CS412 Software Security

Mitigations

Mathias Payer

EPFL, Spring 2019

Model for Control-Flow Hijack Attacks

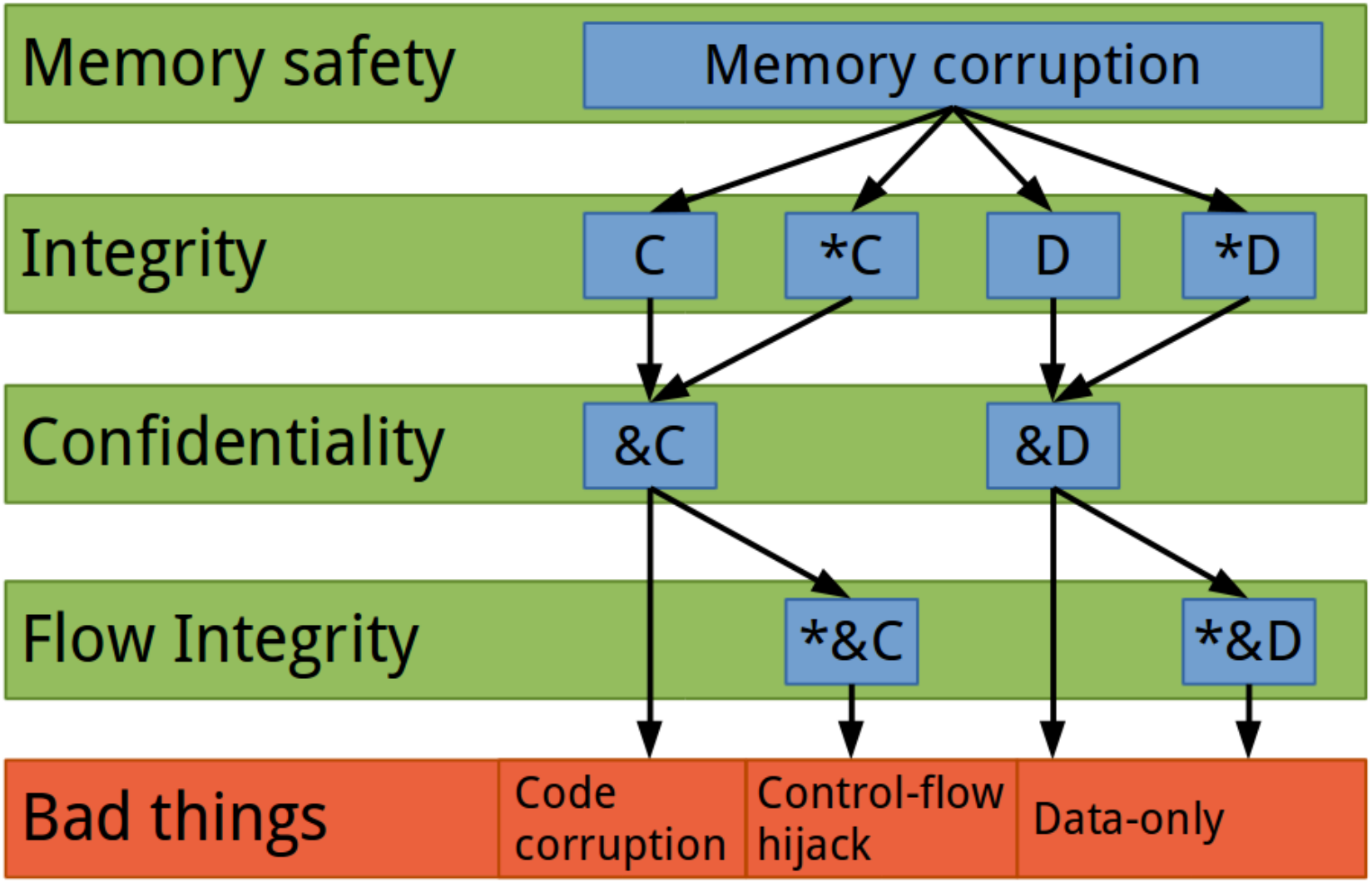

Figure 1

Widely-adopted defense mechanisms

- Hundreds of defense mechanisms were proposed.
- Only few mitigations were adopted.
- What factors increase chances of adoption?
  - Mitigation of the most imminent problem.
  - (Very) low performance overhead.
  - Fits into the development cycle.

Attack vector: code injection

- Simplest form of code execution.
- Generally consists of two steps:
  - Inject code somewhere into the process
  - Redirect control-flow to injected code

Data Execution Prevention (DEP)

- ISAs (e.g., x86, ARM) did not distinguish between code and data.
- Any data in the process could be interpreted as code (code injection: an attacker redirects control-flow to a buffer that contains attacker-controlled data as shellcode).
- *Defense assumption:* if an attacker cannot inject code (as data), then a code execution attack is not possible.

DEP: process layout

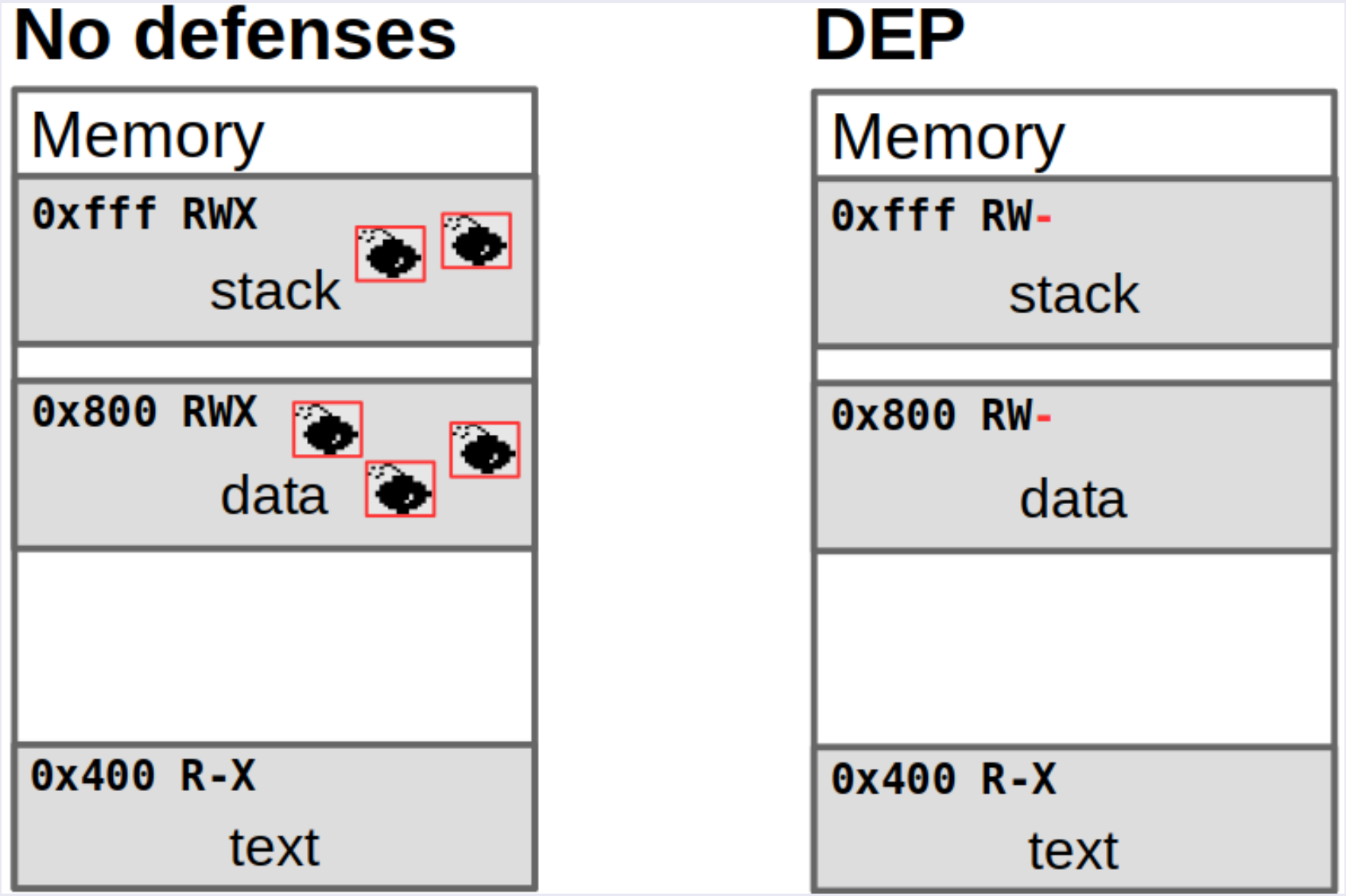

Figure 2

DEP

- Page table extension, introduce NX-bit (No eXecute bit).
  - Intel calls this per-page bit XD (eXecute Disable)
  - AMD calls it Enhanced Virus Protection
  - ARM calls it XN (eXecute Never).
- This is an additional bit for every mapped virtual page. If the bit is set, then data on that page cannot be interpreted as code and the processor will trap if control flow reaches that page.

Old school page table

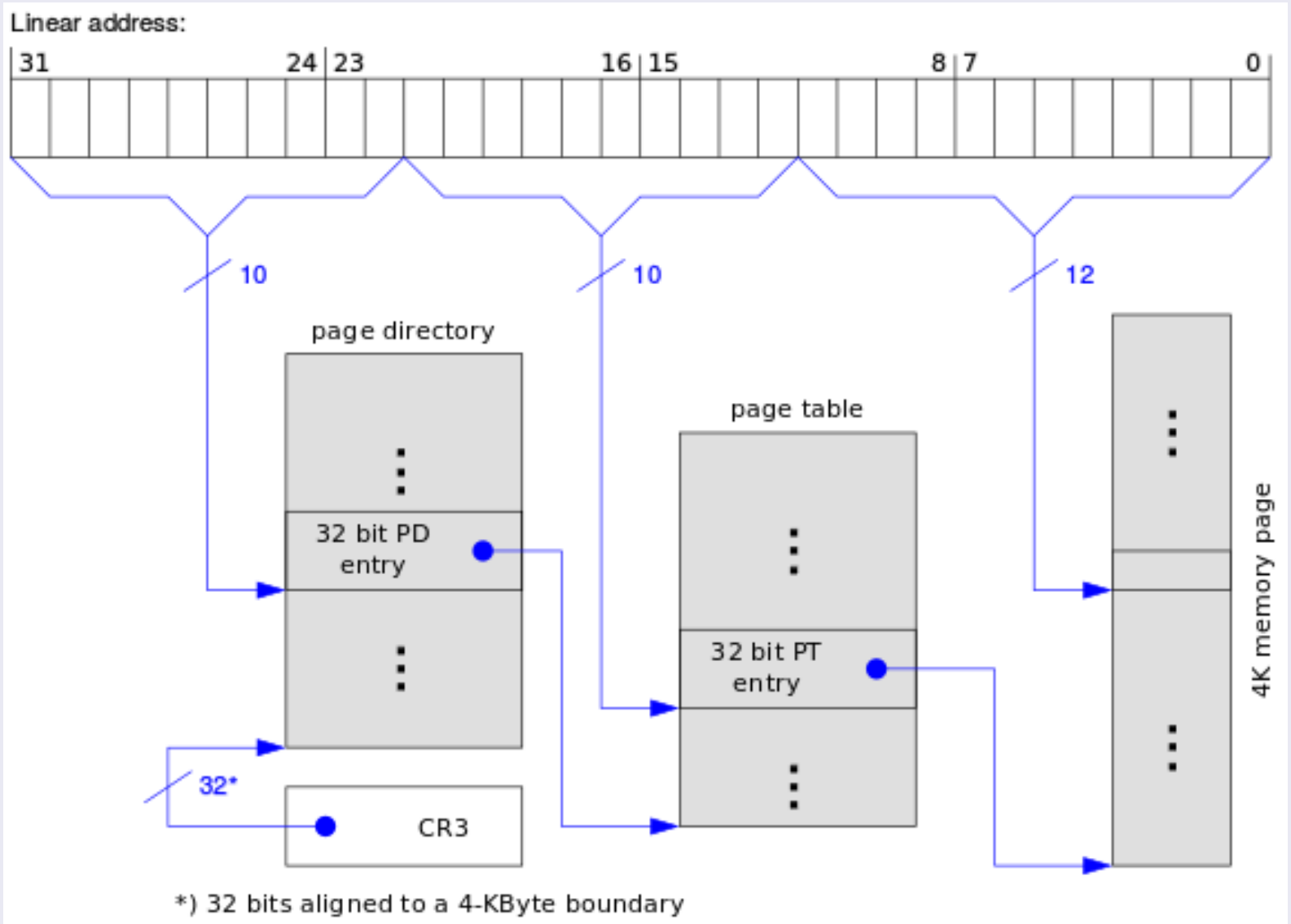

Figure 3

Old school page table entry

Bit	Interpretation
0 (P)	Present; must be 1 to map a 4-KB page
1 (R/W)	Read/write; if 0, writes may not be allowed
2 (U/S)	User/supervisor; if 0 access with CPL=3 not allowed
3 (PWT)	Page-level write-through
4 (PCD)	Page-level cache disable
5 (A)	Accessed
6 (D)	Dirty
7 (PAT)	Page Attribute Table (PAT) indicator or reserved (0)
8 (G)	Global
11:9	Ignored
31:12	Physical address of the 4-KB page

According to Intel 64 and IA-32 manual 3A, Table 4-6.

See, no distinction between code and data!

Physical Address Extension (PAE) page table

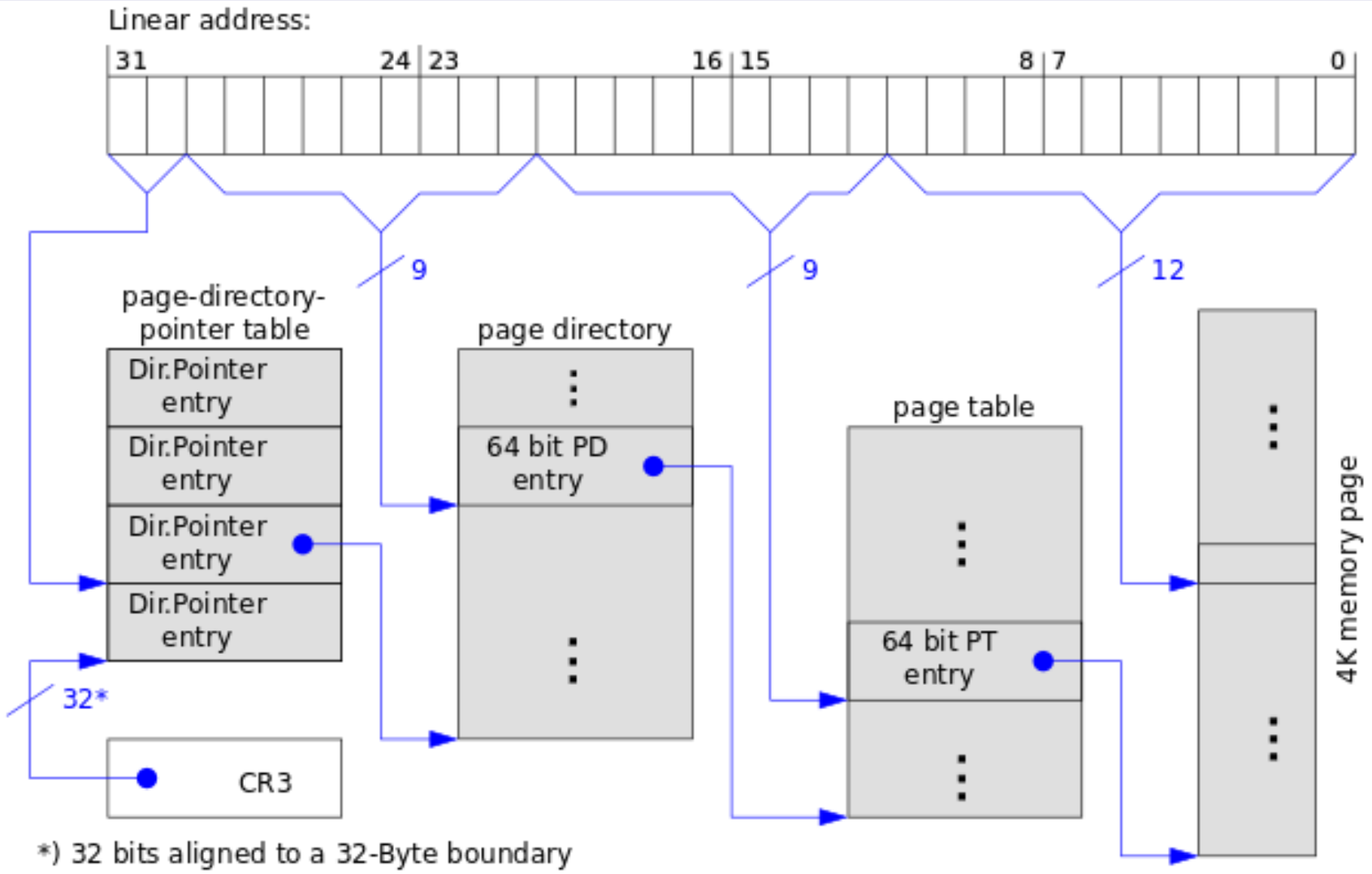

Figure 4

PAE page table entry

Bit	Interpretation
0 (P)	Present; must be 1 to map a 4-KB page
1 (R/W)	Read/write; if 0, writes may not be allowed
2 (U/S)	User/supervisor; if 0 access with CPL=3 not allowed
3 (PWT)	Page-level write-through
4 (PCD)	Page-level cache disable
5 (A)	Accessed
6 (D)	Dirty
7 (PAT)	Page Attribute Table (PAT) indicator or reserved (0)
8 (G)	Global
11:9	Ignored
(M-1):12	Physical address of the 4-KB page
62:M	Reserved (0)
63 (XD)	*If IA32_EFER.NXE = 1, execute-disable if 1*

According to Intel 64 and IA-32 manual 3A, Table 4-11.

x86-64 page table

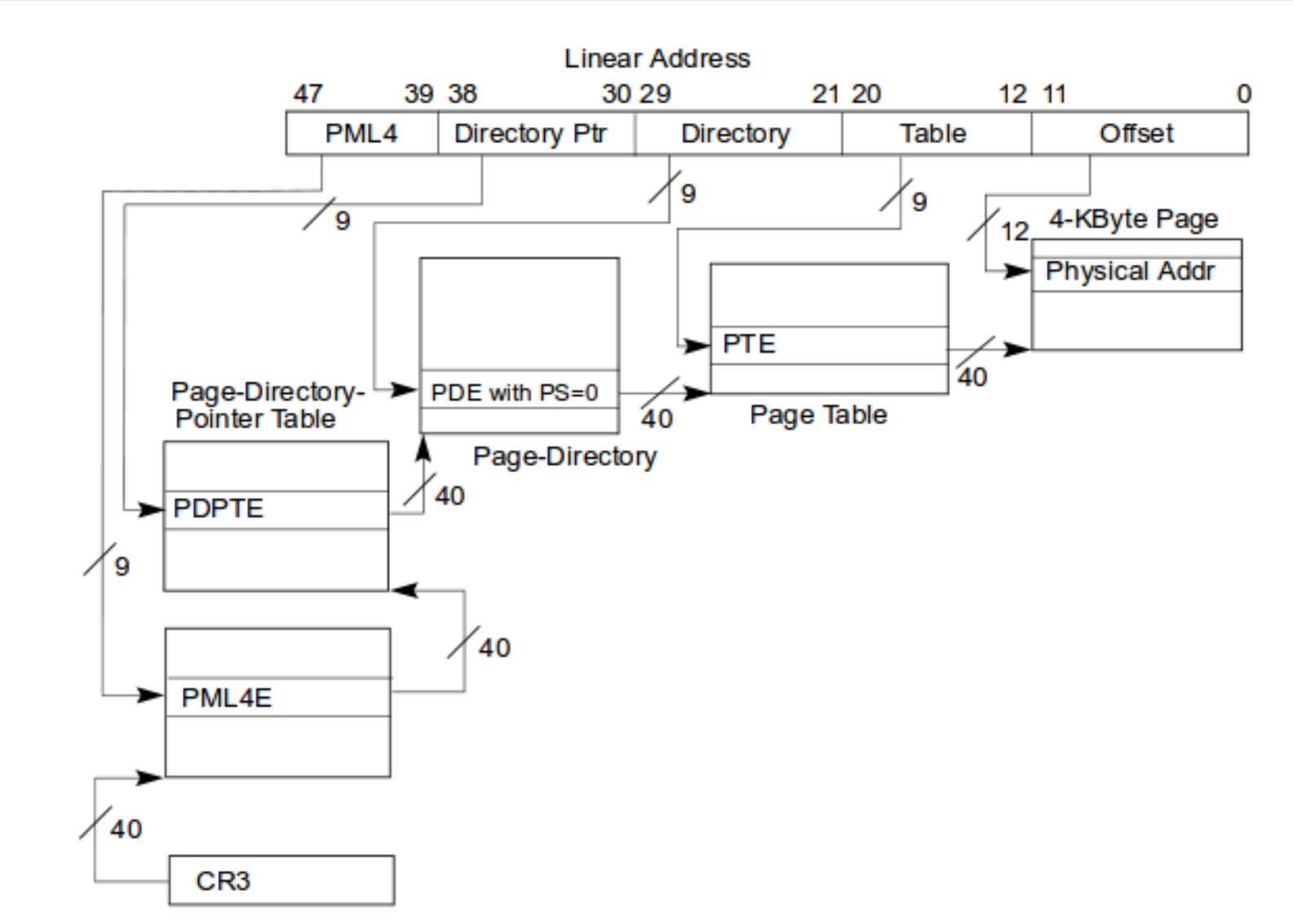

Figure 5

x86-64 page table entry

Bit	Interpretation
0 (P)	Present; must be 1 to map a 4-KB page
1 (R/W)	Read/write; if 0, writes may not be allowed
2 (U/S)	User/supervisor; if 0 access with CPL=3 not allowed
3 (PWT)	Page-level write-through
4 (PCD)	Page-level cache disable
5 (A)	Accessed
6 (D)	Dirty
7 (PAT)	Page Attribute Table (PAT) indicator or reserved (0)
8 (G)	Global
11:9	Ignored
(M-1):12	Physical address of the 4-KB page
51:M	Reserved (0)
62:52	Ignored
63 (XD)	*If IA32_EFER.NXE = 1, execute-disable if 1*

According to Intel 64 and IA-32 ... vol 3A, Table 4-19

Alternate approaches

- Not all hardware supports PAE
- ExecShield by Ingo Molnar used code segment restrictions to limit code execution on x86 for Linux.
- OpenBSD's W^X follows the same idea.
- PaX, also for Linux, is ExecShield on steroids with significant remapping of code region (and may break applications).

DEP summary

- DEP is now enabled widely by default (whenever a hardware support is available such as for x86 and ARM)
- Stops all code injection.
- Check for DEP with checksec.sh
- DEP may be disabled through gcc flags: `-z execstack`

Attacks evolve: from code injection to reuse

- Did DEP solve all code execution attacks?

Attacks evolve: from code injection to reuse

- Did DEP solve all code execution attacks?

- Unfortunately not! Attacks got (much?) harder though.
- A code injection attack consists of two stages:

1. redirecting control flow to
2. injected code. DEP stops the second stage.

- Attackers can still redirect control flow to *existing* code

Code reuse

- The attacker can overwrite a code pointer (e.g., a function pointer, a return pointer on the stack, the vtable pointer to an array of code pointers of a C++ object, or an exception frame for C++)
- Prepare the right parameters on the stack, reuse a full function (or part of a function)

From Code Reuse to full ROP

Instead of targeting a simple function, we can target a gadget.

- Gadgets are a sequence of instructions ending in an indirect control-flow transfer (e.g., return, indirect call, indirect jump)
- Prepare data and environment so that, e.g., pop instructions load data into registers
- A gadget invocation frame consists of a sequence of 0 to n data values and an pointer to the next gadget. The gadget uses the data values and transfers control to the next gadget

Simple ROP tutorial

Understanding ROP: gadgets galore!

- `ROPgadget --binary /bin/bash` ROPGadget

```
...
0x00000000004fe8a9 : xor edi, esp ; call rsi
0x00000000004692ec : xor edx, edx ; jmp 0x4692dd
0x0000000000432960 : xor edx, edx ; mov dword ptr [rax + 8]
0x0000000000460b61 : xor edx, edx ; mov esi, r13d ; call 0x
0x0000000000047bbc0 : xor edx, edx ; test esi, esi ; sete dl
0x00000000004ea831 : xor esi, ebp ; call qword ptr [rbx]
0x00000000004747ee : xor esi, esi ; call 0x470a94
0x00000000004ee0b9 : xor esi, esi ; call qword ptr [rax]
0x00000000004caa49 : xor esi, esi ; call qword ptr [rdi]
0x0000000000485634 : xor esi, esi ; sub rsp, 8 ; call 0x431
0x00000000004c890d : xor esp, dword ptr [rax + rax] ; ret
0x00000000004b3920 : xor esp, esp ; call 0x47ef29

Unique gadgets found: 11699
```

Understanding ROP: shellcode translation

- Shellcode can generally be translated to ROP sequences
  - ```
    xor eax, eax
    xor ebx, ebx
    inc eax
    int $0x80
    ```
- Find gadgets for each instruction:
  - ```
    xor eax, eax; ret
    xor ebx, ebx; ret
    inc eax; ret
    int $0x80
    ```
- Overwrite the stack with these addresses so that they are called in sequence.

Understanding ROP: stack pivots

- Often, a buffer overflow allows only control of one single RIP
- Idea: overwrite RIP *and* stack pointer, relocate to some area controlled by the attacker
- Search for `esp` gadgets, e.g., `add esp, 0x40c; ret`

How to stop code reuse?

- Enforce control-flow integrity?

How to stop code reuse?

- Enforce control-flow integrity?
- Nah, that's too easy! (hard?)

Address Space Randomization (ASR)

- Successful control-flow hijack attacks depend on the attacker overwriting a code pointer with a known alternate target.
- ASR changes (randomizes) the process memory layout.
- If the attacker does not know where a piece of code (or data) is, then it cannot be reused in an attack.
- Attacker must first *learn* or recover the address layout.

ASR effectiveness

The security improvement of ASR depends on (i) the entropy available for each randomized location, (ii) the completeness of randomization (i.e., are all objects randomized), and (iii) the absence of any information leaks.

Candidates for randomization

- Trade-off between overhead, complexity, and security benefit.
- Randomize start of heap
- Randomize start of stack
- Randomize start of code (PIE for executable, PIC each library)
- Randomize mmap allocated regions
- Randomize individual allocations (malloc)
- Randomize the code itself, e.g., gap between functions, order of functions, basic blocks, ...
- Randomize members of structs, e.g., padding, order.

Different forms of fine-grained randomization exist. Software diversity is a related concept.

Address Space *Layout* Randomization (ASLR)

ASLR is a practical form of ASR

- ASLR focuses on blocks of memory
- Heap, stack, code, executable, mmap regions
- ASLR is inherently page-based

ASLR entropy

- Entropy of each section is key to security (if all sections are randomized).
- Attacker follows path of least resistance, i.e., targets the object with the lowest entropy.
- Early ASLR implementations had low entropy on the stack and no entropy on x86 for the main executable (non-PIE executables).
- Linux (through Exec Shield) uses 19 bits of entropy for the stack (on 16 byte period) and 8 bits of mmap entropy (on 4096 byte period).

ASLR changes to the address space

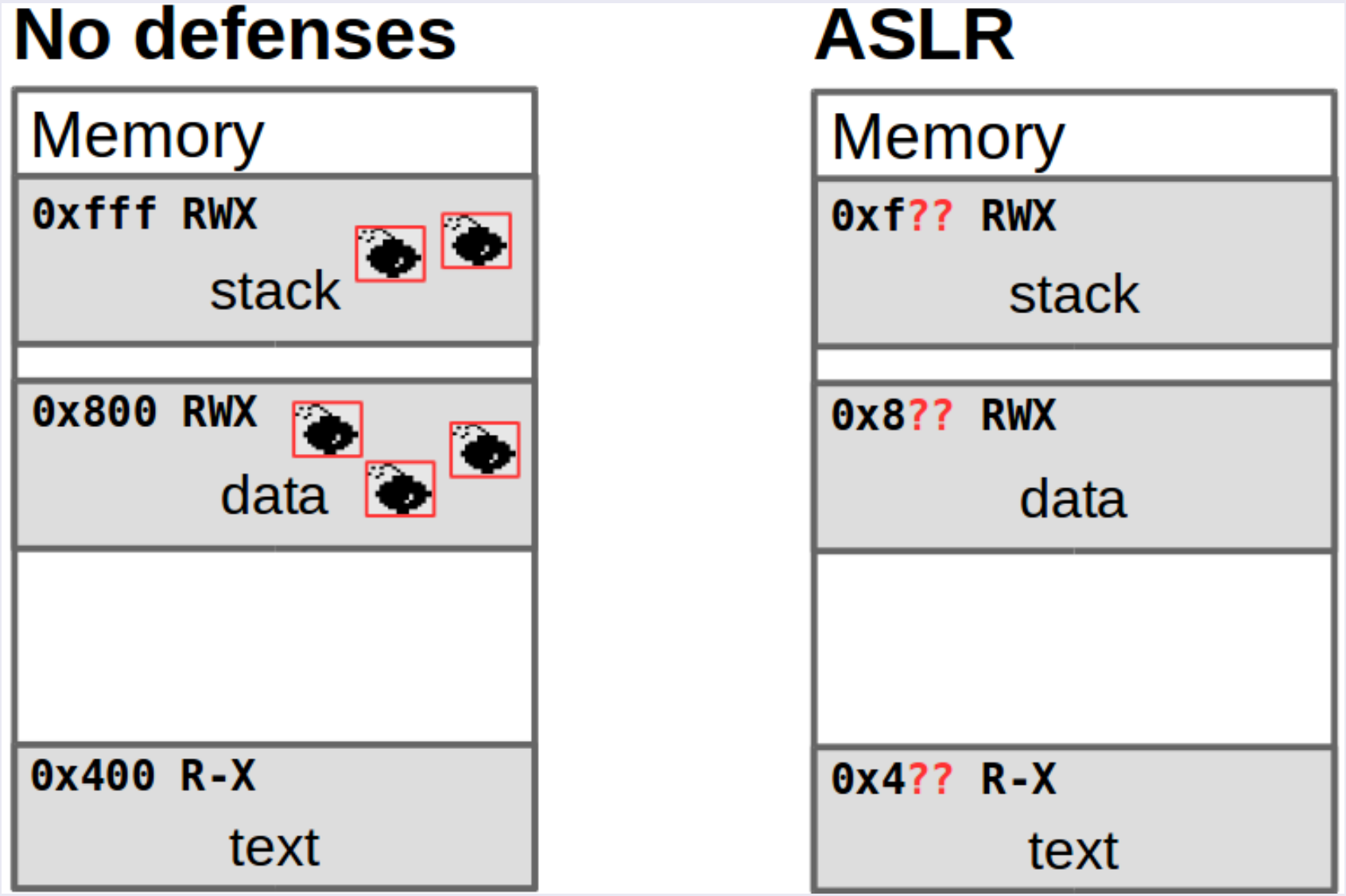

Figure 6

ASLR and DEP changes to the address space

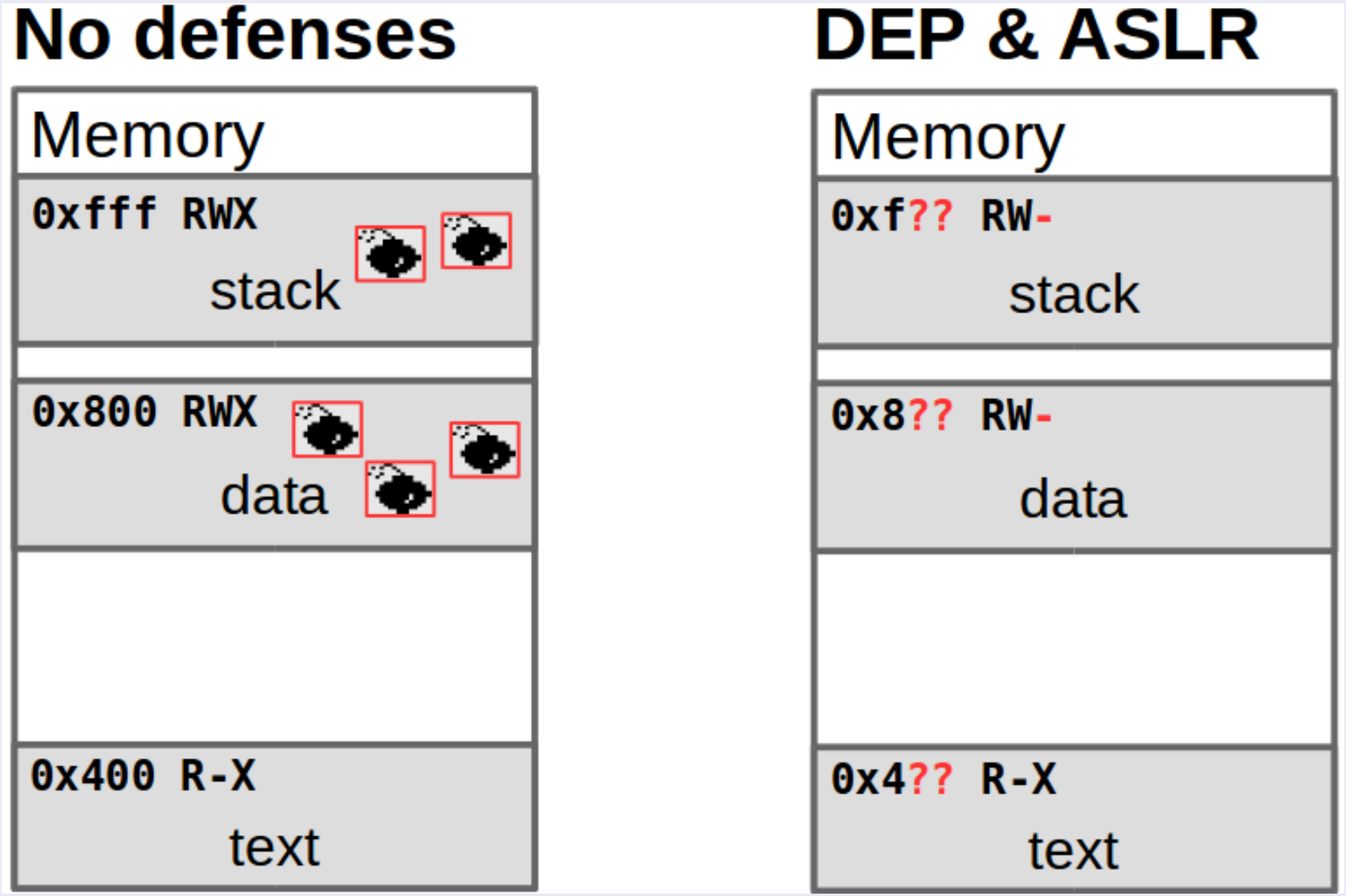

Figure 7

Stack canaries

- Early attacks relied on a stack-based buffer overflow to inject code.
- Memory safety would mitigate this problem but adding full safety checks is not feasible due to high performance overhead.
- Instead of checking each dereference to detect arbitrary buffer overflows we can add a check for the integrity of a certain variable.
- *Assumption:* we only prevent RIP control-flow hijack attacks.
- We therefore only need to protect the integrity of the return instruction pointer.

Key insight: buffer overflows are only possible after pointer arithmetic.

Stack canaries

- Place a canary after a potentially vulnerable buffer
- Check the integrity of the canary before the function returns.
- The compiler may place all buffers at the end of the stack frame and the canary just before the first buffer. This way, all non-buffer local variables are protected as well.
- Limitation: the stack canary only protects against *continuous overwrites* iff the attacker does *not know* the canary.
- An alternative is to encrypt the return instruction pointer by xoring it with a secret.

Stack protector: code

```
char unsafe(char *vuln) {
  char foo[12];
  strcpy(foo, vuln);
  return foo[1];
}

int main(int ac,
    char* av[]) {
  unsafe(argv[0]);
  return 0;
}
```

Stack protector: disabled

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %rdi,-0x18(%rbp)
mov    -0x18(%rbp),%rdx
lea    -0x10(%rbp),%rax
mov    %rdx,%rsi
mov    %rax,%rdi
callq  400410 <strcpy@plt>
movzbl -0xf(%rbp),%eax
leaveq
retq
```

Stack protector: enabled

```
push   %rbp; mov %rsp,%rbp
sub    $0x30,%rsp
mov    %rdi,-0x28(%rbp)
mov    %fs:0x28,%rax
mov    %rax,-0x8(%rbp)
xor    %eax,%eax
mov    -0x28(%rbp),%rdx
lea    -0x20(%rbp),%rax
mov    %rdx,%rsi
mov    %rax,%rdi
callq  <strcpy@plt>
movzbl -0x1f(%rbp),%eax
mov    -0x8(%rbp),%rcx
xor    %fs:0x28,%rcx
je     <out>
callq <__stack_chk_fail@plt>
out: leaveq, retq
```

Stack protector: added code

Prologue:

```
mov     %fs:0x28,%rax
mov     %rax,-0x8(%rbp)
xor     %eax,%eax
```

Epilogue:

```
mov     -0x8(%rbp),%rcx
xor     %fs:0x28,%rcx
je      <out>
callq <__stack_chk_fail@plt>
out:
leaveq
ret
```

Safe Exception Handling (SEH)

- Exceptions are a way of indirect control flow transfer in C++
- A safe alternative to setjmp/longjmp or goto
- Make control-flow semantics explicit in the programming language.
- Exceptions allow handling of special conditions.
- Exception-safe code safely recovers from thrown conditions.

Exceptions in C++

```
double div(double a, double b) {
   if (b == 0)
      throw "Division by zero!";
   return (a/b);
}
...
try {
  result = div(foo, bar);
} catch (const char* msg) {
  ...
}
```

Exception implementation

- Exception handling requires support from the code generator (compiler) and the runtime system (libc or libc++).
  - Implementation is compiler specific (libunwind for LLVM)
- There are two fundamental approaches: (a) inline exception information in stack frame or (b) generate exception tables that are used when an exception is thrown.

Inline exception handling

- The compiler generates code that registers exceptions whenever a function is entered.
- Individual exception frames are linked across stack frames.
- When an exception is thrown, the runtime system traces the chain of exception frames to find the corresponding handler.
- This approach is compact but results in overhead for each function call (as metadata about exceptions has to be allocated).

Exception tables

- Code generation emits per-function or per-object tables that link instruction pointers to program state with respect to exception handling.
- Throwing an exception is translated into a range query in the corresponding table, locating the correct handler for the exception.
- These tables are encoded very efficiently. This encoding may lead to security problems.

Windows exception handling

Microsoft Windows uses a combination of tables and inlined exception handling. Each stack frame records (i) unwinding information, (ii) the set of destructors that need to run, and (iii) the exception handlers if a specific exception is thrown.
When entering a function, a structured exception handling (SEH) record is generated, pointing to a table with address ranges for try-catch blocks and destructors. Handlers are kept in a linked list:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE                     Handler;
} EXCEPTION_REGISTRATION_RECORD,
  *PEXCEPTION_REGISTRATION_RECORD;
```

Attacking Windows exception handling

- An attacker may overwrite the first SEH record on the stack and point the handler to the first gadget.
- Microsoft developed two defenses: SAFESEH and SEHOP.
- SafeSEH forces the compiler to generate a list of allowed targets. If a record points to an unknown target it is rejected.
- SEHOP initializes the chain of registration records with a sentinel. If the sentinel is not present, the handler is not executed.
- What are the limitations of these defenses?

GCC exception handling

- GCC encodes all exception information in external tables.
- When an exception is thrown, the tables are consulted to learn which destructors need to run and what handlers are registered for the current IP location.
- This results in less overhead in the non-exception case (as additional code is only executed *on-demand* but otherwise jumped over.
- The information tables can become large and heavyweight *compression* is used, namely an interpreter that allows on-the-fly construction of the necessary data.
- Interpreter can be (ab-)used for Turing-complete execution

See James Oakley and Sergey Bratus, Exploiting the Hard-Working DWARF, WOOT'11.

Fortify source: format strings

- Format strings allow to generate formatted output.
- Format strings also allow to write to memory through `%n`.
- Format strings allow to jump over arguments and access stack slots out-of-bounds through, e.g., `%2$hn`.
- Format strings can be bad. Don't let the user control the first argument to `printf`.

Format string mitigations

- Deprecate use of `%n` (Windows); this is the sane option.
- Add extra checks for format strings (Linux):
  - Check for buffer overflows if possible.
  - Check if the first argument is in a read-only area. (how do you check if a segment is not writable?)
  - Check if all arguments are used.
- Linux checks the following functions: `mem{cpy,pcpy,move,set}`, `st{r,p,nc}py`, `str{,n}cat`, `{,v}s{,n}printf`.

Fortify source (buffers)

The GCC/GLIBC patch distinguishes between four cases:

- Known correct: do not check.
- Not known if correct, but checkable (i.e., compiler knows length of target): do check.
- Known incorrect: compiler warning, do check.
- Not known if correct, not checkable: no check, overflows may remain undetected.

Fortify source: not known if correct

```
int main(int argc, char *argv[]) {
  char buffer[10];
  char *q = argv[1];
  strcpy(buffer, q);
  return 0;
}
// gcc -O3 -Wall -Wpedantic -Wextra
          -D_FORTIFY_SOURCE=2 test.c
```

Checkable or not?

Fortify source: not known if correct

```
int main(int argc, char *argv[]) {
  char buffer[10];
  char *q = argv[1];
  strcpy(buffer, q);
  return 0;
}
// gcc -O3 -Wall -Wpedantic -Wextra
          -D_FORTIFY_SOURCE=2 test.c
```

Checkable or not?

Checkable! The size of buffer is known.

Fortify source: not known if correct

```
int main(int argc, char *argv[]) {
  char *p = argv[1];
  char *q = argv[0];
  strcpy(p, q);
  return 0;
}
// gcc -O3 -Wall -Wpedantic -Wextra
//       -D_FORTIFY_SOURCE=2 test.c
```

Checkable or not?

Fortify source: not known if correct

```
int main(int argc, char *argv[]) {
  char *p = argv[1];
  char *q = argv[0];
  strcpy(p, q);
  return 0;
}
// gcc -O3 -Wall -Wpedantic -Wextra
//       -D_FORTIFY_SOURCE=2 test.c
```

Checkable or not?

Not checkable!

Going past ROP: Control-Flow Bending

- Data-only attack: Overwriting arguments to `exec()`
- Non-control data attack: Overwriting is admin flag
- Control-Flow Bending (CFB): Modify function pointer to valid alternate target
  - Attacker-controlled execution along valid CFG
  - Generalization of non-control-data attacks
  - Each individual control-flow transfer is valid
  - Execution trace may not match non-exploit case

Control-Flow Bending research paper

Going past CFB: Block-Oriented Programming

- Express payload in C dialect, abstract into sets of constraints
- Find candidate basic blocks in original program
- Synthesize path through all candidate blocks
- Encode attack as state change, injected through bug

Block-oriented programming research paper

Finding computation in odd places

- Bugs enable attackers to break out of the program semantics.
- The existing program code can enable powerful abstractions if executed with different data.
  - Format strings are Turing complete!
  - printbf
  - Without bugs: conditionals in Mario Maker

Deployed mitigations

- Data Execution Prevention: stops code injection
- Address Space Randomization: makes code reuse harder
- Stack Canaries: stops stack-based buffer overflows
- Safe Exception Handling: makes exception hijacking harder
- Fortify Source: stops format string attacks and many `strcpy` errors

Summary and conclusion

- Several defense mechanisms have been adopted in practice. Know their strengths and weaknesses.
- Data Execution Prevention stops code injection attacks, but does not stop code reuse attacks.
- Address Space Randomization is probabilistic, shuffles memory space, prone to information leaks.
- Stack Canaries are probabilistic, do not protect against direct overwrites, prone to information leaks.
- Safe Exception Handling protects exception handlers. Reuse remains possible.
- Fortify source protects static buffers and format strings.