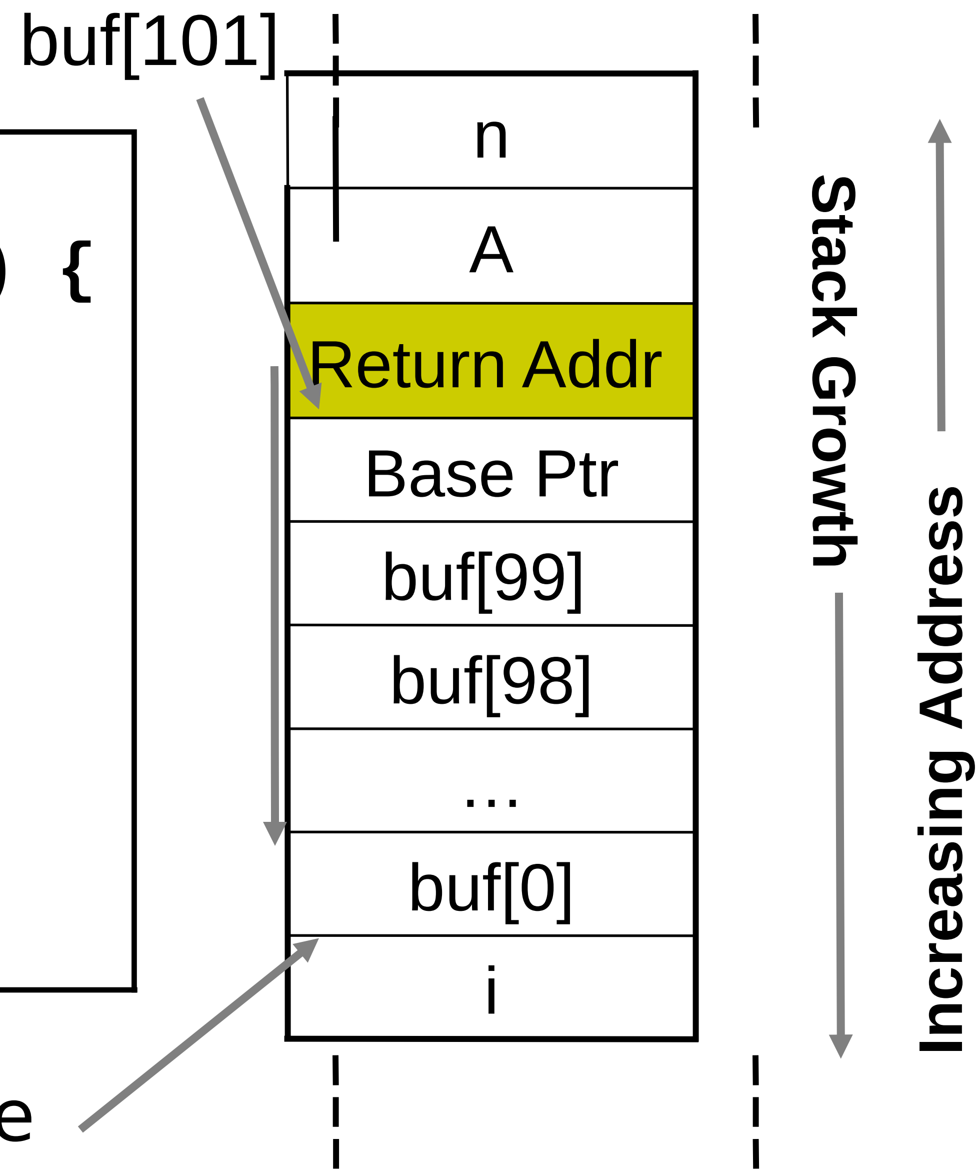
Memory Error Exploits and Defenses

Example: Stack Smashing Attack

```
void
f(const int *A, int n) {
   int buf[100];
   int i = 0;
   while (i < n) {
      buf[i] = A[i++];
   }
   ...
}
```



Stack smashing defenses

- **Canary stored before return value, checked before return**
  - Issues
    - Protecting RA vs Saved BP
    - Random, XOR, null canaries
    - How about data?
  - Weaknesses
    - Brute-force canary, or rely on information leakage attacks
    - Overwrite RA without overwriting canary (e.g., double pointer attacks)
    - Overwrite other code pointers (e.g., function pointer, virtual table pointer, GOT)
- **Storing RA in two places**
    - StackShield, Return address defender (RAD)
    - Issues: compatibility with signals, exceptions, longjmp
- **Propolice**
  - Canary before saved BP + protect local variables by reordering them
    - Simple variables (integers, pointers) located at lower addresses, buffers at higher addresses
      - Buffer overflow cannot corrupt local variables, preventing double pointer attacks
        - But underruns can corrupt these simple (non-buffer) variables
  - Mainstream compilers (gcc, MS) include Propolice like protection
    - Not included for functions with no arrays

Heap Overflows

- **Overflow from one heap block to the next is possible – but not easily exploited**
  - Hard to ensure that critical data worth corrupting will be located in the next block
- **More easily exploitable: overflow that overwrites control metadata stored adjacent to the buffer**
- **One form of attack**
  - Free heap blocks maintained as doubly linked list
  - Heap management code that adds/deletes from this list “trusts” the values of forward and backward links
  - Example: delete blk from free list:
    - blk->prev->next = blk->next
    - If an overflow within blk allows the attacker to overwrite prev field with “a” and next field with “b”, then the above code is equivalent to “*(a+k) = b”, where “k” is the offset of the field “next” within the struct.

Heap Overflows

- **More generally, provides a primitive to write an arbitrary 32-bit value at an arbitrary location**
- **Possible targets**
  - Function pointers
    - Return address on stack
      - Canaries don't help, but second RA copy will detect attack
    - Global Offset Table (GOT)
    - Function pointers in static memory
  - Data pointers
    - Names of programs executed or files opened
    - Application-specific data, e.g., "is_authenticated" flag in a login-like program

Heap Overflow Defenses

- **Heap canaries**
  - “magic numbers” between data and header
- **Separation of metadata from data**
  - In general, separating control data from program data is a good idea
    - Helps prevent data corruption attacks from altering the control-flow of programs
  - Can be applied on the stack as well
    - “Safe stack” holds control-data
      - “safe” data (e.g., local integer-valued variables) can also be located there as they cannot be involved in memory errors
    - All other data moved to a second stack

Format-string Attacks

- **Exploits code of the form**
  - Read variables from untrusted source
  - printf(s)
- **Printf usually reads memory, so how can it be used for memory corruption?**
  - “%n” primitive allows for a memory write
  - Writes the number of characters printed so far (character count)
  - Many implementations (Linux, Windows) allow just the least significant byte of the number of character count
    - you don’t have to print large number of characters to write arbitrary 32-bit values --- just perform 4 separate writes of the LS byte of character count
    - Use field-width specifications to control character count
- **Formatguard: pass in actual number of parameters so the callee can only dereference that many parameters**
  - Not adopted in practice due to compatibility issues

Integer Overflows

- **There are multiple forms**
  - Assignment between variables of different width
    - Assign 32-bit value to 16-bit variable
  - Assignment between variables of different signs
    - Assign an unsigned variable to a signed variable or vice-versa
  - Arithmetic overflows
    - i = j+k
    - i = 4*j
    - Note that i may become smaller than j even if j > 0
- **Exploitation**
  - Allocate less memory than needed, leading to a heap overflow
    - One of the common forms of file-format attacks
  - “Escape” bounds checks
    - If (i < sizeof(buf)) memcpy(buf, src, i);
- **For more info:**
  - http://www.phrack.org/archives/60/p60-0x0a.txt

Memory Errors

- **Although other attack types have emerged, memory errors continue to be the dominant threat**
  - Behind most "critical updates" from Microsoft and other vendors
  - Mechanism of choice in "mass-market" attacks, including worms
  - Evolved to target client (web browsers, email-handlers, word-processors, document/image viewers, media players, …) rather than server applications (e.g., web browsers)
- **A memory error occurs when an object accessed using a pointer expression is different from the one intended**
  - Spatial error
    - Examples
      - Out-of-bounds access due to pointer arithmetic errors
      - Access using a corrupted pointer
      - Uninitialized pointer access
  - Temporal error: access to objects that have been freed (and possibly reallocated)
    - Example: dangling pointer errors
    - applicable to stack and heap allocated data

Use of Memory Errors in Attacks

- **Temporal errors**
  - Not as frequently targeted as spatial errors
- **Spatial errors**
  - Pointer corruption is most popular
  - Out-of-bounds errors are most commonly used to corrupt pointers
    - But some attacks rely on just reads without necessarily corrupting existing data, e.g., heartbleed SSL vulnerability
- **Typically, multiple memory errors (2 to 3) are used in an attack**
  - Stack-smashing relies on out-of-bounds write, plus the use of a corrupted pointer as return address
  - Heap overflow relies on out-of-bounds write, use of corrupted pointer as target of write, and then the use of a corrupted pointer as branch target.

Memory Error Defenses

- **Disrupt exploits**
  - Identify mechanisms used for exploit, block them
    - Disrupt mechanism used for corruption
      - Protect attractive targets against common ways to corrupt them ("guarding" solutions)
    - **Disrupt mechanism used for take-over**
      - Disrupt ways in which the victim program uses corrupted data
      - ***Randomization-based defenses***
    - Disrupt payload delivery mechanism
      - NX, CFI
- **Block memory errors**
  - Bounds-checking (mainly focused on spatial error)
    - Bounds-checking C and CRED, Valgrind memcheck, ...
  - Blocking all memory errors (including temporal)

A. Disrupting Memory Error Exploits

Disrupting mechanisms used for corruption

- **Stackguard and related solutions**
  - Protect RA and saved BP; with ProPolice, some local variables as well
- **Magic cookies and safe linking on heaps**
- **Attacks on GOT**
  - GOT contains function pointers used to call library functions
    - Compiler generates a stub for each library function in a code section called PLT (program linkage table)
    - Stub code for a function f performs an indirect jump using the address stored in the GOT corresponding to f.
  - Defense: hide GOT
    - Not very effective: injected code can search and locate it!
- **Problem: incomplete**
  - Not all targets can be protected
  - Incomplete even for protected targets: some corruption techniques can still succeed, e.g., corrupting RA without disturbing canary.

Disrupting payload delivery mechanisms

- **Prevent control transfer to/execution of injected code**
  - Most OSes enforce W ⊕ X (aka NX or DEP)
    - prevents writable memory from being executable, so can't execute injected code
  - Attackers get around this by reusing existing code
    - return-to-libc: return to the beginning of existing functions
      - Instead of having injected code spawning a shell, simply "return" to the execle function in libc
      - If it is a stack-smash, attacker controls the contents of the stack at this point, so they can control the arguments to execle
    - By constructing multiple frames on the stack, it is possible to chain together multiple fragments of existing code
      - ROP (return-oriented programming) takes this to the extreme
        - Chains together many small fragments of existing code ("gadgets")
        - Each gadget can be thought of as an "instruction" for a "virtual machine"
        - For sufficiently complex binaries, sufficient number and variety of gadgets are available to support Turing-complete computation
      - Most exploits today rely on ROP, due to widespread deployment of W ⊕ X
        - Goal of ROP payload is to invoke mprotect system call to disable W ⊕ X.
  - Control-flow integrity (CFI) is another (partial) defense that limits attacker's freedom in terms of control transfer target
    - Can defeat most injected code and ROP attacks, but skilled attackers may be able to craft attacks that operate despite CFI

Disrupting take-over mechanism

- **Key issue for an attacker:**
  - using attacker-controlled inputs, induce errors with predictable effects
- **Approach: exploit software bugs to overwrite critical data, and the behavior of existing code that uses this data**
  - Relative address attacks (RA)
    - Example: copying data from input into a program buffer without proper range checks
  - Absolute address attacks (AA)
    - Example: store input into an array element whose location is calculated from input.
      - Even if the program performs an upper bound check, this may not have the intended effect due to integer overflows
  - RA+AA attacks: use RA attack to corrupt a pointer *p,* wait for program to perform an operation using **p*
    - Stack-smashing, heap overflows, …

Disrupting take-over: Diversity Based Defenses

- **Software bugs are difficult to detect or fix**
  - Question: Can we make them harder to exploit?
- ***Benign Diversity***
  - Preserve functional behavior
    - On benign inputs, diversified program behaves exactly like the original program
  - Randomize attack behavior
    - On inputs that exercise a bug, diversified program behaves differently from the original

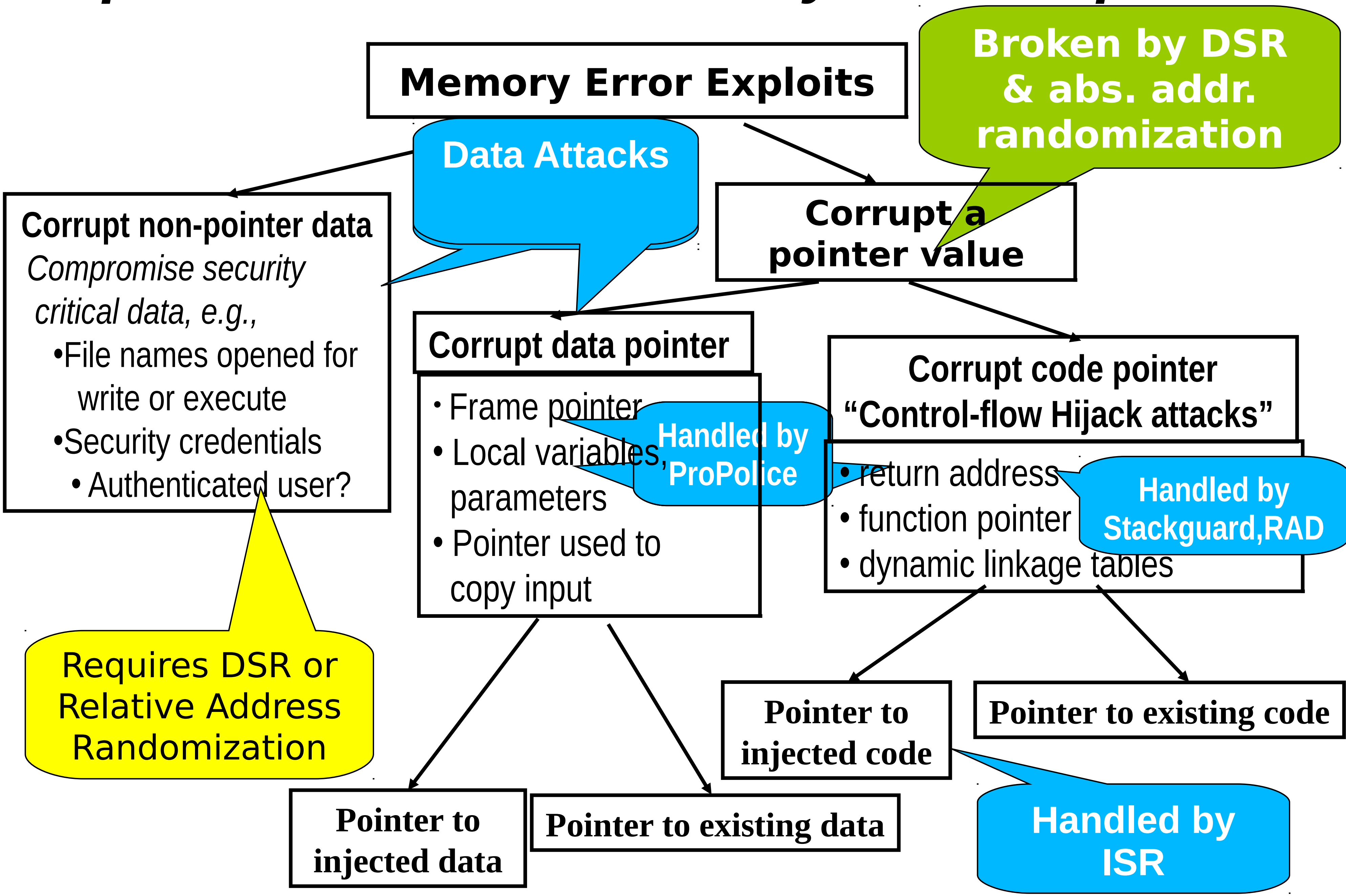
Automated Introduction of Diversity

- **Use transformations that preserve program semantics**
- **Challenge: how to capture intended program semantics?**
    - Relying on manual specifications isn't practical
- **Solution: Instead of focusing on program-specific semantics, rely on *programming language semantics***
  - Randomize aspects of program implementation that aren't specified in the programming language
    - Benefit: programmers don't have to specify any thing
  - Examples
    - Address Space Randomization (ASR)
      - Randomize memory locations of code or data objects
      - Invalid and out-of-bounds pointer dereferences access unpredictable objects
    - Data Space Randomization (DSR)
      - Randomize low-level representation of data objects
      - Invalid copy or overwrite operations result in unpredictable data values
    - Instruction Set Randomization (ISR)
      - Randomize interpretation of low-level code
      - W $\oplus$ X has essentially the same effect, so ISR is not that useful any more

How randomization disrupts take-over

- **Without randomization, memory errors corrupt process memory in a predictable way**
  - Attacker knows what data is corrupted, e.g., return address on the stack
    - Relative address randomization (RAR) takes away this predictability
  - Attacker knows the correct value to be used for corruption, e.g., the location of injected code (in a buffer that contains data read from attacker)
    - Absolute address randomization (AAR) takes away this predictability for pointer-valued data
    - DSR takes away this predictability for all data

Space of Possible Memory Error Exploits



First Generation ASR: Absolute Address Randomization (ASLR)

- **Discovered by PaX project and [Bhatkar et al]**
- **Randomizes base address of data (stack, heap, static memory) and code (libraries and executable) regions**
- **Implemented on many flavors of UNIX & Windows**
  - UNIX implementations usually provide 20+ bits of randomness, 16 bits for Windows
- **Finding its way into mainstream OS distributions**
  - Linux, OpenBSD, …
  - Vista (limited to 8 bits of randomness)
- **Limitations**
  - Brute-force attacks
  - Relative address attacks
    - Non-pointer data attacks, partial pointer overwrites, integer overflows
  - Information leakage attacks

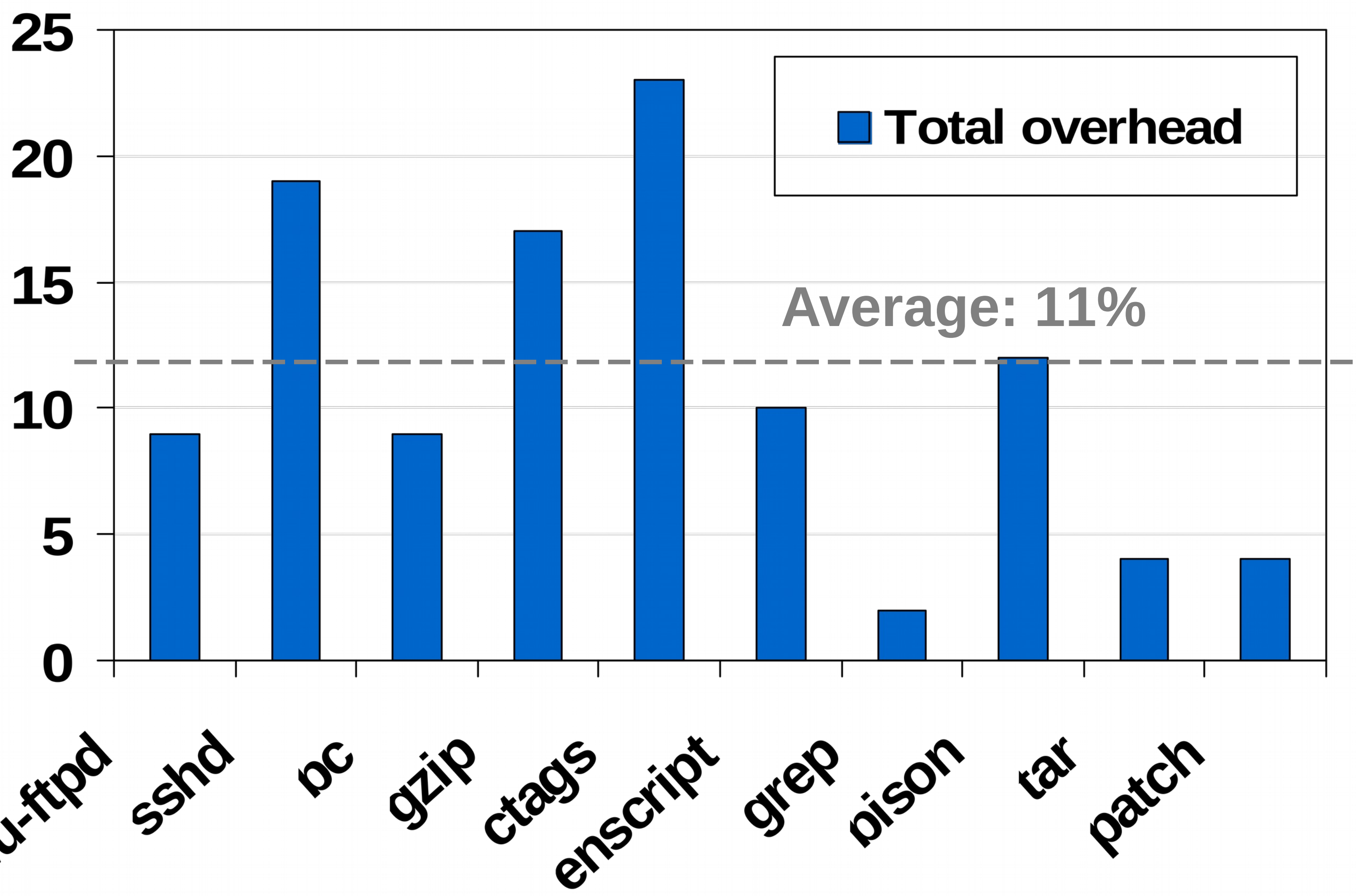
Second Generation ASR: Relative Address Randomization

- **Randomize distances between individual data and code objects**
- **[Bhatkar et al] use code transformation to**
  - permute the relative order of objects in memory
    - Static variables
    - “Unsafe” local variables
      - Safe local variables moved to a “safe” stack (no overwrites possible)
    - Routines (functions)
  - introduce gaps between objects
    - Some gaps may be made inaccessible

Benefits of RAR

- **Defeats the overwrite step, as well the step that uses the overwritten pointer value**
  - Defeats format-string and integer overflow attacks
  - Stack-smashing attacks fail deterministically
- **Higher entropy**
  - Up to 28 bits
  - Knowing the location of one object does not tell you much about the locations of other objects
    - information leakage attacks become difficult
    - heap overflows become more difficult since you need to make two independent guesses

Execution Time Overheads



Data Space Randomization

DSR Technique

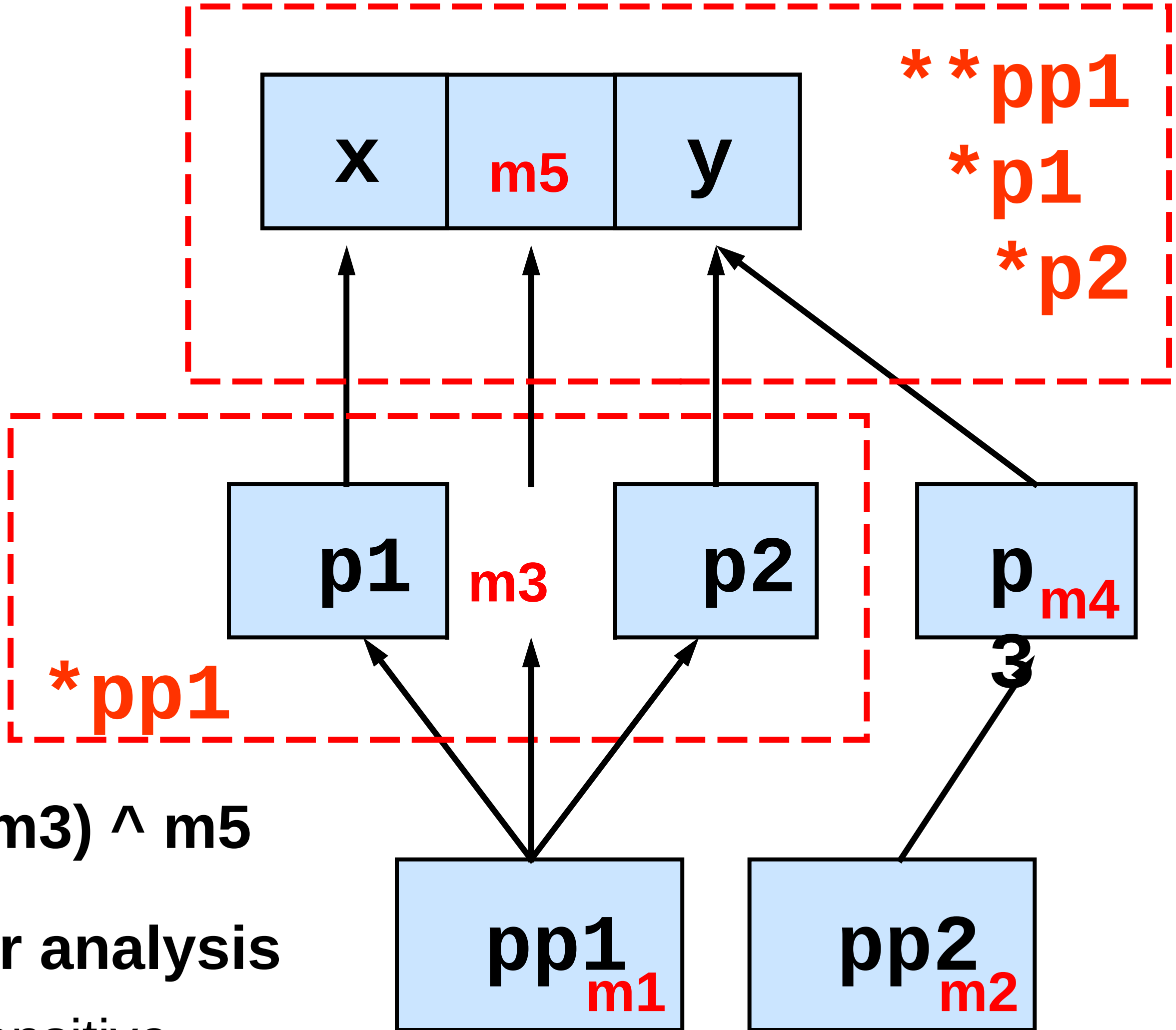
- **Basic idea: Randomize data representation**
  - *Xor* each data object with a *distinct random mask*
  - Effect of data corruption becomes non-deterministic, e.g.,
    - Use out-of-bounds access on array ***a*** to corrupt variable ***x*** with value ***v***
      - Actual value written: mask(***a***) ⊕ ***v***
      - When ***x*** is read, this value is interpreted as mask(***x***) ⊕ (mask(***a***) ⊕ ***v***)
        - Which is different from ***v*** as long as the masks for ***x*** and ***a*** differ.
- **Benefits**
  - Large entropy
    - 32-bits of randomization for integers
    - Masks for different variables can be independent
  - Can address intra-structure overflows
    - Not even addressed by full memory error detection techniques
  - Natural generalization of PointGuard
    - Protects all data, not just pointers
    - Effective against relative address as well as absolute address attacks
    - Different objects can use different masks (resists information leak attacks)

DSR Transformation Approach

- **For each variable** *v,* **introduce another variable** *m_v* **for storing its mask**
- **Randomize values assigned to variables (LHS)**
  - Example: ***x = 5*** → ***x = 5; x = x ^ m_x;***
- **Derandomize used variables (RHS)**
  - Example: (***x + y)*** → ***((x ^ m_x) + (y ^ m_y))***
- **Key problem: aliasing**
  - int *x = &y
  - A value may be assigned to y and dereferenced using *x
    - Both expressions should yield the same value
      - Need to ensure that possibly aliased objects should use the same randomization mask
- **Note**
  - In x = y, it is not necessary to assign same mask to x and y

Pointer Analysis & Mask Assignment

```
int x, y;
int *p1,*p2, *p3;
int **pp1, **pp2;

pp1 = &p1; …
pp1 = &p2; …
pp2 = &p3; …
p1 = &x; …
p2 = &y; …
p3 = &y; …
```



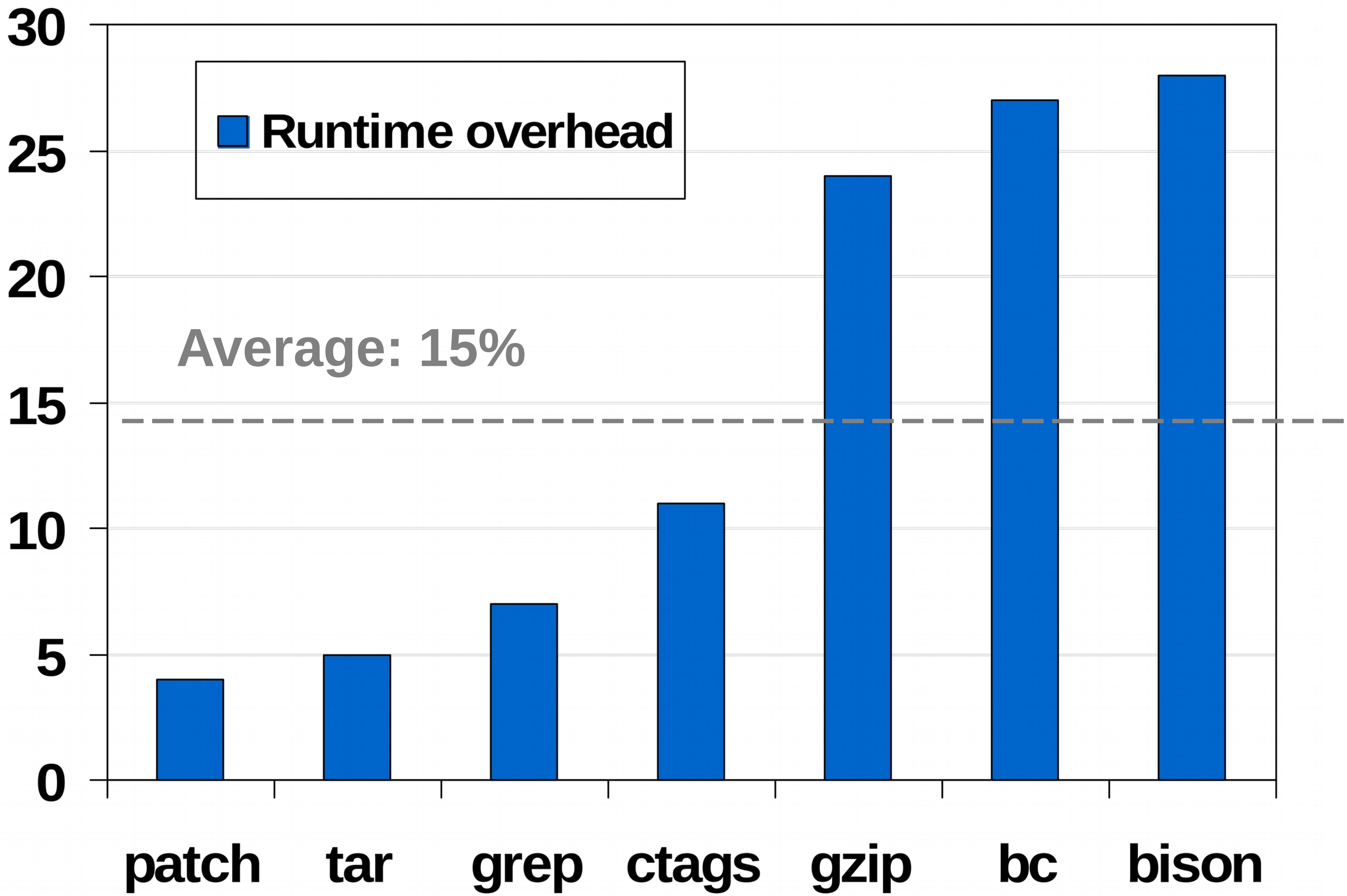
****pp1 => *(*(pp1 ^ m1) ^ m3) ^ m5**

- **Steensgaard's pointer analysis**
  - Flow and context insensitive
  - Efficient (linear time complexity)

Implementation

- **Uses source-to-source transformation**
- **For performance reasons, applies DSR to buffers and pointers only**
  - Non-buffer data is still protected against buffer overflows
- **Attempts to ensure that adjacent buffers won’t have the same mask**
  - Makes it possible to detect all buffer overflows
- **Limitations**
  - Does not yet support field sensitive *points-to* analysis
  - Requires identification of external functions that aren’t transformed

Execution Time Overheads



Limitations of ASR/DSR

- **Interoperability between diversified code and code that is not diversified**
  - Some randomizations need source code
    - e.g., RAR relies on source-code transformations to reorder static variables, functions, etc.
- **Performance**
  - Increased VM usage (insignificant)
  - Increased physical memory usage (insignificant)
  - Runtime overhead (negligible for AAR, small for RAR, DSR)
- **Making debuggers randomization-aware**
- **Biggest security challenge:**
  - Protecting randomization key(s), or in other words, resilience in the face of information leak attacks

Summary of Automated Diversity

- **Transformations that respect programming language semantics are good candidates for automated diversity**
  - But they are typically good for addressing only low-level implementation errors. (We have discussed them only in the context of a specific low-level error, namely, memory corruption.)
- **Automated diversity has been particularly successful in the area of memory error exploit prevention**
  - First generation of randomization-based defenses focused on absolute address based attacks
    - Absolute-address randomization
    - Practical technique with low impact on systems, and hence begun to be deployed widely
  - Second generation defenses provide protection from relative-address dependent attacks
    - Relative address randomization and data-space randomization

State of Exploit defenses and New attacks

- **Most OSes now implement**
  - ProPolice like defenses, plus SEH protection (Microsoft)
  - ASLR
  - DEP/NX (prevent injected code execution)
- **Recent attacks**
  - Exploit incomplete defenses, or use Heapspray for control-flow hijack
    - No ASLR on most executables on Linux, some EXE, DLLs on MS
    - Some libraries don’t enable stack protection, or it is incomplete
    - Heapspray: brute-force attack in the space domain
      - Exploits untrusted code in safe languages (Javascript, Java, Flash,…)
      - Code allocates almost all of memory, fills with exploit code
      - Jump to random location: with high probability, it will contain exploit code
  - Return-oriented programming (ROP) to overcome DEP
  - Rely increasingly on information leak attacks to overcome uncertainty due to ASLR, frequent software updates, and so on
    - Just-in-time-ROP: use information leak vulnerability to scan code at runtime to identify ROP gadgets

B. Preventing Memory Errors

Memory Errors in C

- **Spatial errors: out-of-bounds subscript or pointer**
  - `char *p = malloc(10); *(p+15);`
- **Temporal errors: pointer target no longer valid**
  - Unintialized pointer
  - Dangling pointer
    - `free(p); q = malloc(…); *p;`
    - **Note: target may be reallocated!**
- **Hard to debug, especially temporal errors**
  - Unpredictable delay, unpredictable effect
    - Reallocated pointer errors are the worst kind
  - “Defensive programming” leads to memory leaks

Memory Errors in C

- **Spatial errors: out-of-bounds subscript or pointer**
  - `char *p = malloc(10); *(p+15);`

- **Temporal errors: pointer target no longer valid**
  - Unintialized pointer
  - Dangling pointer
    - `free(p); q = malloc(…); *p;`
    - **Note: target may be reallocated!**
- **Hard to debug, especially temporal errors**
  - Unpredictable delay, unpredictable effect
    - Reallocated pointer errors are the worst kind
  - “Defensive programming” leads to memory leaks

Issues and Constraints

- **Backward compatibility with existing C-code**
  - Casts, unions, address arithmetic
  - Conversion between integers and pointers
- **Compatibility with previously compiled libraries**
  - Can't expect to rebuild the entire system
  - Source code access can be problematic for some libs
- **Temporal Vs Spatial Errors**
  - Detecting reallocated storage
  - Important, since such errors get detected very late, and it is extremely hard to track them down
- **Use of garbage collection**

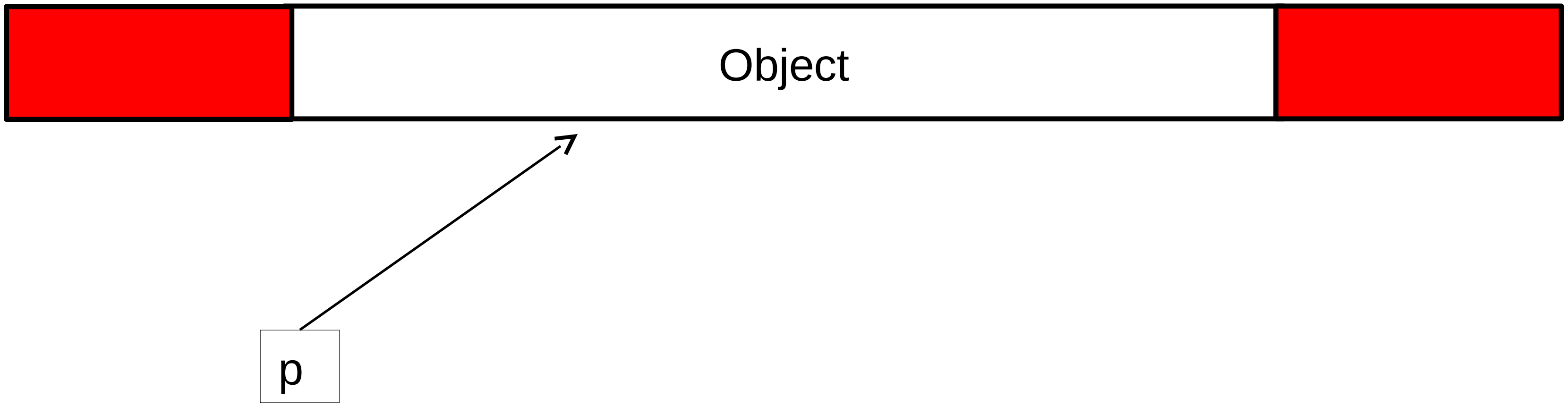
Why Not Garbage Collection?

- **Masks temporal errors**
  - Problematic if the intent is to use memory error-checking only during the testing phase
- **Unpredictable overheads**
  - Problematic for systems with real-time or stringent performance constraints
- **GCs can make mistakes due to free conversion between integers and pointers**
  - Fail to collect inaccessible memory
  - *Collect memory that should not be collected*
  - Problematic for code that relies heavily on such conversions, e.g, OS Kernel

Approaches for Preventing Memory Errors

- **Introduce inter-object gaps, detect access to them (Red zones)**
  - Detect subclass of spatial errors that involve accessing buffers just past their end
    - Purify, Light-weight bounds checking [Hasabnis et al], Address Sanitizer [Serebryany et al]
- **Detect crossing of object boundaries due to pointer arithmetic**
  - Detects spatial errors
  - Backwards-compatible bounds checker [Jones and Kelly 97]
  - Further compatibility improvements achieved by CRED [Ruwase et al]
  - Speed improvements: Baggy [Akritidis et al], Paricheck [Younan et al]
- **Runtime metadata maintenance techniques**
  - Temporal errors: pool-based allocation [Dhurjati et al], Cling [Akritidis et al]
  - Spatial and temporal errors: CMemSafe [Xu et al]
    - Further compatibility improvements: SoftBounds [Nagarakatte et al]
  - Targeted approaches: Code pointer integrity [Kuznetsov et al], protects subset of pointers needed to guarantee the integrity of all code pointers.

Red Zone: LBC Approach

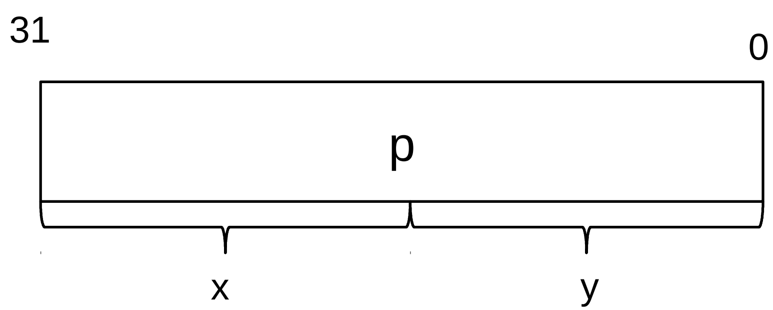


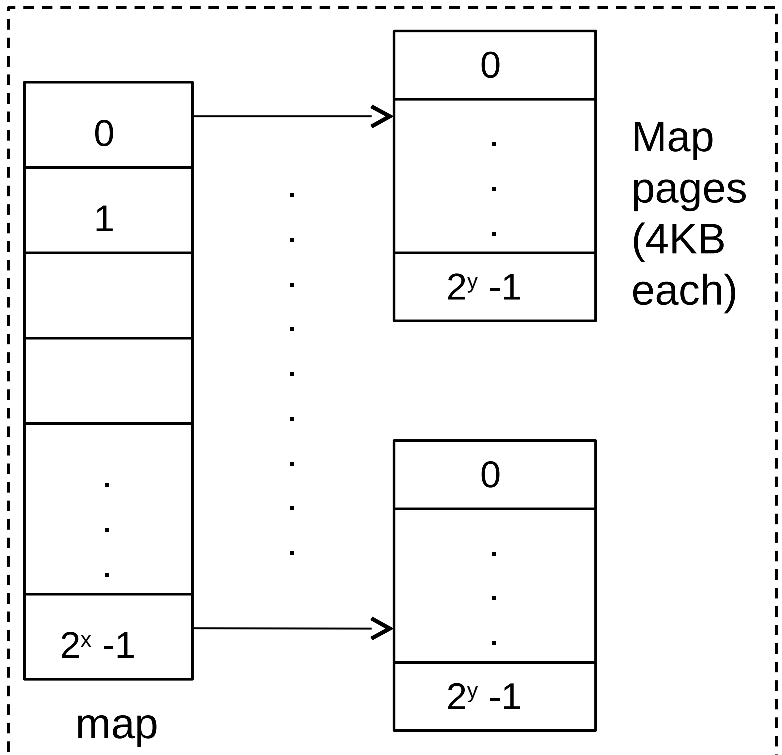

```
((*p == guard_zone_value && slowcheck (p)) ?
   flag_error() : *p
```

Zero metadata operations in most common case saves significant runtime overheads

Slowcheck

- Simple version: guardmap[p] == 1
  - Occupies 1/8th of the address space, even for a program that uses a few bytes of memory –- leads to inefficiencies
- Better version: two-level map
  - Divide 32-bits of p into two parts, x (17 bits) and y (15 bits)
    - Check: map[x] == NULL || map[x][y] == 1
  - Map uses just 0.5MB for programs with small memory use
- Use 3-level map for 64-bit address space
- Address sanitizer uses a similar approach, but without a fast check





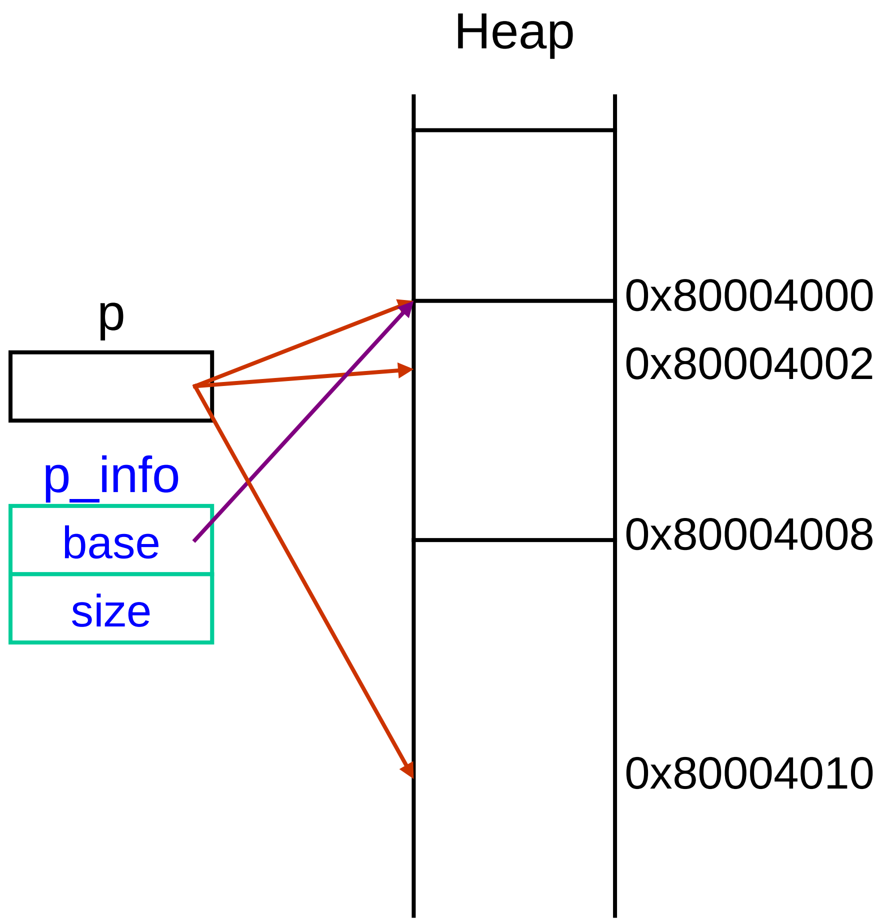
Backwards Compatible Bounds-Checking

- **Enforces object allocation boundaries**
- **All allocations are entered into an efficient data structure for intervals (splay tree)**
- **Checks pointer arithmetic, not dereferences**
- **If *p* is derived through address arithmetic on *q*, then requires that *p* and *q* refer to the same object**
  - If not, p is set to an invalid value (e.g., -1) that will cause memory exception on dereference
- **CRED: improves compatibility in cases where out-of-bounds pointer is created but is not dereferenced before being brought back in bounds**
  - Uses a special data structure to keep track of OOB pointers

CMemSafe: Detecting Spatial Errors Using Metadata

```
Spatial Check:
(p >= p_info.base &&
 p < p_info.base+p_info.size)?
```

```
char * p;

p = malloc(8);

p += 2;

*p;    /* OK */

p += 14;

*p;    /* error */
```

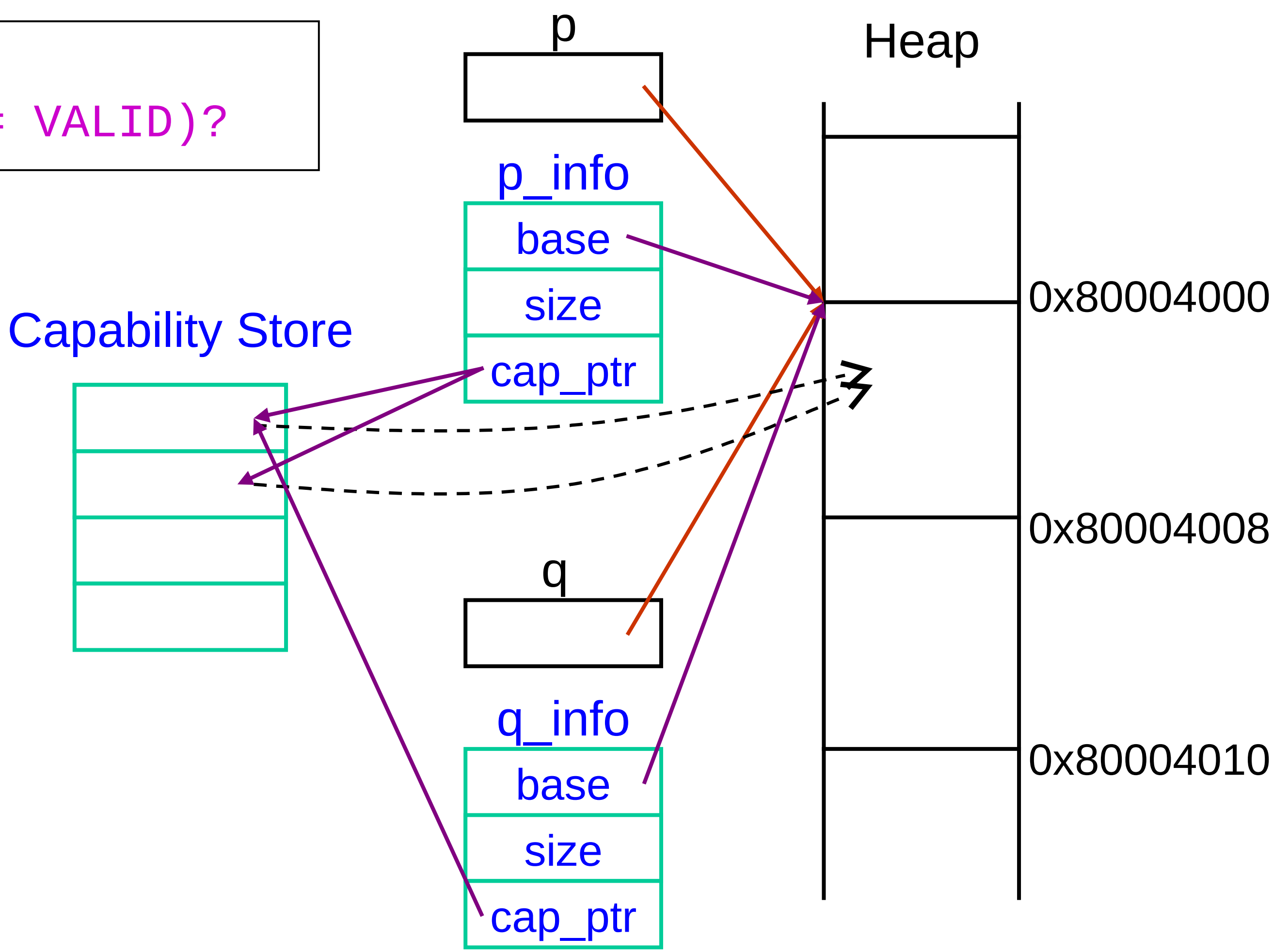


- **base, size: base address and allocated size of the block**

CmemSafe: Detecting Temporal Errors

```
Temporal Check:
(*q_info.cap_ptr == VALID)?
```

```
char * p, *q;

p = malloc(8);

q = p;

*q;   /* OK */

free(p);

*q;   /* error */

p = malloc(16);

*q;   /* error */
```



- **cap_ptr: pointer to unique capability associated with block**
- **Detect erroneous accesses to freed or reallocated memory**

Summary of Memory Error Defenses

- **Static analysis (False positives and false negatives)**
  - Produce false positives (underlying problems are undecidable)
  - Aimed at programmers, who need to investigate reported errors
  - Not very practical because of FPs and FNs, so we did not discuss these
- **Runtime detection of errors (Typically, no FPs)**
  - Exploit detection
    - ASR, canaries, ….
  - Error detection (some incompatibility with legacy code)
    - Metadata for allocations, but no per-pointer metadata
      - Compatible with untransformed libraries
      - Can't detect pointer corruptions or temporal errors
      - Examples: red zones, bounds-checking, CRED
    - Per-pointer metadata
      - Detect pointer arithmetic errors as well as corruption errors, plus temporal errors
      - Compatibility issues: serious with "fat" pointers, significant even otherwise.
- **Hybrid approaches**
  - CCured: static analysis classifies pointers, avoid metadata for most pointers
  - Pool-based allocation: map temporal error effects into those of spatial errors