

WAR GAMES

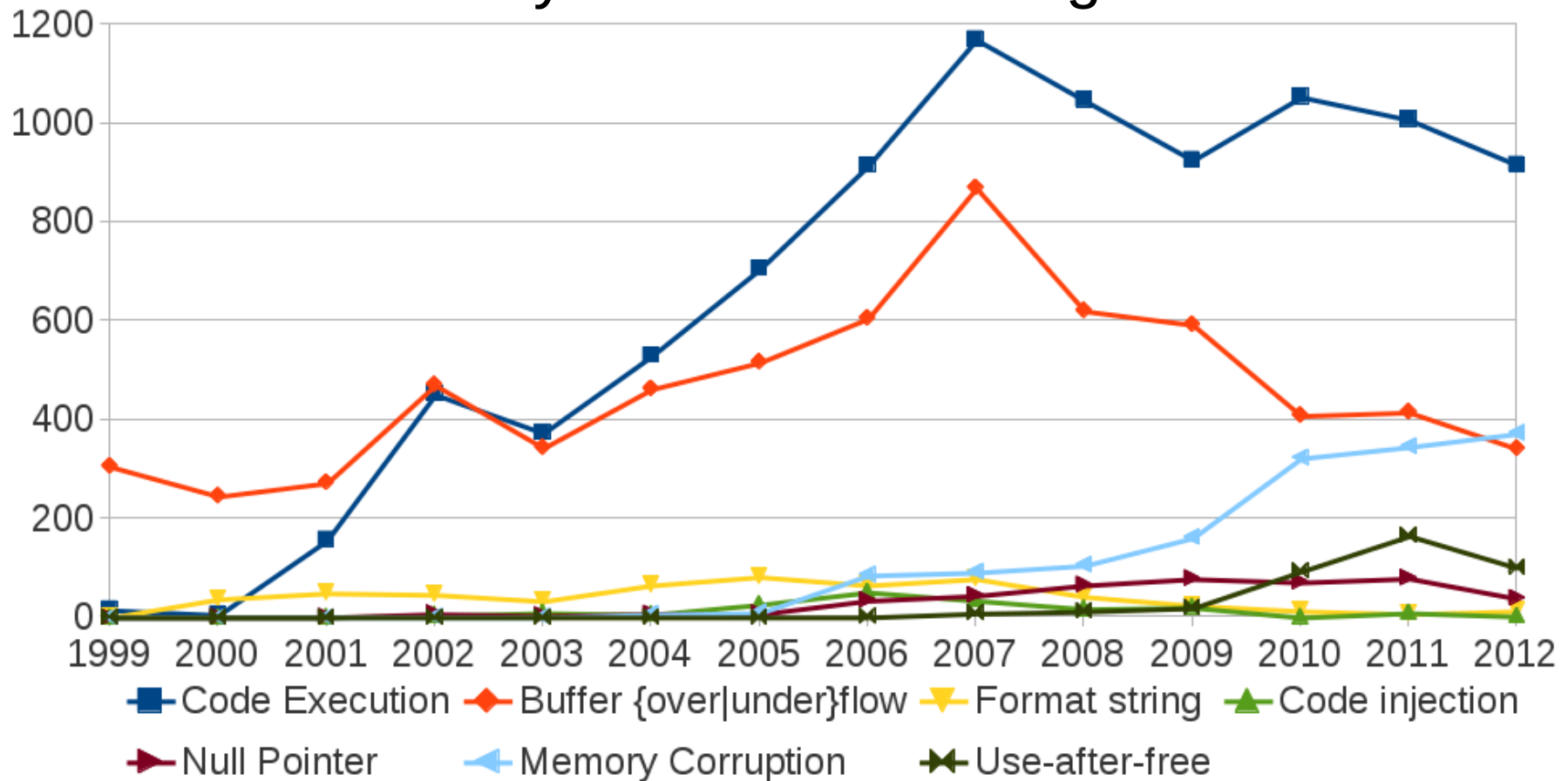
... in memory: an evolution of attacks



Mathias Payer <mathias.payer@nebelwelt.net>
Purdue University

Memory attacks: an ongoing war

Vulnerability classes according to CVE



Memory attacks: an ongoing war

- Low-level languages trade type safety and memory safety for performance
 - Programmer in control of all checks

- Large
written

David Lightman: Hey, I don't believe that any system is totally secure."

- Too many
– Pro

policy

Memory corruption

SHALL WE PLAY A GAME?

Memory corruption

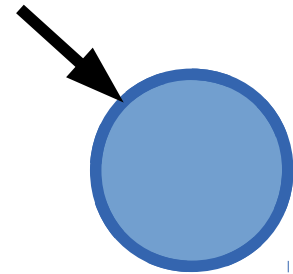
- Unintended modification of memory location due to missing / faulty safety check
 - Exploitable only if address or value input dependent
 - Attacker sees all memory, controls writable memory

```
void vulnerable(int user1, int *array) {  
    // missing bound check for user1  
    array[user1] = 42;  
}
```


Memory safety: temporal error



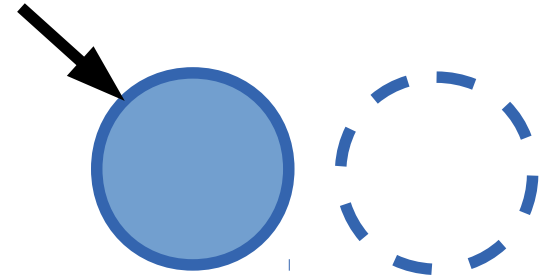
```
void vulnerable(char *buf) {  
    free(buf);  
    buf[12] = 42;  
}
```



Memory safety: spatial error



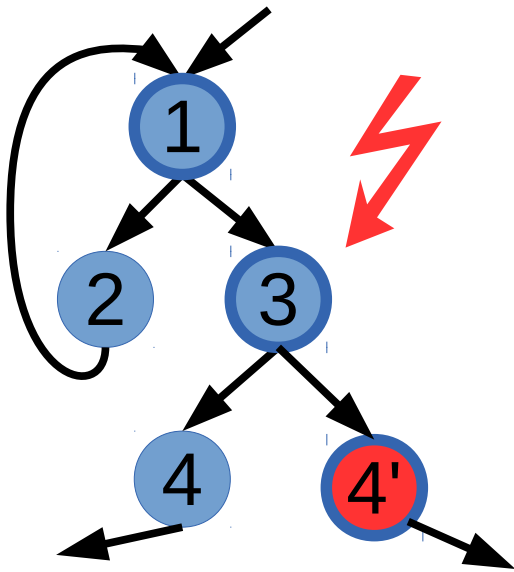
```
void vulnerable() {  
    char buf[12];  
    char *ptr = buf[11];  
    *ptr++ = 10;  
    *ptr = 42;  
}
```





**Control-flow hijacking:
Attack opportunities**

Control-flow hijack attack



- Attacker modifies ***code pointer***
 - Function return
 - Indirect jump
 - Indirect call
- Control-flow leaves static graph
- Reuse existing code
 - Return-oriented programming
 - Jump-oriented programming

Control-flow hijack attack

```
void vuln(char *u1) {  
    // assert(strlen(u1)) < MAX  
    char tmp[MAX];  
    strcpy(tmp, u1);  
    return strcmp(tmp, "foo");  
}  
vuln(&exploit);
```

don't care

don't care

points to &system()

ebp after system call

1st argument to system()

Memory safety Violation

Integrity

*C

Randomization

&C

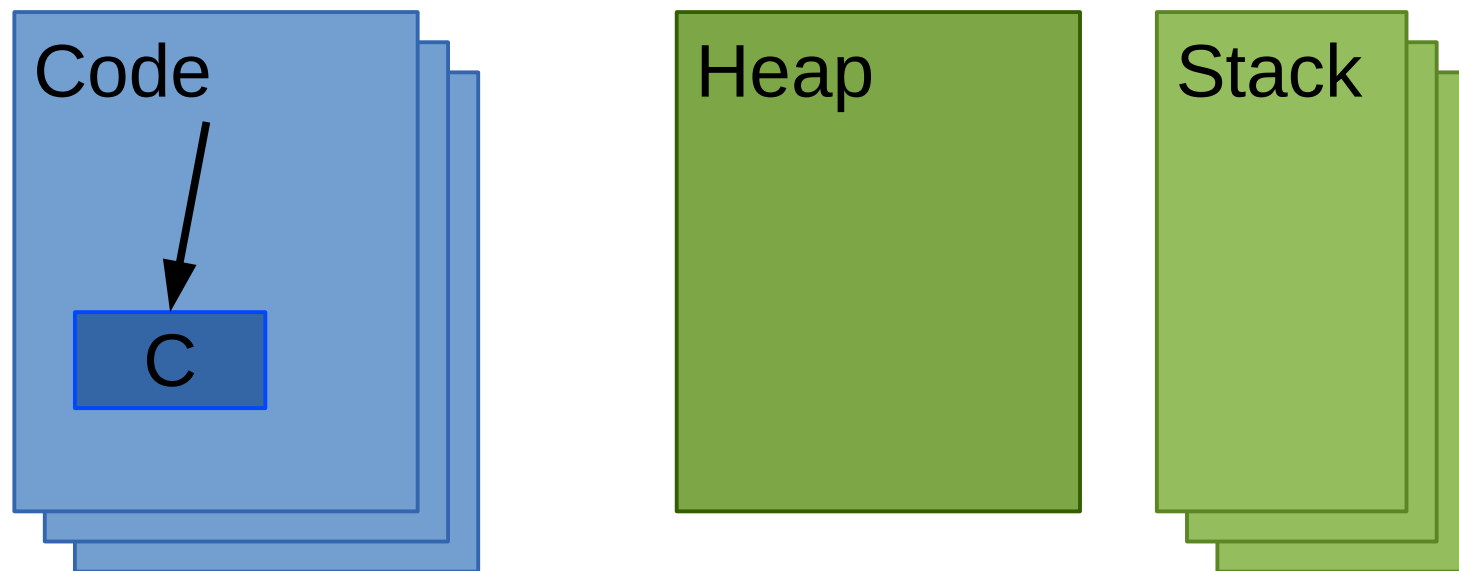
Flow Integrity

*&C

Attack

Control-flow
hijack

Code corruption attack



- Code modified or new code added
- Hardware protection enforces code integrity

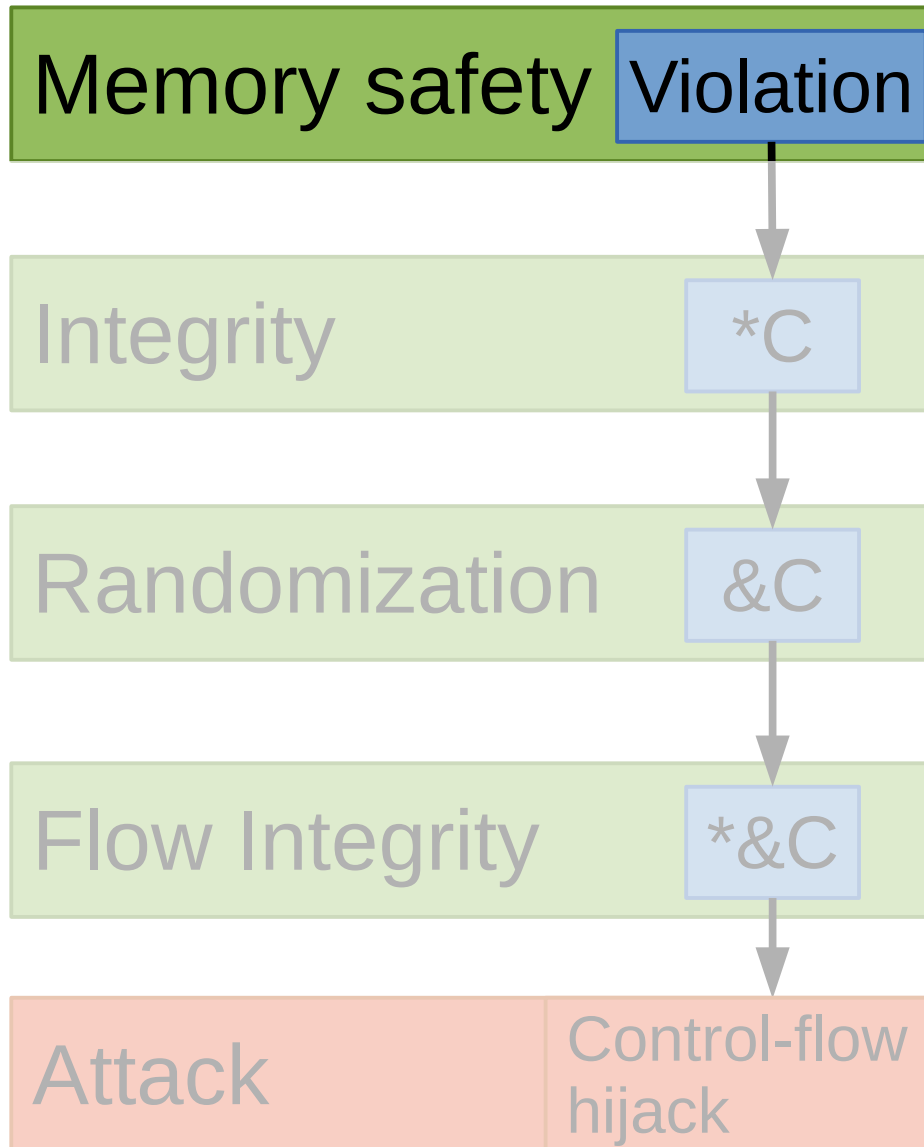
Memory corruption attacks*

- Model allows reasoning and classification
 - Classify security policies and defense mechanisms
 - Reason about power of attacks
- Identify properties that enable wide adoption
 - Low overhead is key (<10%)
 - Compatibility to legacy code and source code
 - Protection against class(es) of attacks



Control-flow hijacking: Defense strategies

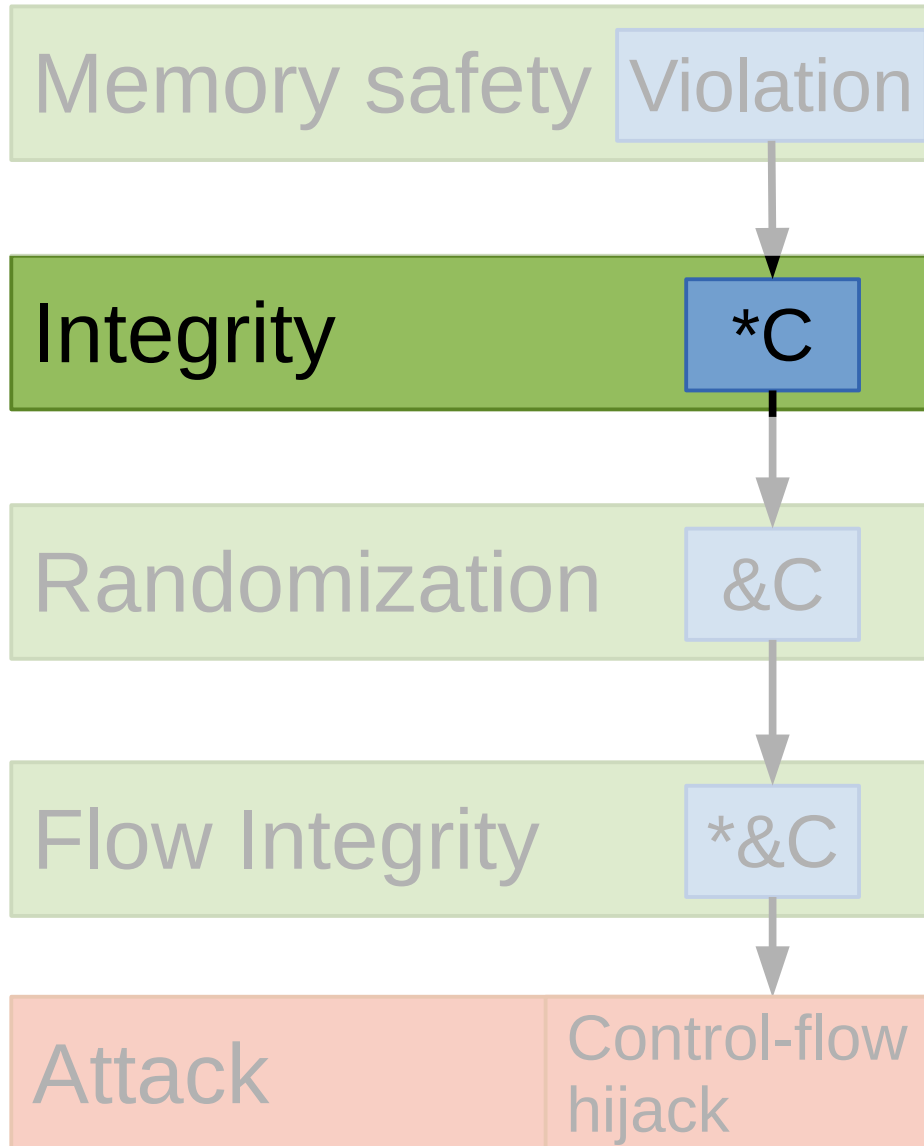
Defense strategies



Stop memory corruption

- Safe dialects of C/C++:
CCured, Cyclone
- Retrofit on C/C++:
SoftBounds+CETS
- Rewrite in safe language:
Java/C#

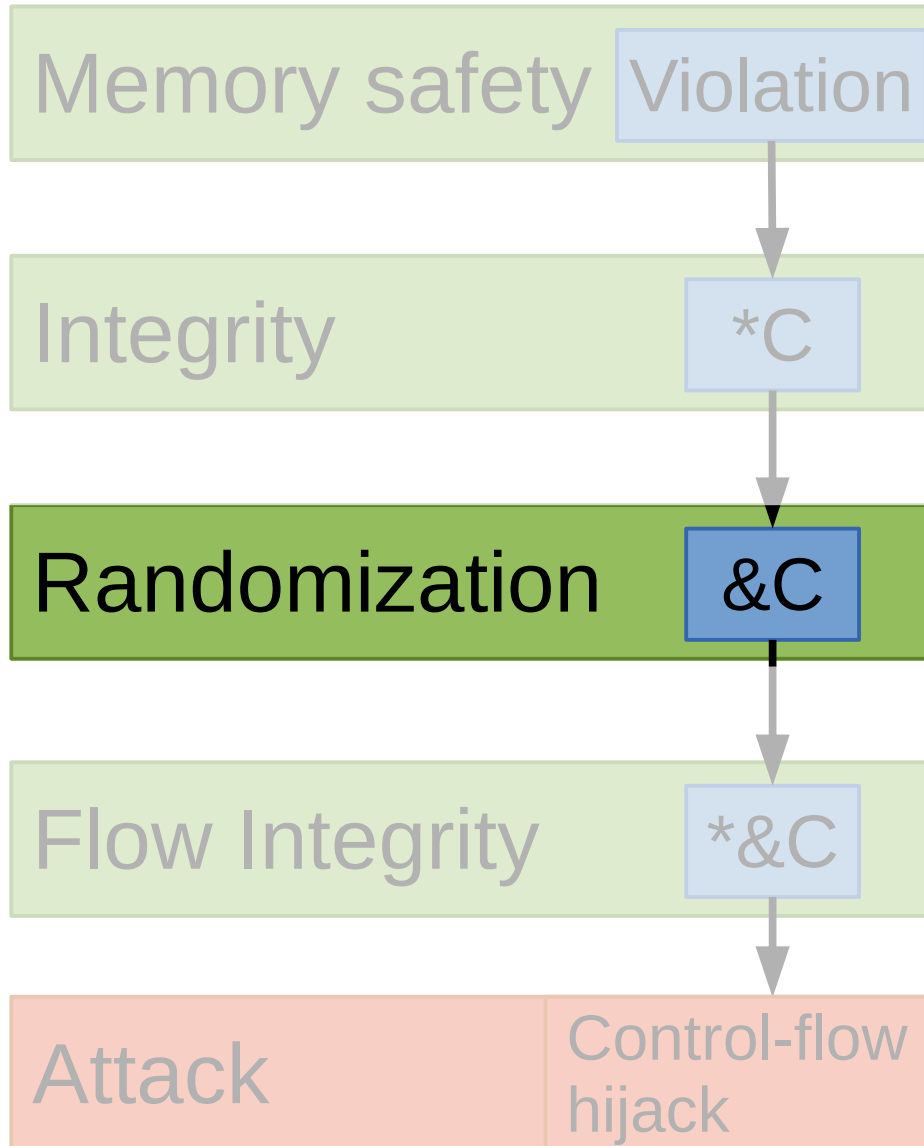
Defense strategies



Enforce integrity of reads/writes

- Write Integrity Testing
- (DEP and W^X for code)

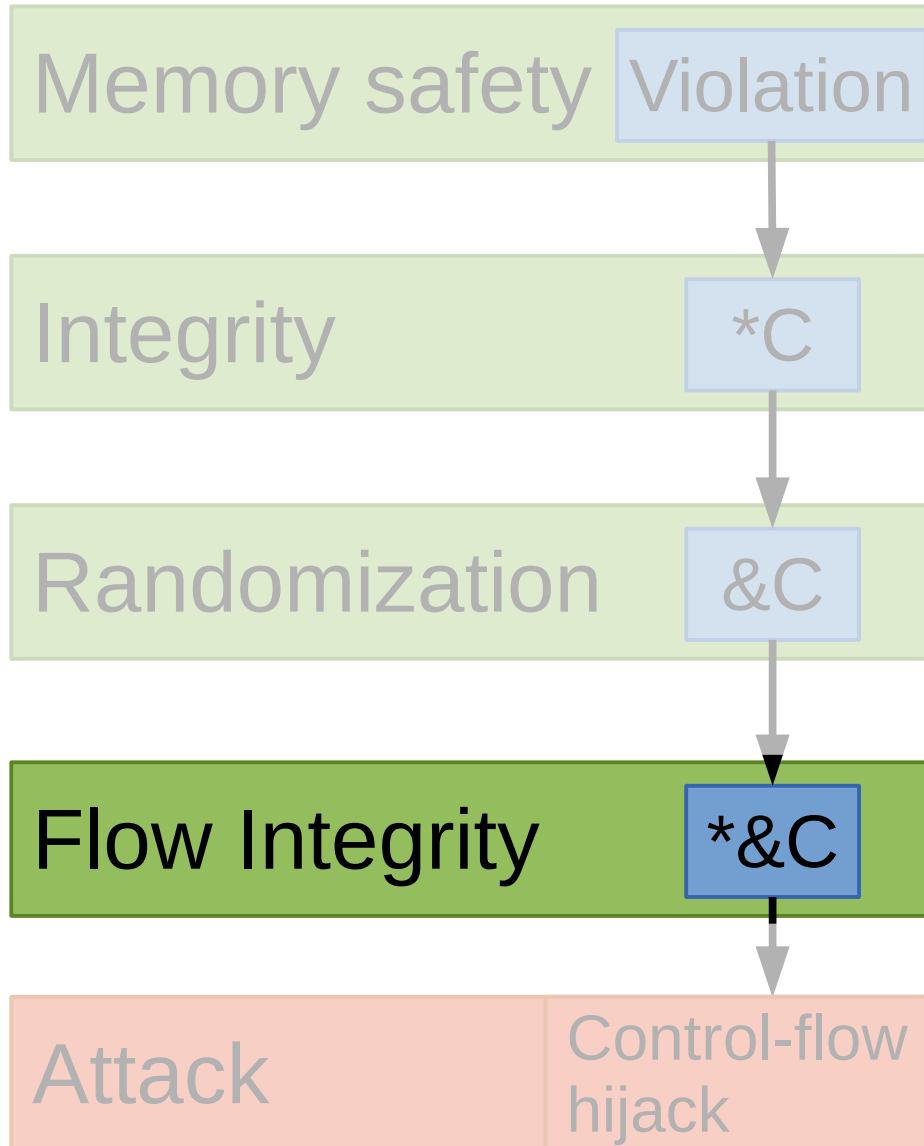
Defense strategies



Probabilistic defenses

- Randomize locations, code, data, or pointer values

Defense strategies

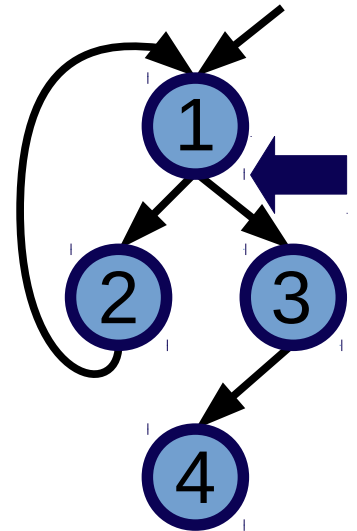


Protect control transfers

- Data-flow integrity
- Control-flow integrity

Control-Flow Integrity

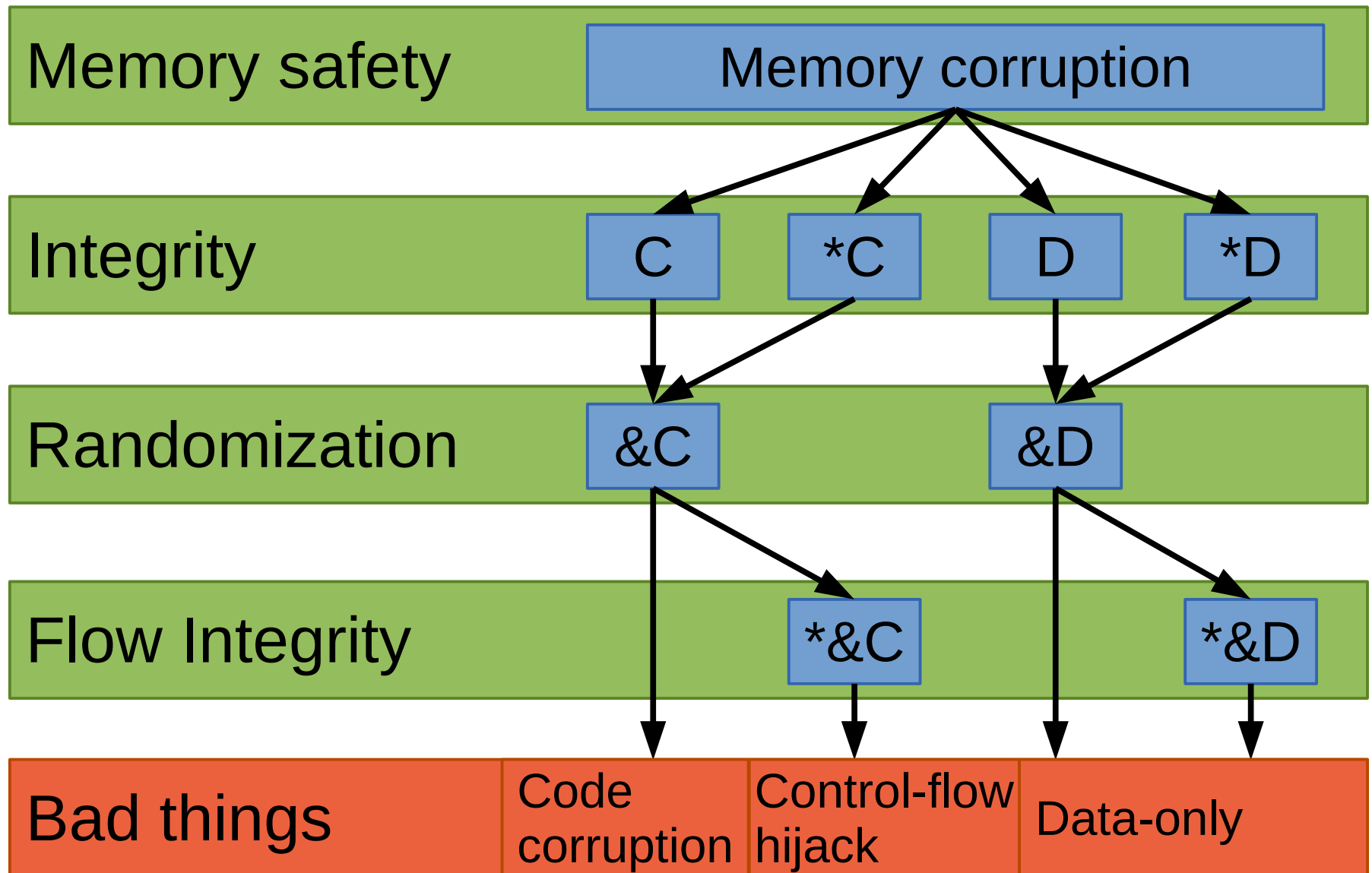
- Dynamic control flow must follow the static control flow graph (CFG)
 - Use points-to analysis to get CFG
 - Runtime check if target in static set
- Current implementations over-approximate
 - Imprecision of static analysis, runtime concerns
 - One set each for indirect calls, jumps, and returns



CFI: Limitations and Drawbacks

- Precision limited by static type analysis
 - Imprecision leads to ambiguities
- Static analysis must “see” all code
 - Support for dynamic libraries challenging
- Performance overhead or imprecision
 - Current implementations (greatly) over-approximate target set to achieve performance and compatibility

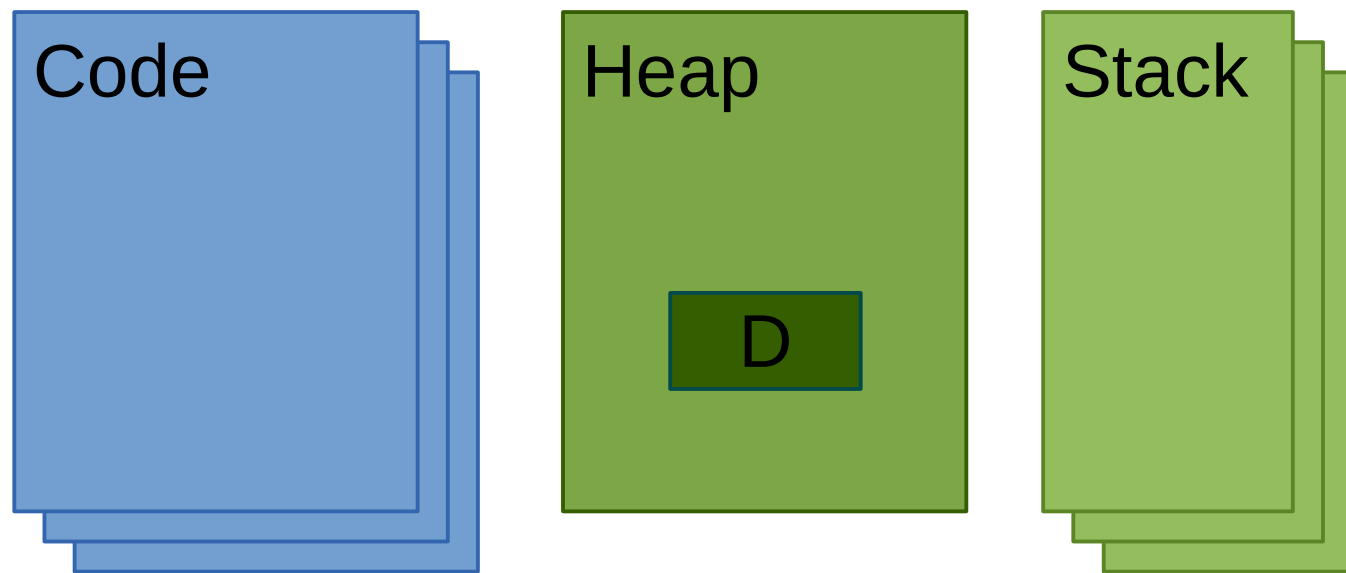
Model for memory attacks



A stylized map of the Great Lakes region, primarily in shades of blue and green. The map shows the outlines of the lakes and surrounding landmasses. Various symbols are scattered across the map, including small blue triangles, clusters of small blue squares, and a single yellow star. Dashed lines cross the map, possibly representing shipping lanes or boundaries. Numbers like 20, 22, 24, 25, 26, and 27 are placed in different areas. Labels for 'GRAND FORKS' and 'LORING' are visible. The text 'Data-only attacks' is overlaid in the center in a large, bold, black font.

Data-only attacks

Data-only attack



- Privileged or informative data changed
 - Simple, powerful and hard to detect

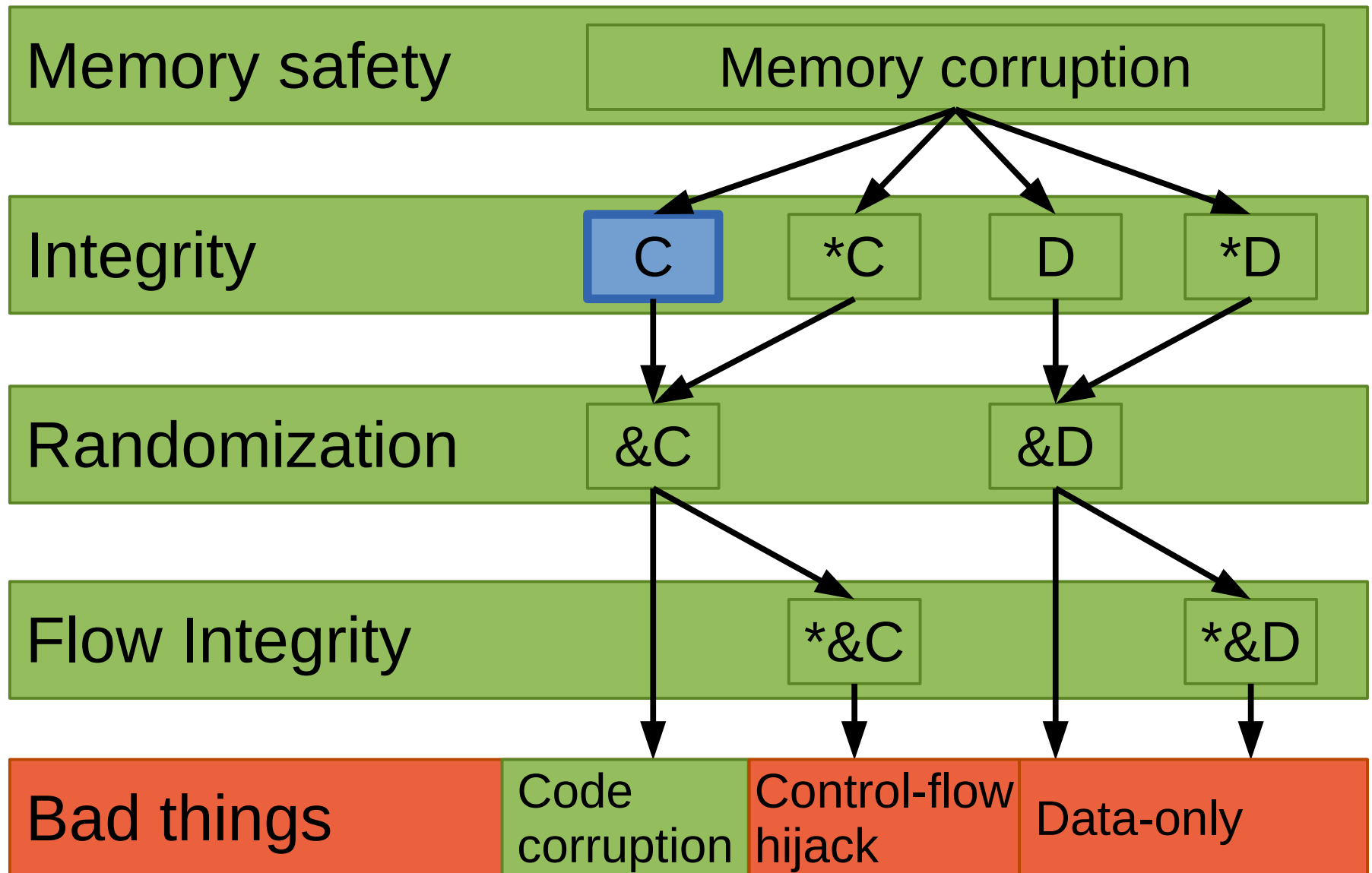


Deployed defenses

Data Execution Prevention

- Enforces code integrity on page granularity
 - Execute code if eXecutable bit set
- W^X ensures write access or executable
 - Mitigates against code corruption attacks
 - Low overhead, hardware enforced, widely deployed
- Weaknesses and limitations
 - No-self modifying code supported

Data Execution Prevention

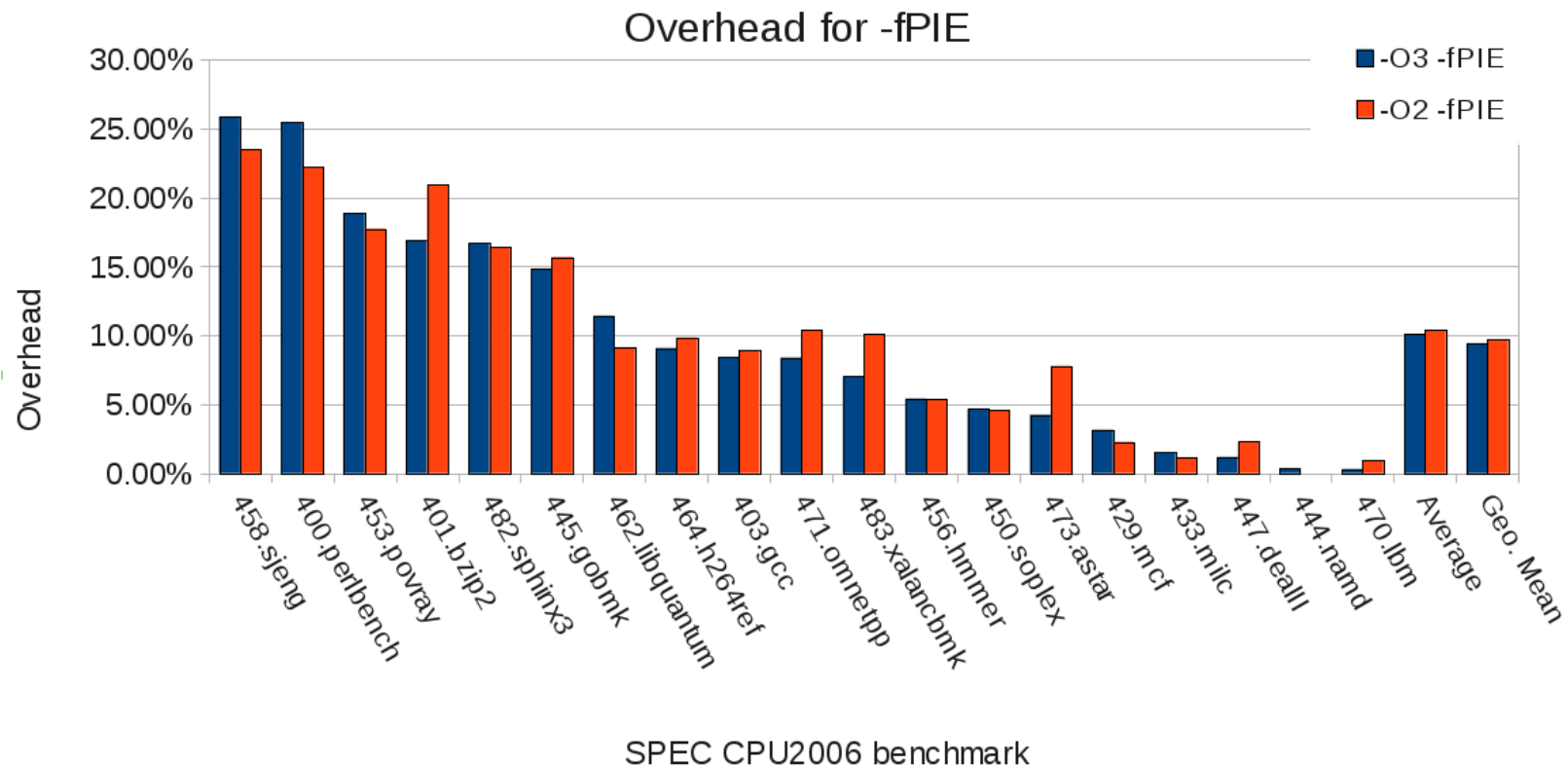


Address Space Layout Randomization

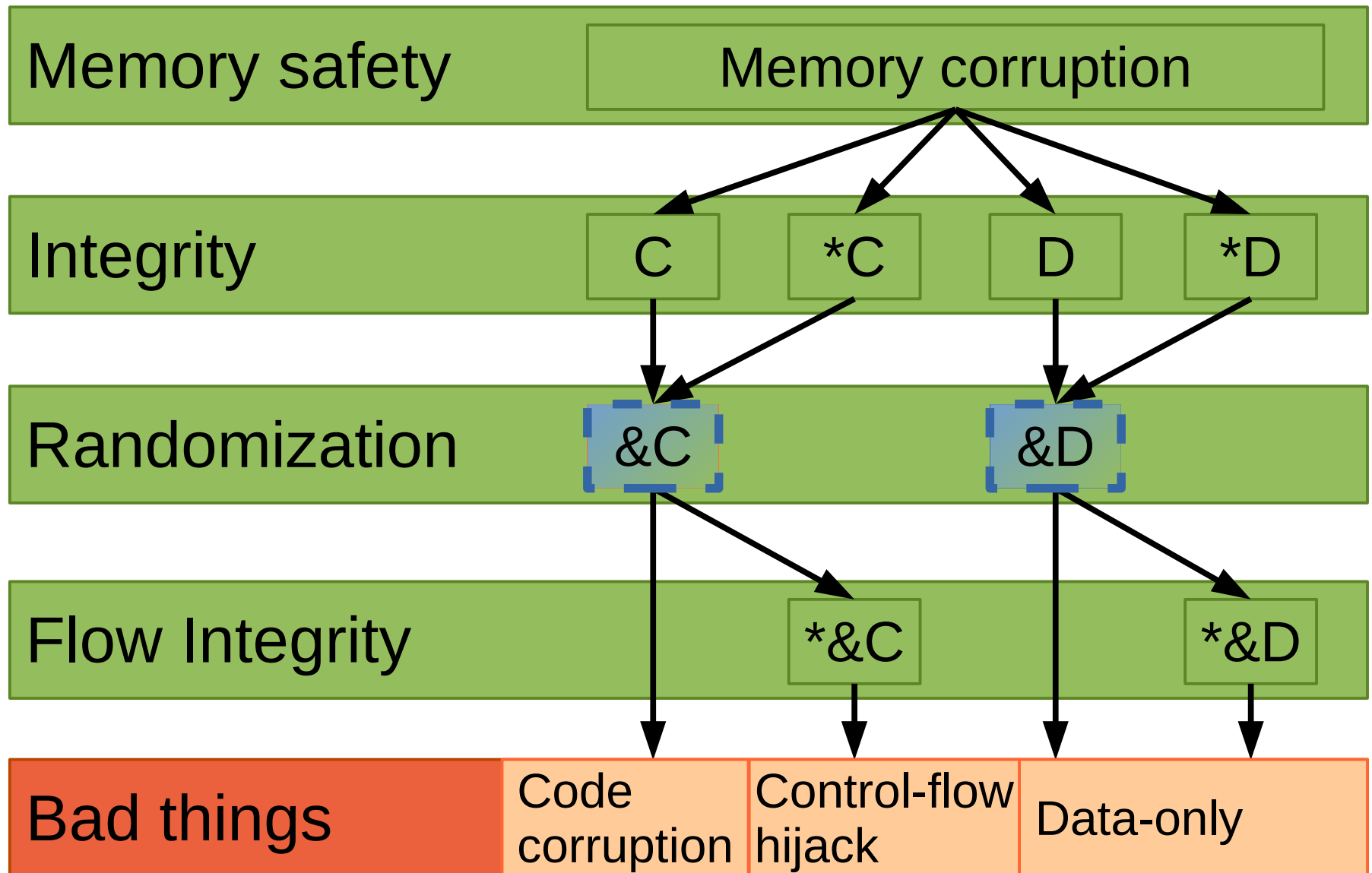
- Randomizes locations of code and data regions
 - Probabilistic defense
 - Depends on loader and OS
- Weaknesses and limitations
 - Prone to information leaks
 - Some regions remain static (on x86)
 - Performance impact (~10%)

ASLR: Performance overhead

- ASLR uses one register for PIC / ASLR code
 - Performance degradation on x86



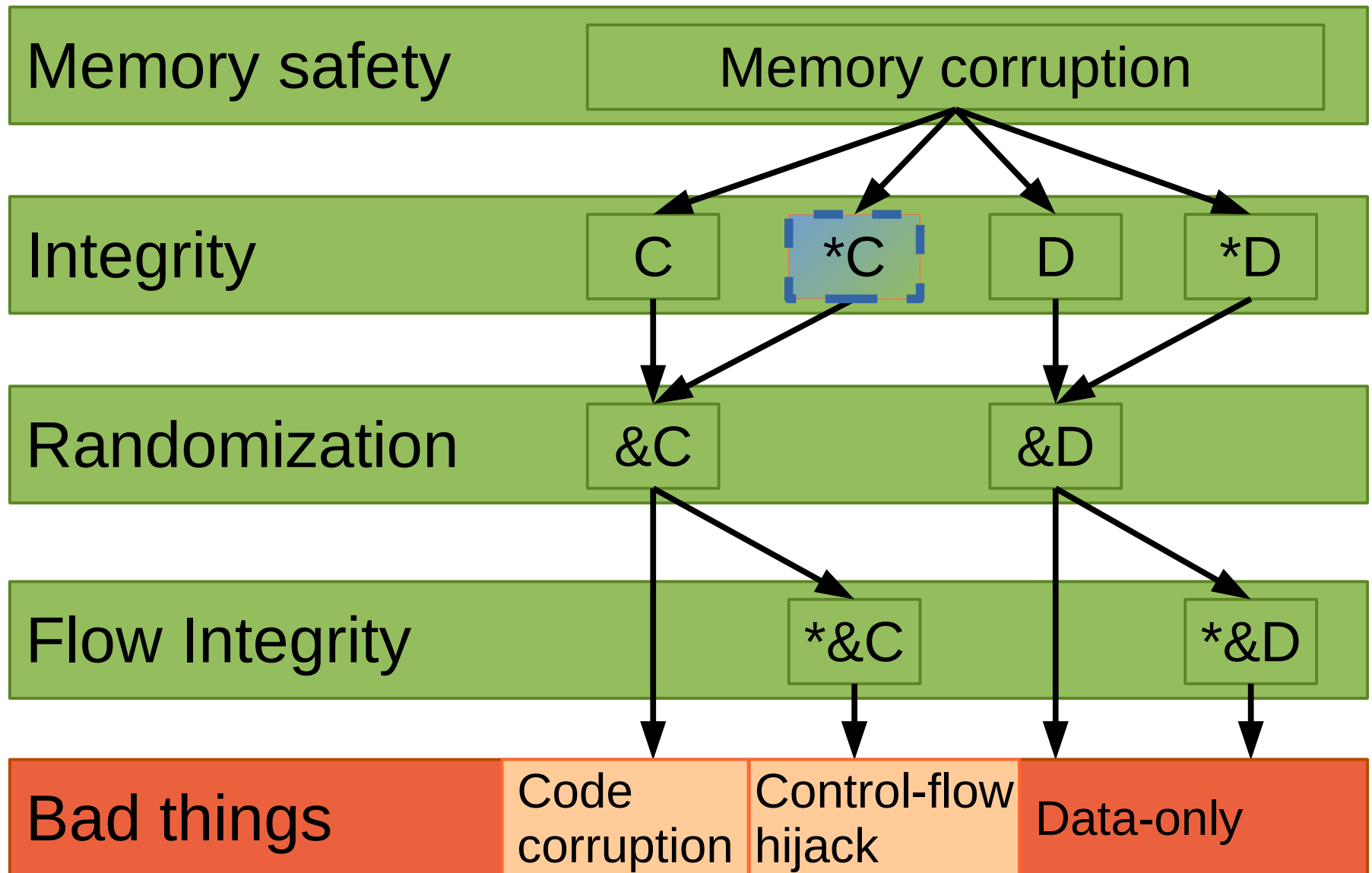
Address Space Layout Randomization



Stack canaries

- Protect return instruction pointer on stack
 - Compiler modifies stack layout
 - Probabilistic protection
- Weaknesses and limitations
 - Prone to information leaks
 - No protection against targeted writes / reads

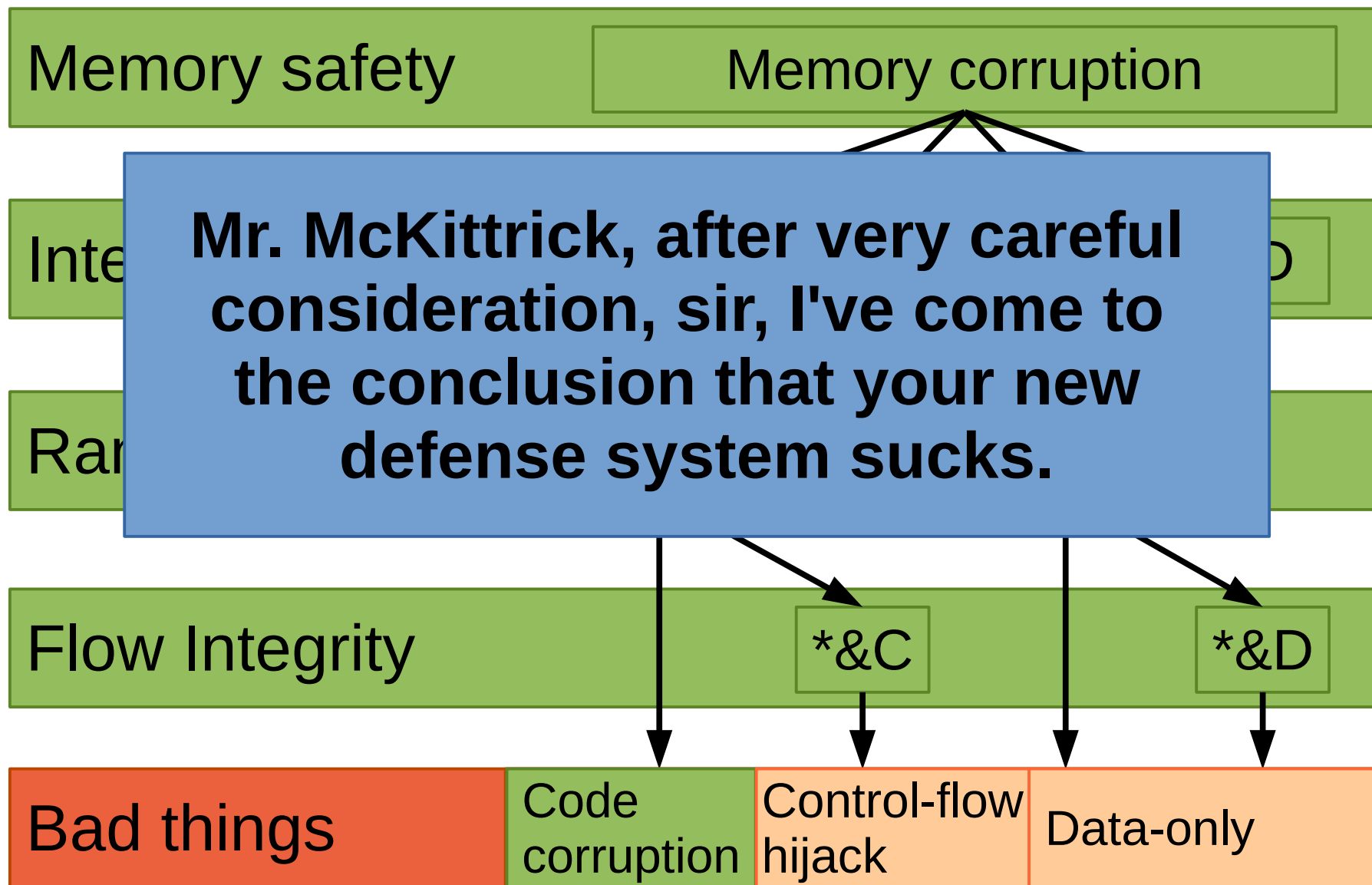
Stack canaries



Widely deployed defenses

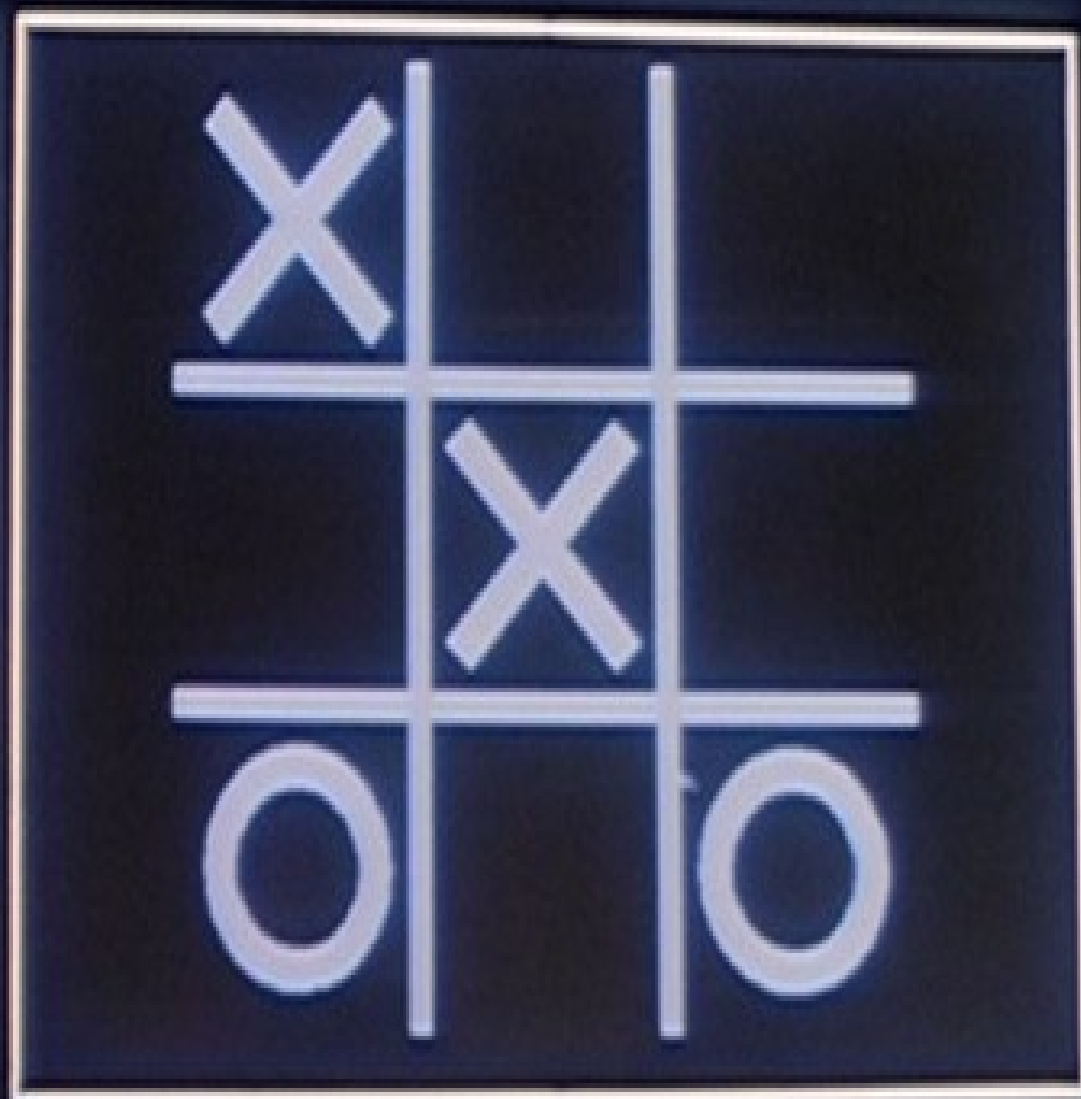
- Memory safety: none
- Integrity: partial
 - Code integrity: W^X
 - Code pointer integrity: canaries and safe exceptions
 - Data integrity: none
- Randomization: partial
 - Address Space Layout Randomization
- Control/Data-flow integrity: none

Widely deployed defenses



Why did stronger defenses fail?

- Too much overhead
 - More than 10% is not feasible
- Compatibility to legacy and source code
 - Shared library support, no code modifications
- Effectiveness against attacks
 - Protection against complete classes of attacks



Onwards?

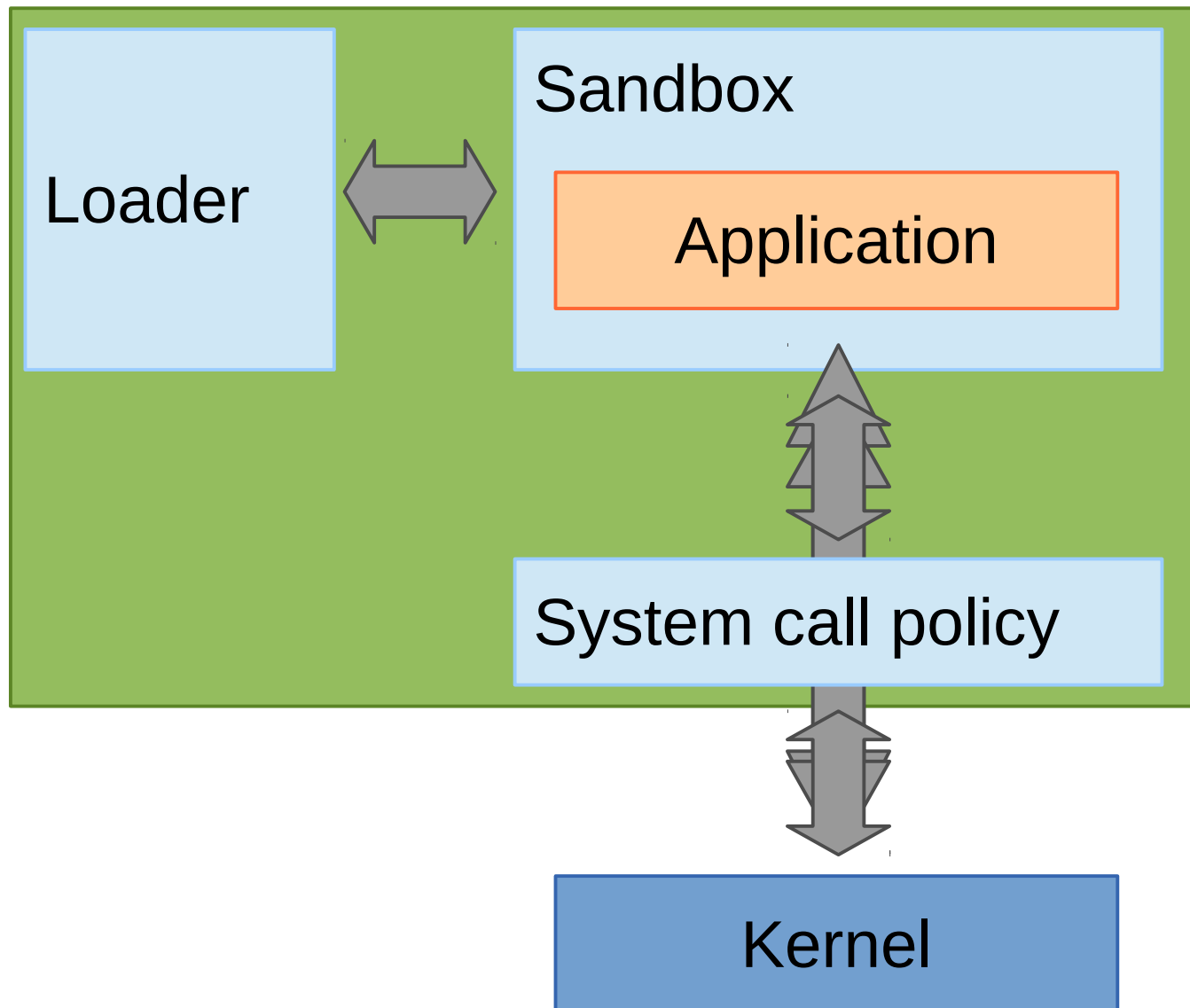
Partial? Data Integrity

- Memory safety stops control-flow hijack attacks
 - ... but memory safety has high overhead
 - SoftBounds+CETS reports up to 250% overhead
- Enforce memory safety for “some” pointers
 - Compiler analysis can help
 - Tricky engineering to make it work

Secure execution platform

- Must support legacy, binary code
- Dynamic binary translation allows virtualization
- Leverage runtime information
 - Enables preciser security checks

Secure execution platform

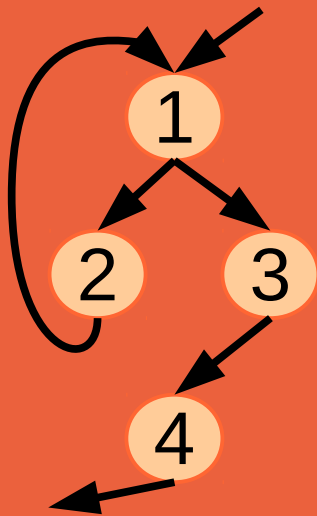


Sandbox implementation

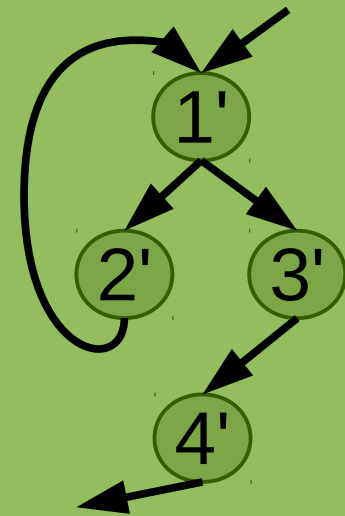
Dynamic binary translator

- Check targets and origins
- Weave guards into code

Original code



Protected code



GREETINGS PROFESSOR FALKEN

HELLO

A STRANGE GAME.

THE ONLY WINNING MOVE IS
NOT TO PLAY.

HOW ABOUT A NICE GAME OF CHESS?

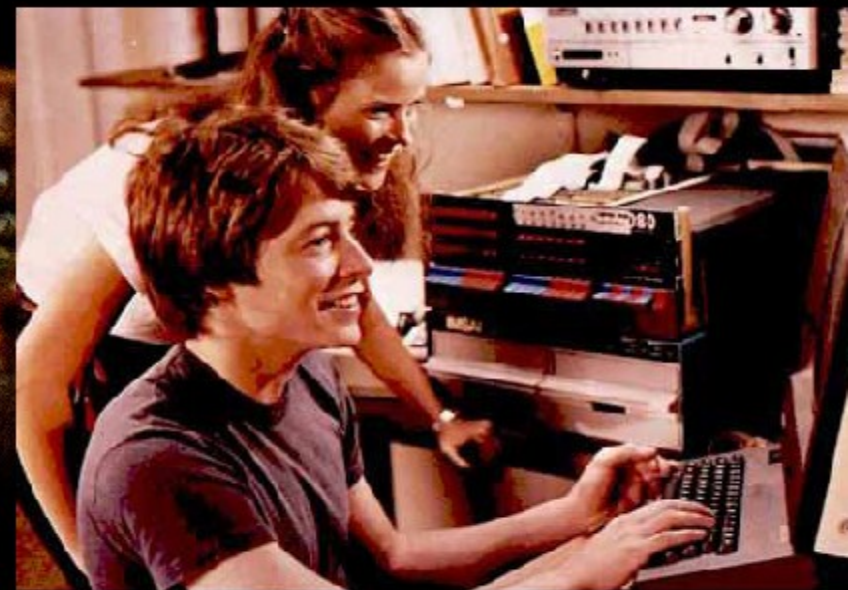
Conclusion

- Low level languages are here to stay
 - We need protection against memory vulnerabilities
 - Performance, legacy, compatibility
- Mitigate control-flow hijack attacks
 - Secure execution platform for legacy code
- Future directions: strong policies for data

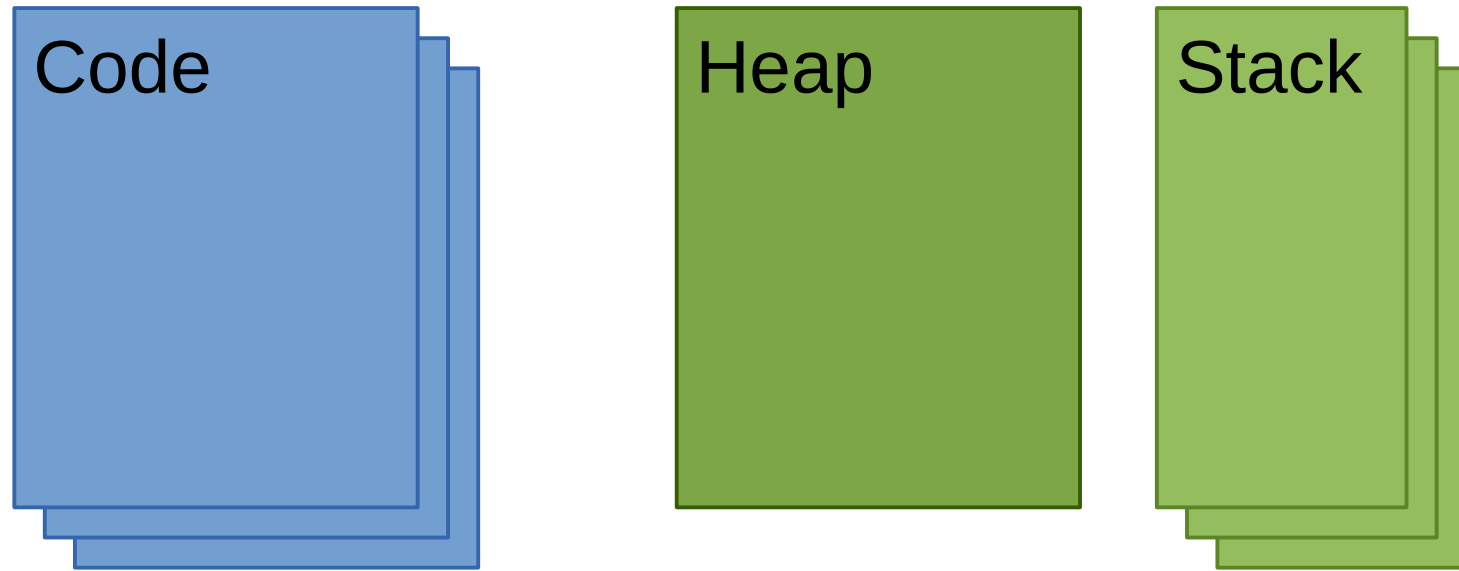


If the winning move is not to
play then we need to change
the rules of the game!

<http://nebelwelt.net>



Address space



- No separation between code and data memory
- Code pointers and data pointers mixed