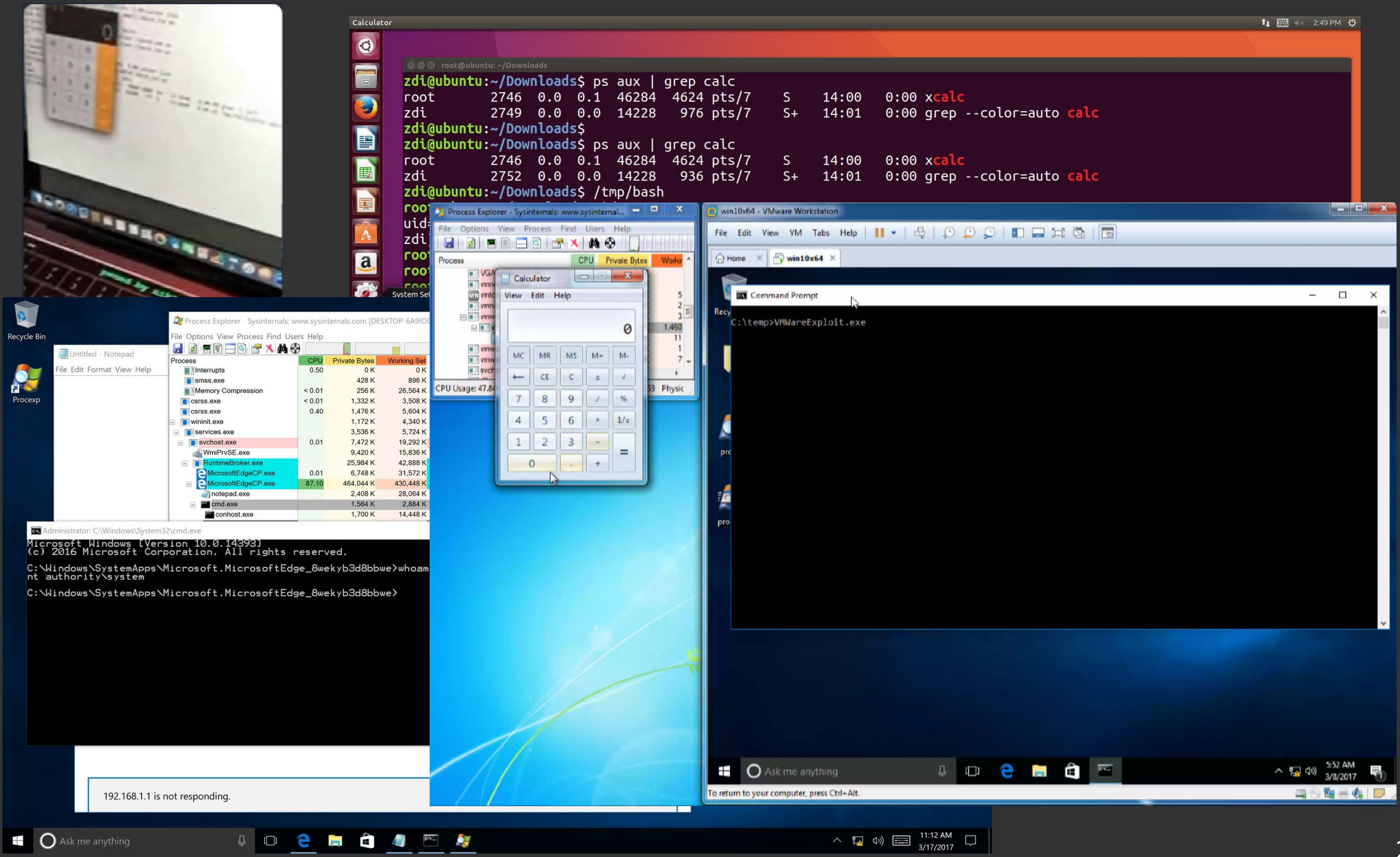
hexhive

Control-Flow Hijacking:
Are We Making Progress?

Mathias Payer, Purdue University
http://hexhive.github.io

Bugs are everywhere?



```
zdi@ubuntu:~/Downloads$ ps aux | grep calc
root        2746  0.0  0.1  46284  4624 pts/7    S    14:00   0:00 xcalc
zdi         2749  0.0  0.0  14228   976 pts/7    S+   14:01   0:00 grep --color=auto calc
zdi@ubuntu:~/Downloads$
zdi@ubuntu:~/Downloads$ ps aux | grep calc
root        2746  0.0  0.1  46284  4624 pts/7    S    14:00   0:00 xcalc
zdi         2752  0.0  0.0  14228   936 pts/7    S+   14:01   0:00 grep --color=auto calc
zdi@ubuntu:~/Downloads$ /tmp/bash
```

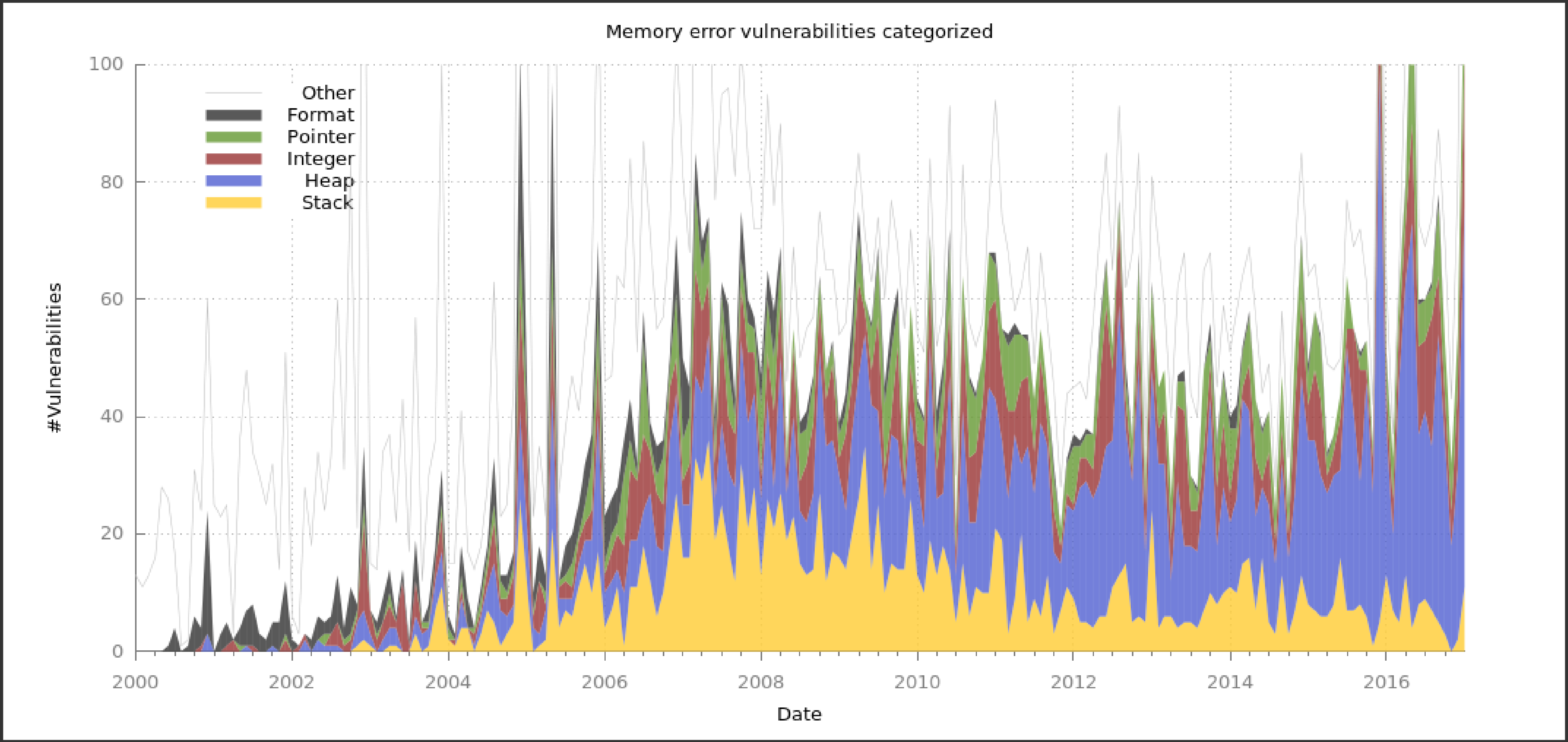
Process	CPU	Private Bytes	Working Set
Interrupts	0.50	0 K	0 K
smss.exe		428 K	896 K
Memory Compression	< 0.01	256 K	26,564 K
csrss.exe	< 0.01	1,332 K	3,508 K
csrss.exe	0.40	1,476 K	5,604 K
wininit.exe		1,172 K	4,340 K
services.exe		3,536 K	5,724 K
svchost.exe	0.01	7,472 K	19,292 K
WmiPrvSE.exe		9,420 K	15,836 K
RuntimeBroker.exe		25,984 K	42,888 K
MicrosoftEdgeCP.exe	0.01	6,748 K	31,572 K
MicrosoftEdgeCP.exe	87.10	464,044 K	430,448 K
notepad.exe		2,408 K	28,064 K
cmd.exe		1,564 K	2,884 K
conhost.exe		1,700 K	14,448 K

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe>whoam
nt authority\system

C:\Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe>
```

192.168.1.1 is not responding.

```
C:\temp>VMWareExploit.exe
```

Trends in Memory Errors*



* Victor van der Veen, https://www.vvdveen.com/memory-errors/, updated Feb. 2017

Software is unsafe and insecure*

- Low-level languages (C/C++) trade type safety and memory safety for performance
  - Our systems are implemented in C/C++
  - Too many bugs to find and fix manually

Google Chrome:	**76 MLoC**
glibc:	**2 MLoC**
Linux kernel:	**14 MLoC**

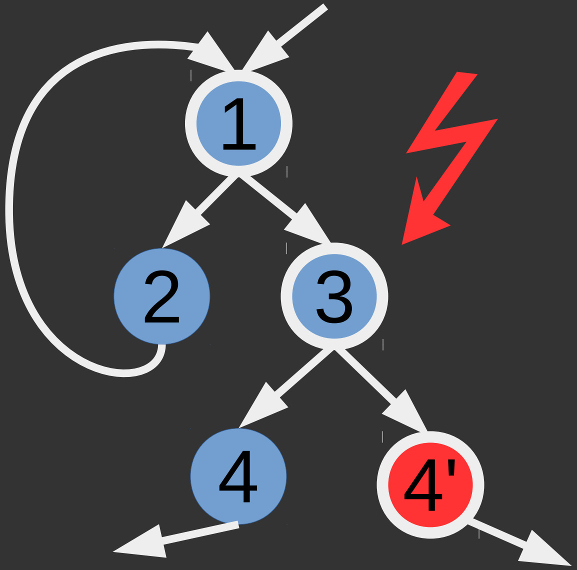
---

* SoK: Eternal War in Memory. Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. In IEEE S&P'13


Vulnerability
Just Ahead

Control-Flow Hijack Attack

Control-flow hijack attack



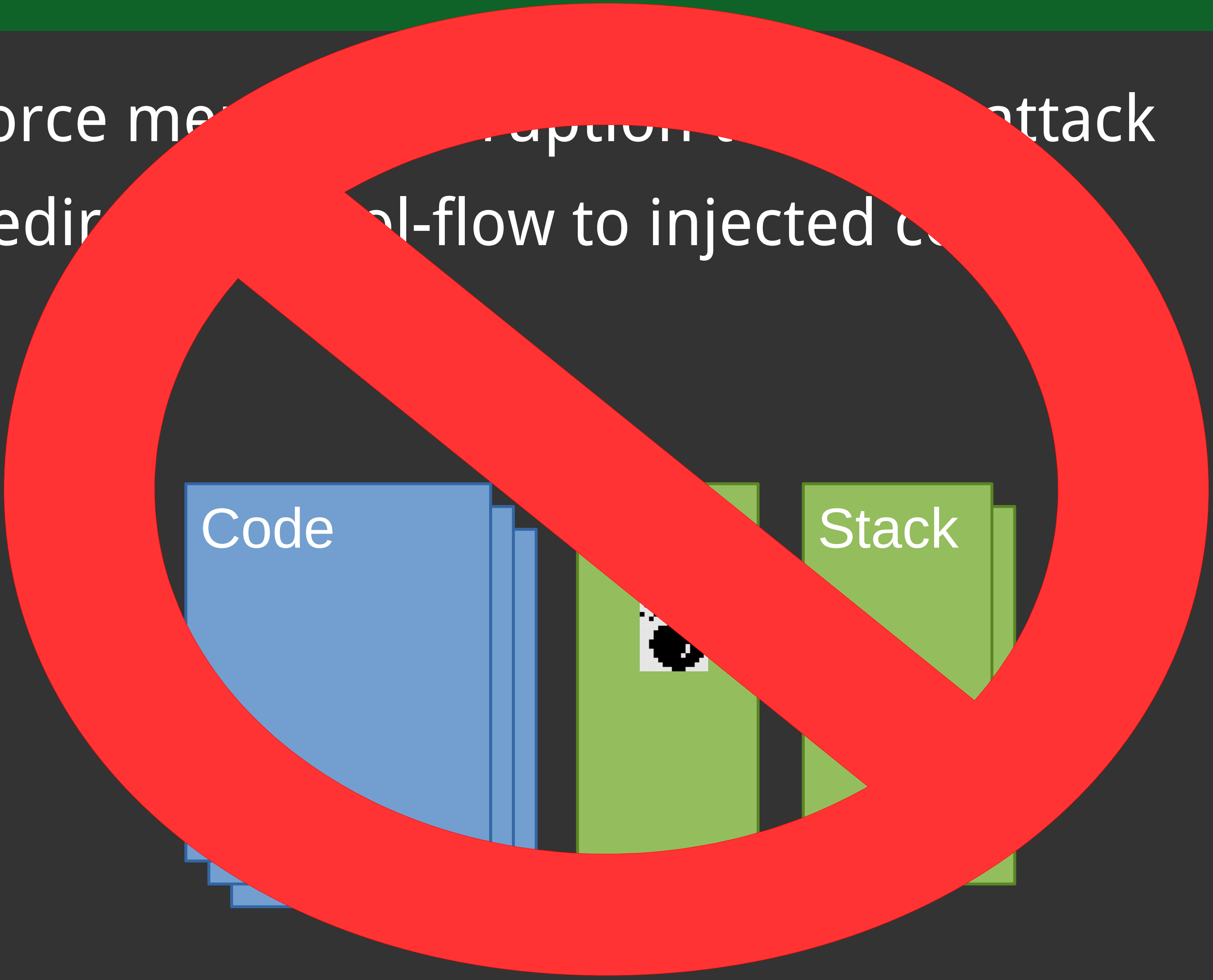
- Attacker modifies ***code pointer***
  - Information leak: target address
  - Memory safety violation: write
- Control-flow leaves ***valid graph***
  - Inject/modify code
  - Reuse existing code

Attack scenario: code injection

- Force memory corruption to set up attack
- Redirect control-flow to injected code

Attack scenario: code injection

- Force me uption attack
- Redir l-flow to injected c



Attack scenario: code reuse

- Find addresses of gadgets
- Force memory corruption to set up attack
- Redirect control-flow to gadget chain

Control-Flow Integrity

Control-Flow Integrity (CFI)*

- Restrict a program’s dynamic control-flow to the static control-flow graph
  - Requires static analysis
  - Dynamic enforcement mechanism

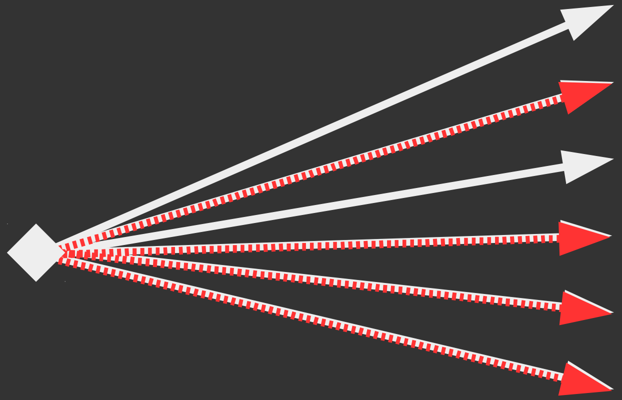
- Forward edge: virtual calls, function pointers
- Backward edge: function returns

---

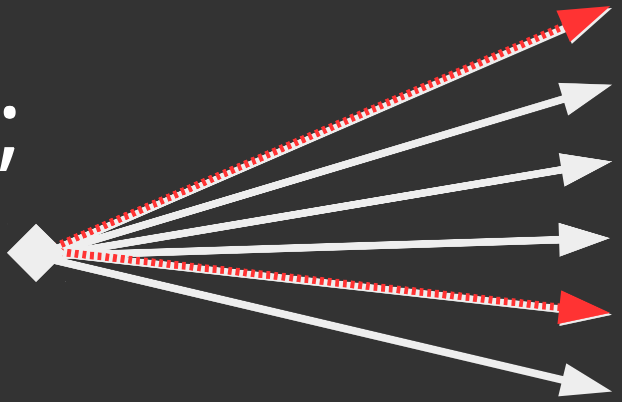
* ***Control-Flow Integrity.*** Martin Abadi, Mihai Budiu, Ulfar Erlingsson, Jay Ligatti. CCS ‘05
Control-Flow Integrity: Protection, Security, and Performance. Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, Mathias Payer. ACM CSUR ‘18, preprint: https://nebelwelt.net/publications/files/18CSUR.pdf

Control-Flow Integrity (CFI)

```
CHECK(fn);
(*fn)(x);
```

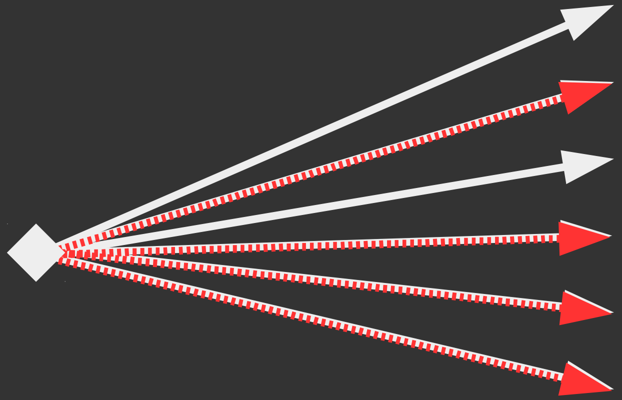


```
CHECK_RET();
return 7
```



Control-Flow Integrity (CFI)

```
CHECK(fn);
(*fn)(x);
```



Attacker may corrupt memory, code ptrs. verified when used

CFI: Limitations

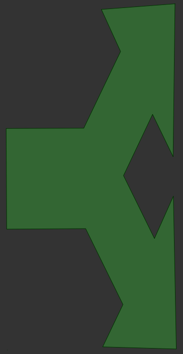
- CFI provides incremental security
- Strength of CFI mechanism depends on the power of the analysis
  - Coarse-grained: all functions are allowed
  - Fine-grained: better than coarse-grained

Qualitative Analysis

- Classes of analysis precision for forward edges
  1) Ad hoc algorithms, labeling
  2) Class-hierarchy analysis
  3) Flow- or context-sensitive analysis
  4) Devirtualize through dynamic analysis

CFI: Strength of Analysis

A *obj = new A();
obj->foo(int b, int c);



🚫 0xf000b400

🚫 int bar1(int b, int c, int d);

🚫 void bar2(int b, int c);

🚫 int bar3(int b, int c);

🚫 int B::foo(int b, int c);

class A :: B {... };
🚫 int B::bar5(int b, int c);

int A::foo(int b, int c);

Qualitative Analysis

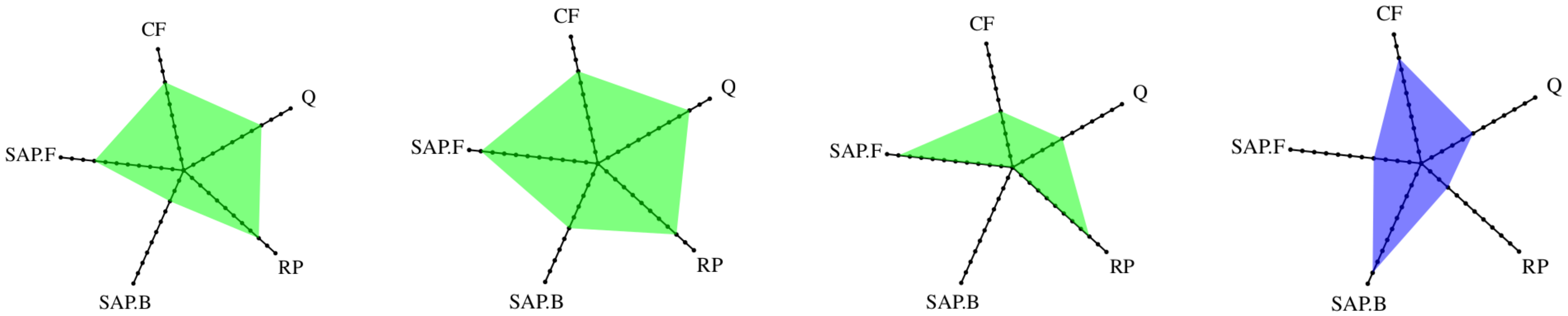
- Backward edge best protected orthogonally
  - Shadow stacks
  - Safe stacks

- In practice:
  - Backward edge excluded (“assume shadow stack”)
  - Reuse forward-edge analysis

Existing Quantitative Metrics

- Average Indirect-target Reduction (AIR)
  - AIR is defined as: $\frac{1}{n}\sum_{j=1}^{n}(1-\frac{|T_j|}{S})$
- Allowing any libc function has 99.9% AIR
  - 2,102 exported functions
  - 1,864,888 bytes of text
- All mechanisms have AIR of 99.9+%

Qualitative Analysis



MCFI

πCFI

LLVM-CFI-3.7 (2015)

Lockdown

Control-flows (CF), quantitative security (Q), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B)

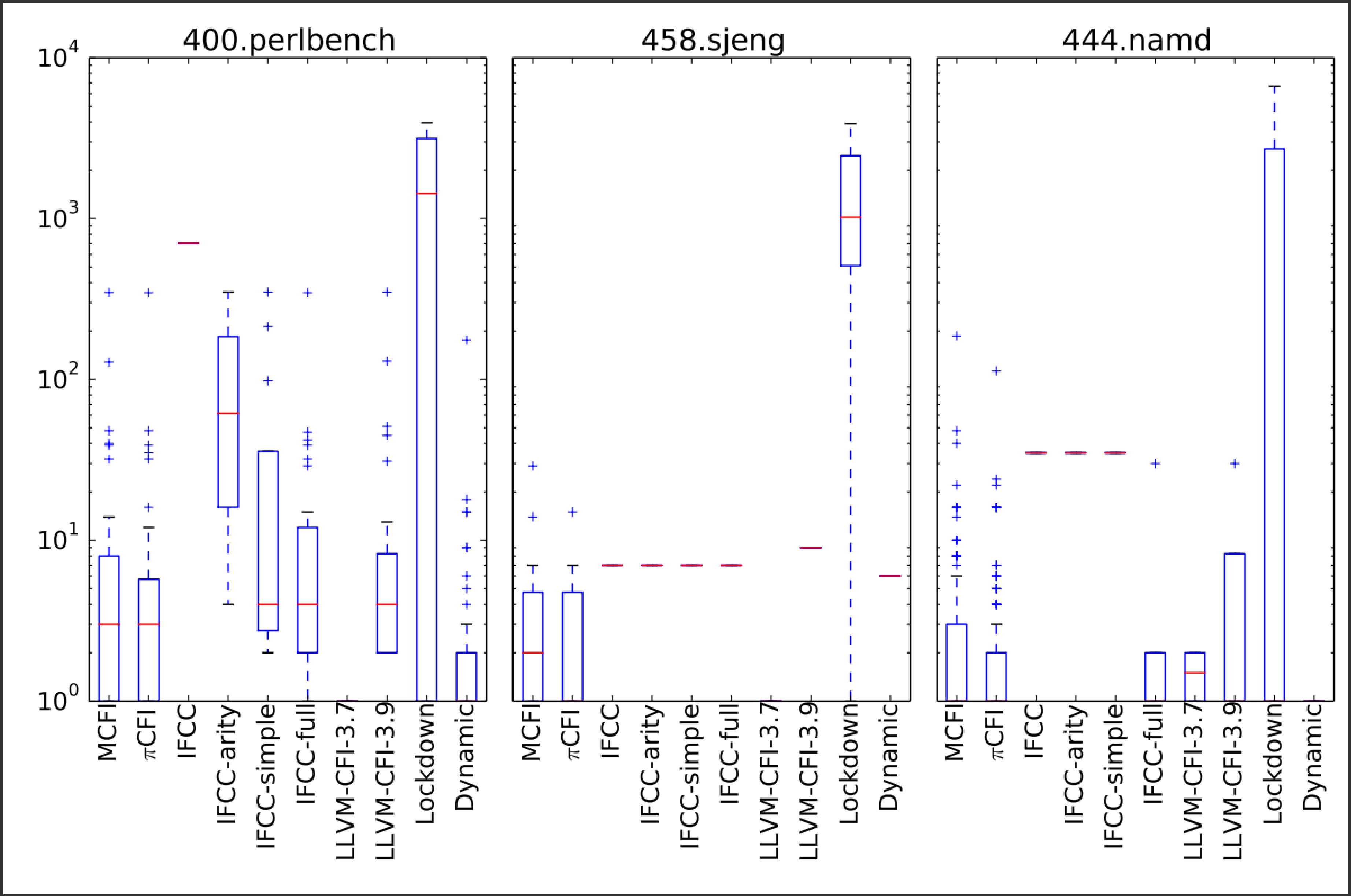
Quantitative Security Analysis

- Compare 5 open-source mechanisms
  - on the same machine
  - with the same benchmarks

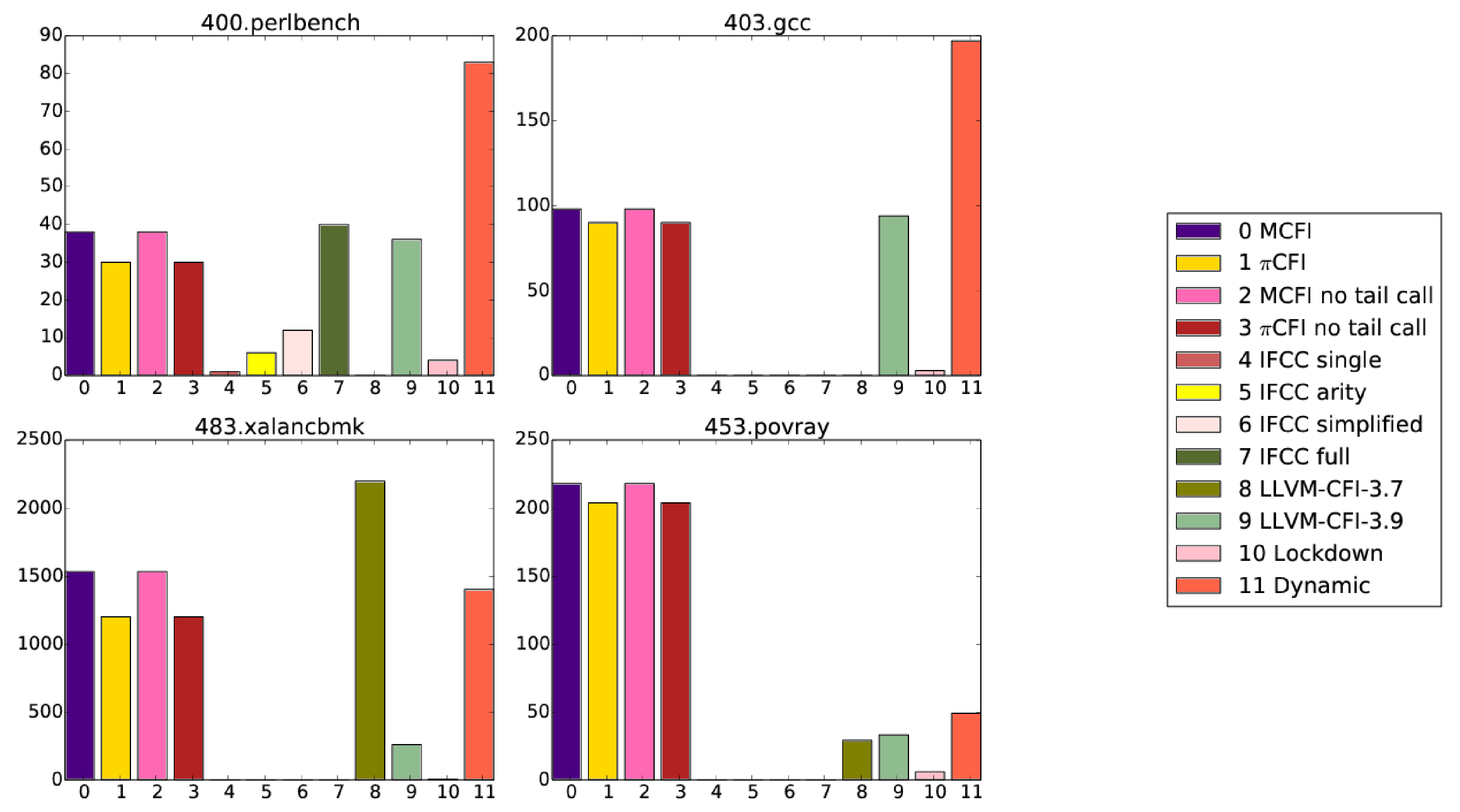
- Define quantitative metrics
  - Number of equivalence classes
  - Size of largest class

- Dynamic profiling bounds required targets

Size of Equivalence Classes

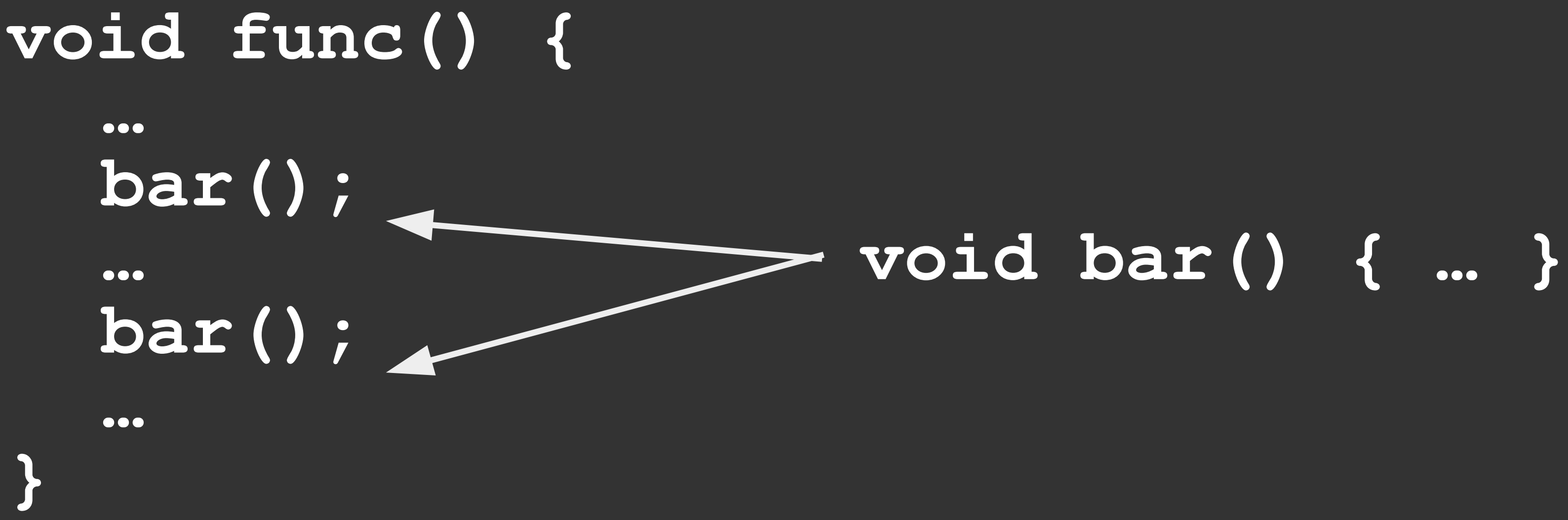


Number of Equivalence Classes



Necessity of shadow stack*

- Defenses without stack integrity are broken
  - Loop through two calls to the same function
  - Choose any caller as return location



```
void func() {
   …
   bar();
   …
   bar();
   …
}
```

```
void bar() { … }
```

---

* Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. In Usenix SEC'15

Necessity of shadow stack*

- Defenses without stack integrity are broken
  - Loop through two calls to the same function
  - Choose any caller as return location

- Shadow stack enforces stack integrity
  - Attacker restricted to arbitrary targets ***on*** the stack
  - Each target can only be called once, in sequence

* Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. In Usenix SEC'15

Code-Pointer Integrity, SafeStack*

- Memory safety stops control-flow hijack attacks
  - … but memory safety has high overhead (250%)
- Enforce memory safety for code pointers only
  - Partition code pointers, check all loads and stores
- Efficient prototype: 5.82% for C/C++ on SPEC
  - (Partially) upstreamed to LLVM
  - HardenedBSD relies on SafeStack (11/28/16)

* Code-Pointer Integrity. Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, Dawn Song, R. Sekar. In Usenix OSDI'14

CFI Summary

- CFI is available and makes attacks harder
    - Microsoft Visual Studio, GCC, LLVM
    - Deployed in Microsoft Edge, Google Chrome

- Potential limitations
    - Large equivalence classes are attack targets
    - Backward edge protection is crucial

- Ongoing work: precision and metrics
    - CFI should use context and flow sensitivity

Type Safety

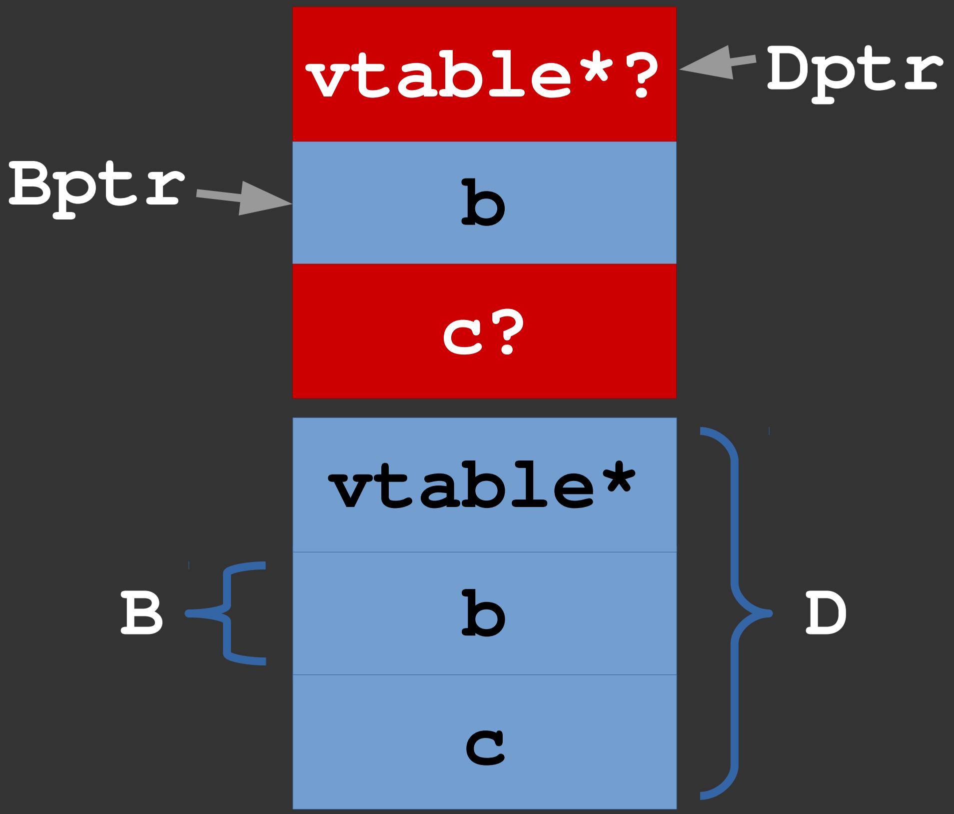
Type Confusion

- Type confusion arises through illegal downcasts
  - Converting a base class pointer to a derived class

- This problem is common in large software
  - Adobe Flash (CVE-2015-3077)
  - Microsoft Internet Explorer (CVE-2015-6184)
  - PHP (CVE-2016-3185)
  - Google Chrome (CVE-2013-0912)

* TypeSanitizer: Practical Type Confusion Detection. Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Herbert Bos, Cristiano Giuffrida, Erik van der Kouwe. In CCS'16

Type Confusion



Dptr

vtable*?

Bptr

b

c?

vtable*

B

b

D

c

```
class B {
   int b;
};
class D: B {
   int c;
   virtual void d() {}
};
…
B *Bptr = new B;
D *Dptr = static_cast<D*>B;
Dptr->c = 0x43; // Type confusion!
Dptr->d();      // Type confusion!
```

Type Confusion Detection

- static_cast<type> uses compile-time check
  - Fast but no runtime guarantees
- dynamic_cast<type> uses runtime check
  - High overhead
  - Only possible for polymorphic classes

- TypeSan approach:
  - Make type verification explicit, check ***all*** cast
  - Challenge: low overhead

Conclusion

Are we making progress?

2007

2017

Conclusion

- We are making progress!
  - Attacks are much harder
  - Require teams, not just single players

- CFI makes attacks harder
  - Some attack surface remains
  - Stack integrity, X⊕W, ASLR complementary

- Ongoing work:
  - Precision, type safety, memory safety

hexhive

Thank you!

Questions?

Mathias Payer, Purdue University
http://hexhive.github.io

Qualitative Analysis

- Classes of analysis precision for forward edges
  1) Ad hoc algorithms, labeling
  2) Class-hierarchy analysis
  3) Rapid-type analysis
  4) Flow or context sensitive analysis
  5) Context and flow sensitive analysis
  6) Devirtualize through dynamic analysis

Flow and Context Sensitivity

Flow insensitive:

Flow sensitive:

```
Object *o;
o = new A();
…
o = new B();
```

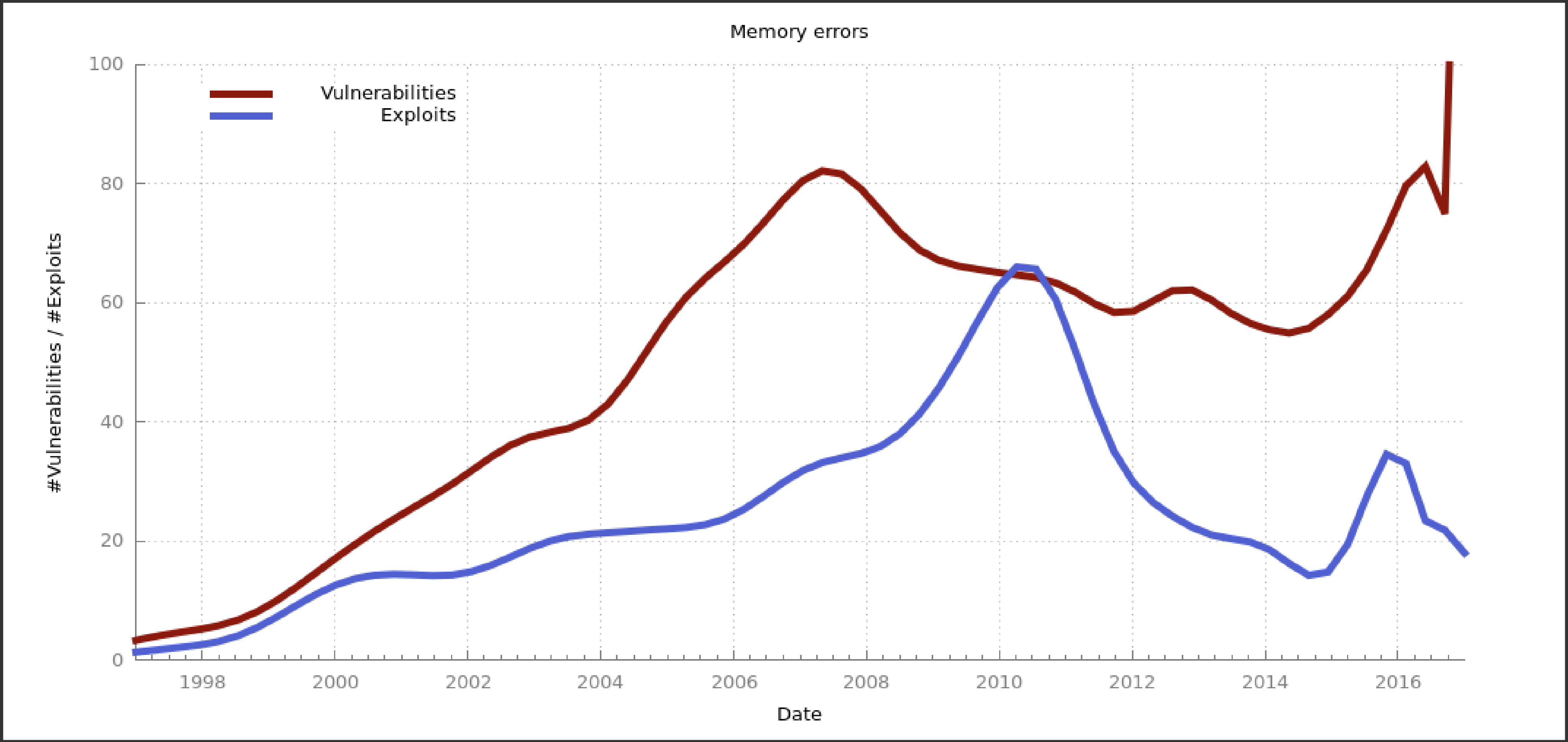
Flow and Context Sensitivity

```
Object *id(Object *o) { return o; }
Object *x, *y, *a, *b;
```

Context insensitive: *Context sensitive:*

```
x = new A();
y = new B();
a = id(x);
b = id(y);
```

Trends in Memory Errors*



* Victor van der Veen, https://www.vvdveen.com/memory-errors/, updated Feb. 2017