CSE 127: Computer Security

Heap Corruption and CFI

Deian Stefan

Slides adopted from Stefan Savage

Now you know about return-to-libc and ROP, what is a reasonable defense strategy?

W^X + ASL not enough?

- Let’s suppose we have shadow/safe stack
  - Are we safe now? A: yes, **B: no**

Attacks via the heap!

Memory management in C/C++

- How do we allocate/deallocate memory?
  - malloc/new
  - free/delete/delete[]
- How do we access memory?
  - through pointers!

Why is this error prone?

- We may:
  - Write/read memory we shouldn't have access to
  - Forget to free memory
  - Free already freed objects
  - Use pointers that point to freed object
- What if the attacker can cause the program to

Heap corruption

- Can bypass security checks (data-only attacks)
  - ➤ .
- Can overwrite function pointers
  - ➤ d
  - ➤

Heap corruption

- Can bypass security checks (data-only attacks)
  - E.g., isAuthenticated, buffer_size, isAdmin, etc.
- Can overwrite function pointers
  -
  - d
  -

Heap corruption

- Can bypass security checks (data-only attacks)
  - E.g., isAuthenticated, buffer_size, isAdmin, etc.
- Can overwrite function pointers
  - Direct transfer of control when function is called
  - What's an example? vTables

Heap corruption

- Use after free
  - free(p); p->foo();
  - free(p); q = malloc(n); memcpy(p, buf, k);
- Double free
  - free(p); free(p); q = malloc(n); r = malloc(n);
  - free(p); q = malloc(n); free(p);

vTables

- How do virtual function calls work in C++?
- How is it implemented?
  - E.g., what does bar compile to?
  - *(obj->vtable[0])(obj)

```
class Base {
  public: virtual void foo() {cout << “Hi\n";}
};

class Derived: public Base  {

 public: void foo()  {cout << "Bye\n";}
};

void bar(Base* obj)  { obj->foo(); }

int main(int argc, char* argv[])
{
          Base *b = new Base();
          Derived *d = new Derived();

          bar(b);
          bar(d);
 }
```

vTables

- Each object contains pointer to table
- vtable is array of function pointers
  - one entry per function
- Based on class + func compiler knows which offset to use

```
class Base {
  public: virtual void foo() {cout << “Hi\n";}
};

class Derived: public Base  {

 public: void foo()  {cout << "Bye\n";}
};

void bar(Base* obj)  { obj->foo(); }

int main(int argc, char* argv[])
{
          Base *b = new Base();
          Derived *d = new Derived();

          bar(b);
          bar(d);
 }
```

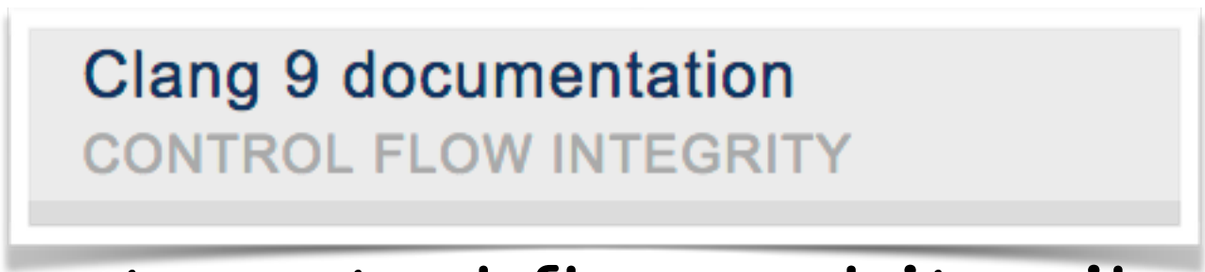
Control Flow Integrity

Clang 9 documentation
CONTROL FLOW INTEGRITY

- **Problem:** we can redirect control flow arbitrarily
- **Idea:** restrict control-flow to legitimate paths
- **Approach:** target

destinations

Control Flow Integrity



- **Problem:** we can redirect control flow arbitrarily
- **Idea:** restrict control-flow to legitimate paths
- **Approach:** [illegible]
  target

  destinations

Control Flow Integrity

Clang 9 documentation
CONTROL FLOW INTEGRITY

- **Problem:** we can redirect control flow arbitrarily
- **Idea:** restrict control-flow to legitimate paths
- **Approach:** Match jump, call, return sites to target

destinations

Direct control flow

- Q: do we need to protect direct control flow transfer (i.e., direct jumps/calls)?
  - A: yes, **B: no**
- Q: Why/why not?
  -

Direct control flow

- Q: do we need to protect direct control flow transfer (i.e., direct jumps/calls)?
  - A: yes, **B: no**
- Q: Why/why not?
  - Address is hard-coded in instruction

Indirect control flow transfer

- Jumping to (or calling function at) an address in register or memory
  - What's an example of this?
- Do we need to only worry about where we're jumping to?
  -

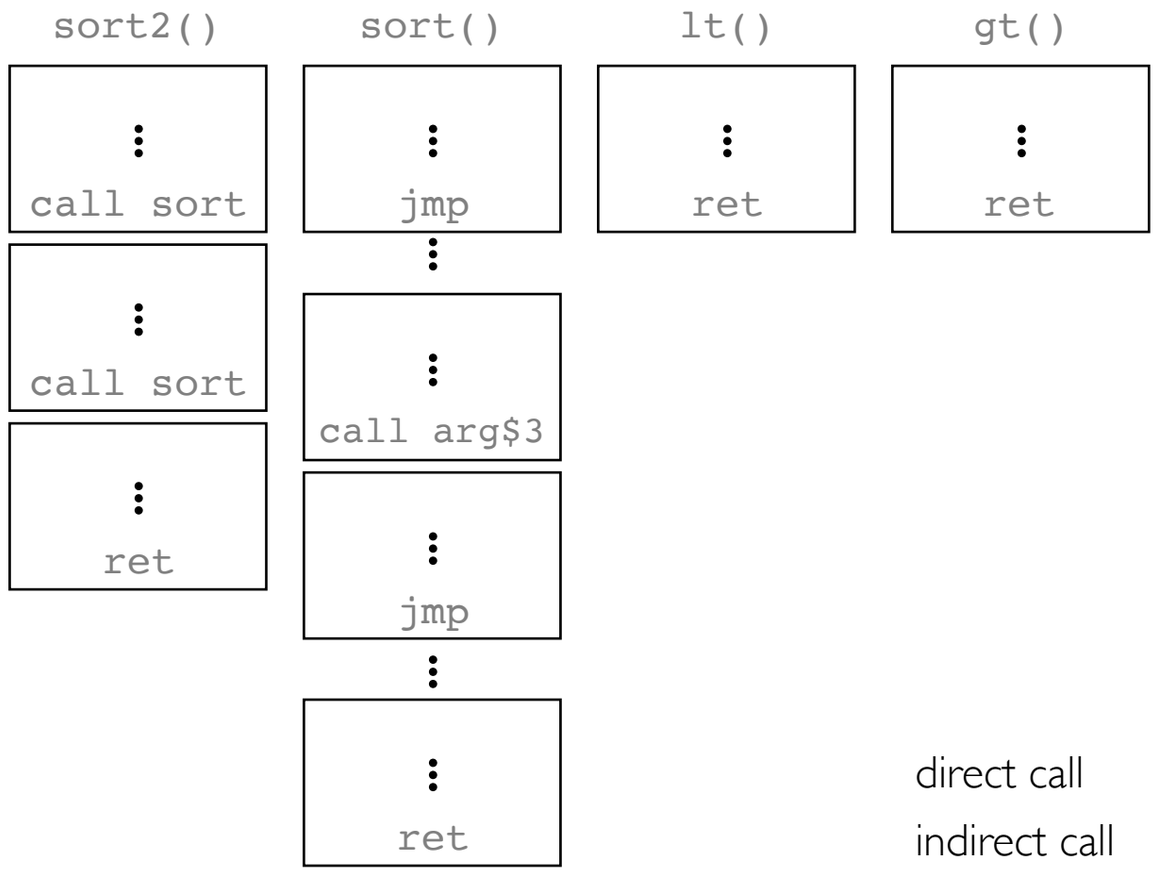
Indirect control flow transfer

- Jumping to (or calling function at) an address in register or memory
  - What's an example of this?
- Do we need to only worry about where we're jumping to?
  - A: yes, **B: no**

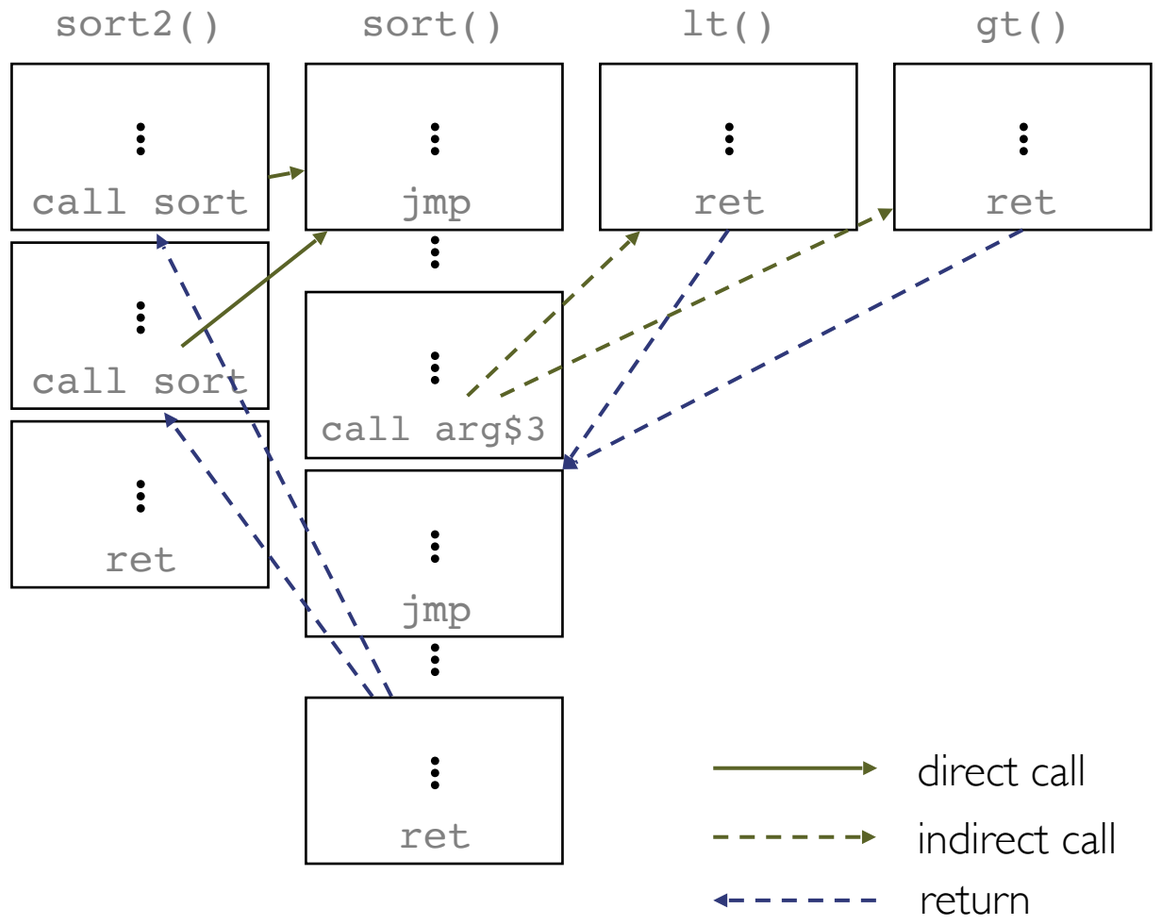
Examples of indirect calls?

- In C: qsort(...fptr...), interrupt handlers
- In C++: virtual functions
- In Wasm: call_indirect

```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y {

  return x < y;

}

bool gt(int x, int y) {

  return x > y;

}
```



```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y {

  return x < y;

}

bool gt(int x, int y) {

  return x > y;

}
```



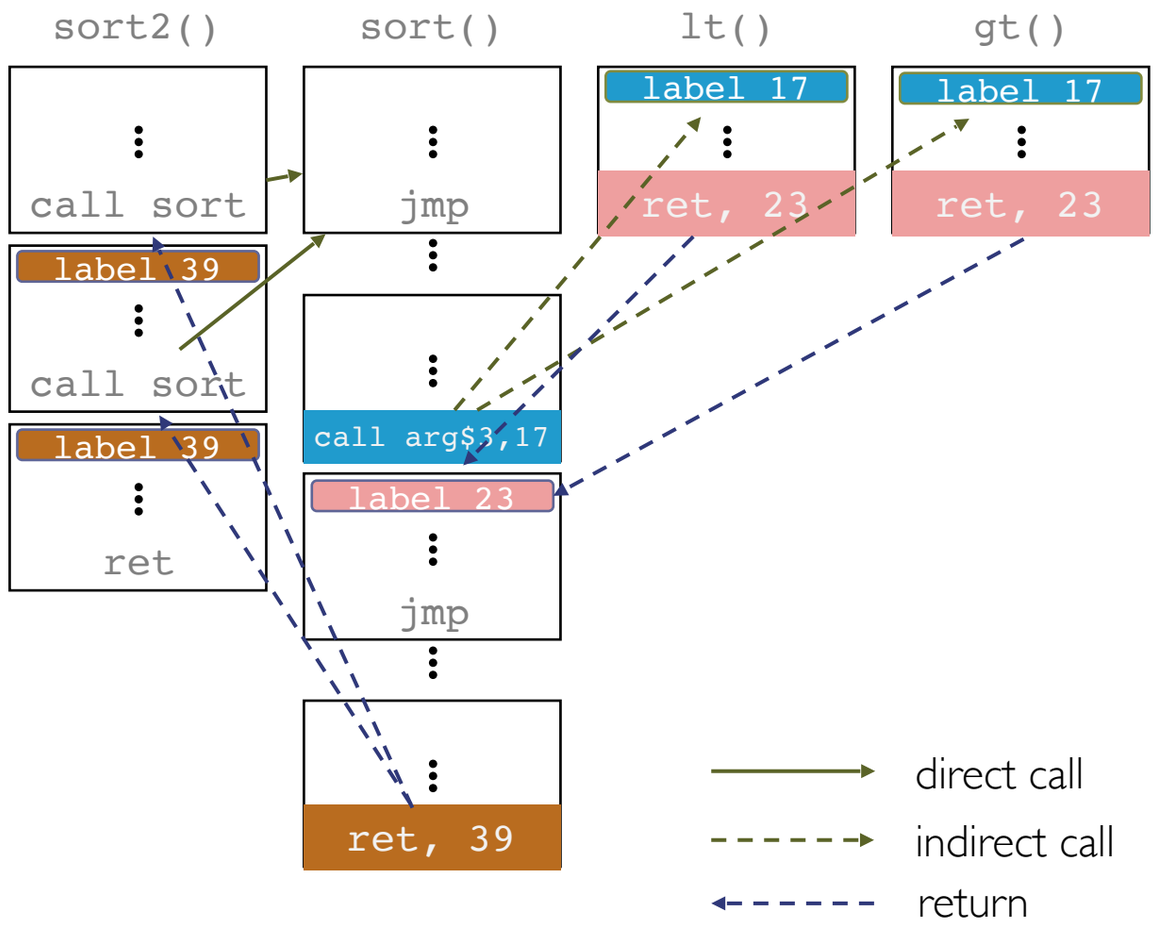
How are we going to match targets?

- Assign labels to all direct jumps and their targets
- After taking an indirect jump:
  - Validate that target label matches jump site
  - Recall stack canaries

Fine grained CFI (Abadi et al.)

- Statically compute control flow graph
- Dynamically ensure program never deviates
  - Assign label to each destination of indirect CF
  - Instrument indirect CT transfers:
    compare label @ dest w/ constant to ensure target is valid

```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y {

  return x < y;

}

bool gt(int x, int y) {

  return x > y;

}
```



Example of labels

Original code

Source			Destination		
Opcode bytes	Instructions		Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst

Example of labels

Original code

Source		Destination	
Opcode bytes	Instructions	Opcode bytes	Instructions
FF E1	jmp ecx ; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst

Instrumented code

```
B8 77 56 34 12     mov  eax, 12345677h    ; load ID-1          3E 0F 18 05    prefetchnta          ; label
40                 inc  eax               ; add 1 for ID       78 56 34 12       [12345678h]       ;    ID
39 41 04           cmp  [ecx+4], eax      ; compare w/dst      8B 44 24 04    mov  eax, [esp+4]    ; dst
75 13              jne  error_label       ; if != fail         ...
FF E1              jmp  ecx               ; jump to label
```

Example of labels

Original code

Source			Destination		
Opcode bytes	Instructions		Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst

Instrumented code

```
B8 77 56 34 12      mov  eax, 12345677h    ; load ID-1          3E 0F 18 05    prefetchnta            ; label
40                  inc  eax               ; add 1 for ID       78 56 34 12       [12345678h]         ;    ID
39 41 04            cmp  [ecx+4], eax      ; compare w/dst      8B 44 24 04    mov  eax, [esp+4]      ; dst
75 13               jne  error_label       ; if != fail         ...
FF E1               jmp  ecx               ; jump to label
```

Abuse an x86 assembly instruction to insert “12345678” tag into the binary

Example of labels

Original code

Source			Destination		
Opcode bytes	Instructions		Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst

Instrumented code

```
B8 77 56 34 12      mov  eax, 12345677h    ; load ID-1
40                  inc  eax               ; add 1 for ID
39 41 04            cmp  [ecx+4], eax      ; compare w/dst
75 13               jne  error_label       ; if != fail
FF E1               jmp  ecx               ; jump to label
```

```
3E 0F 18 05     prefetchnta                ; label
78 56 34 12        [12345678h]             ;    ID
8B 44 24 04     mov  eax, [esp+4]          ; dst
...
```

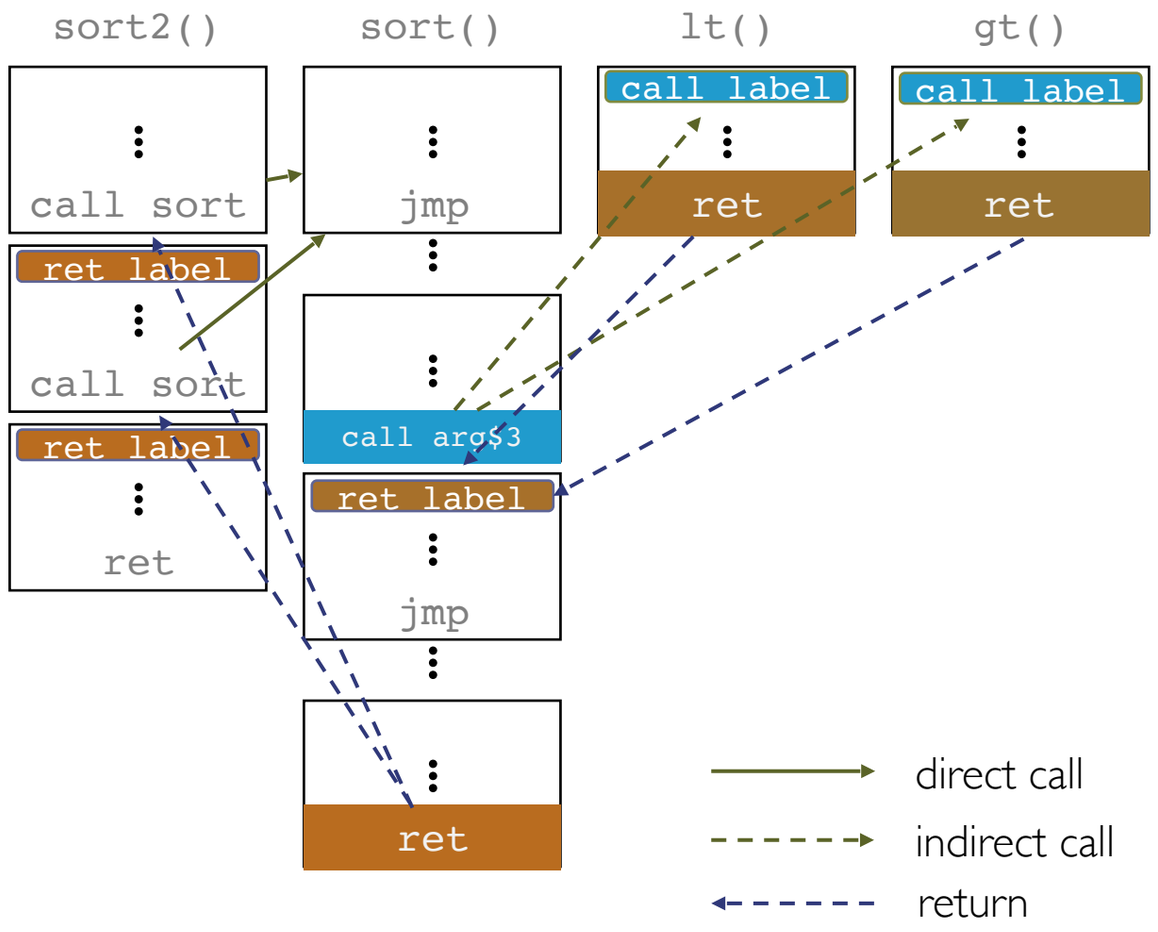
Jump to the destination only if the tag is equal to “12345678”

Abuse an x86 assembly instruction to insert “12345678” tag into the binary

Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Make sure that every call lands @ function entry
- Label for destination of return and indirect jumps
  - Make sure every indirect jump lands at start of BB

```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y {

  return x < y;

}

bool gt(int x, int y) {

  return x > y;

}
```



Why not just do fine-grained CFI?

How else can you choose labels?

$$\frac{tf = t_1^* \to t_2^* \qquad C_{\mathsf{table}} = n}{C \vdash \textbf{call\_indirect}\ tf : t_1^*\ \mathsf{i32} \to t_2^*}$$

$$\begin{array}{lcll} s; (\textbf{i32.const}\ j)\ \textbf{call\_indirect}\ tf & \hookrightarrow_i & \textbf{call}\ s_{\mathsf{tab}}(i,j) & \text{if } s_{\mathsf{tab}}(i,j)_{\mathsf{code}} = (\textbf{func}\ tf\ \textbf{local}\ t^*\ e^*) \\ s; (\textbf{i32.const}\ j)\ \textbf{call\_indirect}\ tf & \hookrightarrow_i & \textbf{trap} & \text{otherwise} \end{array}$$

How else can you choose labels?

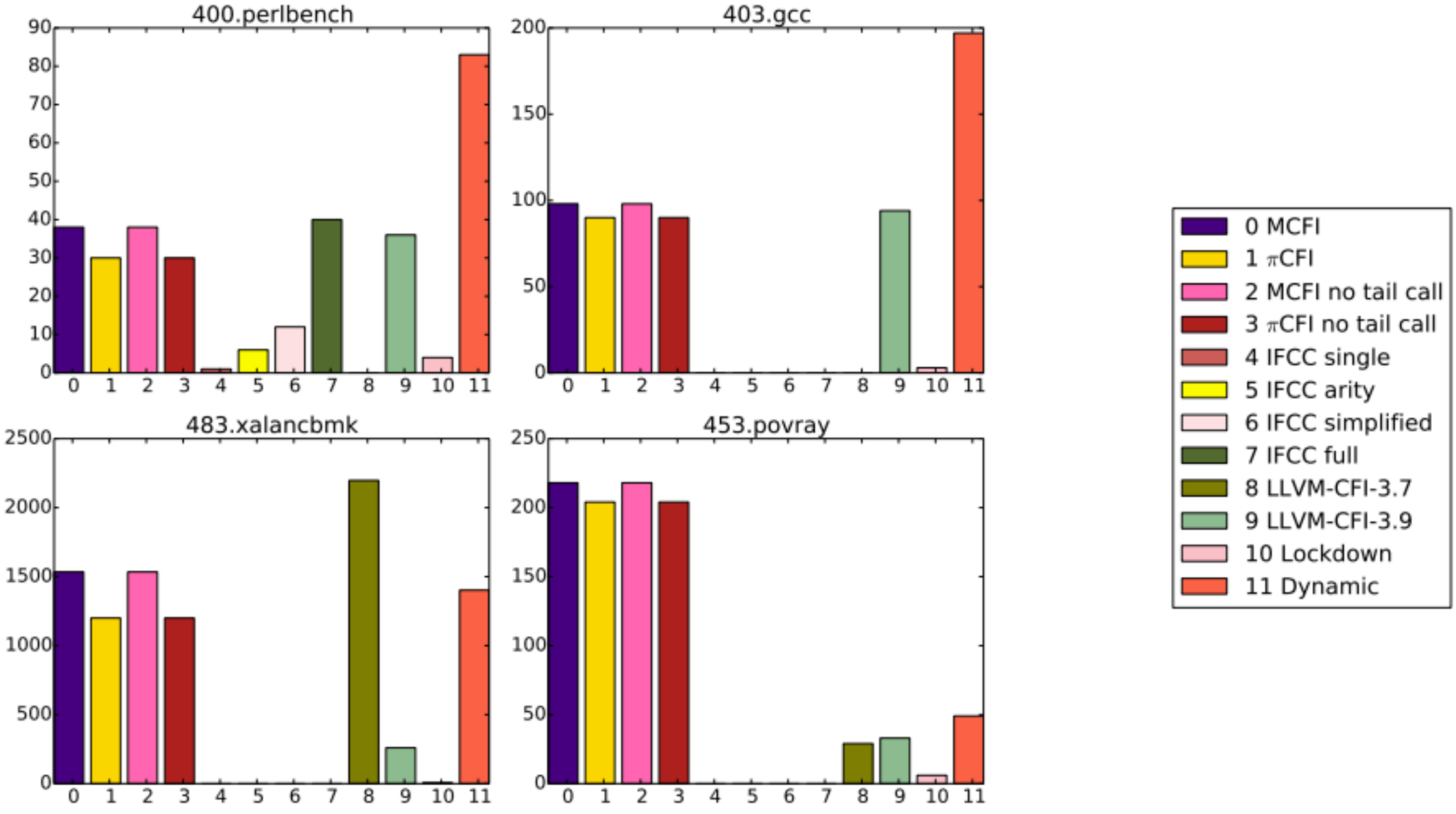


Fig. 4: Total number of forward-edge equivalence classes when running SPEC CPU2006 (higher is better).

Burrow et al.

What's the problem w/ equivalent classes?

What's the problem w/ equivalent classes?

```
int system(const char *command);
```

```
int myFunFunction(const char *command);
```

Limitations

- Precision tradeoff can lead to adverse effects
  - Can create gadgets if you don't bind flow for all indirect control transfers
  - Lot of gadgets on return path w/o shadow stack
    - One way to see this: safely generating labels at run time (this way only return to the function that called)

Limitations

- Overhead
  -
  -
- Scope
  -
  -
  -

Limitations

- Overhead
  - Runtime: every indirect branch instruction
  - Size: code before branch + label @ dst
- Scope
  -
  -
  -

Limitations

- Overhead
  - Runtime: every indirect branch instruction
  - Size: code before branch + label @ dst
- Scope
  - Data is not protected!
  - CFI does not protect against data-only attacks
  - Needs reliable W^X/DEP

Control-Flow Integrity: Precision, Security, and Performance

Nathan Burow, Purdue University
Scott A. Carr, Purdue University
Joseph Nash, University of California, Irvine
Per Larsen, University of California, Irvine
Michael Franz, University of California, Irvine
Stefan Brunthaler, Paderborn University & SBA Research
Mathias Payer, Purdue University