CSE 484/M584: Computer Security (and Privacy)

Spring 2025

David Kohlbrenner
dkohlbre@cs

UW Instruction Team: David Kohlbrenner, Yoshi Kohno, Franziska Roesner, Nirvan Tyagi. Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials

Admin

- Lab 1a due Wednesday
  - Exploits 1+2, along with writeups
  - Remember: clarifications on Ed
  - Optional reading: frame pointer + printf doc
  - Exploit 6 (Lab 1b) significantly harder/different. Start early.

- DRS accommodations
  - If you haven't gotten an email from me about them, email me ASAP
  - In-class extensions will be applied _at the end of the quarter_ (so please refer to our email agreement rather than the gradescope deadlines)

- Final exam
  - Reminder that it exists, scheduling/accommodations/etc.

Binary exploitation part 3

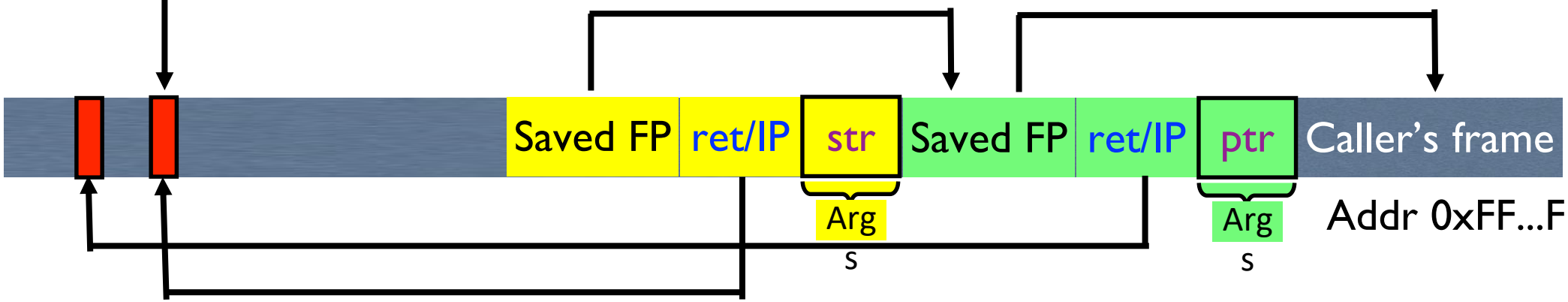
Frame pointers (and saved frame pointers)

Review: Stack Buffers – bar() calls foo()

```
void bar(char *ptr) {
    foo(ptr);
    ptr++;
}
void foo(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

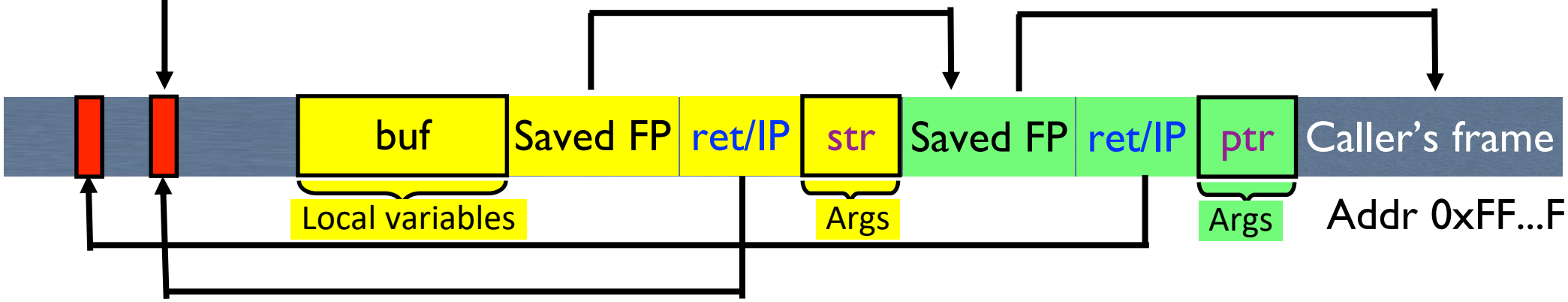
Review: Stack Buffers – bar() calls foo()

```
void bar(char *ptr) {
    foo(ptr);
    ptr++;
}
void foo(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```



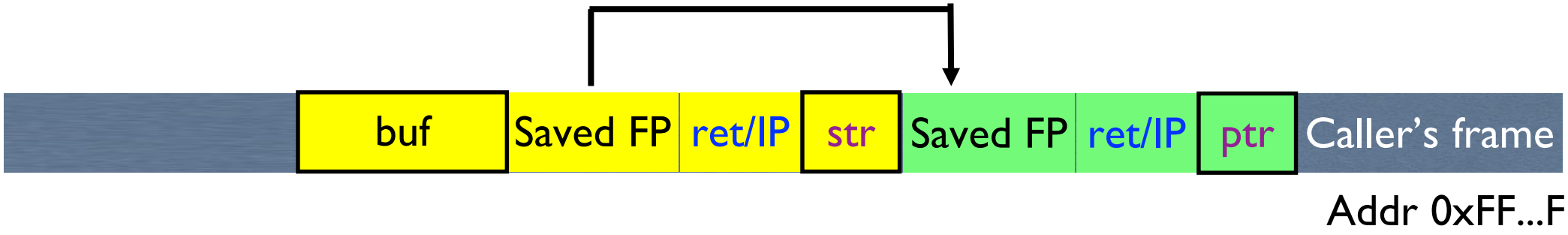
Review: Stack Buffers – bar() calls foo()

```
void bar(char *ptr) {
    foo(ptr);
    ptr++;
}
void foo(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```



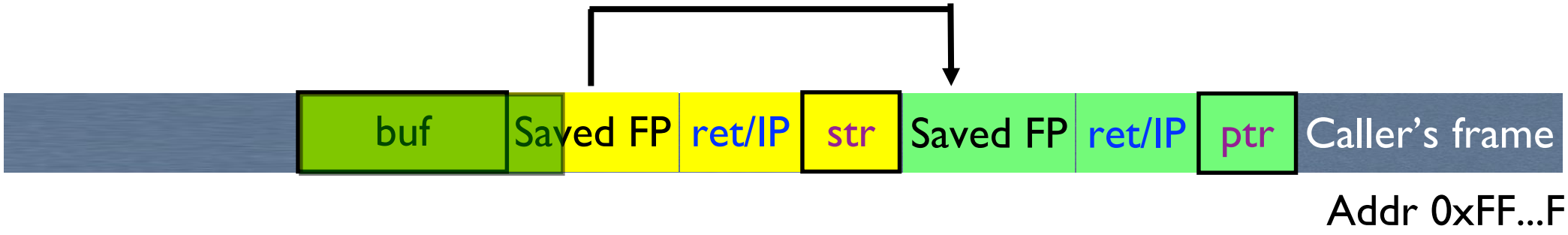
Frame Pointer Overflow

```
void bar(char *ptr) {
    foo(ptr);
    ptr++;
}
void foo(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```



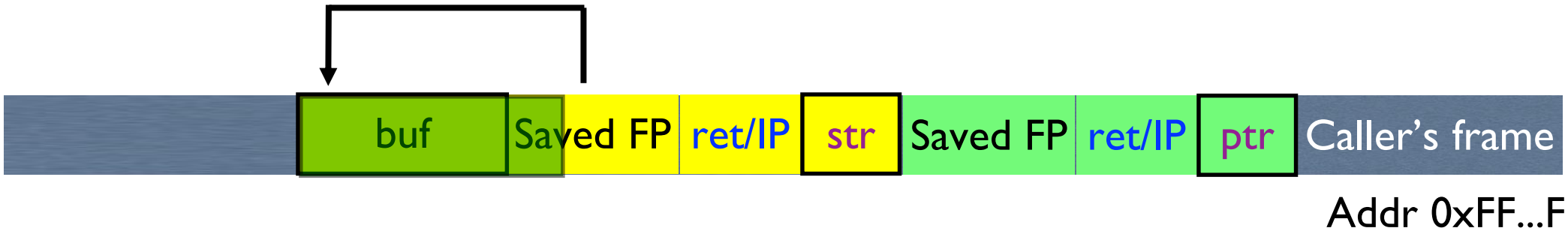
Frame Pointer Overflow

```
void bar(char *ptr) {
    foo(ptr);
    ptr++;
}
void foo(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```



Frame Pointer Overflow

```
void bar(char *ptr) {
    foo(ptr);
    ptr++;
}
void foo(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```



Variable argument exploitation

Variable Argument Functions in C (varargs)

- In C, can define a function with a variable number of arguments
  - E.g. `void printf(const char* format, …)`

- Examples of usage:

```
printf("hello, world");
printf("length of '%s' = %d\n", str, strlen(str));
printf("invalid fd %d\n", fd);
```

How does it know?

```
void printf(const char* format, …)
```

- printf uses ‘format’ to define what arguments are needed!
  - ‘%’ specifiers tell printf “expect another argument to this function
  - %i, %d, %x, etc. expect *integer* arguments
  - %p expects a *pointer* argument
  - %s expects a *pointer to a string* argument

Printf usage

- Reasonable usage

```
int foo = 1234;
printf("foo = %d in decimal, %X in hex", foo, foo);

"foo = 1234 in decimal, 4D2 in hex"
```

- Sloppy usage

```
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

What happens if buf contains % format specifiers??

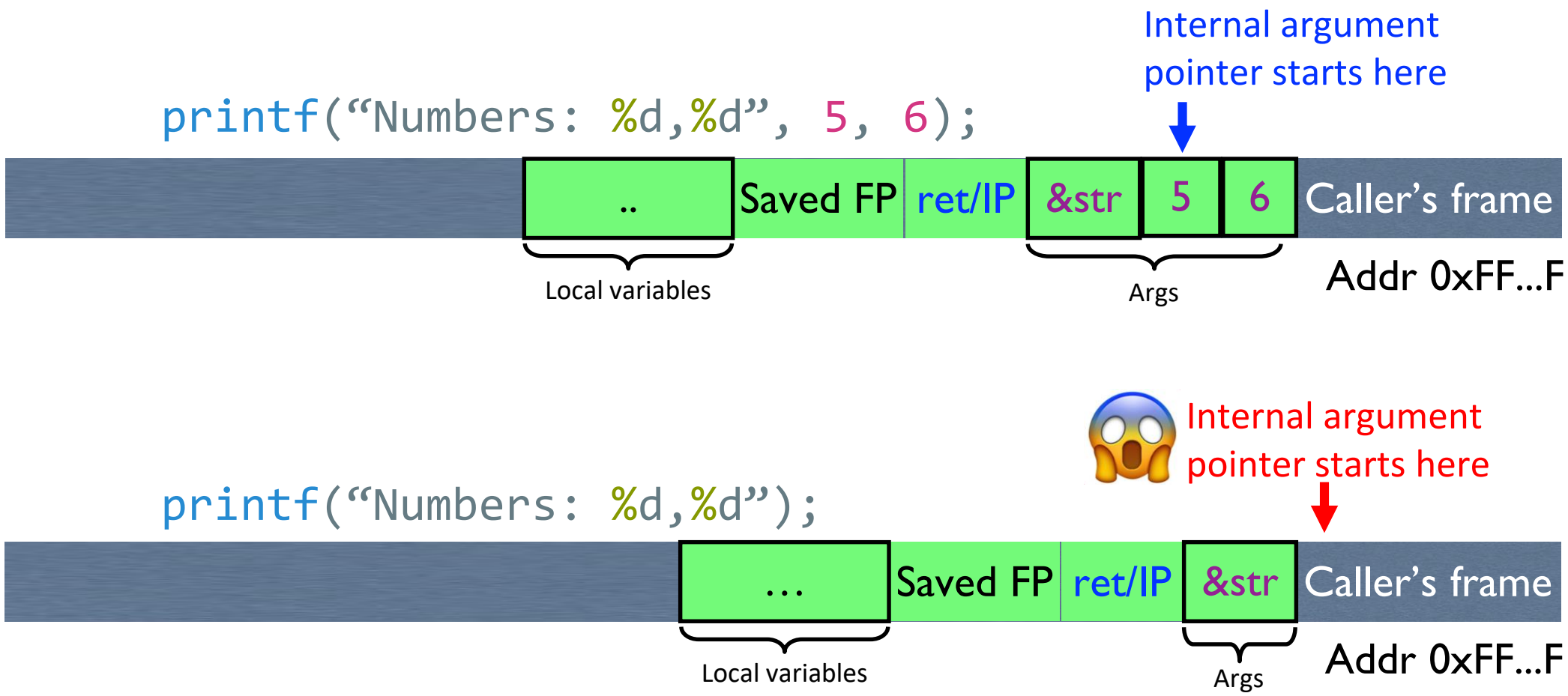
Implementation of Variable Args

- Special functions va_start, va_arg, va_end compute arguments at run-time

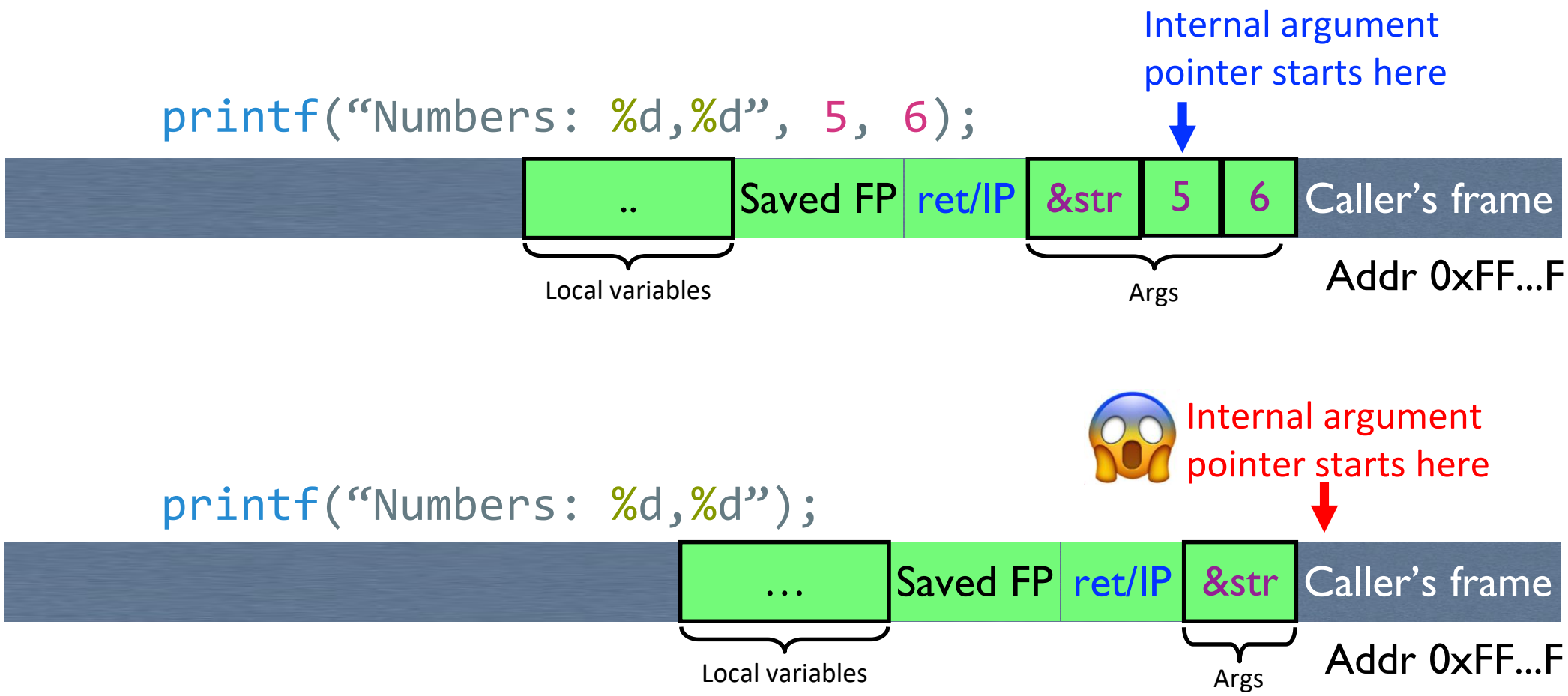
```
void printf(const char * format, ...)
{
  int i; char c; char * s; double d;
  va_list ap; /* declare an "argument pointer" to a variable arg list */
  va_start(ap, format); /*initialize arg pointer using last known arg */
  for (char *p = format; *p != '\0'; p++) {
    if (*p == '%') {
      switch (*++p) {
        case 'd':
          i = va_arg(ap, int); break;
        case 's':
          s = va_arg(ap, char*); break;
        case 'c':
          c = va_arg(ap, char); break;
        ... /* etc for each % specification */
      }
    }
  }
  ...
  va_end(ap); /* restore any special stack manipulations */
}
```

This is simplified code, e.g., handles %d but not %10d

Closer Look at the Stack

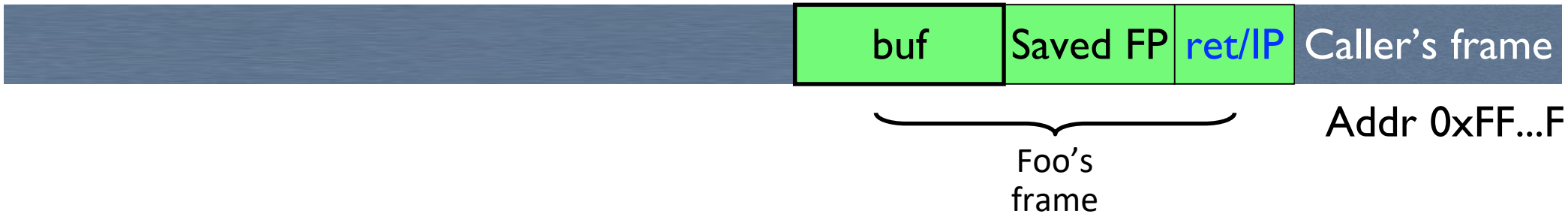


Gradescope: Brainstorm printf exploitation



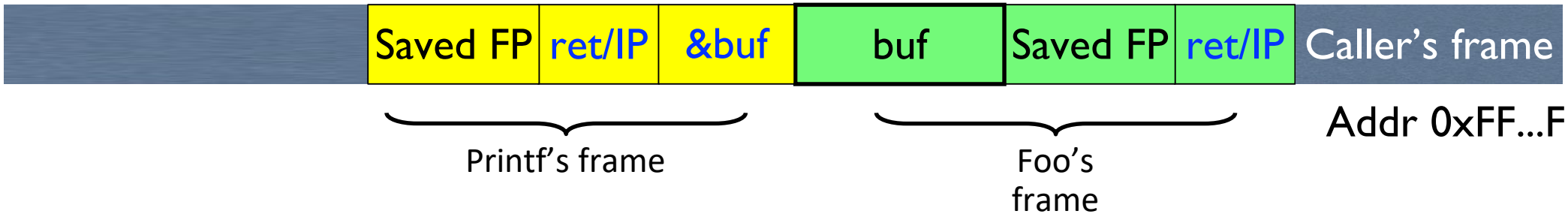
Format String Attacks

```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```



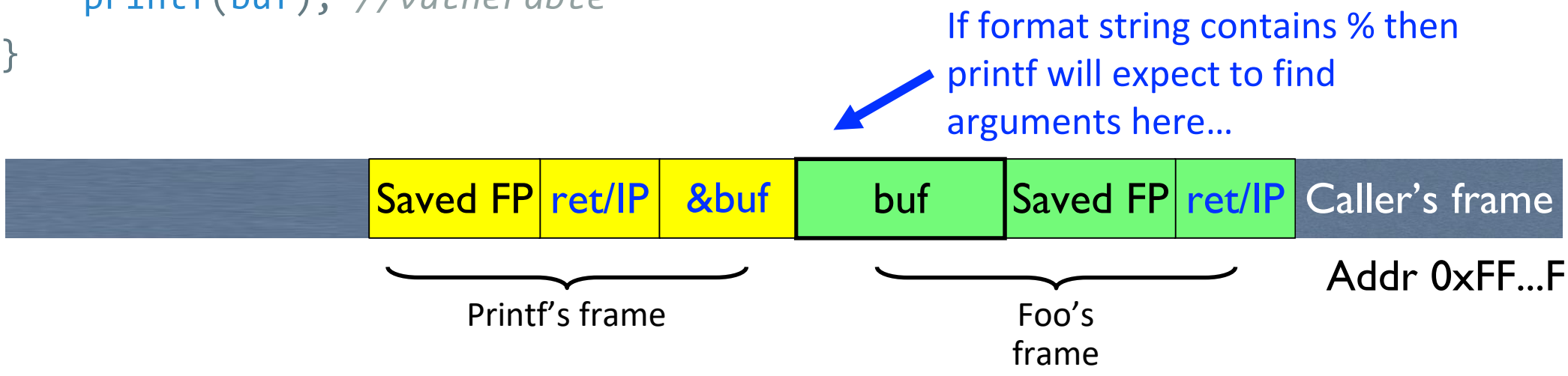
Format String Attacks

```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```



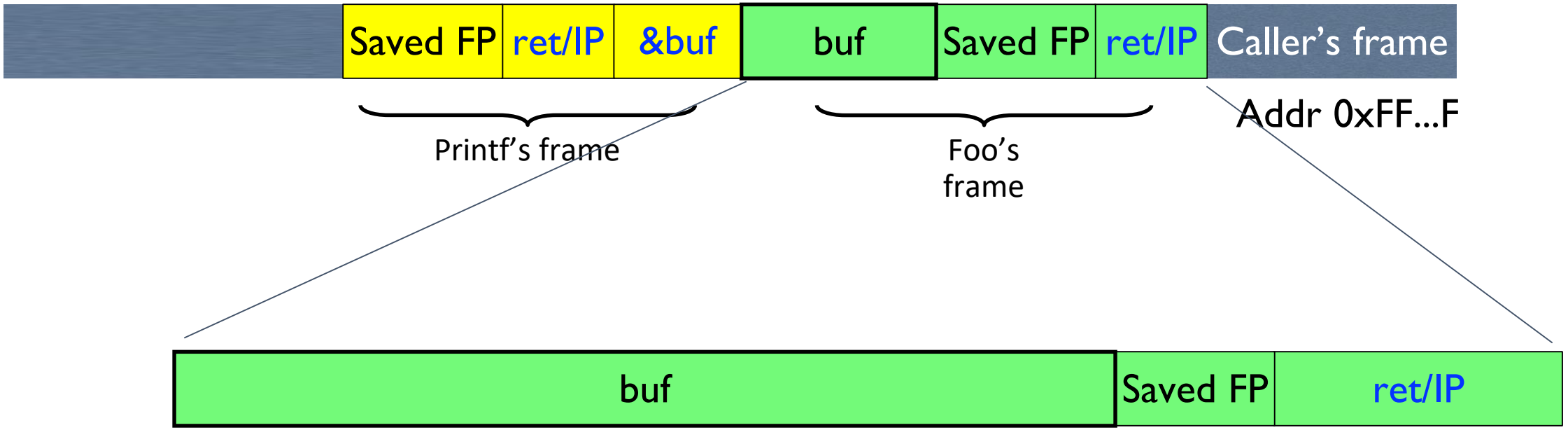
Format String Attacks

```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```

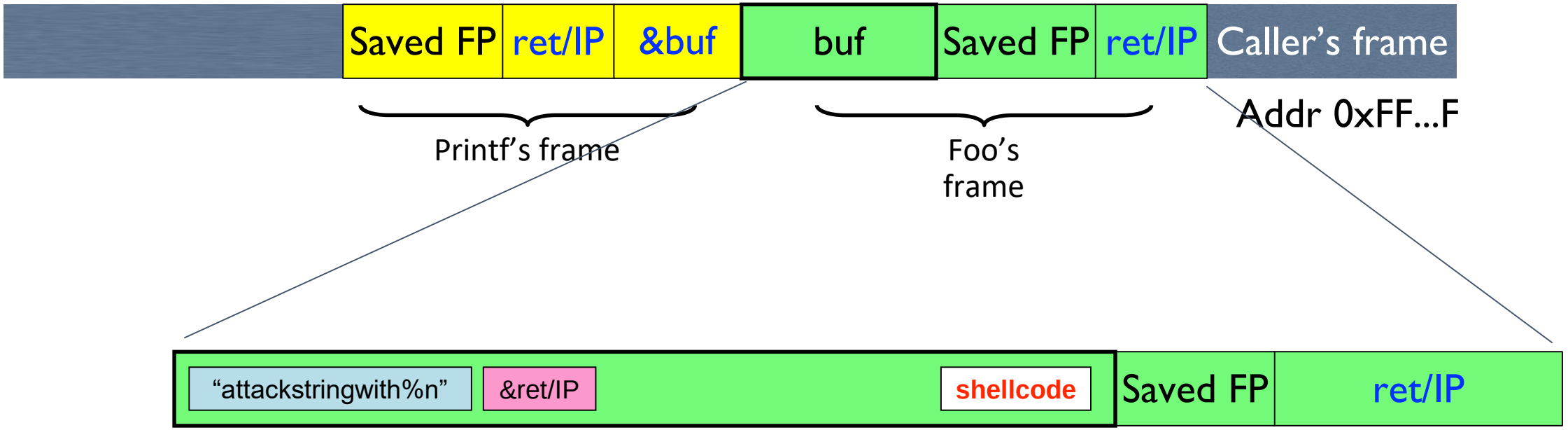


What should the string returned by readUntrustedInput() contain??

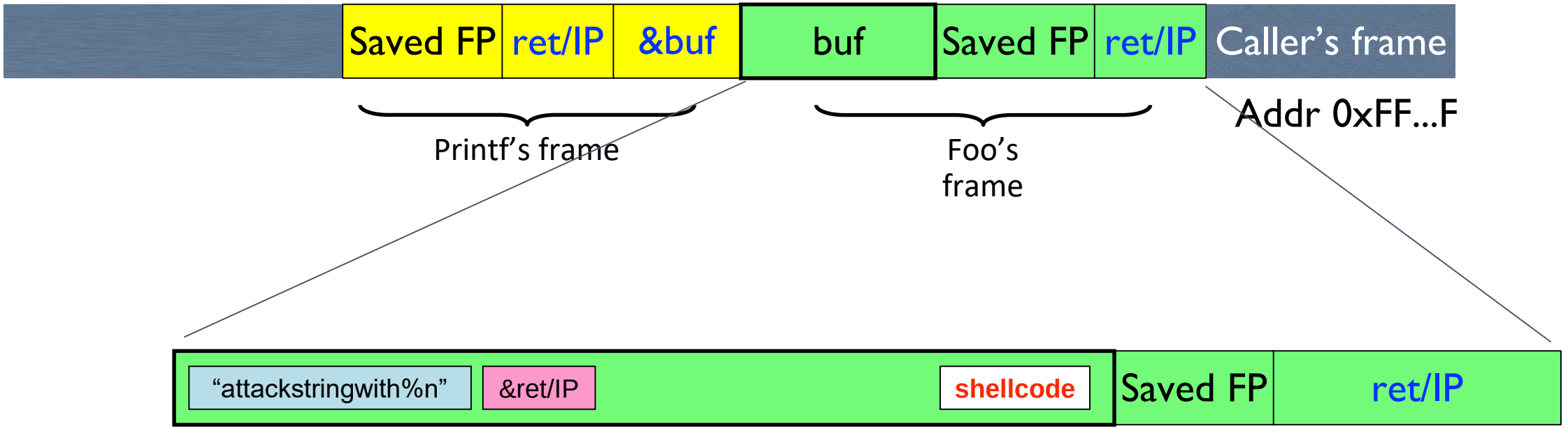
```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```



```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```

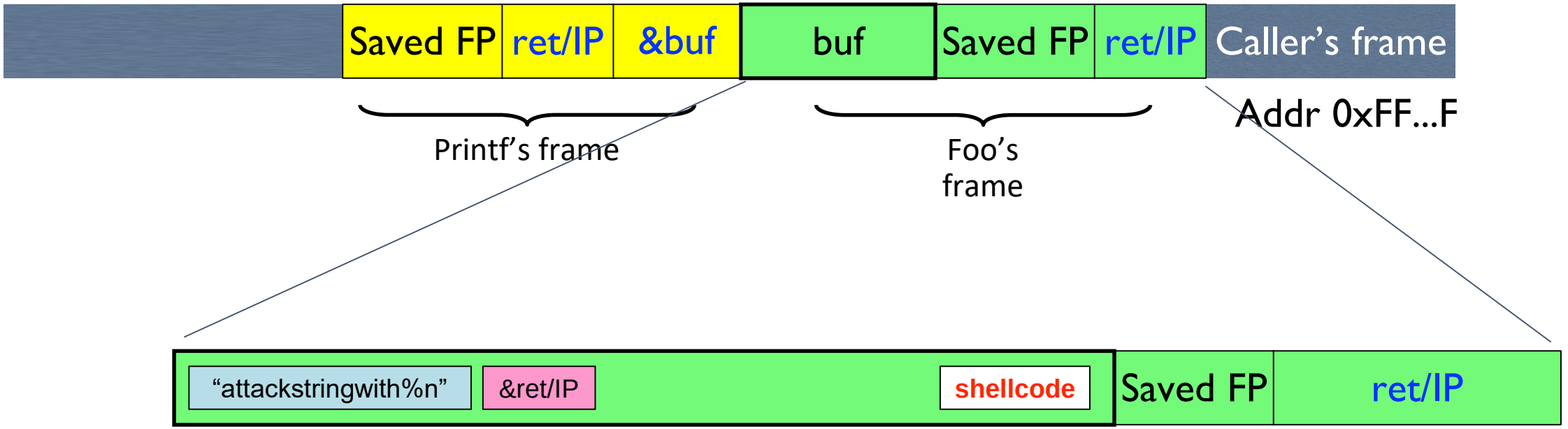


```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```



Pointer to shellcode is large! Can't fit that many characters!

```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```

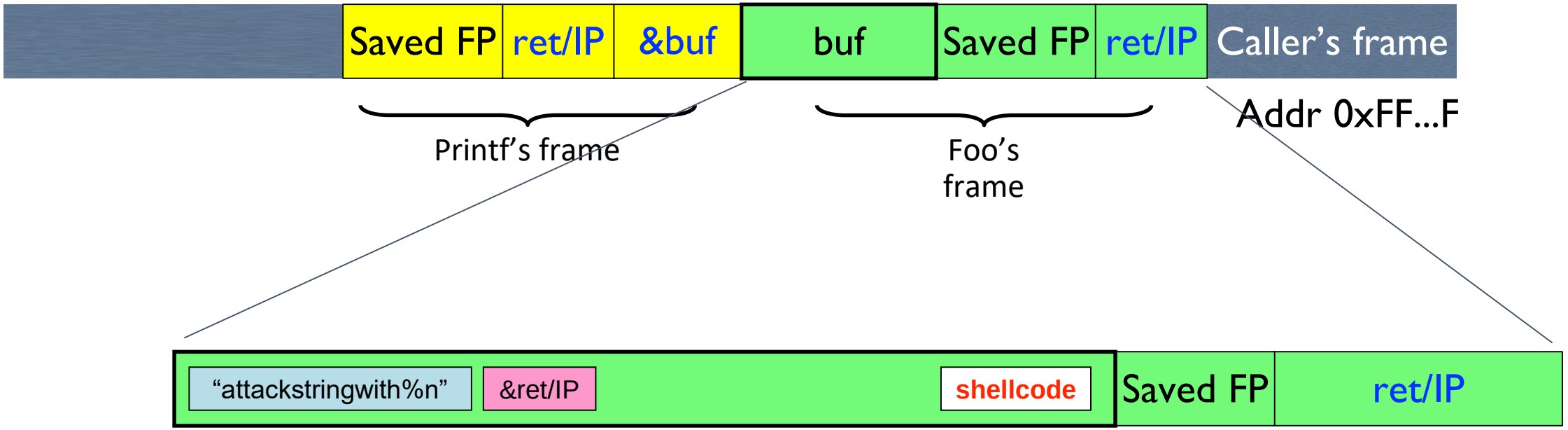


Pointer to shellcode is large! Can't fit that many characters!

Helpful hint: C allows you to concisely specify the "width" to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf("%5d%n", 10) will print three spaces followed by the integer: " 10"
That is, the %n will write 5, not 2.

```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```



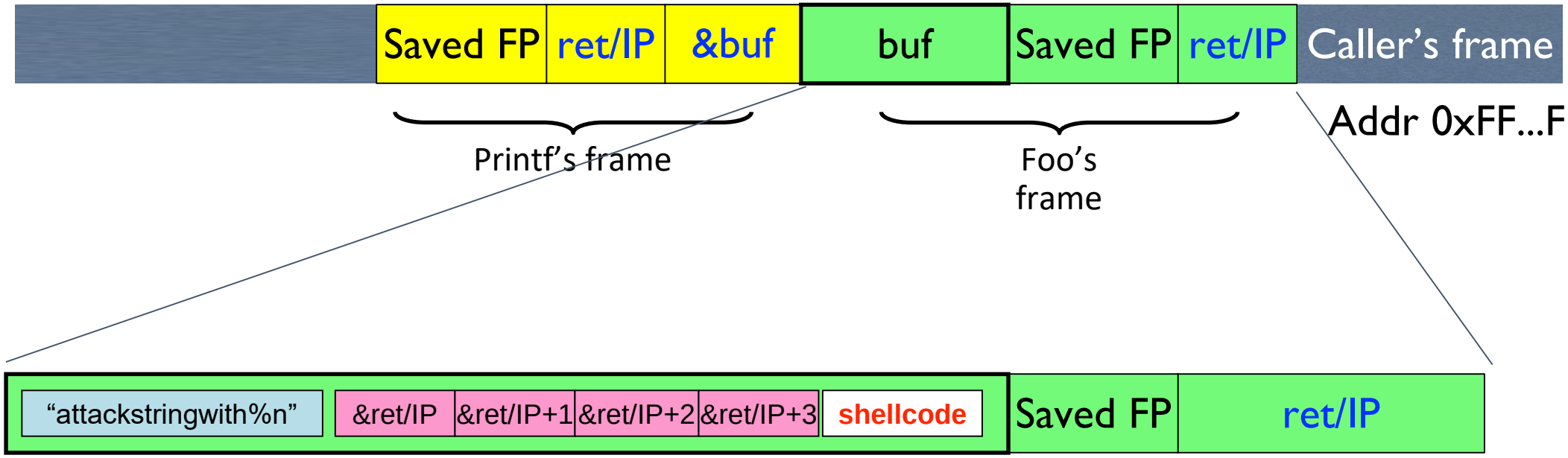
Pointer to shellcode is large! Can't print that many characters!

Helpful hint: C allows you to concisely specify the "width" to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf("%5d%n", 10) will print three spaces followed by the integer: " 10"
That is, the %n will write 5, not 2.

Still not enough!

```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```



Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte. (4x %n to write into &RET, &RET+1, &RET+2, &RET+3)