SFICOMPCERT

Software Fault Isolation
Machine code-level sandboxing

sfiCompCert

Security context :

- Untrusted C code (attacker could write anything)
- Compiled on a special compiler
- sfiCC prevents accesses outside a specific address range
- Hardware is not accessible

Example usage

- Securing low-trust applications even in case of hardware vulnerabilities (e.g. AOC)
- Running 3rd party libraries within trusted code
- ...

APSYS
AUGMENTED TRUST

What C needs to be restricted ?

- `a[i]`
- `p = \&a`
- `fnp = \&func;`
- `qsort(a, n, s, cmp_fn);`
- `open("/etc/passwd");`

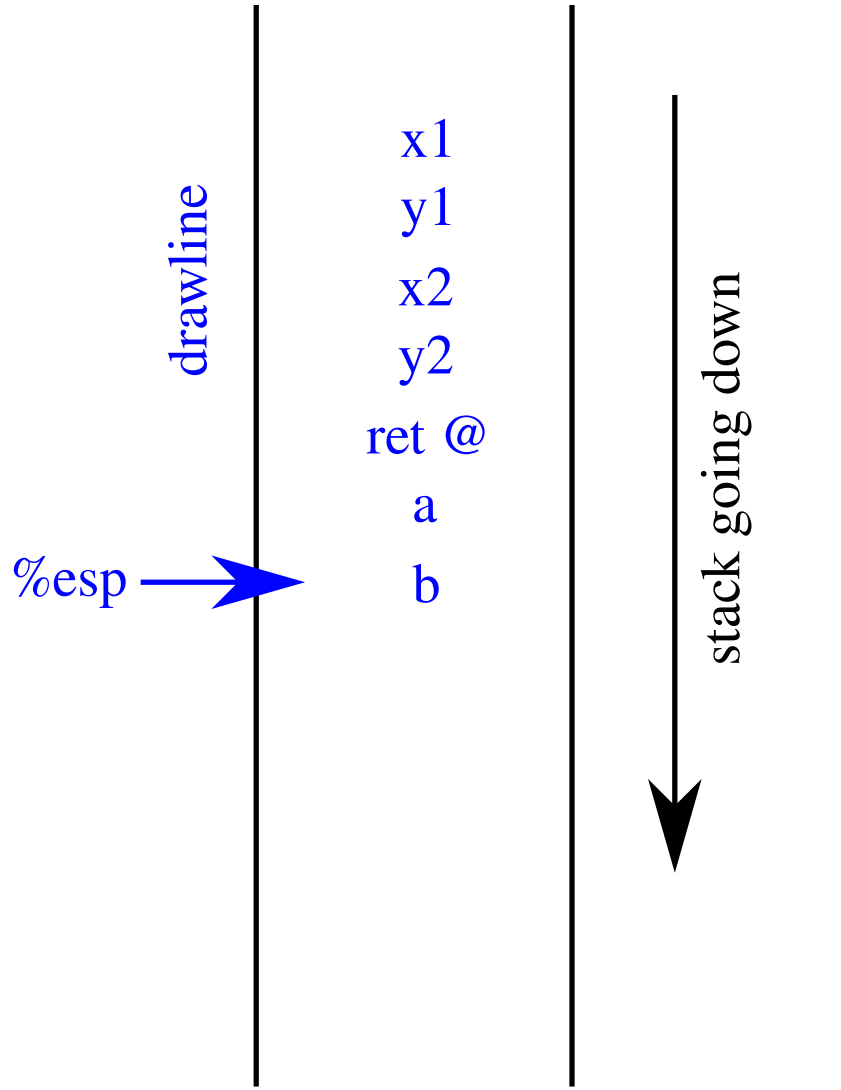
Principle

- Memory area dedicated to sandbox data (e.g. 0x80000000—0x80040000)
- Static variables allocated in sandbox
- malloc() allocates in sandbox
- Addresses for reads and writes are checked
- Local variables remain on normal stack
- ... except if their address is taken, then they are stored on a 'shadow stack' in the sandbox area
- Function pointers are replaced with table indexes
- External calls controled through symbol manipulations

APSYS
AUGMENTED TRUST

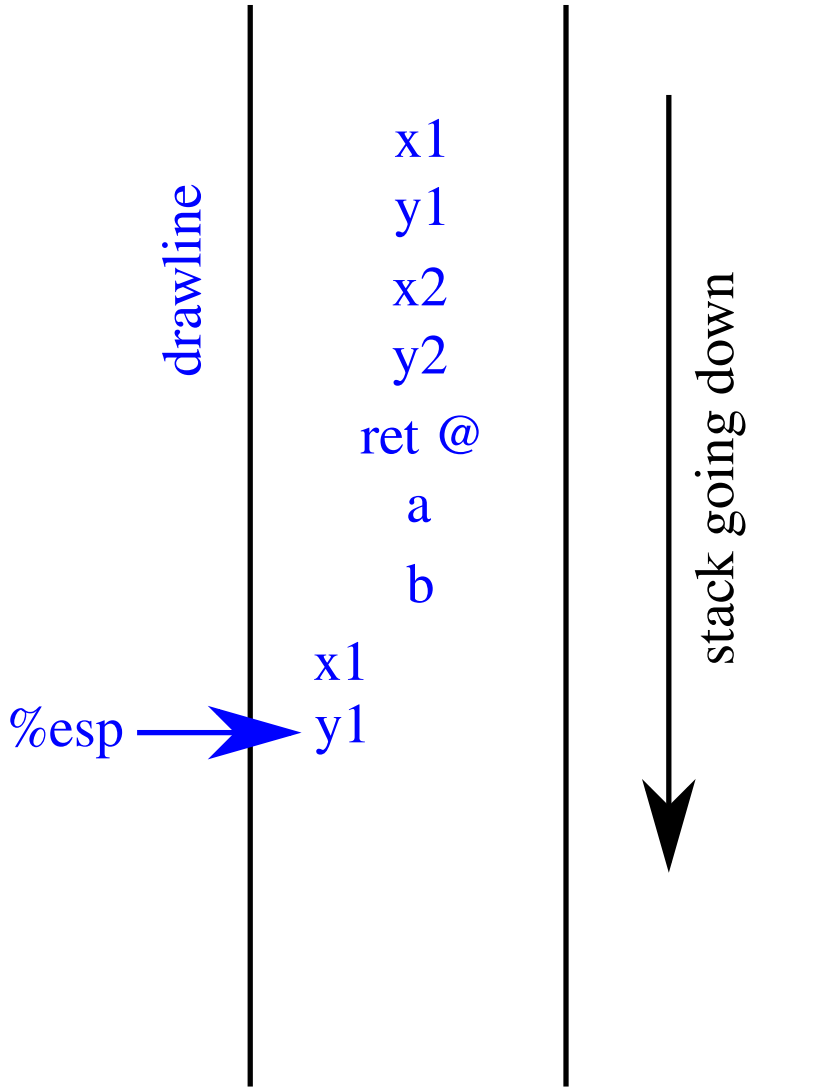
Function calls and the stack

```
void drawpoint(int x, int y) {
    int i, j;
    ...
    return;
}

void drawline(int x1, int y1, int x2, int y2) {
    int a, b;
    ...
    drawpoint(x1,y1);
    return;
}
```



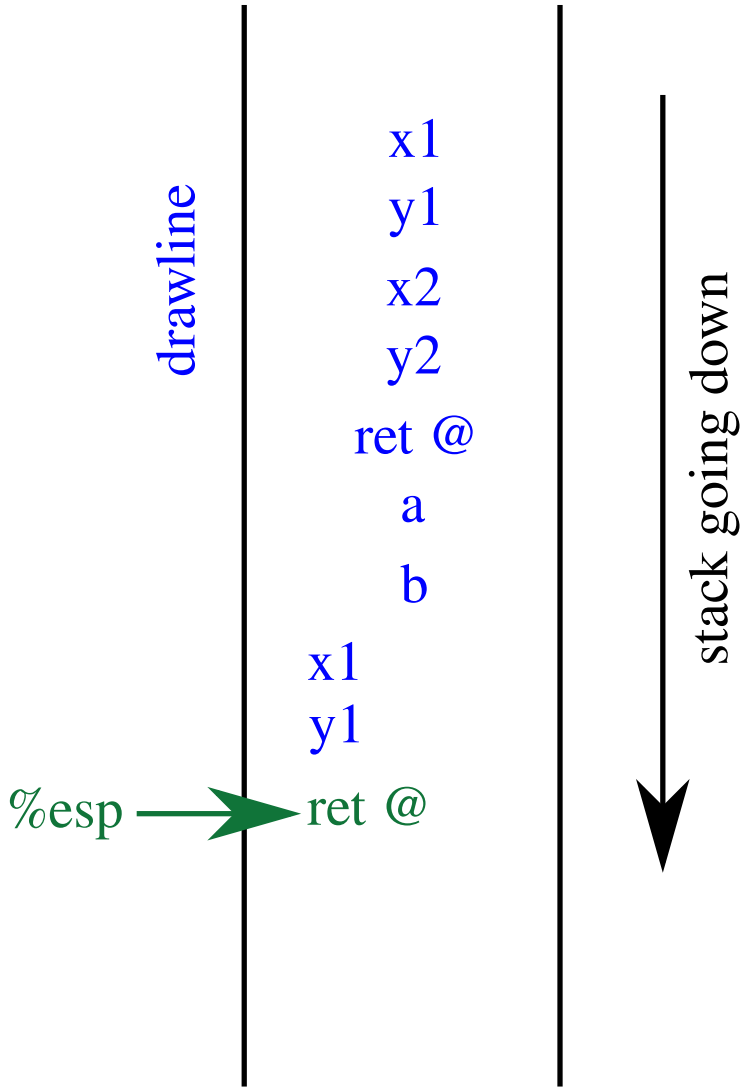
Function calls and the stack

```
void drawpoint(int x, int y) {
    int i, j;
    ...
    return;
}

void drawline(int x1, int y1, int x2, int y2) {
    int a, b;
    ...
    drawpoint(x1,y1);
    return;
}
```



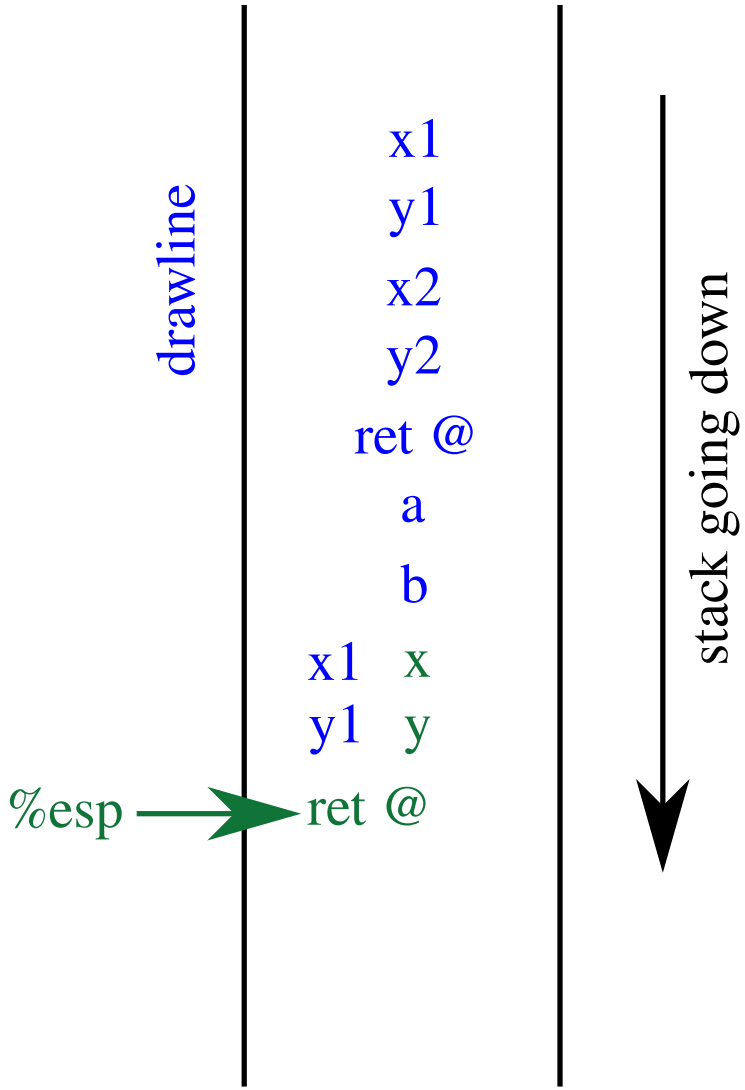
Function calls and the stack

```
void drawpoint(int x, int y) {
    int i, j;
    ...
    return;
}

void drawline(int x1, int y1, int x2, int y2) {
    int a, b;
    ...
    drawpoint(x1,y1);
    return;
}
```



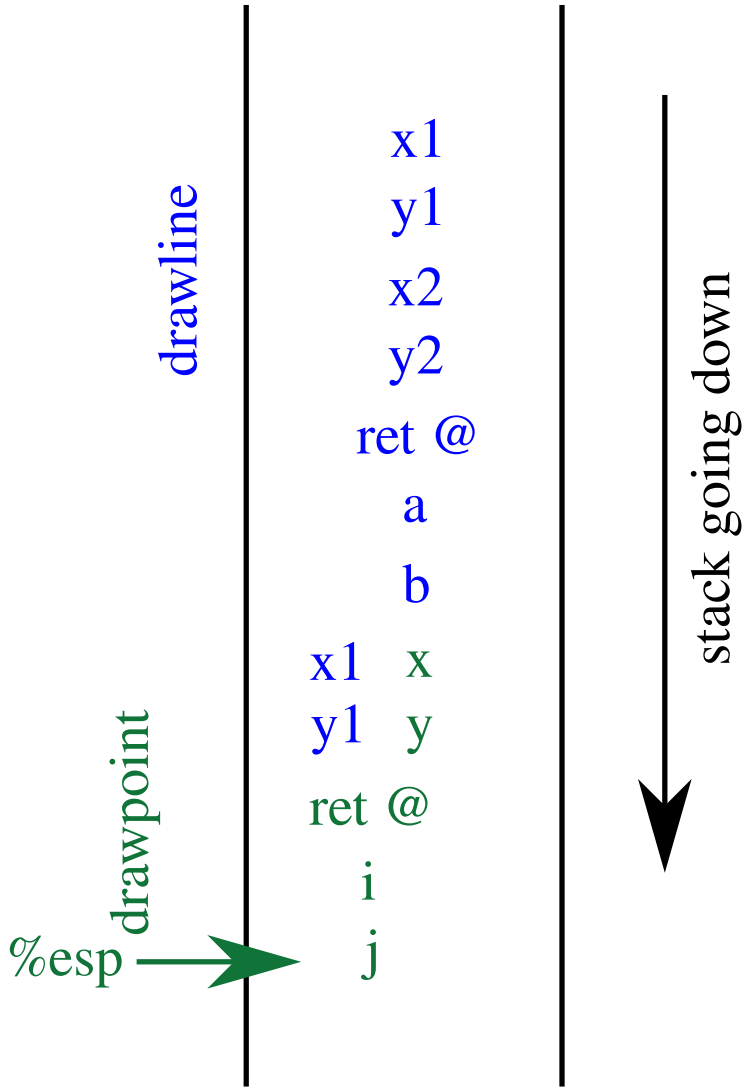
Function calls and the stack

```
void drawpoint(int x, int y) {
    int i, j;
    ...
    return;
}

void drawline(int x1, int y1, int x2, int y2) {
    int a, b;
    ...
    drawpoint(x1,y1);
    return;
}
```



Function calls and the stack

```
void drawpoint(int x, int y) {
    int i, j;
    ...
    return;
}

void drawline(int x1, int y1, int x2, int y2) {
    int a, b;
    ...
    drawpoint(x1,y1);
    return;
}
```



Reading static without sfi

```
int read_global_var(int idx)
{
    return statique[idx];
}

read_global_var:
#int read_global_var(int idx)
        subl    $12, %esp
        leal    16(%esp), %eax
        movl    %eax, 0(%esp)
        movl    0(%eax), %eax
.L127:
#   return statique[idx];
        movl    statique(,%eax,4), %eax
.L128:
        addl    $12, %esp
```

Reading/Writing global

```
read_global_var:
#int read_global_var(int idx)
        subl    $12, %esp
        leal    16(%esp), %eax
        movl    %eax, 0(%esp)
        movl    0(%eax), %ecx
.L130:
#       return statique[idx];
        leal    0x80000014(,%ecx,4), %eax
        andl    $0x0003FFFC, %eax
        movl    0x80000000(%eax), %eax
        addl    $12, %esp
.L131:
        ret
```

Buffer overflows remain within sandbox

Reading/writing locals

- Locals whose address is not taken are kept on normal stack
- Locals whose address is taken are stored on a shadow stack
- Shadow stack pointer (as offset) located at 0x80000000
- Function prologue / epilogue updates *0x80000000
- (Experimental thread support with shadow stack created for each thread by `pthread_create()`, shadow stack pointer passed as argument to each function)

Contrived example

For local allocation and function pointers...

```
int recurs_add_fn(int (*fn)(int), int* inc,
                  int a[], int n)
{
    int s;
    if (n == 0) return 0;

    (inc[n])++;
    s = fn(a[0]) + *inc;
    return s + recurs_add_fn(fn, &s,
                             &a[1], n - 1);
}

i = 0;
i = recurs_add_fn(&sqr, &i,
                  statique, ARRAY_SIZE(statique));
```

Locals : allocation

```
recurs_add_fn:
#int recurs_add_fn(int (*fn)(int), int* inc, int a[],
        subl    $60, %esp
        leal    64(%esp), %eax
        [... save registers ...]
        [... read parameters ...]
        movl    0x80000000, %edx
        leal    8(%edx), %edx
        movl    %edx, 0x80000000
        leal    -8(%edx), %ebx
```

Locals : writing

```
#       s = fn(a[0]) + *inc;
            [... compute right-side to %eax ...]
            leal    0x80000000(%ebx), %edx
            movl    %edx, %ecx
            andl    $0x0003FFFC, %ecx
            movl    %eax, 0x80000000(%ecx)
```

Locals : indirect reading and writing

- %edi : inc
- %esi : n

```
#      (inc[n])++;
       leal        0(%edi,%esi,4), %eax
       andl        $0x0003FFFC, %eax
       movl        0x80000000(%eax), %edx
       leal        1(%edx), %ecx
       movl        %ecx, 0x80000000(%eax)
```

Locals : passing address and epilogue

```
.L120:
#     return s + recurs_add_fn(fn, &s, &a[1], n - 1);
        [...]
        leal    0x80000000(%ebx), %edx

        movl    %edx, 4(%esp)
        call    recurs_add_fn
        movl    0x80000000, %ecx  # release stack
        leal    -8(%ecx), %ecx
        movl    %ecx, 0x80000000
        leal    0x80000000(%ebx), %ecx
        andl    $0x0003FFFC, %ecx
        movl    0x80000000(%ecx), %edx
        leal    0(%edx,%eax,1), %eax
```

Referencing function pointer

sfiCompCert builds tables of function pointers

```
array$i_i:
        .long   __compcert_va_int32
        .long   printf$i_i
        .long   atoi
        .long   sb_malloc
        .long   sqr
        .long   read_local_var
        .long   read_global_var
        .long   __compcert_va_int32
```

Referencing function pointer

```
#     i = recurs_add_fn(&sqr, &i,
#           statique, ARRAY_SIZE(statique));
      movl    $16, %esi
      movl    $0x80000014, %ecx
      movl    $10, %edi
      movl    %edi, 12(%esp)
      movl    %ecx, 8(%esp)
      movl    %eax, 4(%esp)
      movl    %esi, 0(%esp)
      call    recurs_add_fn
```

Dereferencing function pointer

%edi : &a[0]

```
#     s = fn(a[0]) + *inc;
# 40(%esp) is function pointer fn
        movl    40(%esp), %ecx
        andl    $28, %ecx
        movl    array$i_i(%ecx), %edx
        # Stack a[0]
        movl    %edi, %eax
        andl    $0x0003FFFC, %eax
        movl    0x80000000(%eax), %eax
        movl    %eax, 0(%esp)
        call    *%edx
```

Get me out !

- In/Out : Buffers in sandbox

Get me out !

- In/Out : Buffers in sandbox
- Filter external calls : `sed s/open/sb_open/ in.c > out.c`

Get me out !

- In/Out : Buffers in sandbox
- Filter external calls : `sed s/open/sb_open/ in.c > out.c`
- Always :

Get me out !

- In/Out : Buffers in sandbox
- Filter external calls : `sed s/open/sb_open/ in.c > out.c`
- Always :
  - Pointers point inside sandbox

Get me out !

- In/Out : Buffers in sandbox
- Filter external calls : `sed s/open/sb_open/ in.c > out.c`
- Always :
  - Pointers point inside sandbox
  - Swap function indexes with their addresses (e.g. `qsort`)

Get me out !

- In/Out : Buffers in sandbox
- Filter external calls : `sed s/open/sb_open/ in.c > out.c`
- Always :
  - Pointers point inside sandbox
  - Swap function indexes with their addresses (e.g. `qsort`)
- Optionally, tailored checks : open only “input.txt”, only `O_RDONLY`...

Questions ?