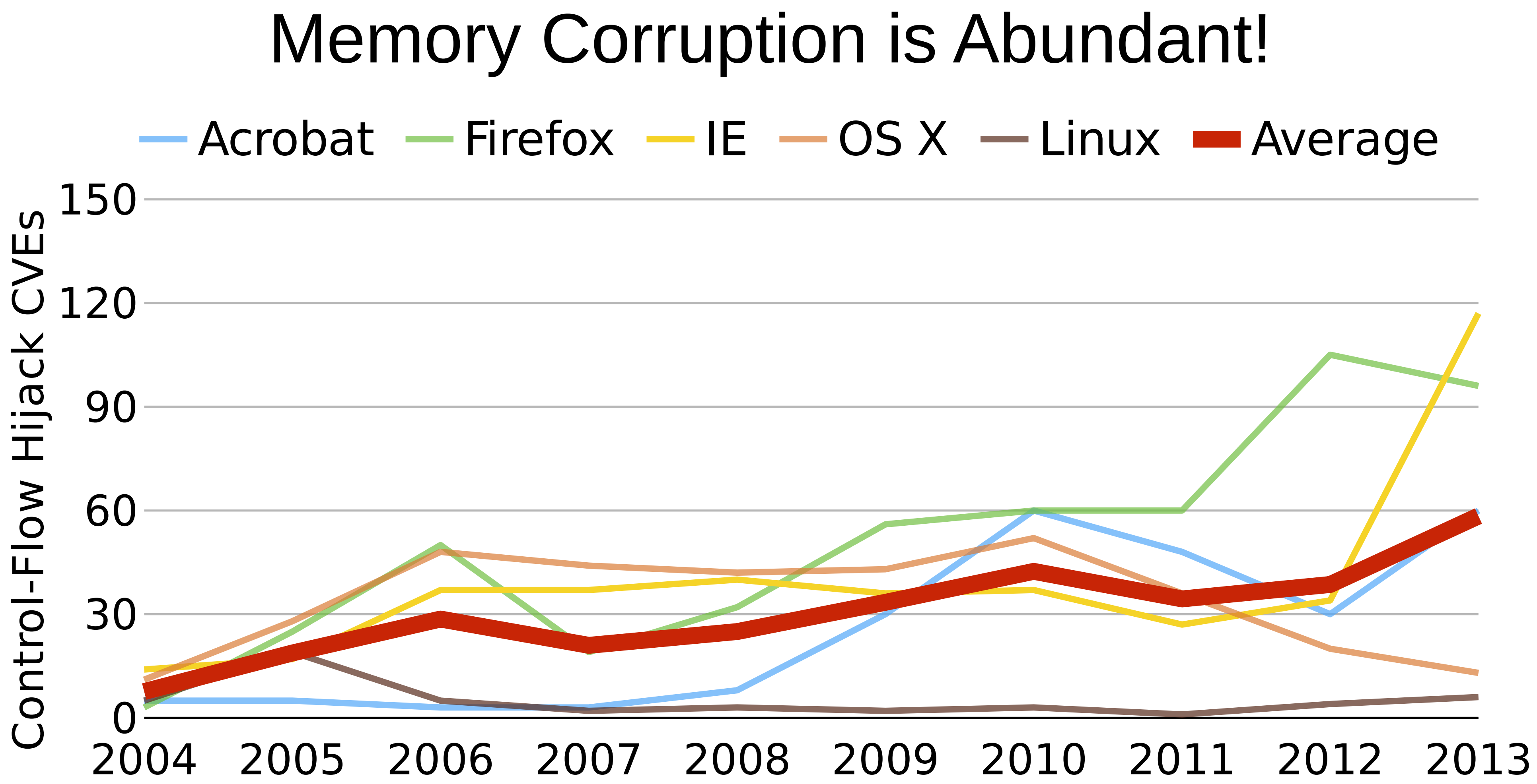
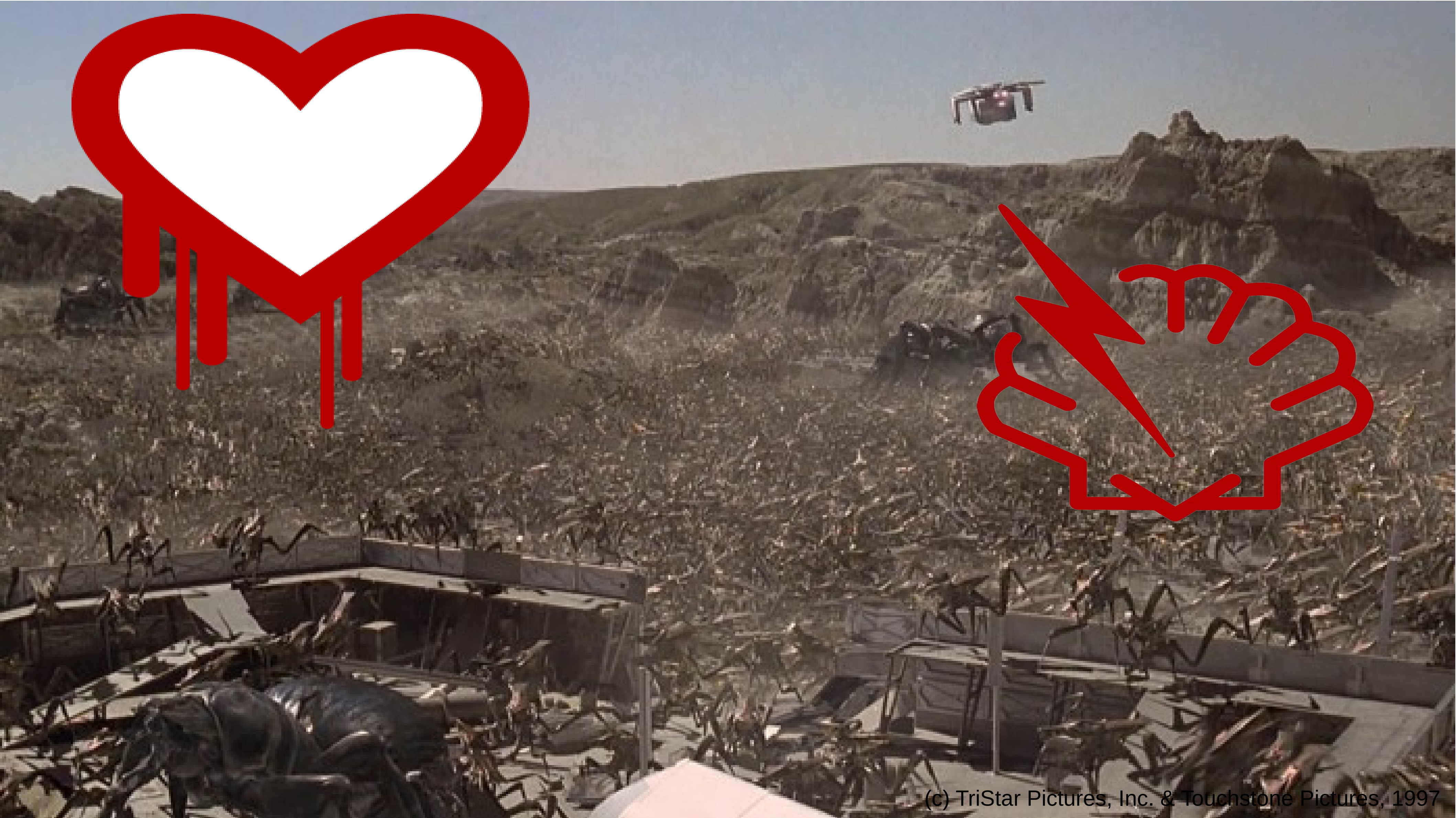
Code-Pointer Integrity

Volodmyr Kuzentsov, László Szekeres,
Mathias Payer, George Candea,
R. Sekar, and Dawn Song

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Stony Brook University
PURDUE UNIVERSITY
THE UNIVERSITY OF CALIFORNIA BERKELEY


Memory Corruption is Abundant!
Acrobat
Firefox
IE
OS X
Linux
Average
Control-Flow Hijack CVEs
150
120
90
60
30
0
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013

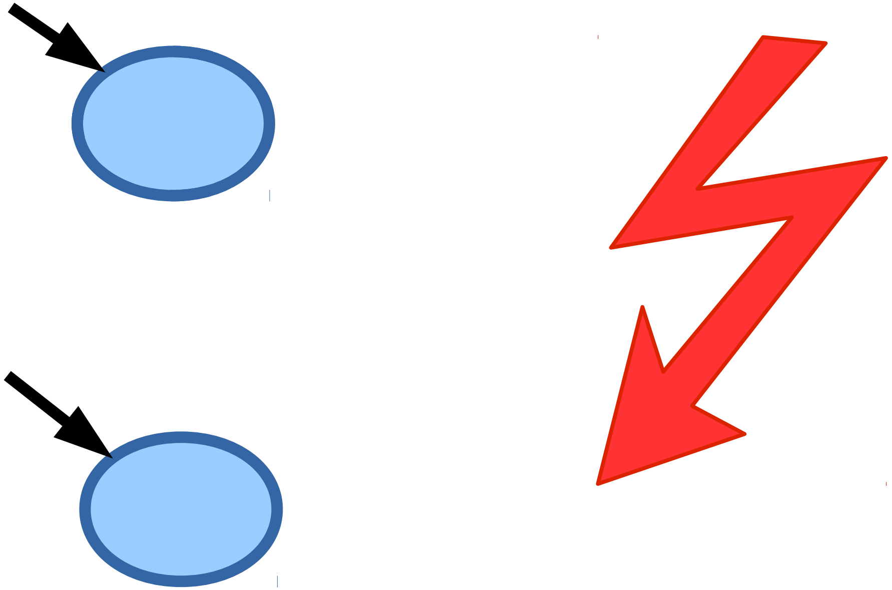

(c) TriStar Pictures, Inc. & Touchstone Pictures, 1997

Memory safety: invalid dereference

- Violation iff
  - Pointer is read
  - Pointer is written
  - Pointer is freed
- No violation
  - Otherwise

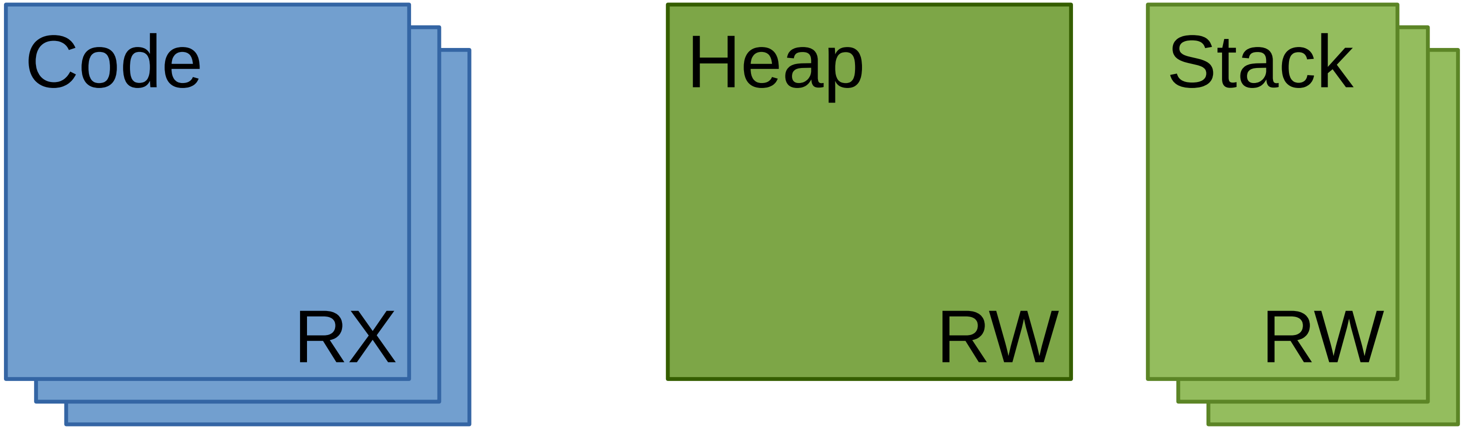
Dangling pointer: (temporal)

Out-of-bounds pointer: (spatial)



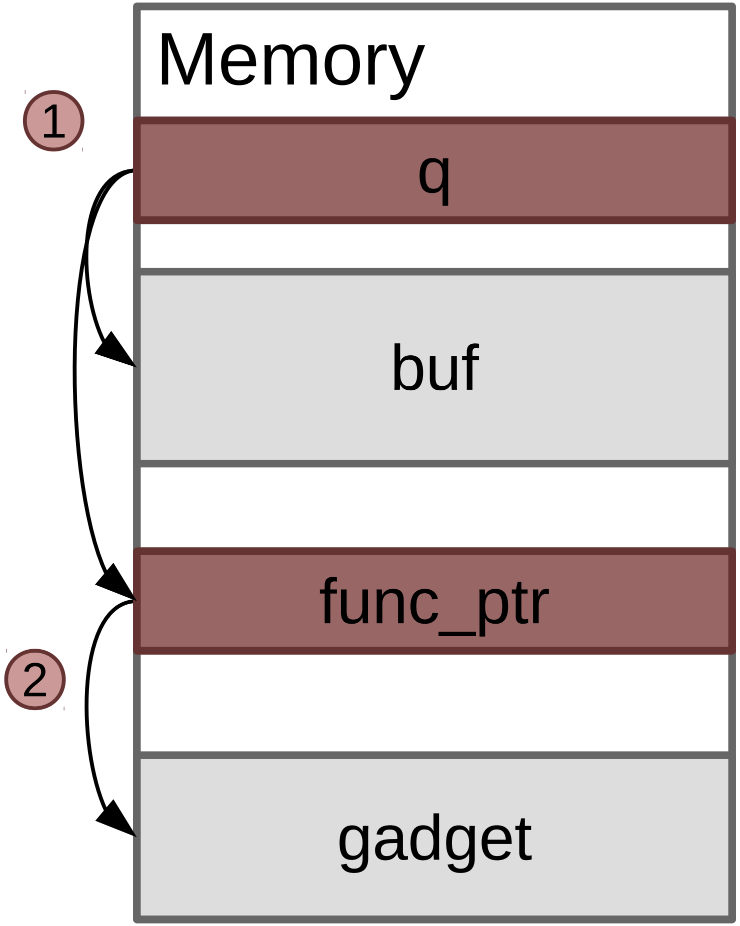
Threat Model

- Attacker can read/write data, read code
- Attacker cannot
  - Modify program code
  - Influence program loading

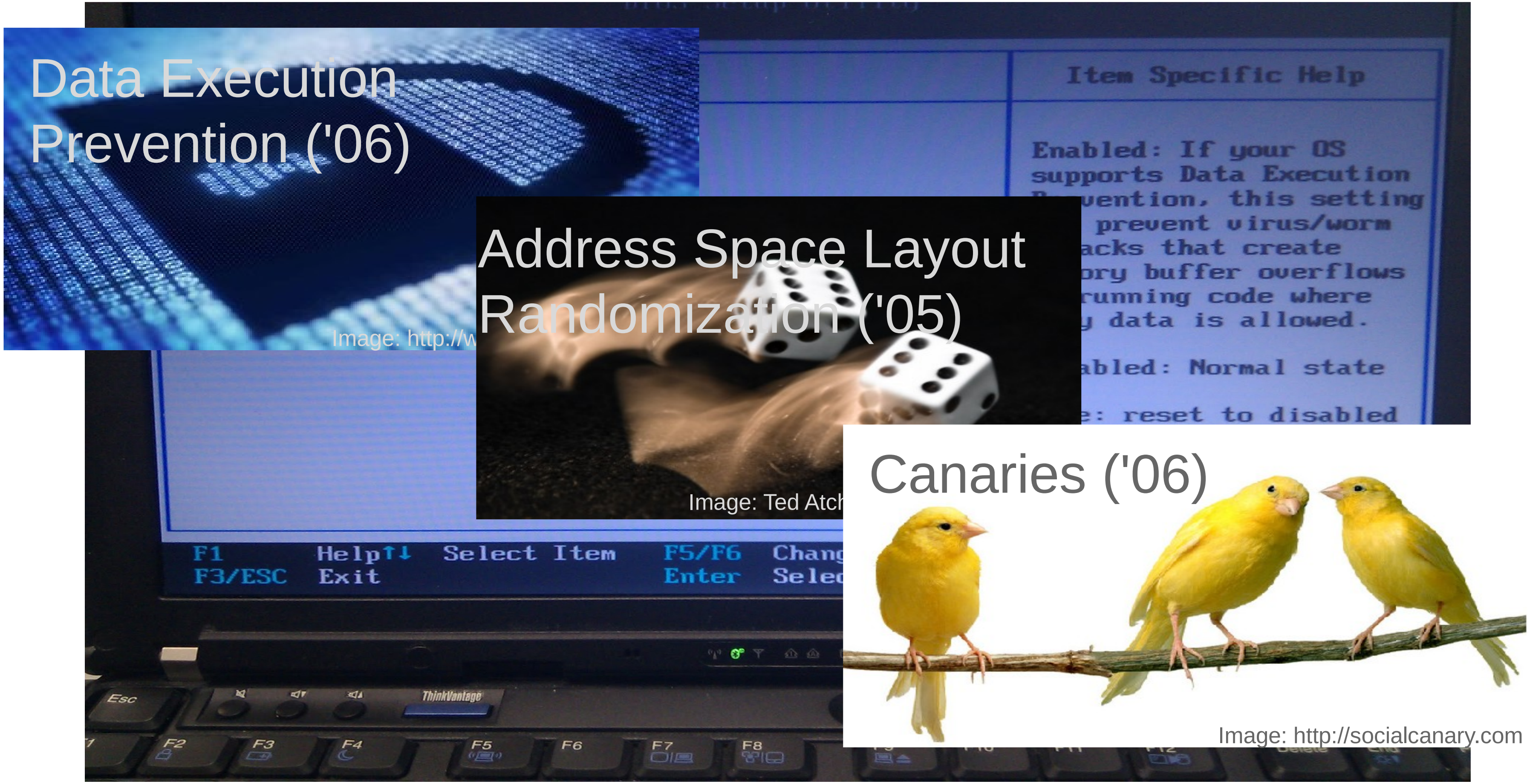


Control-Flow Hijack Attack

```
    void *(func_ptr)();
1   int *q = buf + input;
    …
    func_ptr = &foo;
    …
2   *q = input2;
    …
3   (*func_ptr)();
```



What about existing defenses?



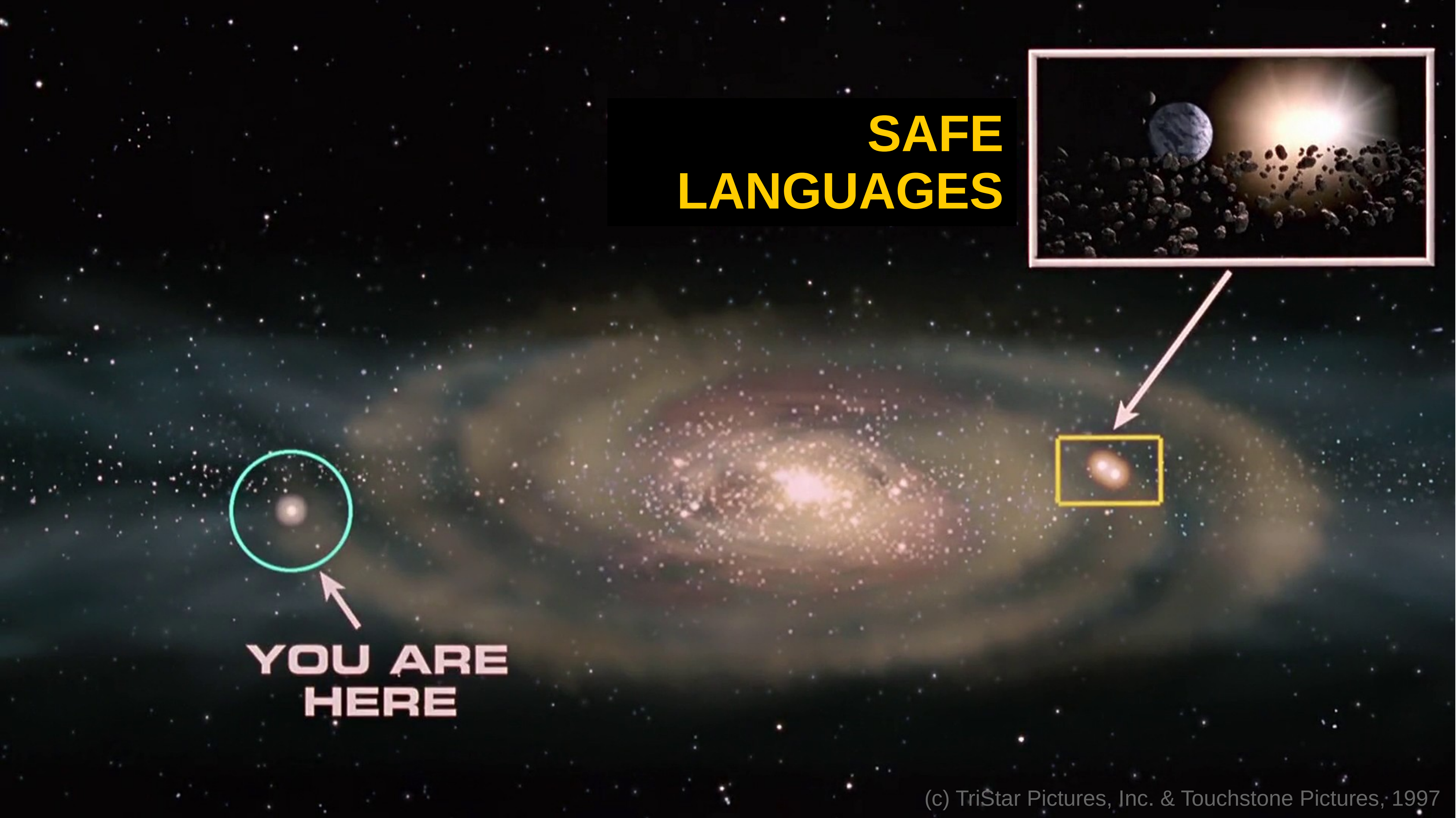

(c) TriStar Pictures, Inc. & Touchstone Pictures, 1997
WARGAMES
MISSILE WARNING
SUB-LAUNCH DETECTION
... in memory: an evolution of attacks
Mathias Payer <mathias.payer@nebelwelt.net>
UC Berkeley
Images (c) MGM, WarGames, 1983
Presented at 30c3
http://youtu.be/CQbXevkR4us
WOULD YOU LIKE TO KNOW MORE?

Memory Safety to the Rescue

python™ Java™ C# Swift

Python code	~3 kloc
Python runtime	~500 kloc
libc	~2,500 kloc
Linux kernel	~15,803 kloc

Unsafe!


SAFE
LANGUAGES
YOU ARE
HERE

Retrofit Memory Safety

C/C++	Overhead
SoftBound+CETS	116%
CCured	56%
AddressSanitizer	73%

Memory Safety

```
char *buf = malloc(10);
b
…
c
c
i  (q   q_lo || q     q_up)
  abort();
*q = input2;
…
(*func_ptr)();
```

1. Assign meta-data

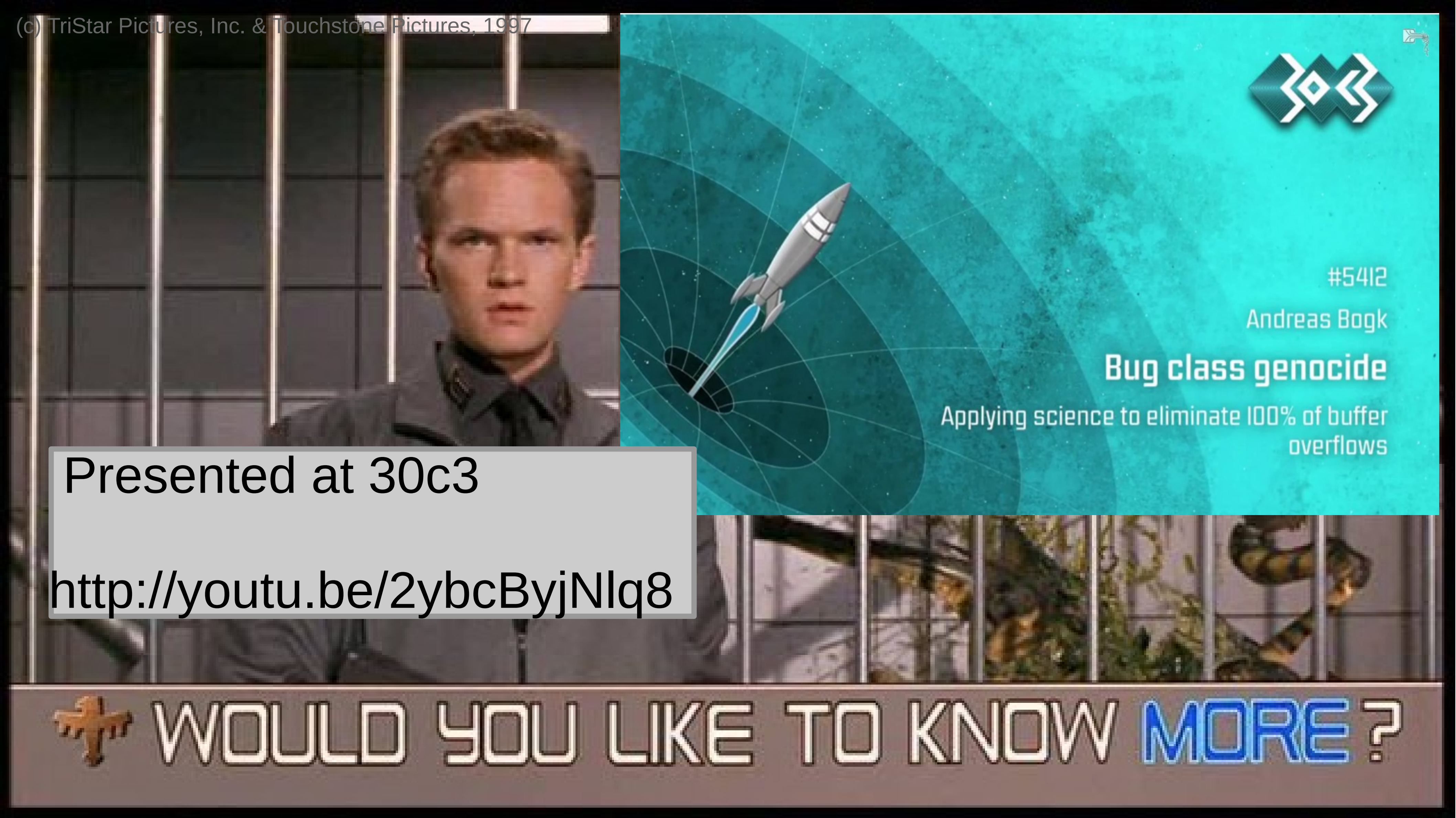
ata

3. Check meta-data

116% performance overhead

(Nagarakatte et al., PLDI'09 and ISMM'10)


Safety
vs.
Flexibility and
Performance
(c) TriStar Pictures, Inc. & Touchstone Pictures, 1997

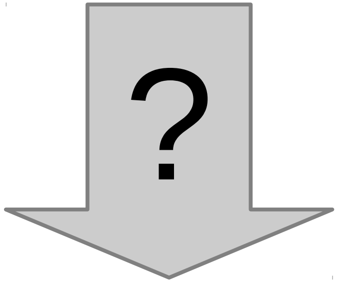

#5412
Andreas Bogk
Bug class genocide
Applying science to eliminate 100% of buffer overflows
Presented at 30c3
http://youtu.be/2ybcByjNlq8
WOULD YOU LIKE TO KNOW MORE?

New Approach: Protect Select Data

Instead of
protecting everything a little
protect a little completely

Strong protection for a select subset of data
Attacker may modify any unprotected data

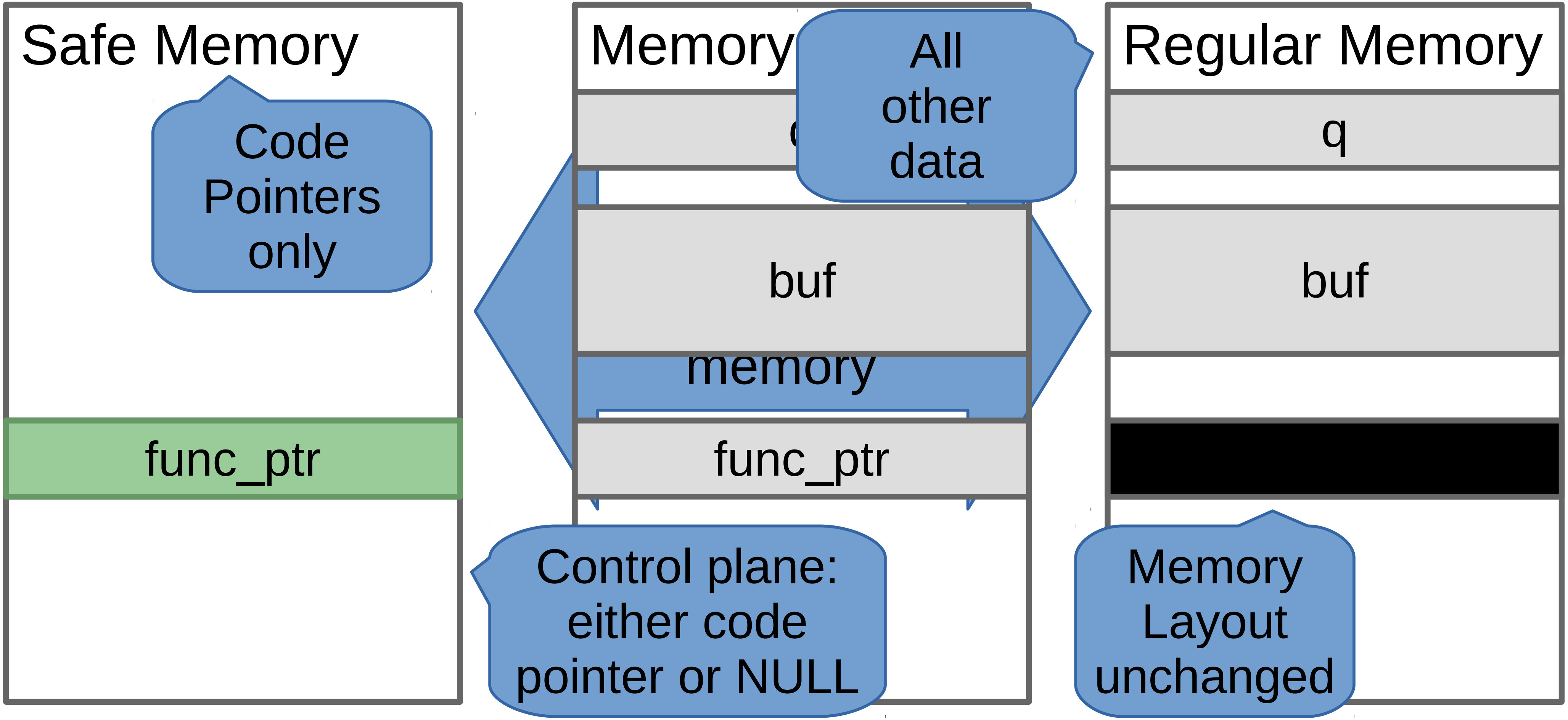
Memory Safety
(116% performance overhead)



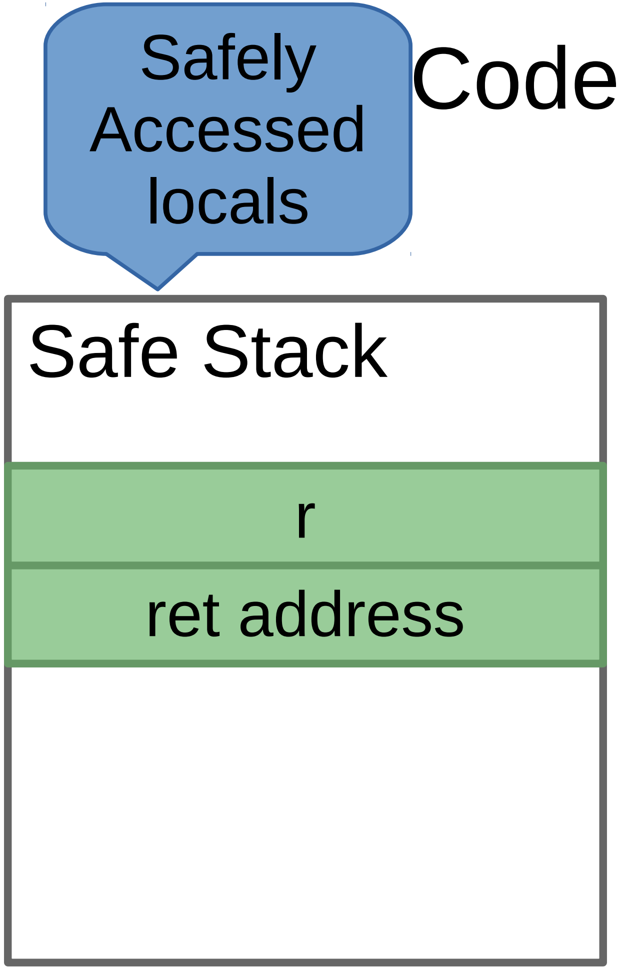
Protect only
Code Pointers

Control-Flow Hijack Protection
(1.9% or 8.4% performance overhead)

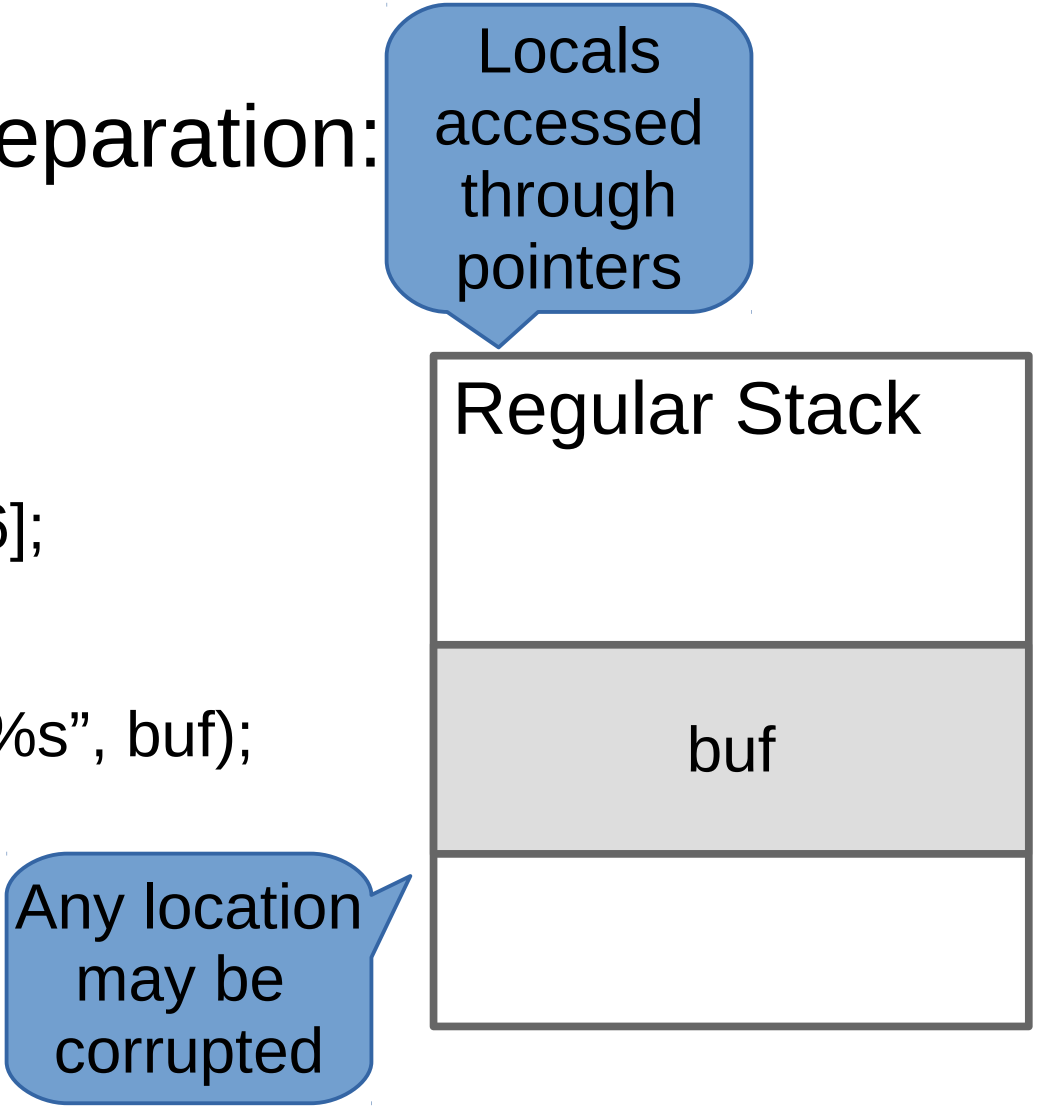
Code-Pointer Separation: Heap



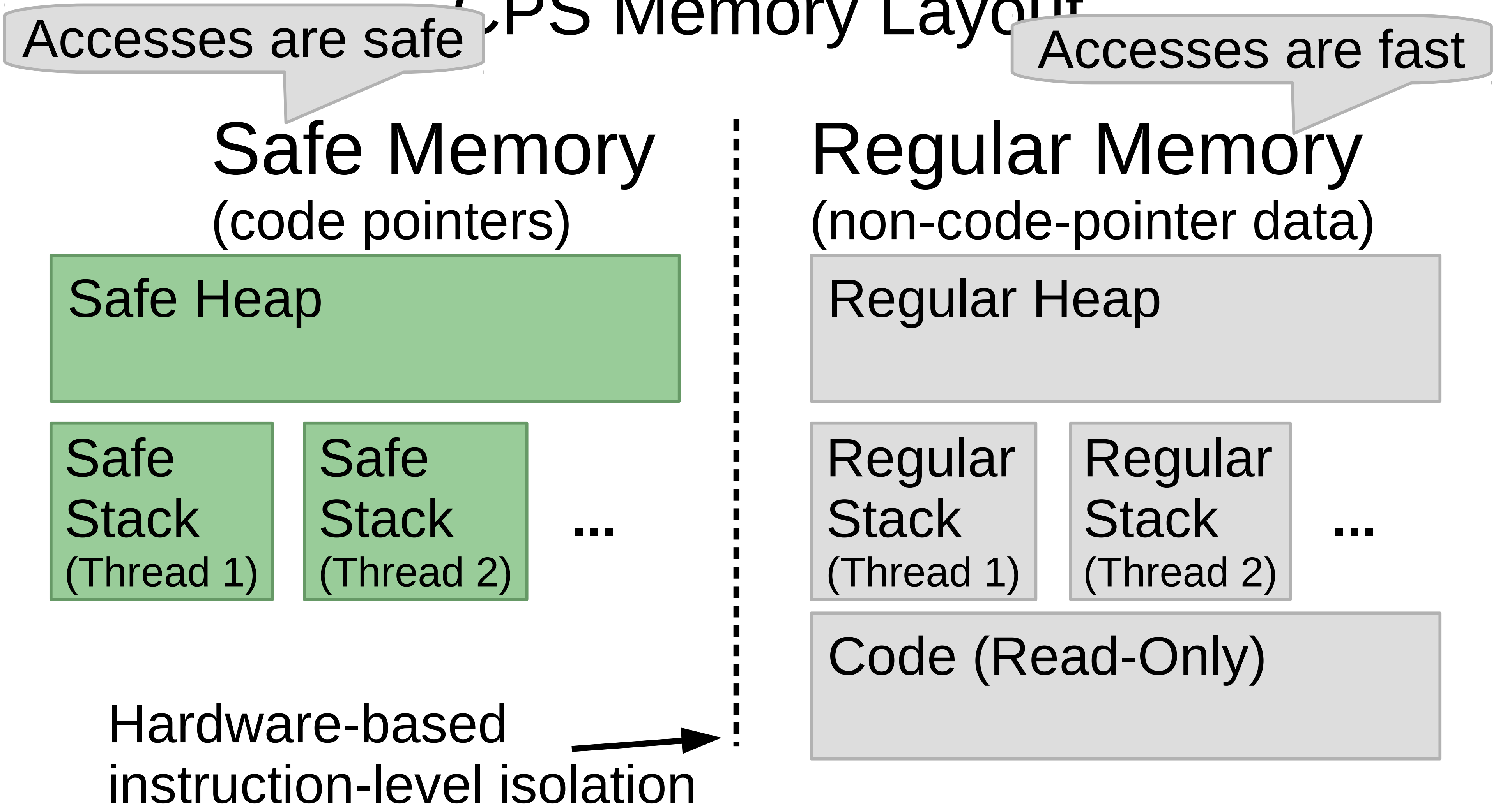
Code-Pointer Separation:




```
int foo() {
 char buf[16];
 int r;
 r = scanf("%s", buf);
 return r;
}
```

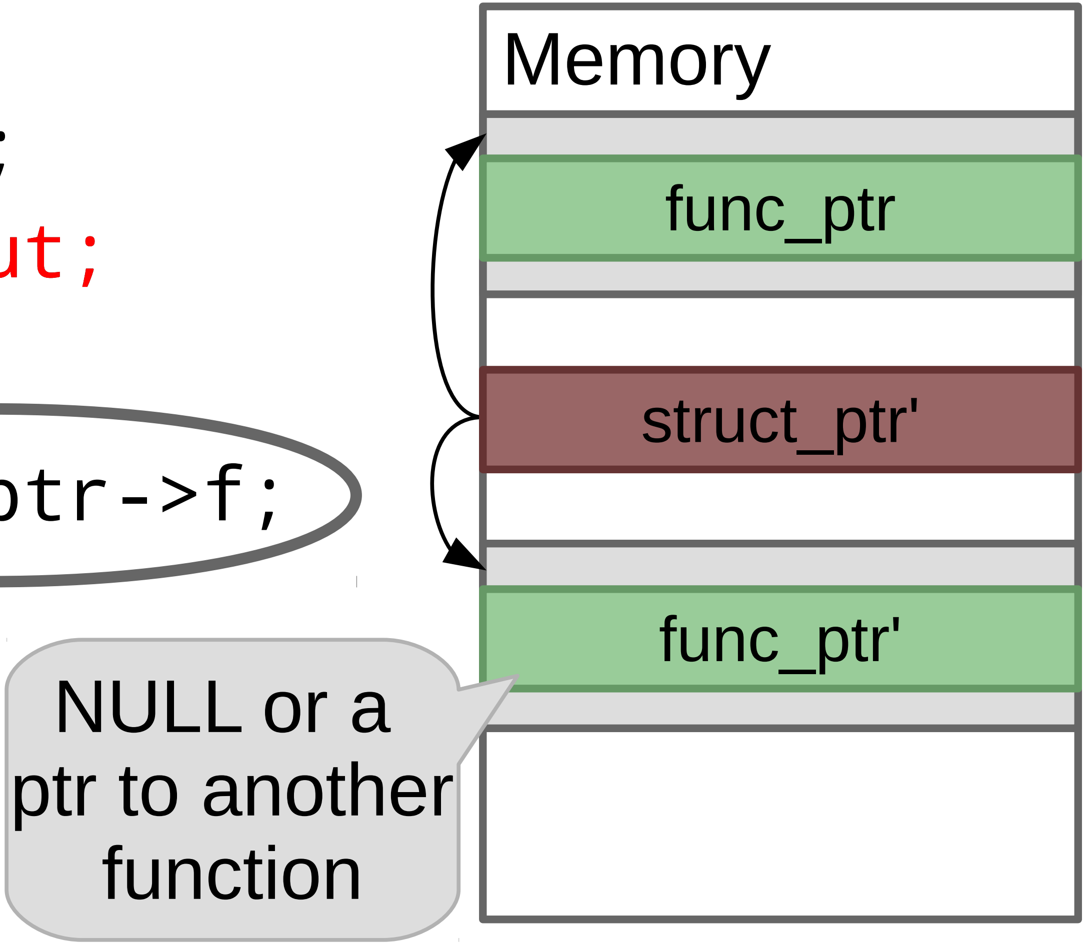


CPS Memory Layout



Attacking Code-Pointer Separation

```
void *(func_ptr)();
int *q = buf + input;
*q = input2;
…
func_ptr = struct_ptr->f;
…
(*func_ptr)();
```



Code-Pointer Separation

- Identify Code-Pointer accesses using static type-based analysis
- Separate using instruction-level isolation (e.g., segmentation)

- CPS security guarantees
  - An attacker cannot forge new code pointers
  - Code-Pointer is either immediate or assigned from code pointer
  - An attacker can only replace existing functions through indirection: e.g., foo->bar->func() vs. foo->baz->func2()

Code-Pointer Integrity (CPI)

Sensitive Pointers = code pointers and
pointers used to access sensitive pointers

- CPI identifies all sensitive pointers using an over-approximate type-based static analysis:

```
is_sensitive(v) = is_sensitive_type(type of v)
```

- Over-approximation only affects performance

  On SPEC2006 <= 6.5% accesses are sensitive

Attacking Code-Pointer Integrity

```
void *(func_ptr)();
int *q = buf + input;
q_lo = buf_lo; q_up = buf_up;
if (q < q_lo || q >= q_up)
 abort();
*q = input2;
func_ptr = struct_ptr->f;
…
(*func_ptr)();
```

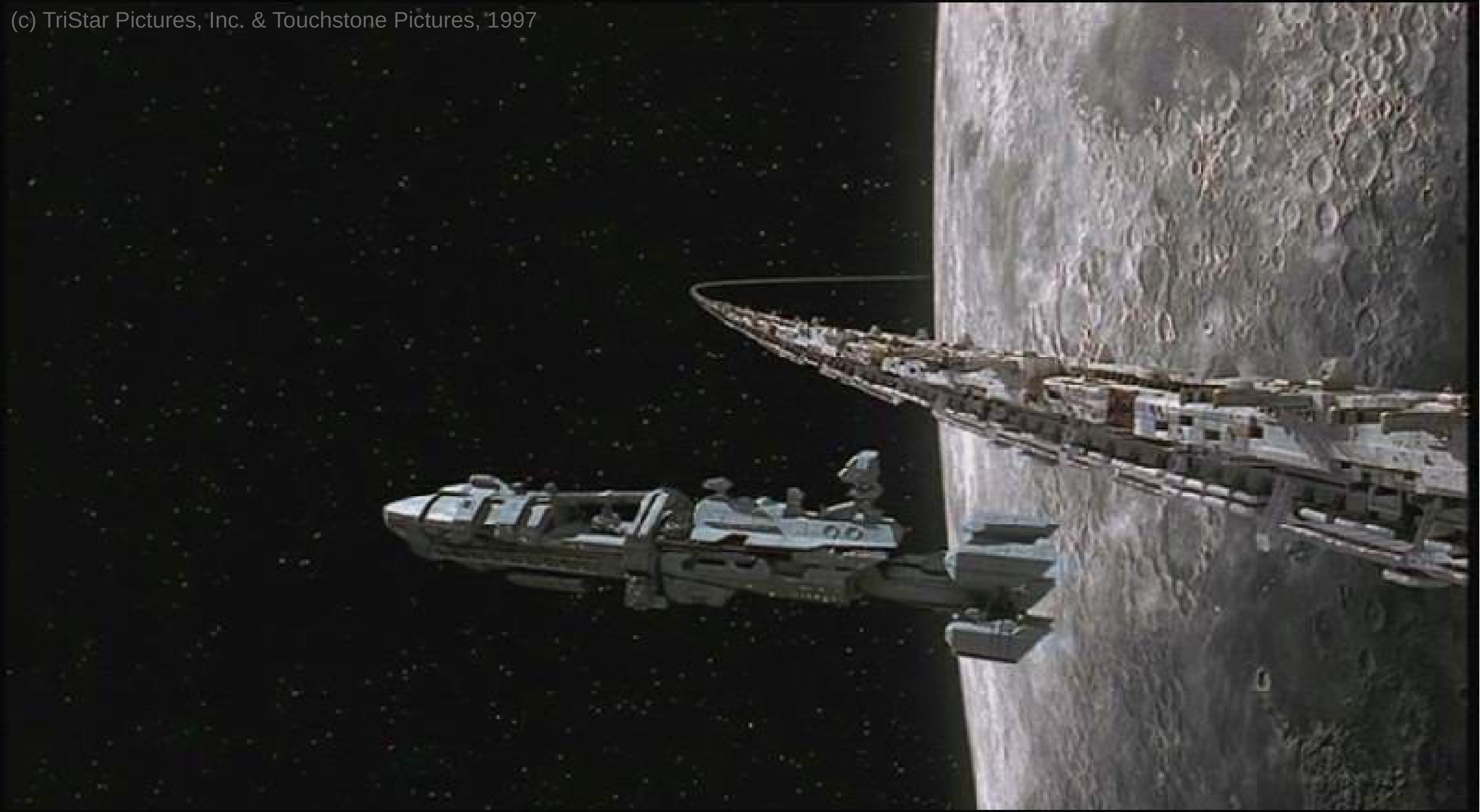
Exception when dereferenced

Code-Pointer Integrity vs. Separation

- Separate sensitive pointers from regular data
  - Type-based static analysis
  - Sensitive pointers = code pointers **+ pointers to sensitive pointers**
- Accessing sensitive pointers is **<u>safe</u>**
  - Separation **+ runtime (bounds) checks**
- Accessing regular data is **<u>fast</u>**
  - Instruction-level safe region isolation

Security Guarantees

- Code-Pointer Integrity: formally guaranteed protection
  - 8.4% to 10.5% overhead (~6.5% of memory accesses)
- Code-Pointer Separation: strong protection in practice
  - 0.5% to 1.9% overhead (~2.5% of memory accesses)
- Safe Stack: full ROP protection
  - Negligible overhead



Implementation

- LLVM-based prototype
  - Front end (clang): collect type information
  - Back-end (llvm): CPI/CPS/SafeStack instrumentation pass
  - Runtime support: safe heap and stack management
  - Supported ISA's: x64 and x86 (partial)
  - Supported systems: Mac OSX, FreeBSD, Linux

Current status

- Great support for CPI on Mac OSX and FreeBSD on x64
- Upstreaming in progress
  - Safe Stack coming to LLVM soon
  - Fork it on GitHub now: https://github.com/cpi-llvm
- Code-review of CPS/CPI in process
  - Play with the prototype: http://levee.epfl.ch/levee-early-preview-0.2.tgz
  - Will release more packages soon
- Some changes to super complex build systems needed
  - Adapt Makefiles for FreeBSD

Is It Practical?

- Recompiled entire FreeBSD userpsace
- … and more than 100 packages


DO
YOUR
PART!

Conclusion

- CPI/CPS offers strong control-flow hijack protection
  - Key insight: memory safety for code pointers only
- Working prototype
  - Supports unmodified C/C++, low overhead in practice
  - Upstreaming patches in progress, SafeStack available soon!
  - Homepage: http://levee.epfl.ch
  - GitHub: https://github.com/cpi-llvm


(c) TriStar Pictures, Inc. & Touchstone Pictures, 1997
?
http://levee.epfl.ch
http://nebelwelt.net/publications/14OSDI/