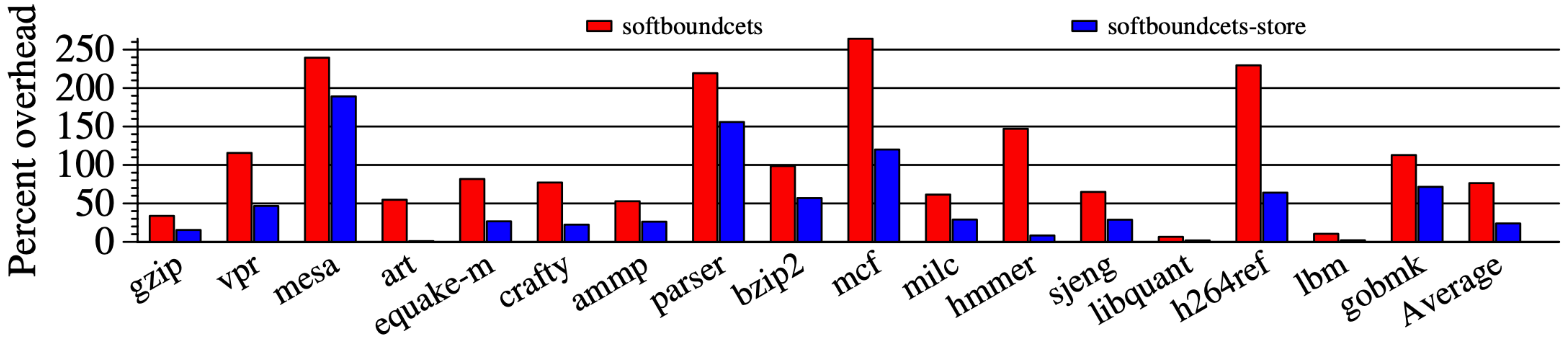


Figure 6 Runtime execution time overheads of the SoftBoundCETS compiler prototype with comprehensive memory safety checking (left bar of each stack) and while checking only stores (right bar of each stack) on a Intel Haswell machine. Smaller bars are better as they represent lower runtime overheads.

Why is this approach slow?

SoftBoundCETS' Metadata Management

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

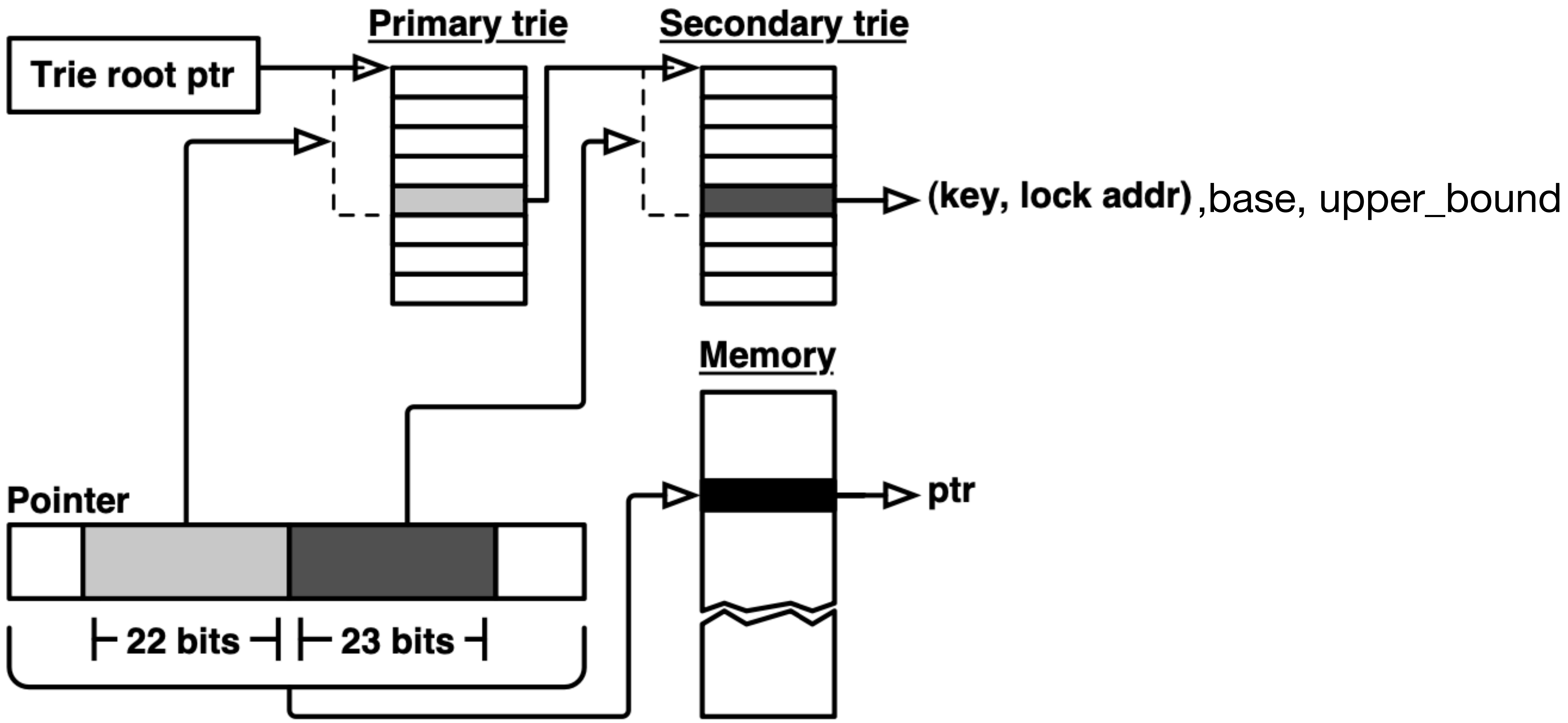
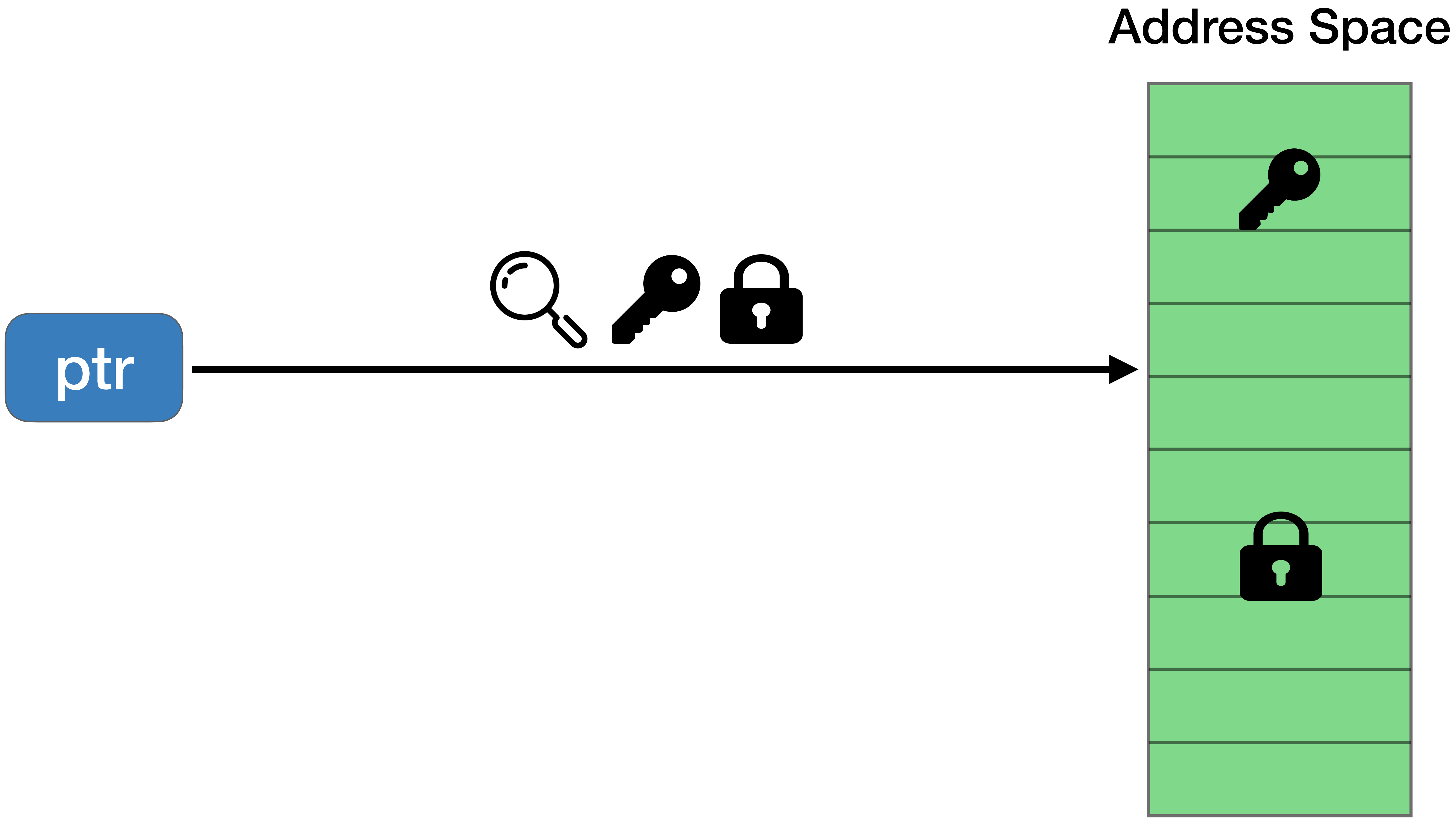


Figure 3. Organization of the trie for pointer metadata

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

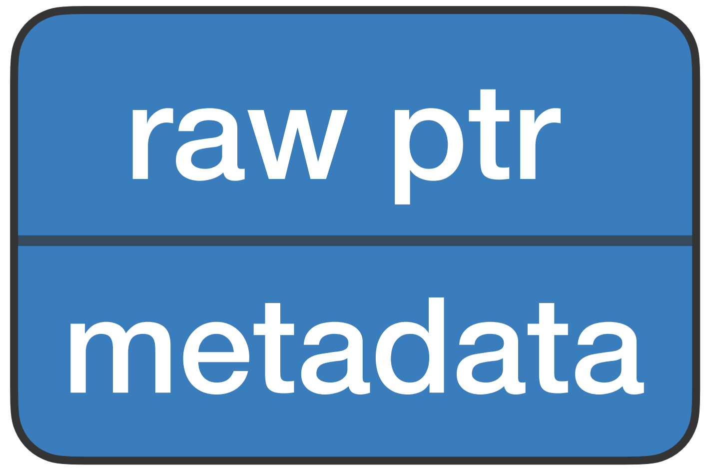
Disjoint Metadata Management is Slow



Fat Pointers

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

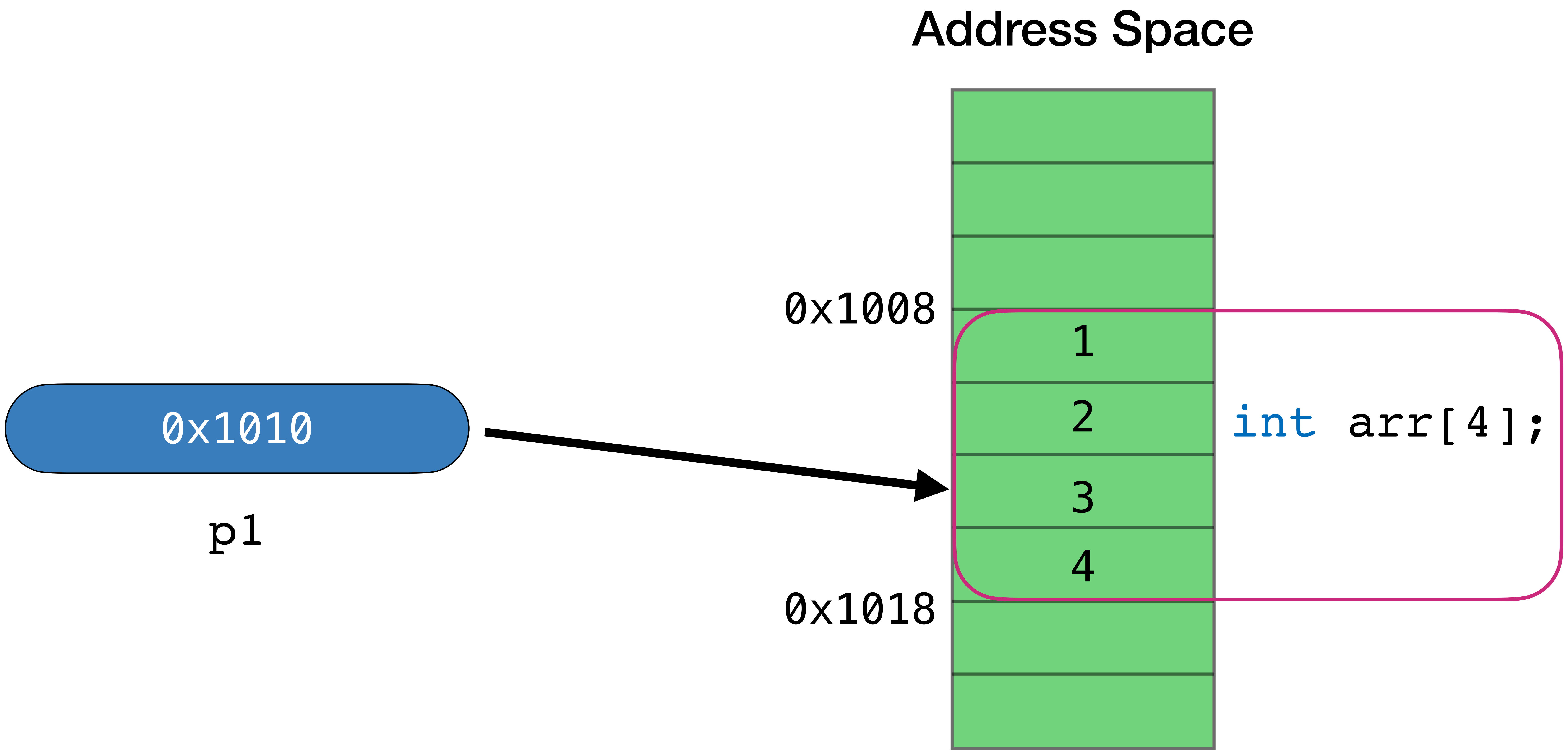
Fat pointer:



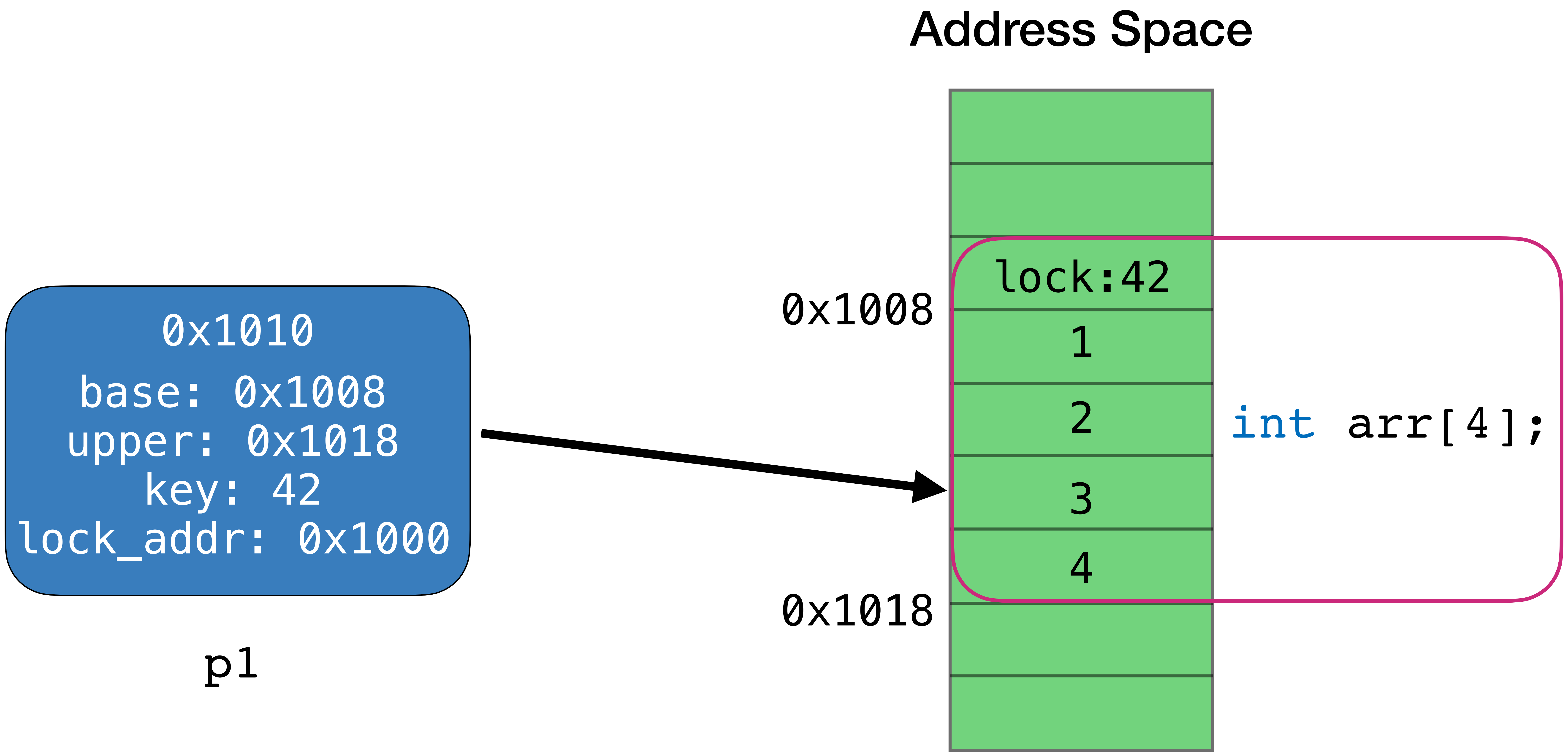
- One implementation

```
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;
};
```

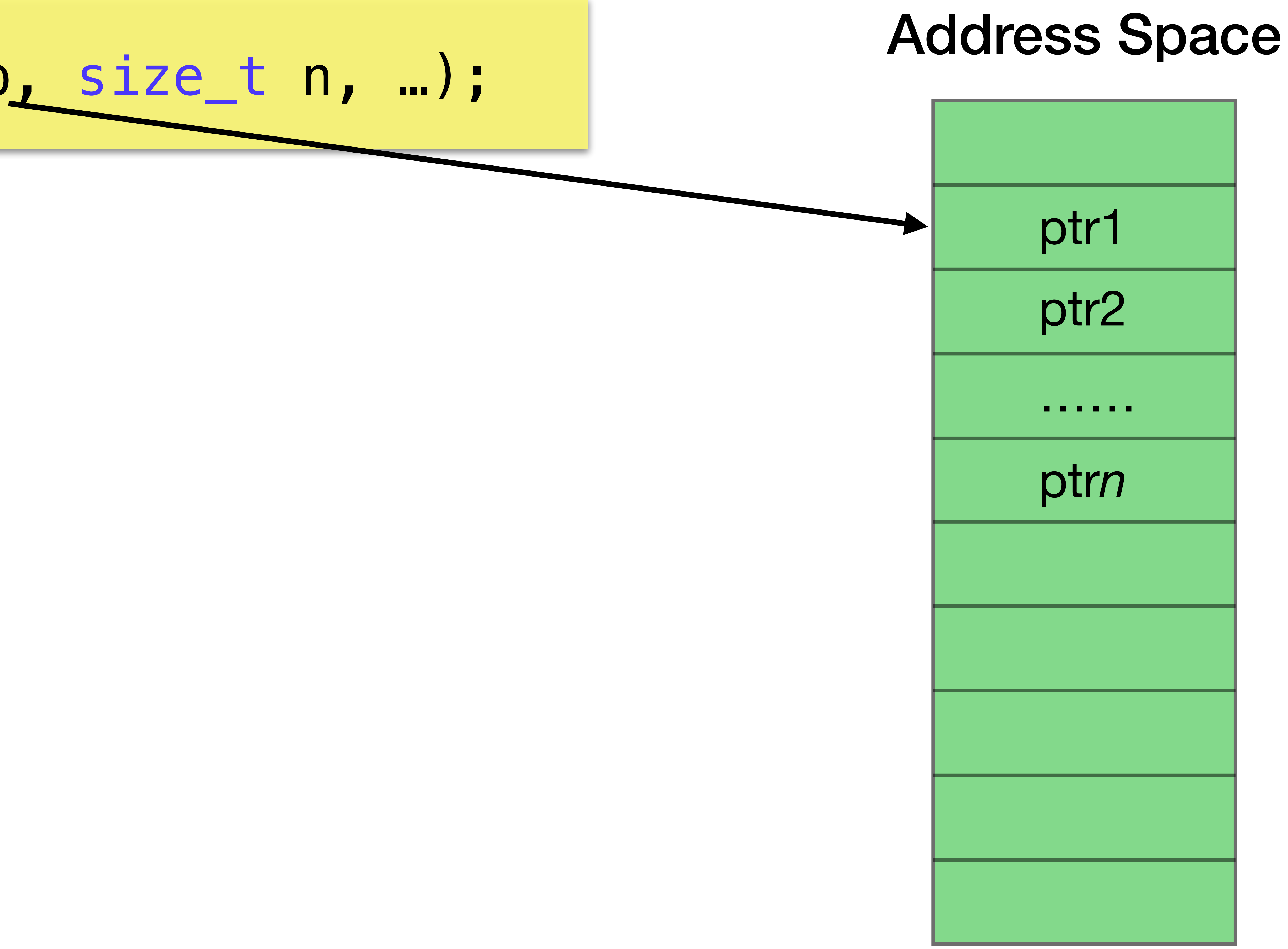
```
char *strchr(const char *s, int c);
```

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

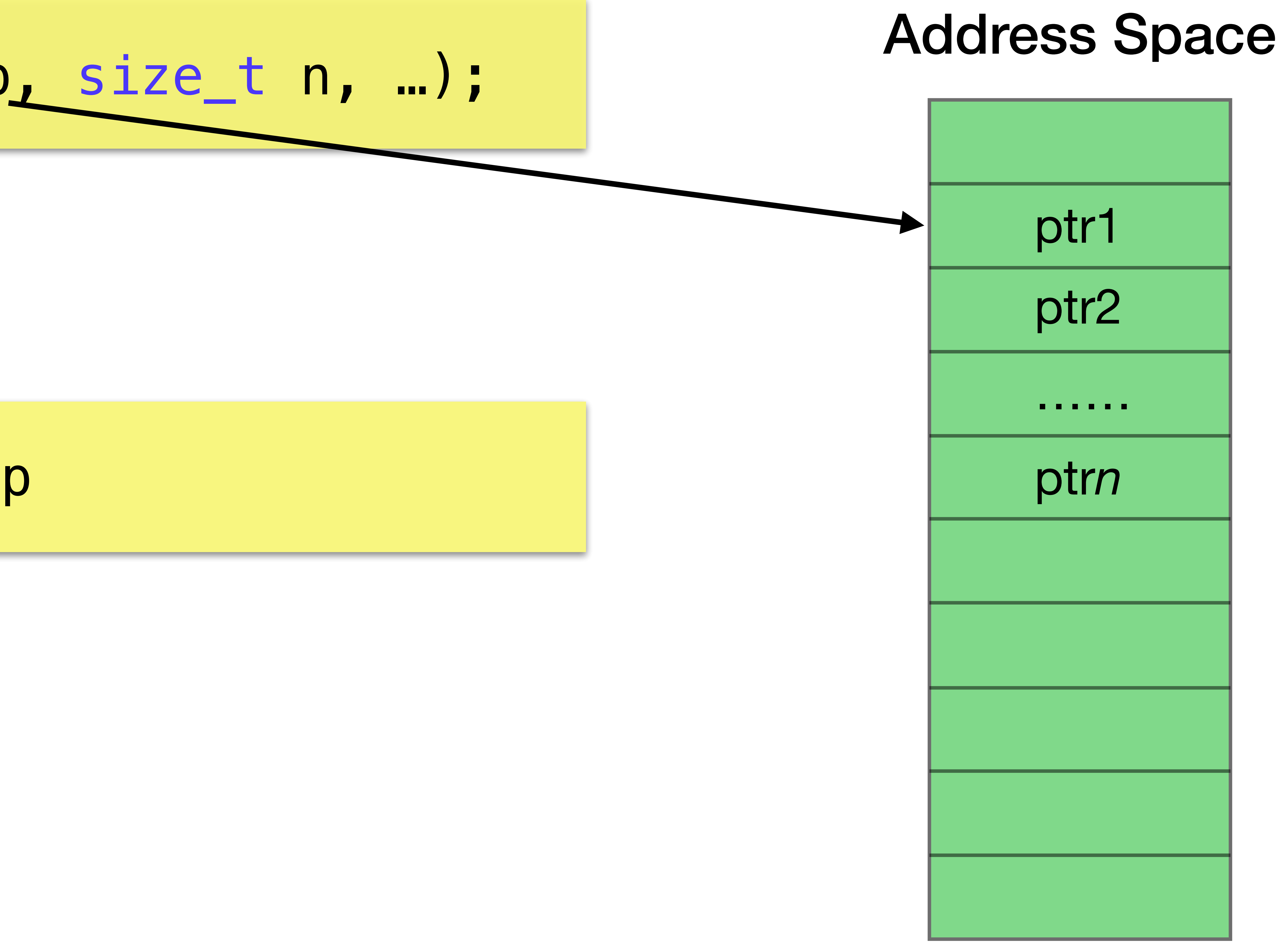
```
char *lib_foo(char **p, size_t n, …);
```



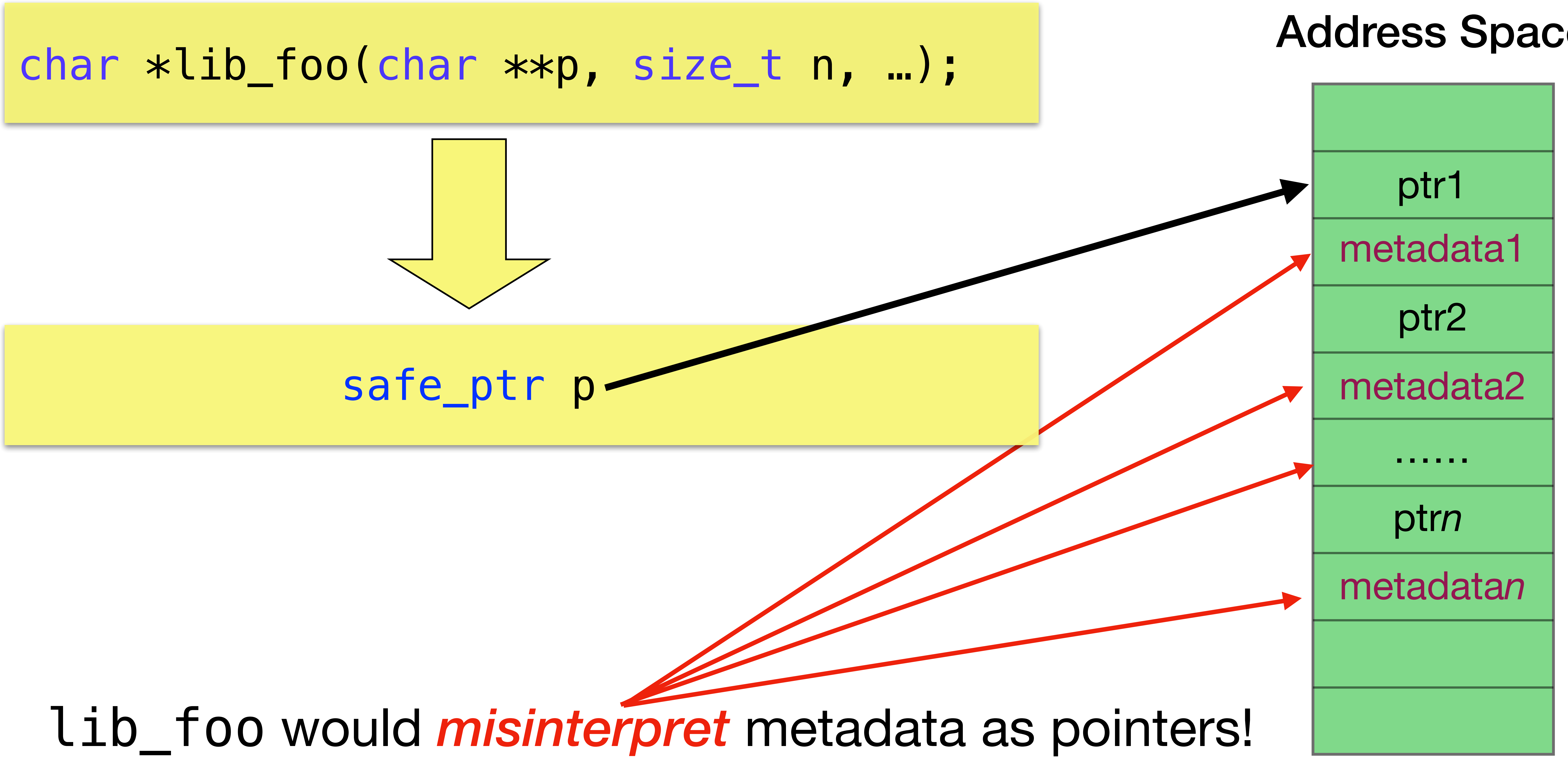
Memory Layout Compatibility with Unchanged Code

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

```
safe_ptr p
```



Memory Layout Compatibility with Unchanged Code

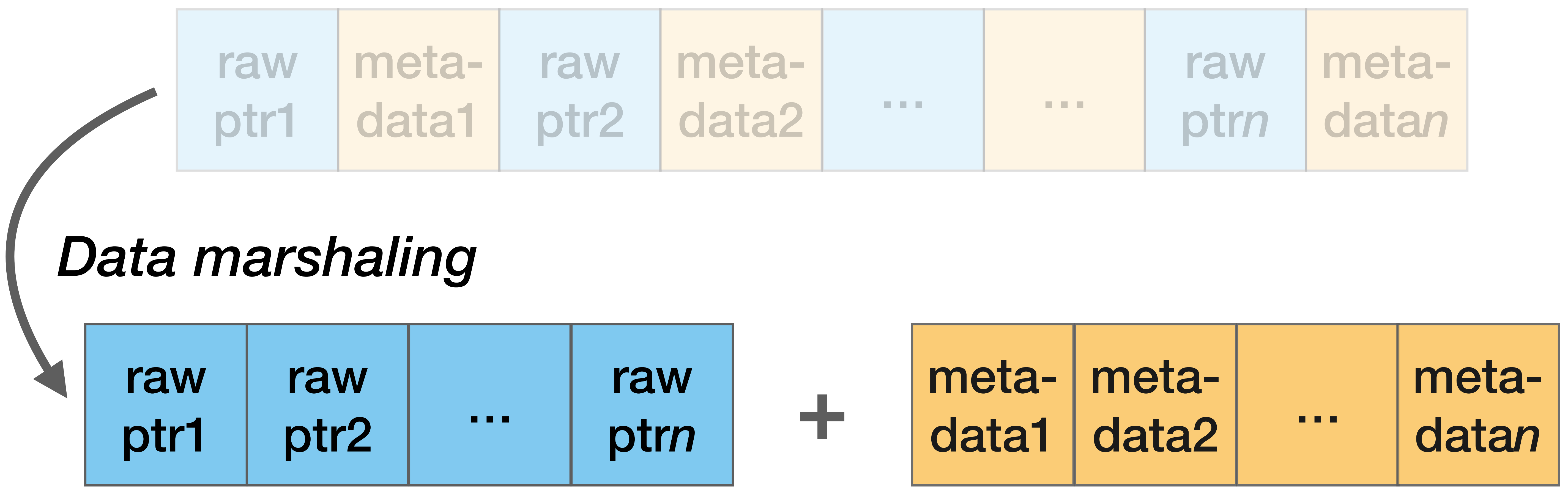


Solutions to Backward Compatibility Issues

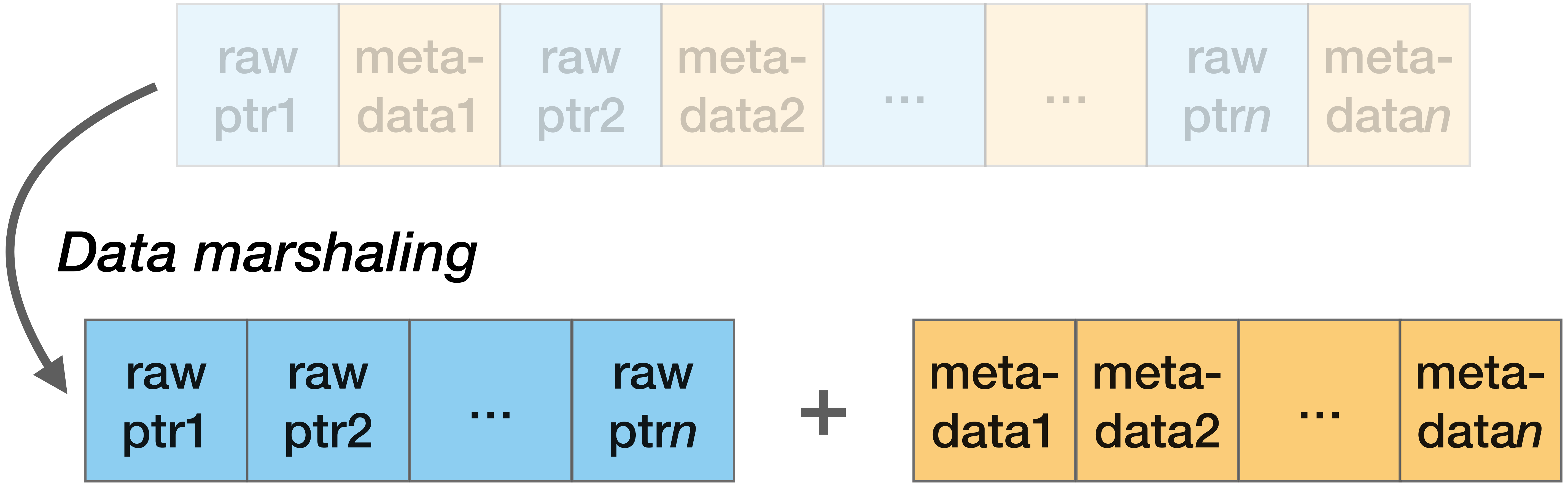
- Porting library functions using fat pointers
  - Pros: complete and secure
  - Cons: high programmer effort; not always viable
- Mixing using raw pointers and fat pointers
  - Pros: easy to implement
  - Cons: security hazards
- Hybrid of disjoint and in-place metadata
  - Pros: secure
  - Cons: complex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

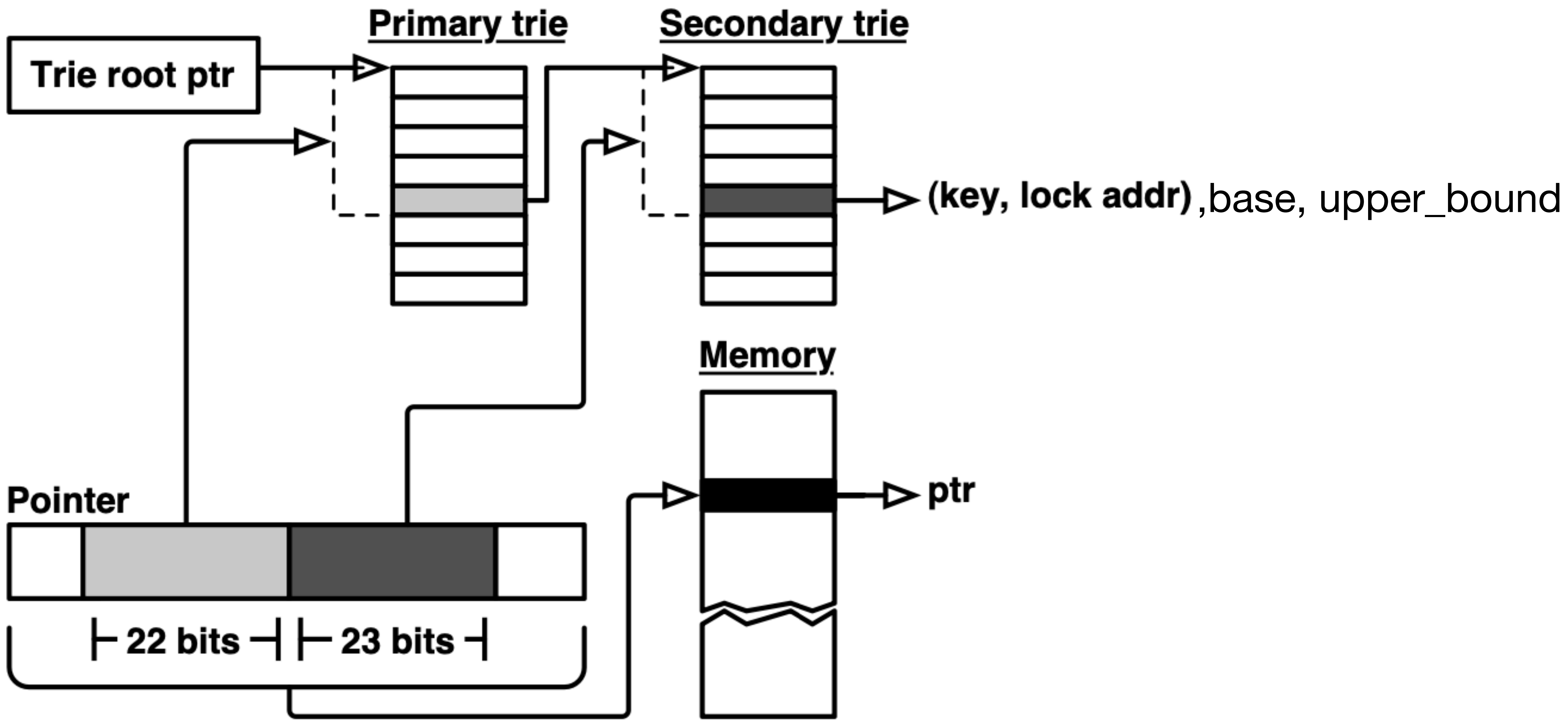


Figure 3. Organization of the trie for pointer metadata

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 *));
```

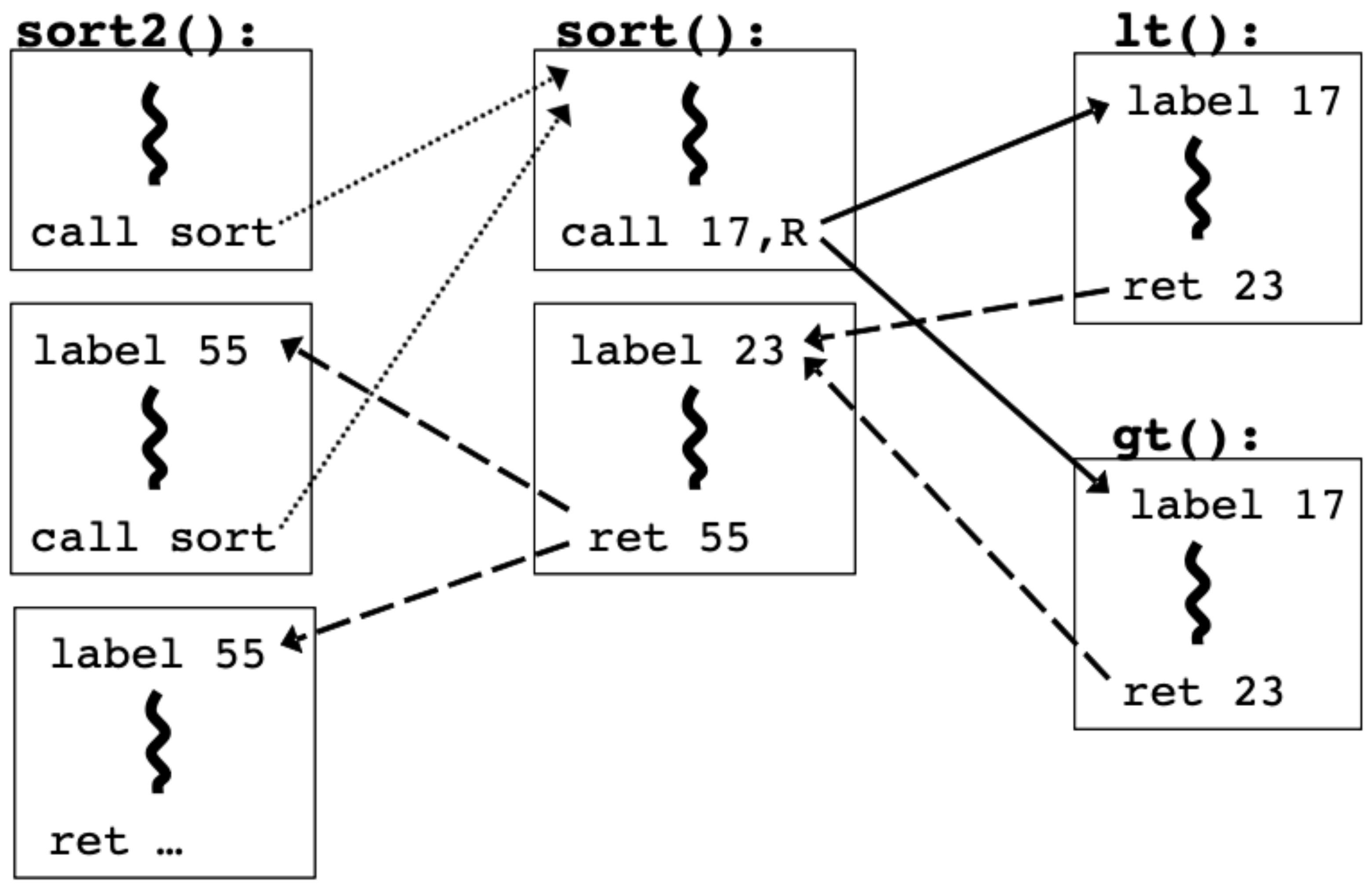
- `qsort`: Sort an array of n item, each of size `width`, using function `compar`.
  - E.g., Use a `greater_than()` function to sort an array of n integers.

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