KU LEUVEN

DISTRINET RESEARCH GROUP

New Technologies for System Software Security

Prof. Frank PIESSENS

SECAPPDEV 2017

iMinds CONNECT.INNOVATE.CREATE

Introduction

- System software is often programmed in C-like languages
  - Another session has covered the security consequences and the raging attacker-defender race
- The purpose of this lecture is to give you a taste of some recent advances in this area:
  - Systems-level compartimentalization mechanisms:
    - We look at protected module architectures like the new Intel SGX
  - Alternative **safe** systems-programming languages:
    - The Rust language from Mozilla and the Go language from Google
  - Advanced compiler-based countermeasures:
    - Control-Flow Integrity
    - Pointer-based checking

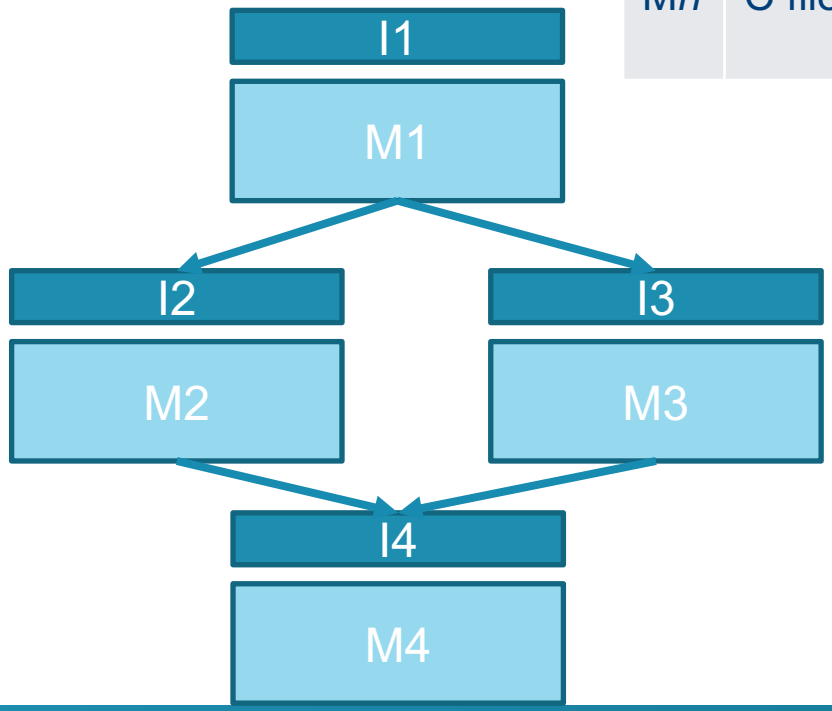
Overview

- Protected module architectures
  - Fine grained isolation at machine code level
  - Supported in the recent Intel Skylake processors under the name Intel Software Guard eXtentions (Intel SGX)
- Safe systems programming languages
  - Compiled languages with low-level control over memory, but with strong safety assurance
  - Supported in the Rust and Go programming languages
- Advanced compiler based countermeasures
  - Control-flow integrity (CFI)
  - Pointer-based checking

Problem statement

Consider a program consisting of a number of modules, and their dependencies.

	C	Java	ML
I*n*	Header file	(Roughly) Interfaces	Signature
M*n*	C file	(Roughly) Classes	Structure / Functor

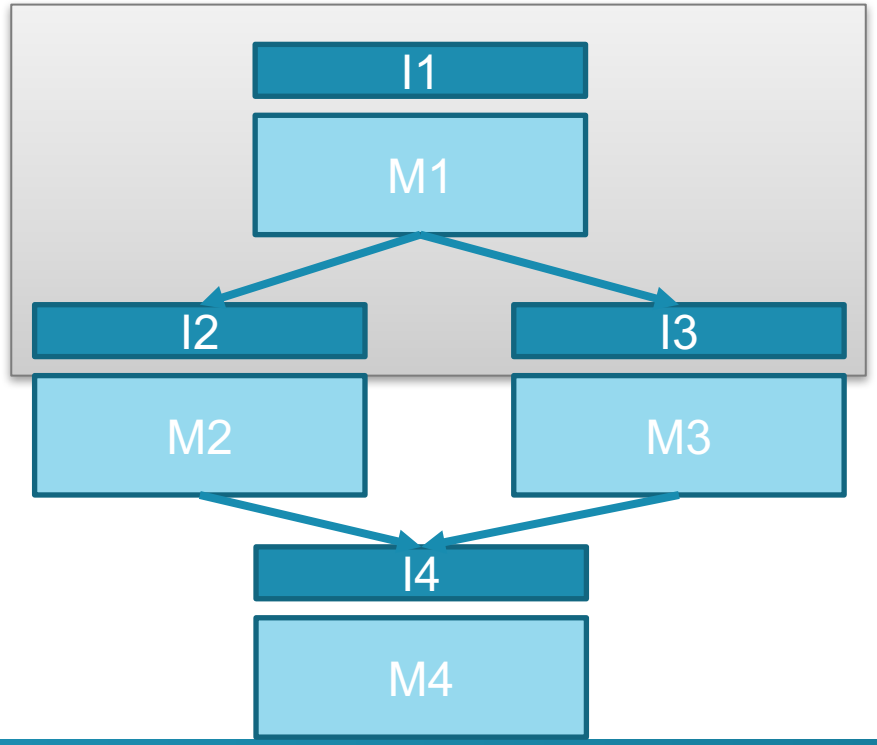


iMinds CONNECT.INNOVATE.CREATE
KU LEUVEN DISTRINET RESEARCH GROUP

Problem statement

Suppose you have proven a (security) property of module M1 by modular reasoning. E.g.:

- Some invariant holds on the module's state
- Some data in the module remains confidential towards other modules
- The integrity of some data in the module is protected from other modules

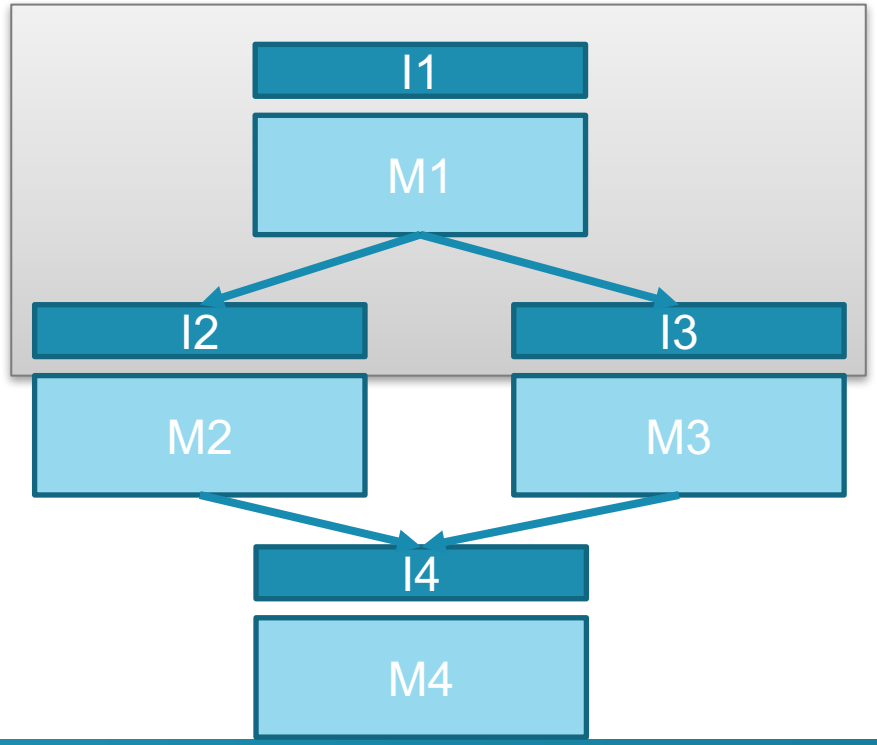


iMinds CONNECT.INNOVATE.CREATE
KU LEUVEN DISTRINET RESEARCH GROUP

Problem statement

Suppose you have proven a (security) property of module M1 by modular reasoning. E.g.:

- Some invariant holds on the module's state
- Some data in the module remains confidential towards other modules
- The integrity of some data in the module is protected from other modules



KEY QUESTION:

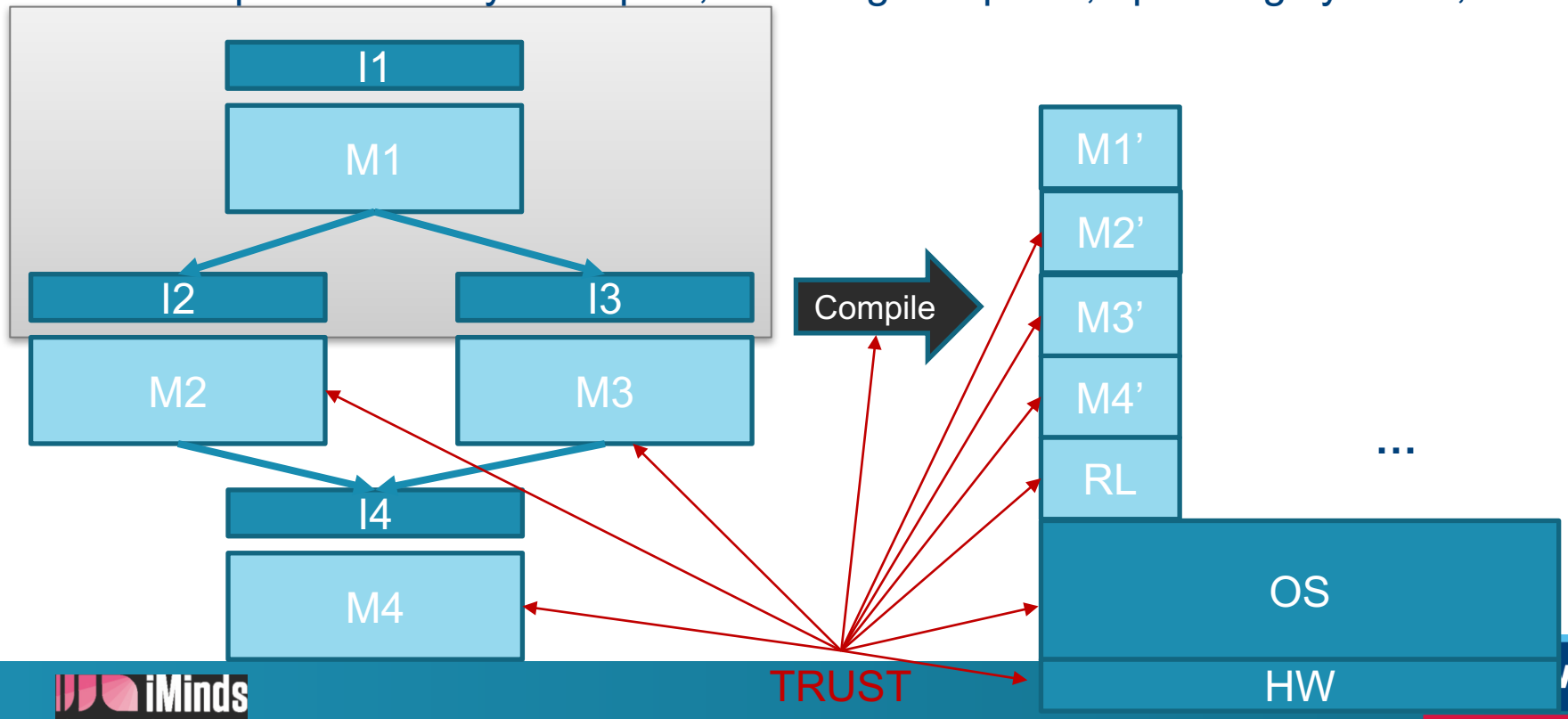
What do you need to trust to be sure that this property will hold at run time?

iMinds CONNECT.INNOVATE.CREATE
KU LEUVEN DISTRINET RESEARCH GROUP

Problem statement

You have to trust at least:

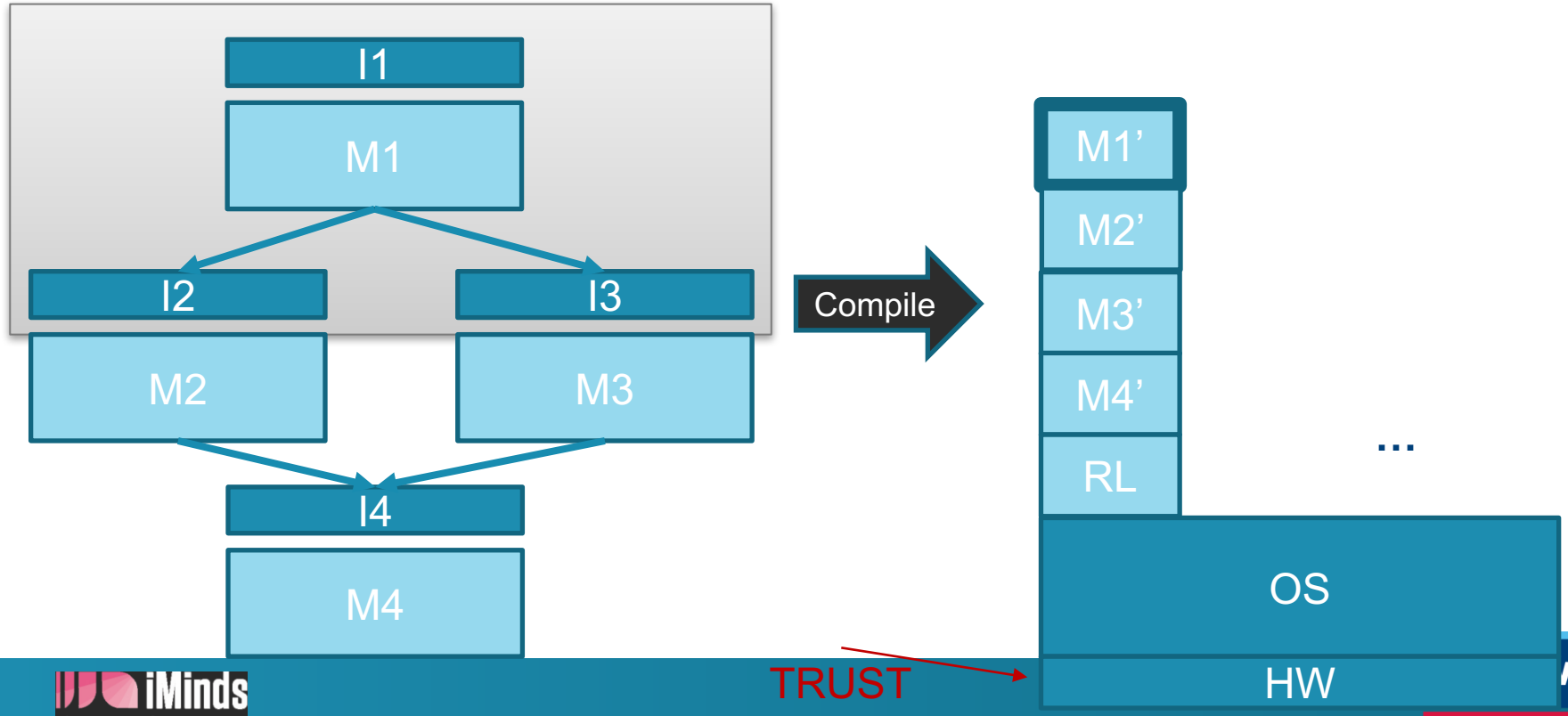
- Your reasoning (e.g. the soundness of the verification tool)
- The implementations of the unverified modules of your program
- The execution infrastructure
  - Potentially “simple” : an interpreter on bare hardware
  - In practice always complex, including compilers, operating systems, …



iMinds CONNECT.INNOVATE.CREATE
DISTRINET RESEARCH GROUP

Problem statement

Can we reduce the TCB to just the hardware, while maintaining backward compatibility with legacy OS's and applications?

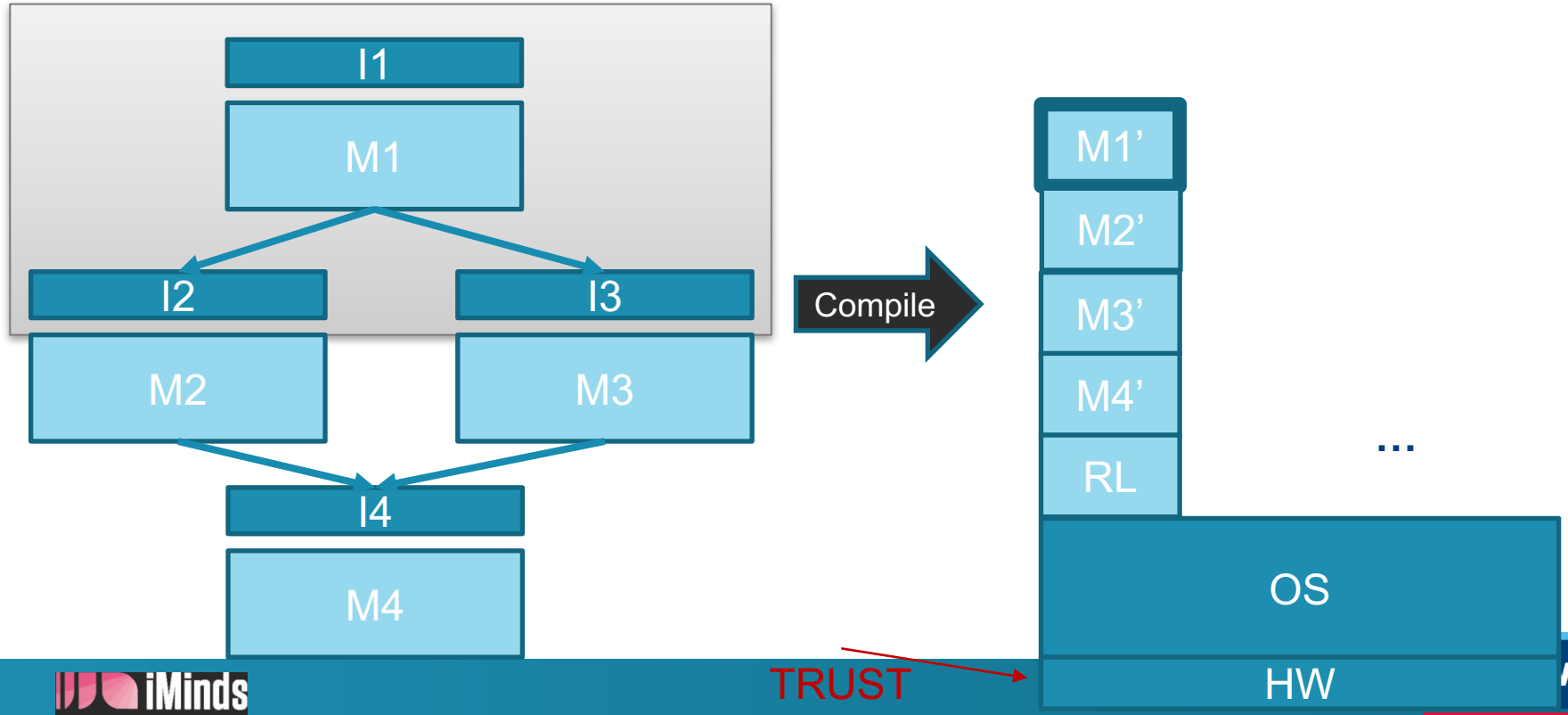


iMinds CONNECT.INNOVATE.CREATE
DISTRINET RESEARCH GROUP

Problem statement

Focus today only on:

- **The creation and attestation of isolated / protected modules within a legacy system**
  - Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, Frank Piessens, *Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base*, USENIX Security 2013



iMinds CONNECT.INNOVATE.CREATE
DISTRINET RESEARCH GROUP

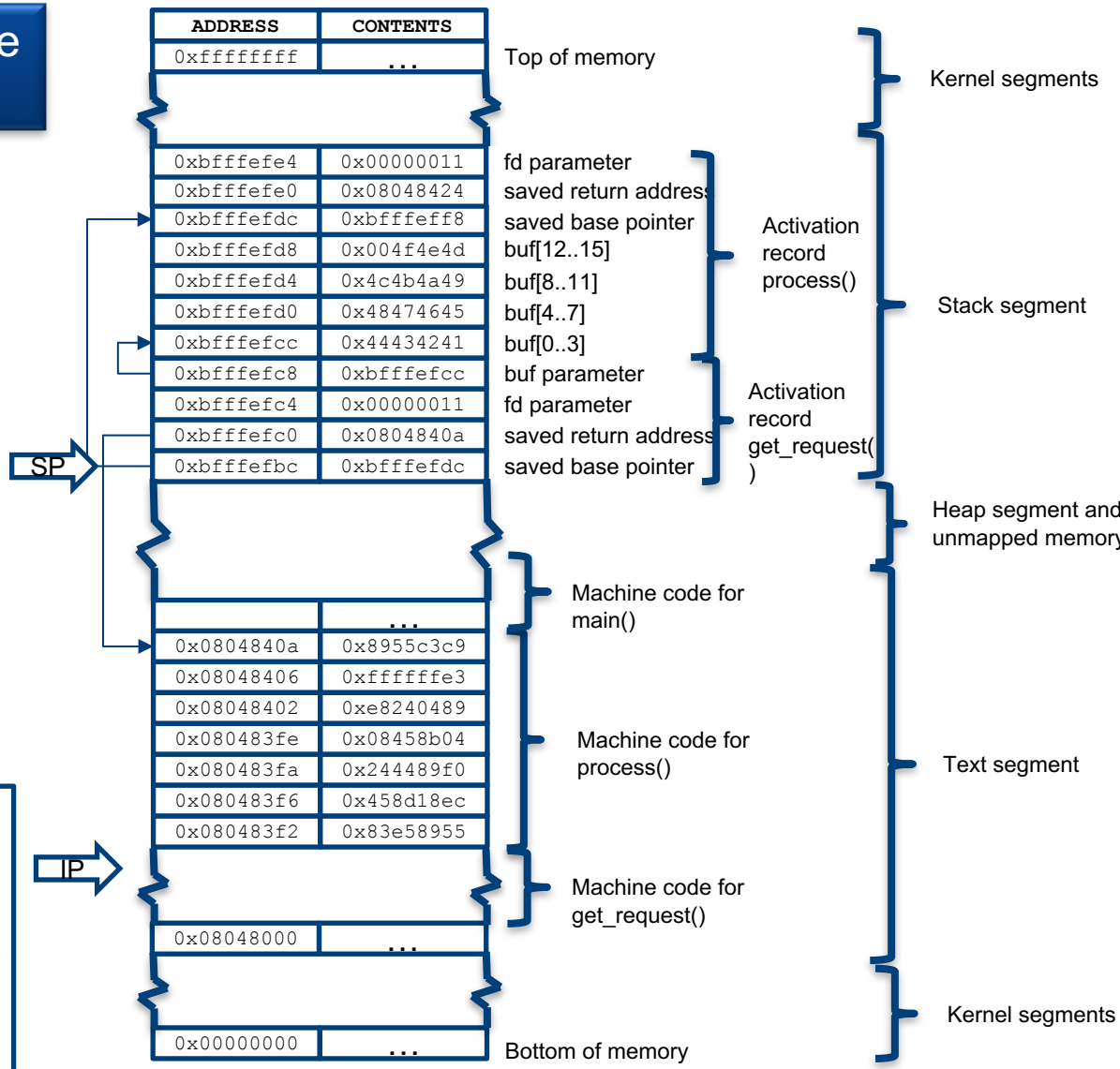
Remember the run-time machine state for executing C programs

```
void get_request(int fd, char buf[]) {
  read(fd,buf,16);
}

void process(int fd) {
  char buf[16];
  get_request(fd,buf);
  // Process the request (code not shown)
}
void main() {
  int fd ;
  // Initialize server, wait for a connection
  // Accept connection, with file descriptor fd
  // Finally, process the request:
  process(fd);
}
```

(a) Program source code

```
55              push   %ebp             ; save base pointer
89 e5           mov    %esp,%ebp        ; set new base pointer
83 ec 18        sub    $0x18,%esp       ; allocate stack record
8d 45 f0        lea    -0x10(%ebp),%eax ; put buf in %eax
89 44 24 04     mov    %eax,0x4(%esp)   ; and push on the stack
8b 45 08        mov    0x8(%ebp),%eax   ; put fd parameter in %eax
89 04 24        mov    %eax,(%esp)      ; and push on the stack
e8 e3 ff ff ff  call   0x80483ed        ; call get_request
c9              leave                   ; deallocate stack frame
c3              ret                     ; return
```

(b) Machine code for process() function



(c) Run-time machine state on entering get_request()

Using PMA's against memory scraping

secret.h

```
int get_secret(int provided_pin)
```

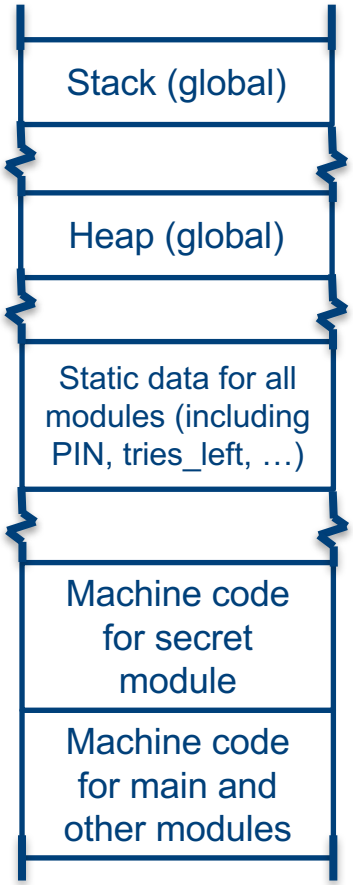
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
      tries_left = 3;
    return secret;}
  else { tries_left-- ; return 0; }; }
else return 0; }
```

(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
…
}
```

(b) Other modules of the program



(c) Run-time memory contents

Using PMA's against memory scraping

secret.h

```
int get_secret(int provided_pin)
```

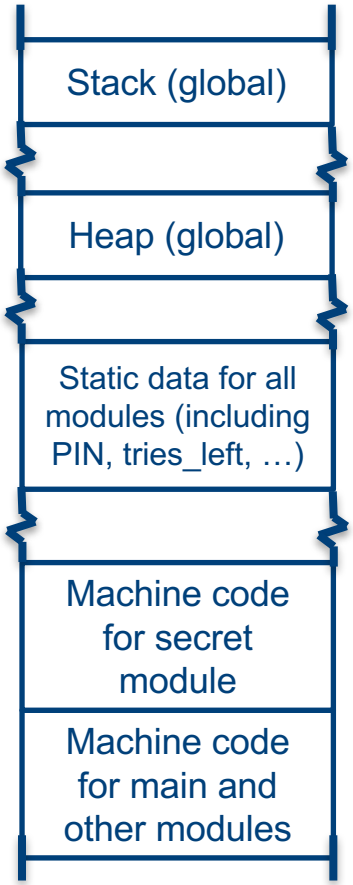
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
      tries_left = 3;
    return secret;}
  else { tries_left-- ; return 0; }; }
else return 0; }
```

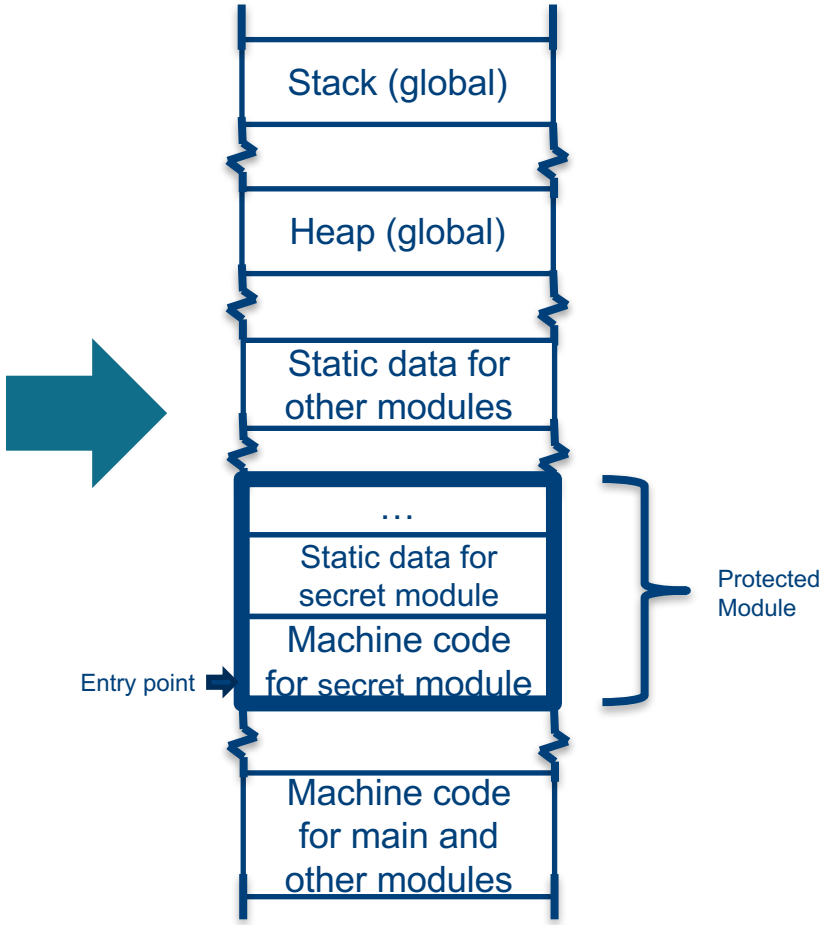
(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
…
}
```

(b) Other modules of the program

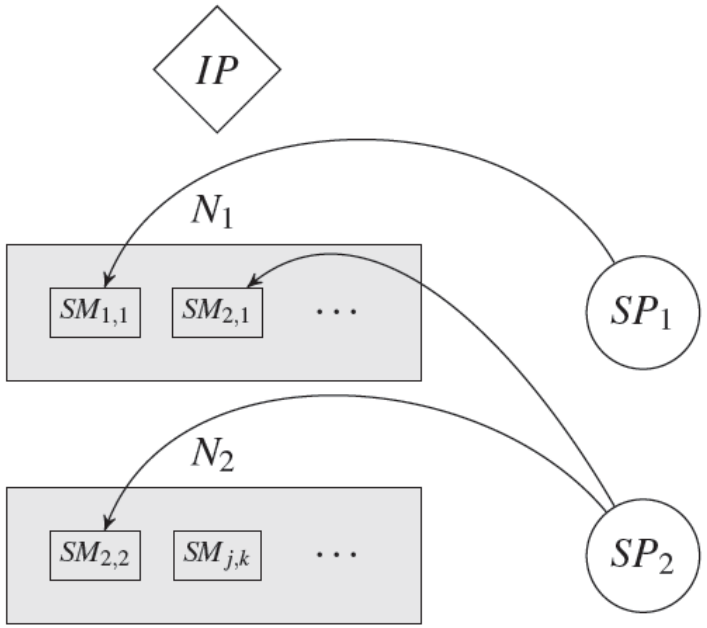


(c) Run-time memory contents



System model

- A network of low-end nodes N managed by an infrastructure provider IP
- Software providers SP deploy software modules SM on these nodes



iMinds
CONNECT.INNOVATE.CREATE

Attacker model and security properties

- Attackers can:
  - Manipulate **all** the SW on nodes
  - Control the network as a Dolev-Yao attacker
  - NOT mess with the hardware
- In the presence of such attackers we guarantee:
  - Software module isolation
  - Remote attestation
  - Secure remote communication
  - [Secure linking]

iMinds CONNECT.INNOVATE.CREATE
KU LEUVEN DISTRINET RESEARCH GROUP

Protected software modules

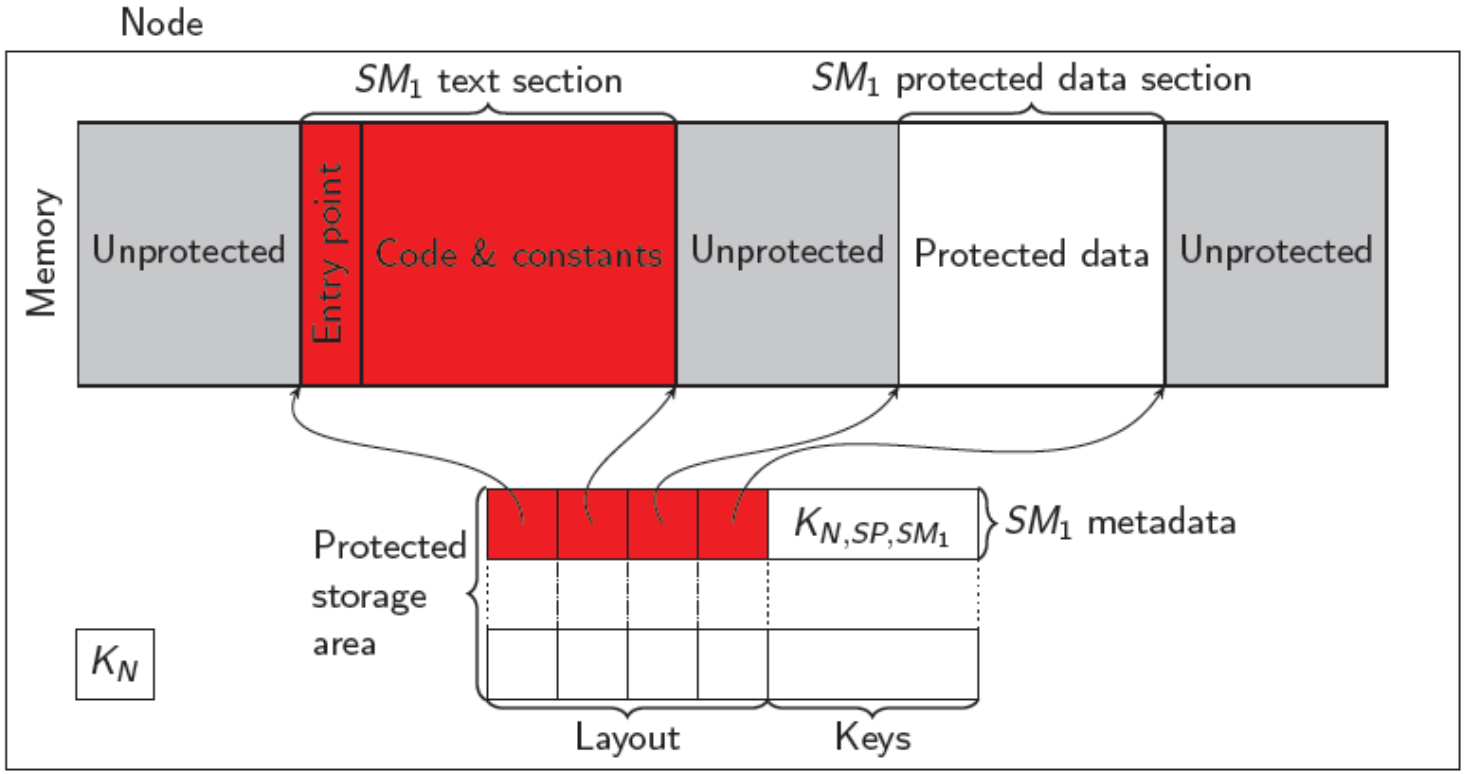
- Standard SW modules, defining memory sections
  - Public text section
    - Code and constants
  - Private data section
    - Runtime data that needs to be protected
  - Optional unprotected sections
- **Layout** of a module:
  - The load addresses of public and private sections
- **Identity** of a module:
  - Layout + contents of text section

iMinds CONNECT.INNOVATE.CREATE
KU LEUVEN DISTRINET RESEARCH GROUP

Isolation

- By PC-based access control:

from \ to	*Protected*			*Unprotected*
	Entry point	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x



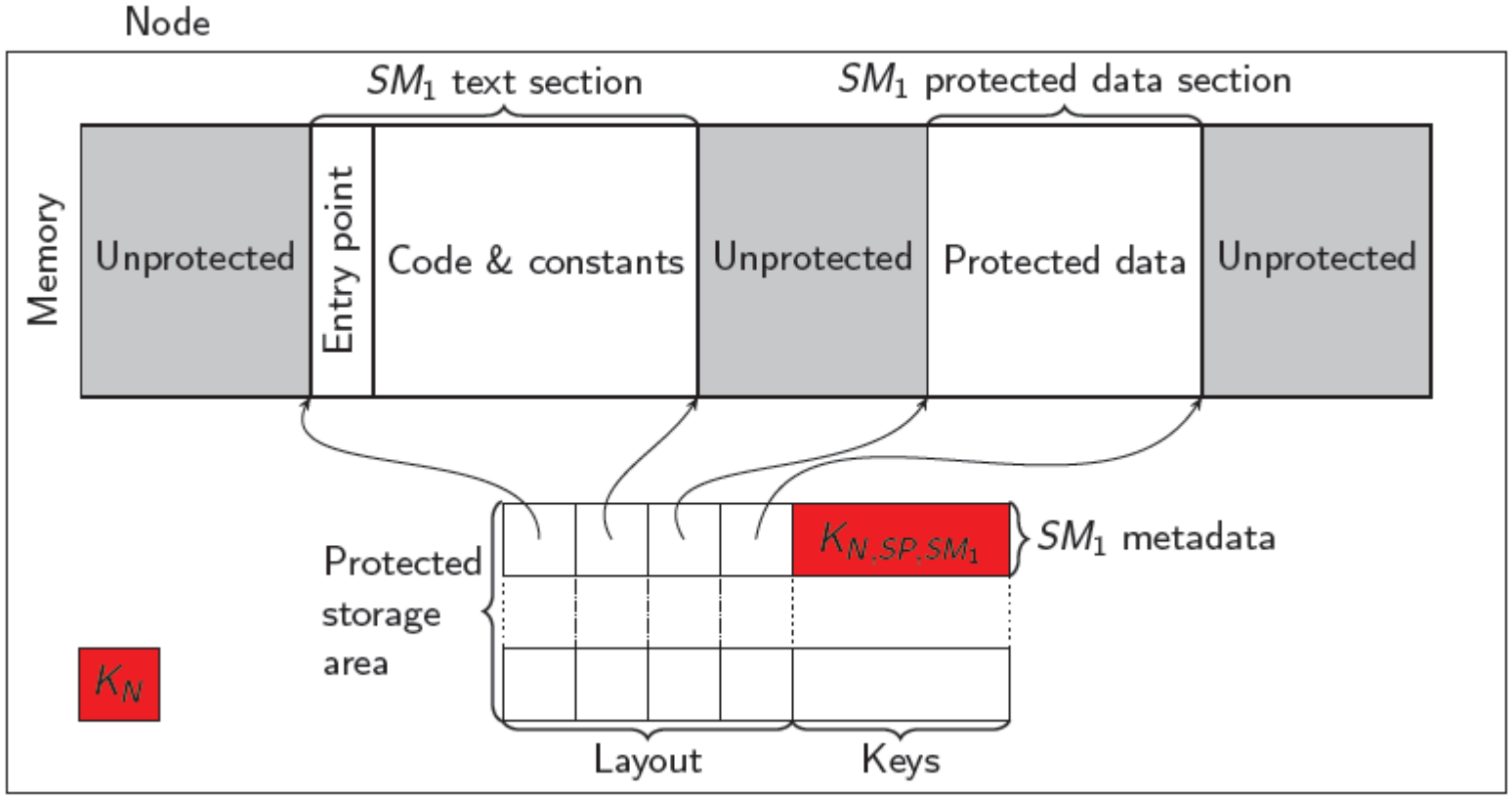
iMinds CONNECT.INNOVATE.CREATE
KU LEUVEN DISTRINET RESEARCH GROUP

Key management

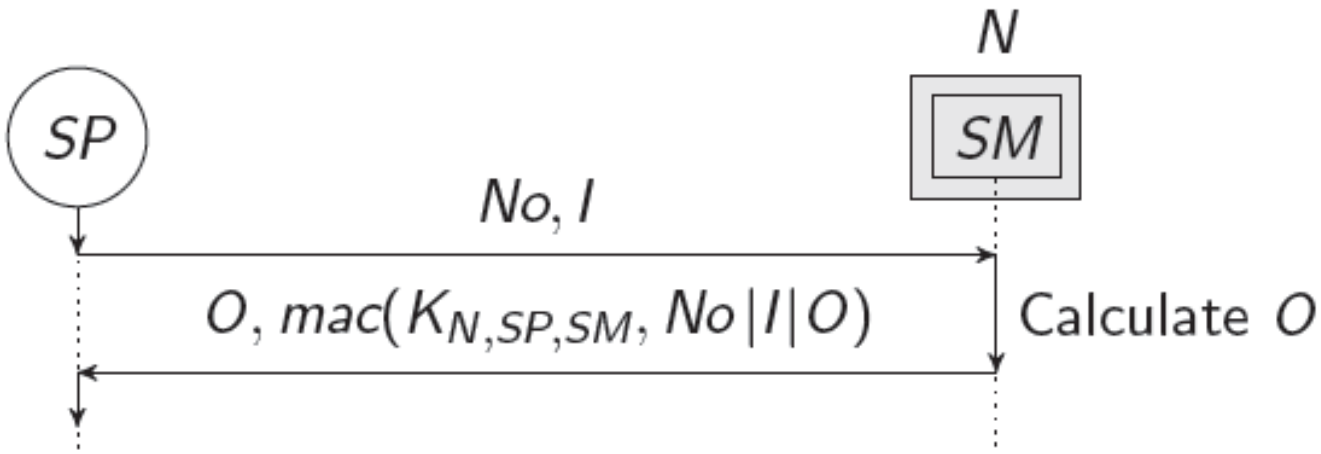
- Strictly symmetric key for performance reasons
- Three types of keys:
  - Node master keys K_N : shared between IP and N
  - Provider keys $K_{N,SP}$: shared between IP, SP and N
  - Module keys $K_{N,SP,SM}$: shared between IP, SP and SM on N
- Nodes are initialized with their master key on production
- All other keys are derived by means of key derivation functions
  - $K_{N,SP} = kdf(K_N , SP)$
  - $K_{N,SP,SM} = kdf(K_{N,SP} , SM)$

Keys on the device managed by HW

- Only computed after enabling isolation
  - **protect** *layout*, SP
- Only usable through special HW instructions
  - **mac-seal** start-address, length, result-address



Remote attestation and secure communication



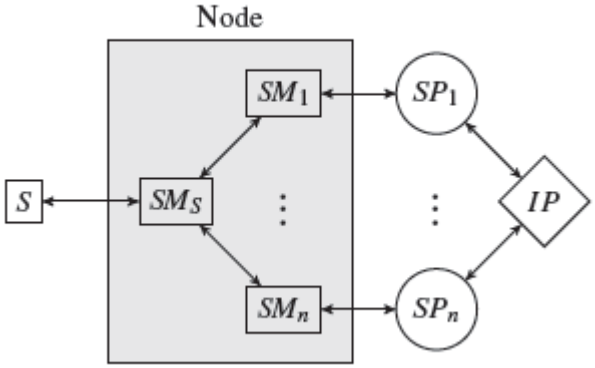
MAC is calculated by a `mac-seal` instruction

Using the key of the calling *SM*

MAC can be recalculated by *SP*...

He knows the *correct* $K_{N,SP,SM}$

An example (simplified) scenario



- Node manages a sensor S by means of an IP provided module SM_S
- Various SP's can install SM's:
  1. The SP contacts IP to get a $K_{N,SP}$
  2. SP creates SM, and calculates $K_{N,SP,SM}$
  3. SM is deployed on N using untrusted OS services
  4. SM is protected with the instruction:
     - **protect** *layout*, SP
     - This creates $K_{N,SP,SM}$ and enables memory protection on SM
  5. SP sends a request to SM (including a nonce No)
  6. SM computes a response (possibly calling SM_S and including No) and signs it using the instruction:
     - **MAC-seal**
     - This creates a MAC of the response using $K_{N,SP,SM}$

Some implementation details

- Built as an extension of an open-source MSP430 implementation
- Main changes:
  - Memory access logic that implements PC-based access control
  - Hardware implementations of an authenticated encryption primitive
  - The new instructions
- Available for download at:
  - https://distrinet.cs.kuleuven.be/software/sancus/

Recap

- Sancus is a low-cost security architecture for networked embedded systems
  - Module isolation though program-counter based access control
  - Key management through a hierarchical symmetric-key ID-based key derivation
  - Remote attestation and secure communication by hardware-guarded access to keys
  - [Secure linking]
  - [A secure compiler supporting the development of modules]
- Intel's recent Skylake processors include a similar security architecture, called Software Guard eXtensions (Intel SGX)

Using PMA's against memory scraping

secret.h

```
int get_secret(int provided_pin)
```

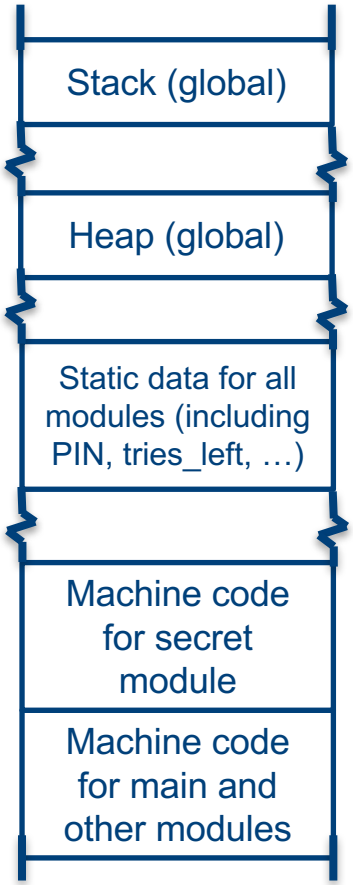
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
      tries_left = 3;
    return secret;}
  else { tries_left-- ; return 0; }; }
else return 0; }
```

(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
…
}
```

(b) Other modules of the program



(c) Run-time memory contents

Using PMA's against memory scraping

secret.h

```
int get_secret(int provided_pin)
```

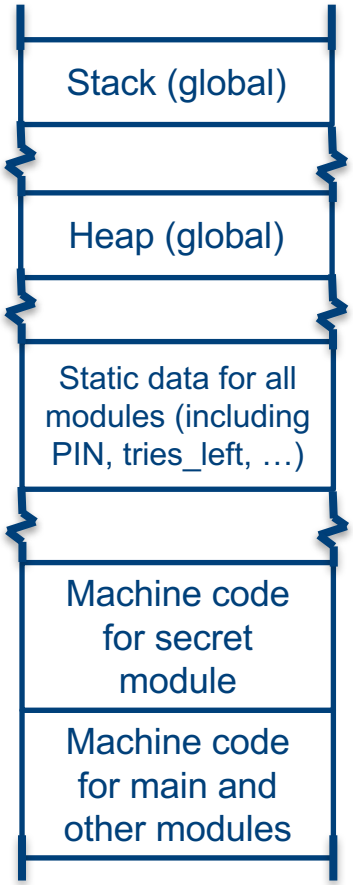
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
      tries_left = 3;
    return secret;}
  else { tries_left-- ; return 0; }; }
else return 0; }
```

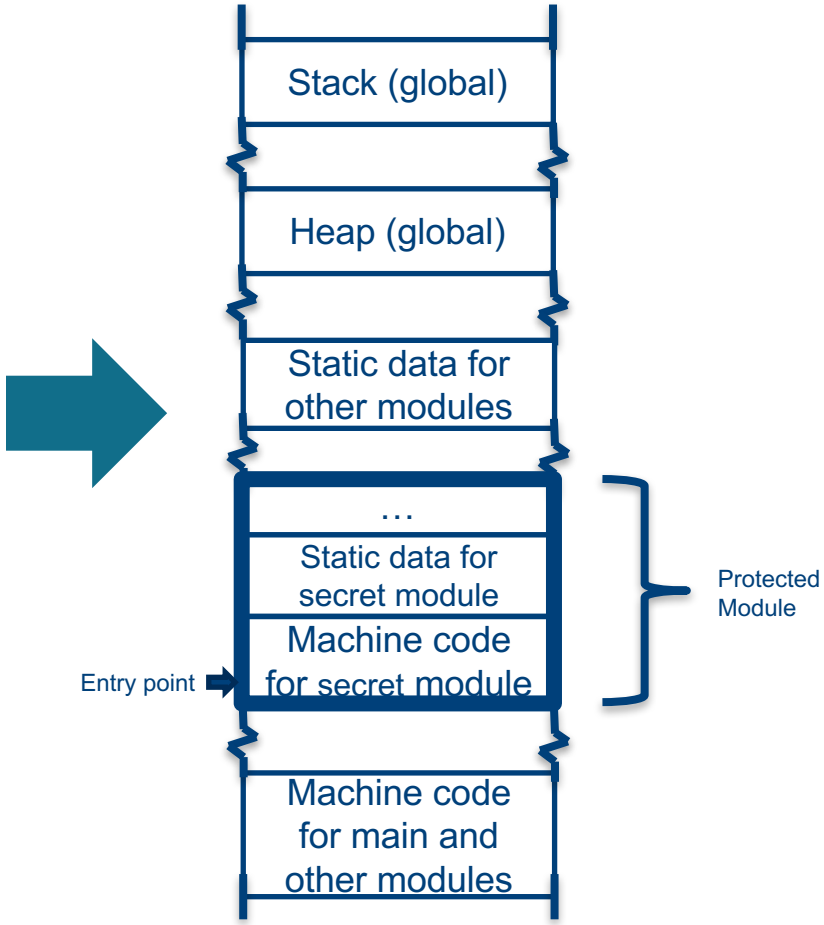
(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
…
}
```

(b) Other modules of the program



(c) Run-time memory contents



The need for secure compilation

secret.h

```
int get_secret(int get_pin())
```

secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int get_pin()) {
 if (tries_left > 0) {
   if (PIN == get_pin()) {
      tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

Conclusions

- Protected Module Architectures are a very promising new system security technology
  - They essentially allow the dynamic creation of Trusted Execution Environments (TEEs) within a legacy, untrusted infrastructure.
- But many interesting questions remain:
  - Secure compilation to such architectures
  - Providing secure persistent storage
  - Making sure attackers can not use them to hide malware
  - …

Overview

- Protected module architectures
  - Fine grained isolation at machine code level
  - Supported in the recent Intel Skylake processors under the name Intel Software Guard eXtentions (Intel SGX)
- Safe systems programming languages
  - Compiled languages with low-level control over memory, but with strong safety assurance
  - Supported in the Rust and Go programming languages
- Advanced compiler based countermeasures
  - Control-flow integrity (CFI)
  - Pointer-based checking

The trade-off between safety and low-level control

- From a security point of view, safe languages like Java, C#, Scala, … are **significantly** better
- Why has C not disappeared?
- There are several reasons for this:
  - C is very “light-weight”
    - Very good performance
    - But also: no need for a “runtime” or “virtual machine”
  - C gives the programmer control over low-level details
    - What is allocated on stack versus heap
    - How are data structures laid out in memory
- There are several new contenders in this arena

Documentation Community Downloads Contribute

Recommended Version:
1.6.0 (Windows installer)

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

Show me!

Install

Other Downloads

Featuring

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

Run

```
// This code is editable and runnable!
fn main() {
    // A simple integer calculator:
    // `+` or `-` means add or subtract by 1
    // `*` or `/` means multiply or divide by 2

    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
             program, accumulator);
}
```

Try Go

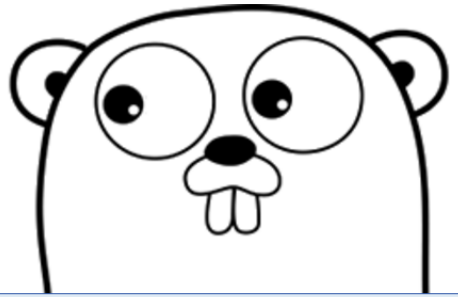
Pop-out

```
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

func main() {
        fmt.Println("Hello, 世界")
}
```

Hello, World! ▼

Run | Share | Tour

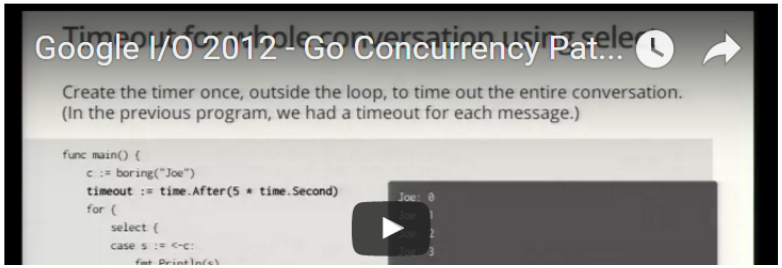
Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.



Download Go

Binary distributions available for
Linux, Mac OS X, Windows, and more.

Featured video



Featured articles

Participate in the 2016 Go User Survey and Company Questionnaire

The Go project wants to hear from you! Our goal is to create the best language for developing simple, reliable, scalable software. We are asking you to help by participating in a survey and if applicable, a company questionnaire.

Our focus

- Rust and Go have many interesting features
- But we focus on Rust's most innovative / most complex feature:
  - Ownership and borrowing
- This is an important new approach to avoiding temporal memory safety errors without garbage collection
- It also addresses important concurrency related errors, but we do not focus on this

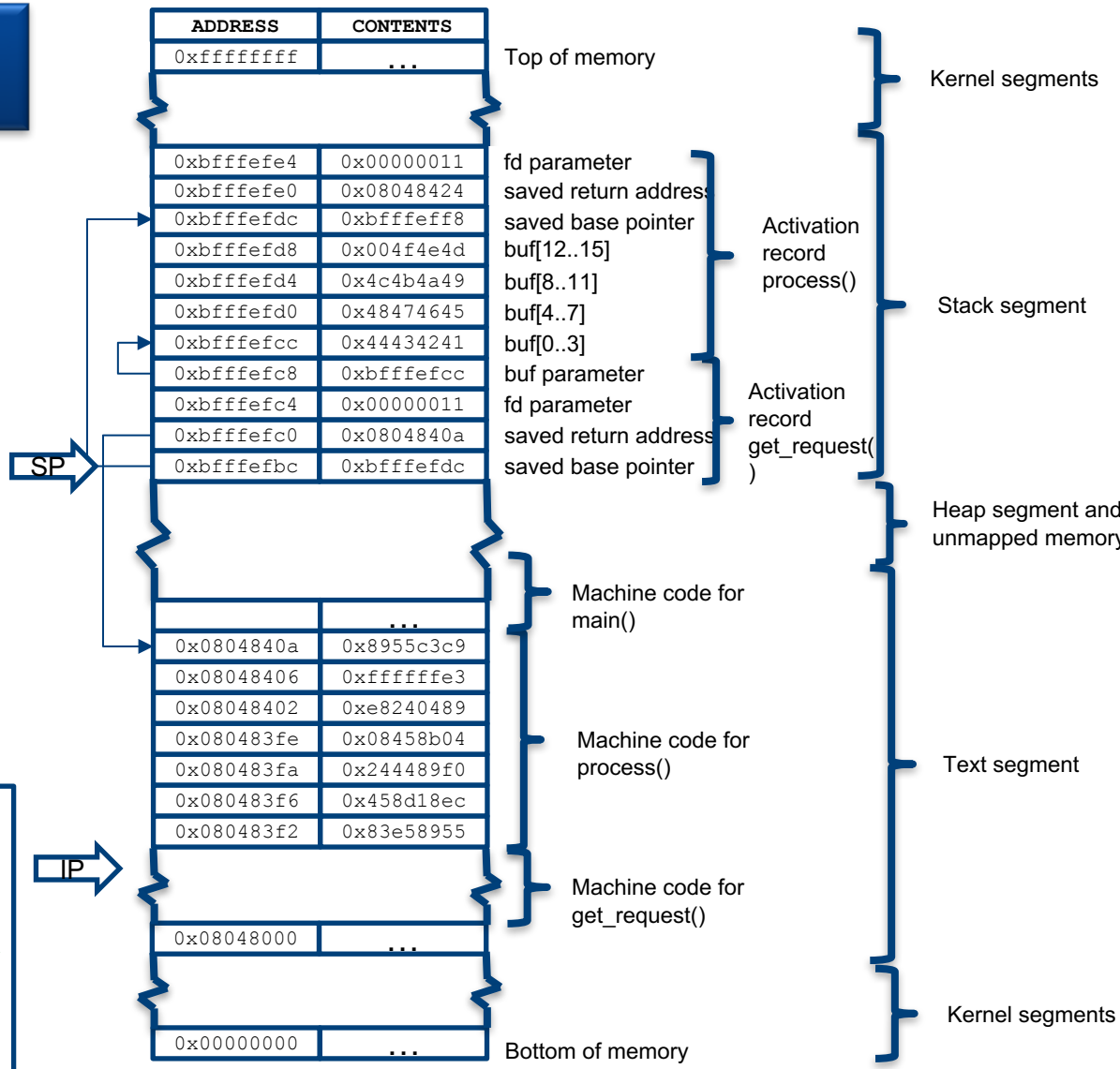
What part of memory should be writable by the program?

```
void get_request(int fd, char buf[]) {
  read(fd,buf,16);
}

void process(int fd) {
  char buf[16];
  get_request(fd,buf);
  // Process the request (code not shown)
}
void main() {
  int fd ;
  // Initialize server, wait for a connection
  // Accept connection, with file descriptor fd
  // Finally, process the request:
  process(fd);
}
```

(a) Program source code

```
55              push   %ebp             ; save base pointer
89 e5           mov    %esp,%ebp        ; set new base pointer
83 ec 18        sub    $0x18,%esp       ; allocate stack record
8d 45 f0        lea    -0x10(%ebp),%eax ; put buf in %eax
89 44 24 04     mov    %eax,0x4(%esp)   ; and push on the stack
8b 45 08        mov    0x8(%ebp),%eax   ; put fd parameter in %eax
89 04 24        mov    %eax,(%esp)      ; and push on the stack
e8 e3 ff ff ff  call   0x80483ed        ; call get_request
c9              leave                   ; deallocate stack frame
c3              ret                     ; return
```

(b) Machine code for process() function



(c) Run-time machine state on entering get_request()

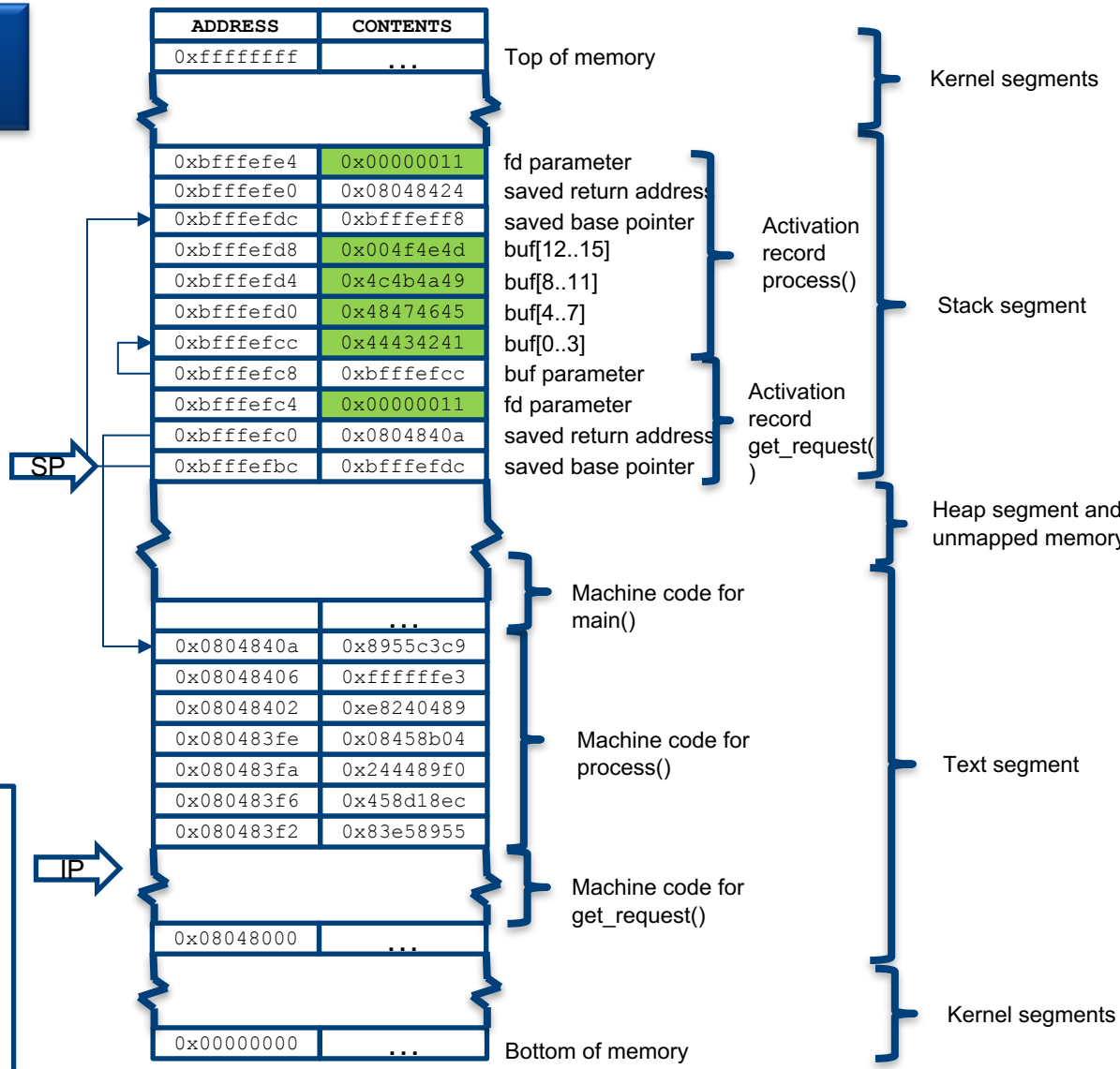
What part of memory should be writable by the program?

```
void get_request(int fd, char buf[]) {
  read(fd,buf,16);
}

void process(int fd) {
  char buf[16];
  get_request(fd,buf);
  // Process the request (code not shown)
}
void main() {
  int fd ;
  // Initialize server, wait for a connection
  // Accept connection, with file descriptor fd
  // Finally, process the request:
  process(fd);
}
```

(a) Program source code

```
55              push   %ebp             ; save base pointer
89 e5           mov    %esp,%ebp        ; set new base pointer
83 ec 18        sub    $0x18,%esp       ; allocate stack record
8d 45 f0        lea    -0x10(%ebp),%eax ; put buf in %eax
89 44 24 04     mov    %eax,0x4(%esp)   ; and push on the stack
8b 45 08        mov    0x8(%ebp),%eax   ; put fd parameter in %eax
89 04 24        mov    %eax,(%esp)      ; and push on the stack
e8 e3 ff ff ff  call   0x80483ed        ; call get_request
c9              leave                   ; deallocate stack frame
c3              ret                     ; return
```

(b) Machine code for process() function



(c) Run-time machine state on entering get_request()

Essentially, only 4 ways things can go wrong

- **Spatial memory safety errors**: a blob of allocated memory is accessed out of bounds
- **Temporal memory safety errors**: a blob of memory is accessed after it has been deallocated
- **Pointer forging**: creating an invalid pointer value
  - By invalid casts
  - By use of uninitialized memory
- **Unsafe primitive API functions**:
  - Like C's printf() function

Spatial memory safety

- Examples: indexing an array, indexing a struct, pointer arithmetic

```
void f1(int a[]) {
    a[5] = 10;
}

void f2(int *a) {
    *(a+5) = 10;
}
```

```
struct S {
    int x;
    int y; };

void f3(struct S p) {
    p.x = 20;
}
```

- How could the compiler protect against spatial memory safety errors?

Enforcing spatial memory safety

- Through type checking for structs and arrays with statically known bounds
  - E.g. Java type system will make sure that you can not access a non-existing field of an object
- Through run-time bounds checking otherwise
  - E.g. Java throws ArrayIndexOutOfBoundsException
  - E.g. “Fat” pointers in C or C++

Temporal memory safety

- How long are pointers valid?
  This depends on how the pointer is created.

```
int c;

int* f(int x) {
    int i;
    int *p1 = &c;
    int *p2 = malloc(sizeof(int));
    int *p3 = &x;
    int *p4 = &i;

    return p1; // or p2? or p3? or p4?
}
```

A simple example

```
typedef struct  {
    int len;
    int cap;
    int* data;
} vec;
```

```
vec newvec() {
    vec v;
    v.len = 0;
    v.cap = 2;
    v.data = malloc(2*sizeof(int));
    return v;
}
```

```
void push(vec* v, int i) {
    if (v->len >= v->cap) {
        v->cap *= 2;
        int *new = malloc(v->cap * sizeof(int));
        memcpy(new,v->data, v->len * sizeof(int));
        free(v->data);
        v->data = new;
    }
    v->data[v->len++] = i;
}
```

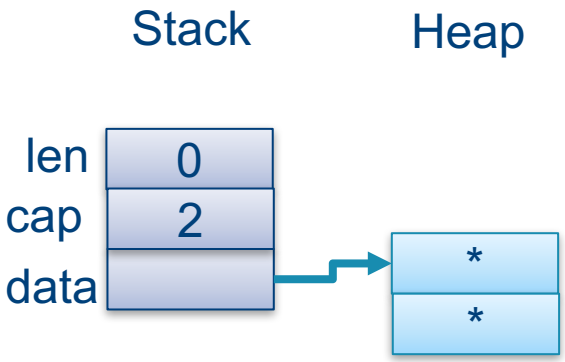
```
void printvec(vec v) {
    int i;
    for (int i = 0; i < v.len; i++) {
        printf("%d\n", v.data[i]);
    }
}
```

```
int* get(vec* v, int i) {
    return v->data + i;
}
```

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

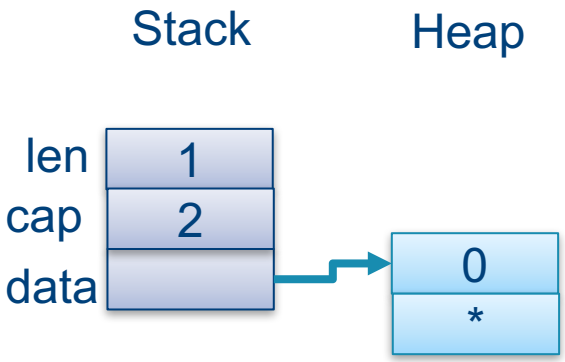
A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



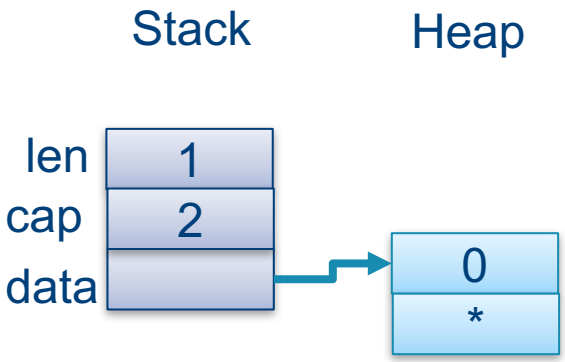
A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



A simple example

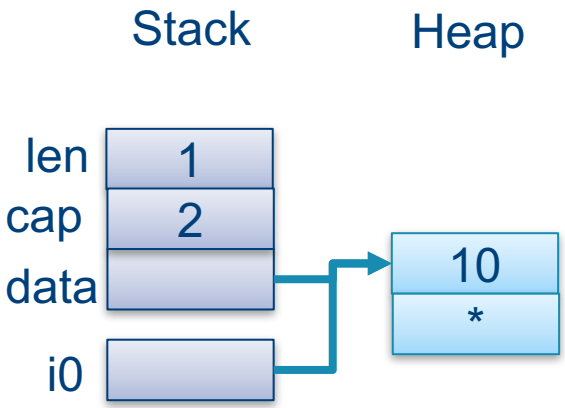
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



Output:
0

A simple example

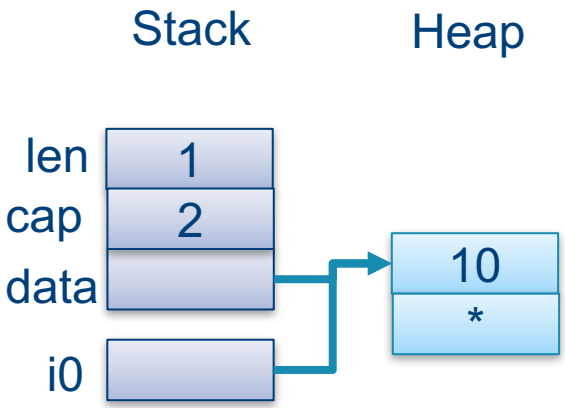
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



Output:
0

A simple example

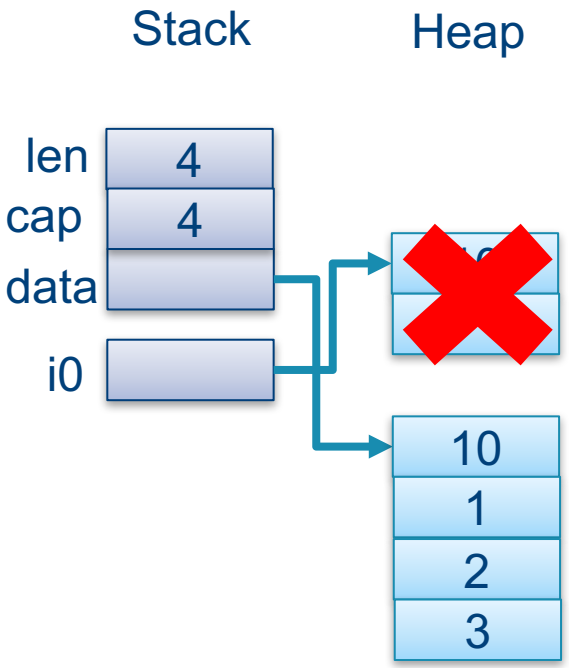
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



Output:
0
10

A simple example

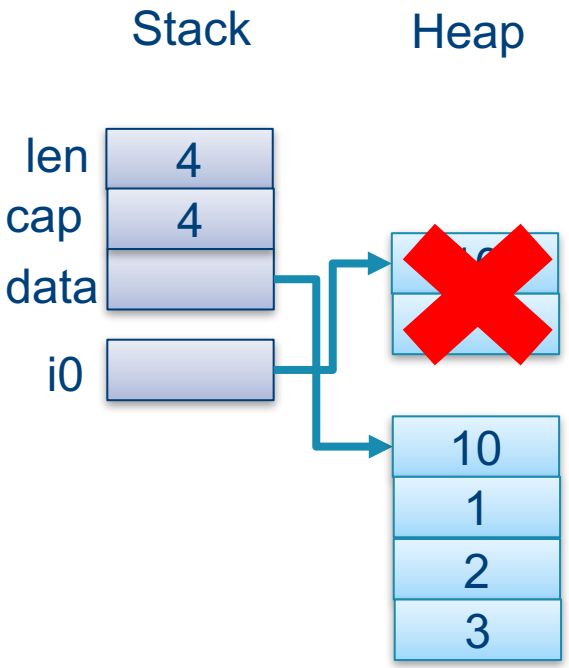
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



Output:
0
10

A simple example

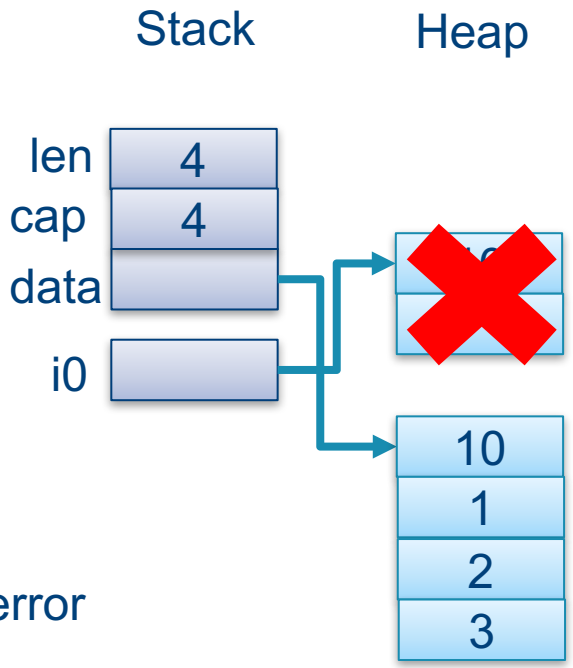
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



Output:
0
10
10
1
2
3

A simple example

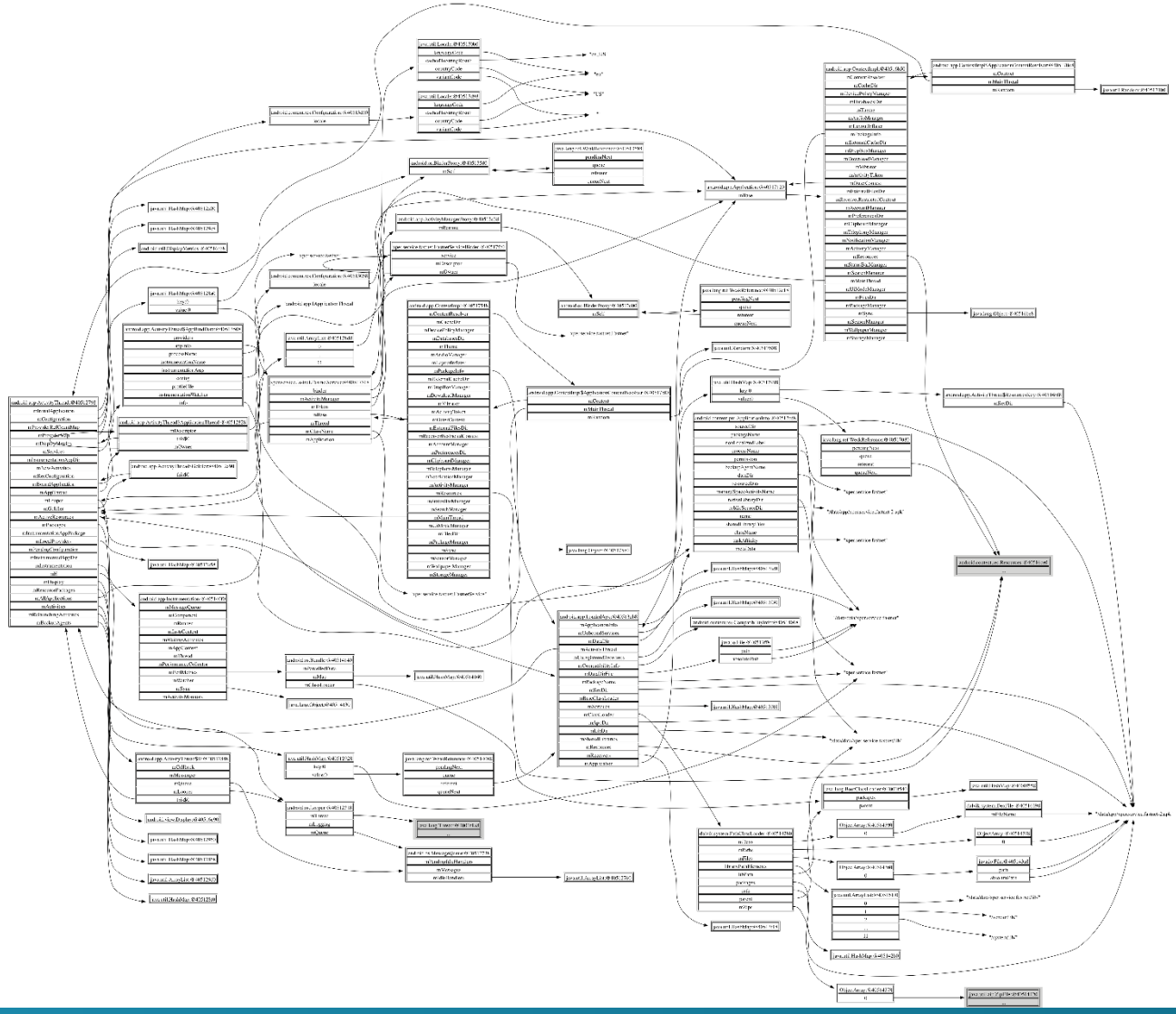
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



Output:
0
10
10
1
2
3

Temporal memory safety error

Real heap looks more complicated…



Enforcing temporal memory safety

- Allocate everything on the heap, and do **garbage collection**:
  - Programmer can not do explicit deallocation
    - I.e. no free()
  - At regular intervals, the program will be halted and the run-time system will clean up unused memory
    - Basic idea: check what memory is reachable from the current program state, and deallocate all the rest
    - Many different strategies to implement this with different pros and cons
- Important disadvantages for systems programming:
  - Less precise control over memory
  - Unpredictable timing

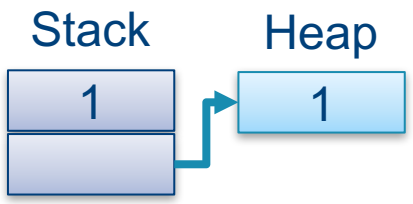
Enforcing temporal memory safety

- New approach: **ownership types** and **borrowing**
- Basic idea:
  - There is at all times a unique **owning** pointer to each allocated blob of memory
  - Memory is deallocated when the owning pointer disappears
    - Because it goes out of scope
    - Or because it is overwritten
    - Or because it was part of a data structure that is being deallocated
- We discuss the implementation of this idea in **Rust**

Memory management in Rust

- Programmer controls:
  - At what time memory is allocated
  - And where it is allocated (stack / heap)
- Deallocated when owner goes out of scope

```
fn main() {
    let x = 1; // allocated on the stack
    let y = Box::new(1); // allocated on the heap
}
```



No use after free is possible

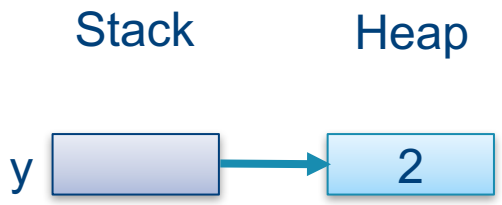
- There was only a single pointer, and it has gone out of scope

```
fn main() {

 {
 let x = Box::new(1);    // alloc x
 println!("x = {}", *x);
 }                       // free x
// ERROR: println!("x = {}", *x);
}
```

Move semantics

- Pointers are not copied but **moved**

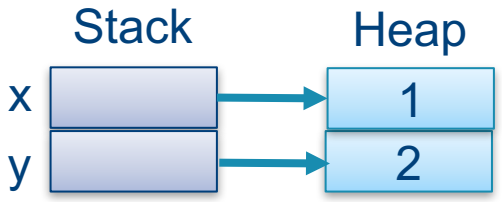
```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```



Move semantics

- Pointers are not copied but **moved**

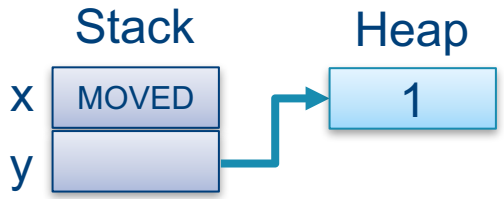
```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```



Move semantics

- Pointers are not copied but **moved**

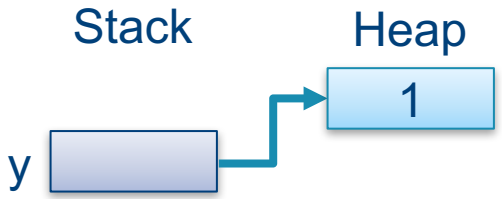
```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```



Move semantics

- Pointers are not copied but **moved**
  - Hence: there is always a **unique** owning pointer

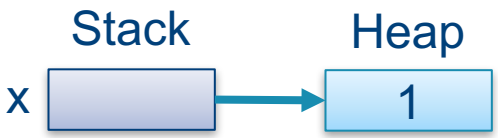
```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```



Pointers move into functions too

- Ownership moves from argument to formal parameter
- So when is the allocated memory freed in the program below?

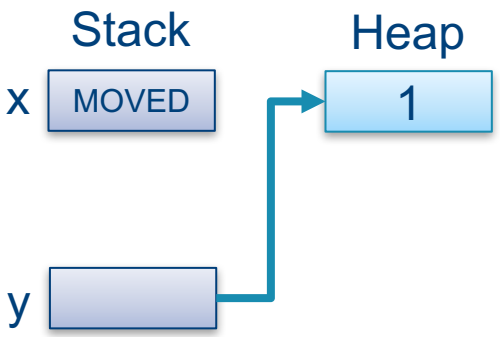
```
fn main() {
    let x = Box::new(1);
    println!("x = {}", *x);
    f(x);
    // ERROR: println!("x = {}", *x);
}

fn f(y : Box<i32>) {
    println!("y = {}", *y);
}
```



Pointers move into functions too

- Ownership moves from argument to formal parameter
- So when is the allocated memory freed in the program below?

```
fn main() {
    let x = Box::new(1);
    println!("x = {}", *x);
    f(x);
    // ERROR: println!("x = {}", *x);
}

fn f(y : Box<i32>) {
    println!("y = {}", *y);
}
```



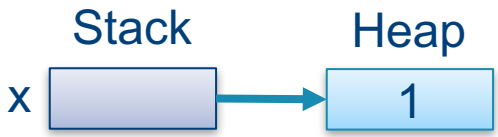
Pointers move into functions too

- Ownership moves from argument to formal parameter
- So when is the allocated memory freed in the program below?

```
fn main() {
    let x = Box::new(1);
    println!("x = {}", *x);
    f(x);
    // ERROR: println!("x = {}", *x);
}

fn f(y : Box<i32>) {
    println!("y = {}", *y);
}
```

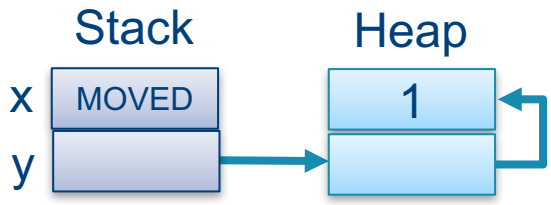
Pointers can also move into Boxes and structs

```
fn main() {
  let mut x = Box::new(1);
  let mut y = Box::new(x);
  let mut z = Box::new(y);
  println!("Val:{}",***z);
}
```



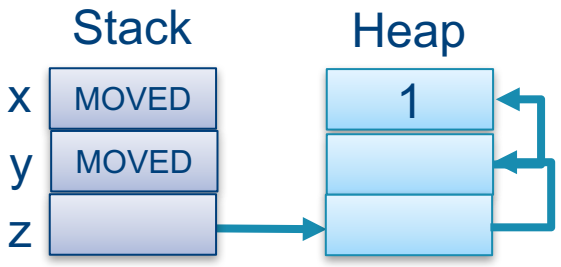
Pointers can also move into Boxes and structs

```
fn main() {
  let mut x = Box::new(1);
  let mut y = Box::new(x);
  let mut z = Box::new(y);
  println!("Val:{}",***z);
}
```



Pointers can also move into Boxes and structs

```
fn main() {
    let mut x = Box::new(1);
    let mut y = Box::new(x);
    let mut z = Box::new(y);
    println!("Val:{}",***z);
}
```



Moving into a box can extend life

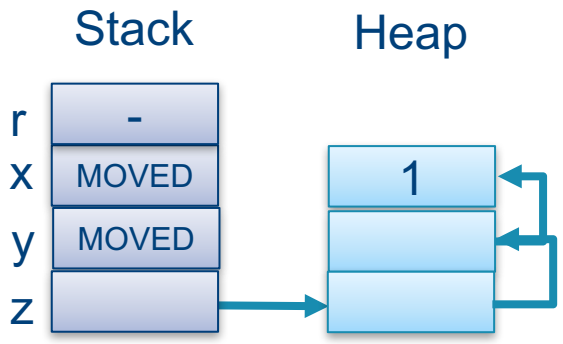
```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```

Moving into a box can extend life

```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```

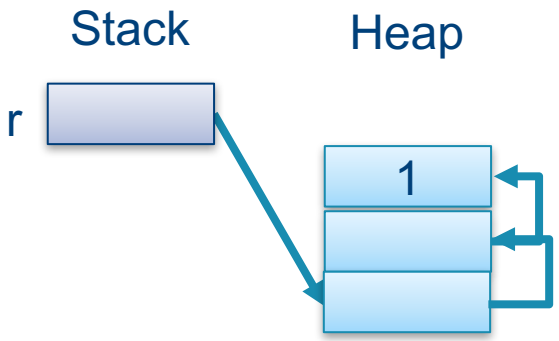
Moving into a box can extend life

```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```



Moving into a box can extend life

```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```



Enforcing unique ownership simplifies the heap

- The heap is a forest (set of trees), with allocated blobs of memory as nodes, and owning references as arrows.
- Roots of the trees are on the stack:
  - local variables of Box type
- If a local variable goes out of scope, that tree gets deallocated
  - We know that there are no other owners, because of uniqueness of ownership
- Uniqueness of ownership is maintained with the move semantics of pointers

Borrowing

- Move semantics is sometimes too limiting / annoying

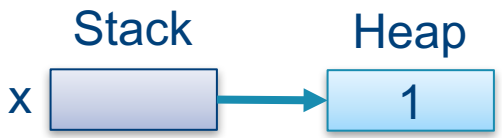
```
fn main() {
  let mut x = Box::new(1);
  print(x);
  *x = 2;
  print(x);
}

fn print(y:  Box<i32>) {
  println!("Value:{}", *y);
}
```

ERROR

- Rust supports “borrowing” of references to address this

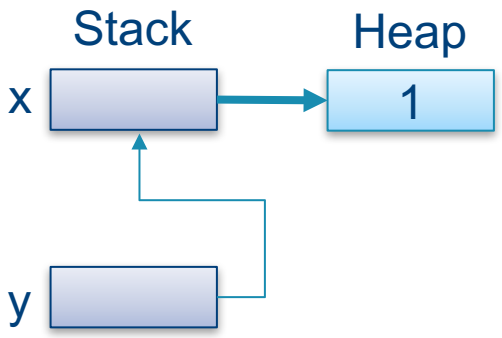
Borrowing

```
fn main() {
 let mut x = Box::new(1);
 print(&x);
 *x = 2;
 print(&x);
 }

fn print(y: &Box<i32>) {
 println!("Value: {}", **y);
}
```



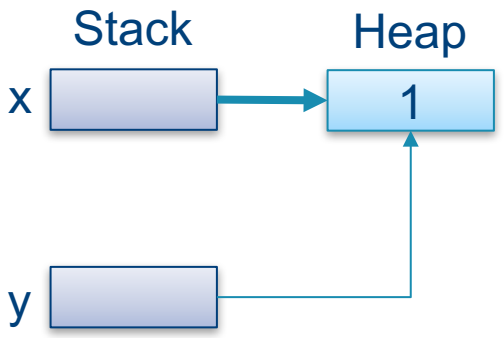
Borrowing

```
fn main() {
 let mut x = Box::new(1);
 print(&x);
 *x = 2;
 print(&x);
 }

fn print(y: &Box<i32>) {
 println!("Value: {}", **y);
}
```



Borrowing

```
fn main() {
  let mut x = Box::new(1);
  print(& *x);
  *x = 2;
  print(& *x);
}

fn print(y:  &i32) {
  println!("Value:{}", *y);
}
```



Borrowing rules

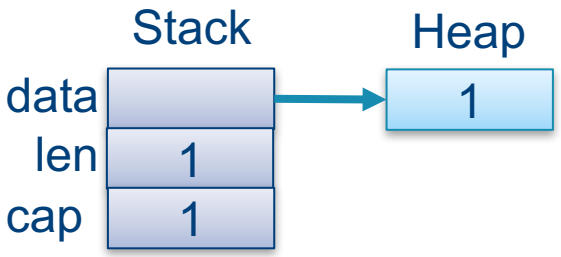
- To avoid introducing temporal safety errors, borrowing and ownership follow some rules:
  - The *lifetime* of a borrow should always be included in the lifetime of the owner from which it is borrowed
    - Otherwise, if the owner dies, the borrowed reference would be dangling

```
fn main() {
  let mut x = Box::new(1);
  let mut y = &x;
  {
   let mut z = Box::new(2);
   y = &z;
   }
}
```

```
6:9: 6:10 error: `z` does not live long enough
6     y = &z;
            ^
```

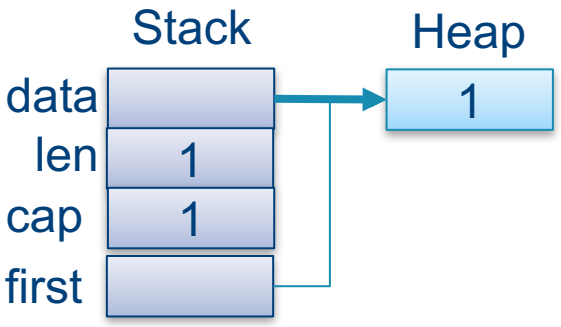
Borrowing should also forbid mutation

```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```



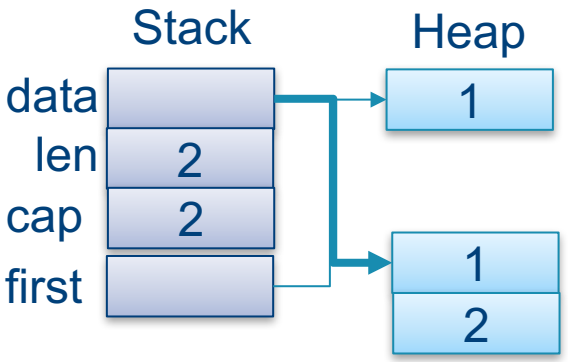
Borrowing should also forbid mutation

```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```



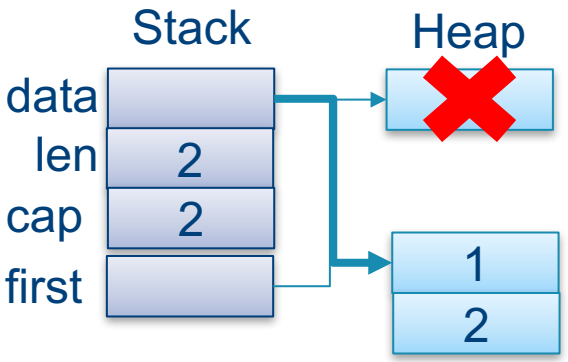
Borrowing should also forbid mutation

```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```



Borrowing should also forbid mutation

```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```



Borrowing rules

- Rust supports borrowing:
  - Either: an arbitrary number of immutable references
  - Or: a single mutable reference
- To ensure safety, Rust ensures:
  - Modification through the owner is disallowed while borrows are outstanding
  - Lifetimes of borrowed references are always strictly included in the lifetime of the owner

Summary: Ownership and borrowing

- Together these concepts:
  - Can guarantee temporal memory safety statically
    - By ruling out simultaneous aliasing + mutation
  - Allow relatively flexible pointer manipulating programs
- Many advantages:
  - No need for a run-time (no garbage collection)
  - Also helps in avoiding data races (concurrency errors)
- Some disadvantages:
  - Non-trivial to use
  - Not as flexible as C

The Rust programming language

- Is one of the fastest growing languages at the moment
- Since Firefox 48 (August 2016), there is Rust code in Firefox
- The language has many other interesting features that we did not discuss
  - Pattern matching
  - Traits
  - Generics
  - …
- See:
  - https://www.rust-lang.org/

Comparison

- Java/C#/JavaScript/…
  - Runtime = virtual machine + JIT compiler + GC + …
  - Garbage collection can induce substantial latency
    - “Stop-the-world”
- Go
  - Runtime = GC
  - Low-latency garbage collection
    - Focus on GC algorithms that can keep the program running
- Rust
  - “Runtime” = just a set of libraries

Overview

- Protected module architectures
  - Fine grained isolation at machine code level
  - Supported in the recent Intel Skylake processors under the name Intel Software Guard eXtentions (Intel SGX)
- Safe systems programming languages
  - Compiled languages with low-level control over memory, but with strong safety assurance
  - Supported in the Rust and Go programming languages
- Advanced compiler based countermeasures
  - Control-flow integrity (CFI)
  - Pointer-based checking

Control-flow integrity

- Most low-level attacks break the control flow as it is encoded in the source program
  - E.g. At the source code level, one always expects a function to return to its call site
- The idea of control-flow integrity is to instrument the code to check the “sanity” of the control-flow at runtime

Remember the heap-based buffer overflow

- Example vulnerable program:

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

Example CFI at the source level

- The following code explicitly checks whether the cmp function pointer points to one of two known functions:

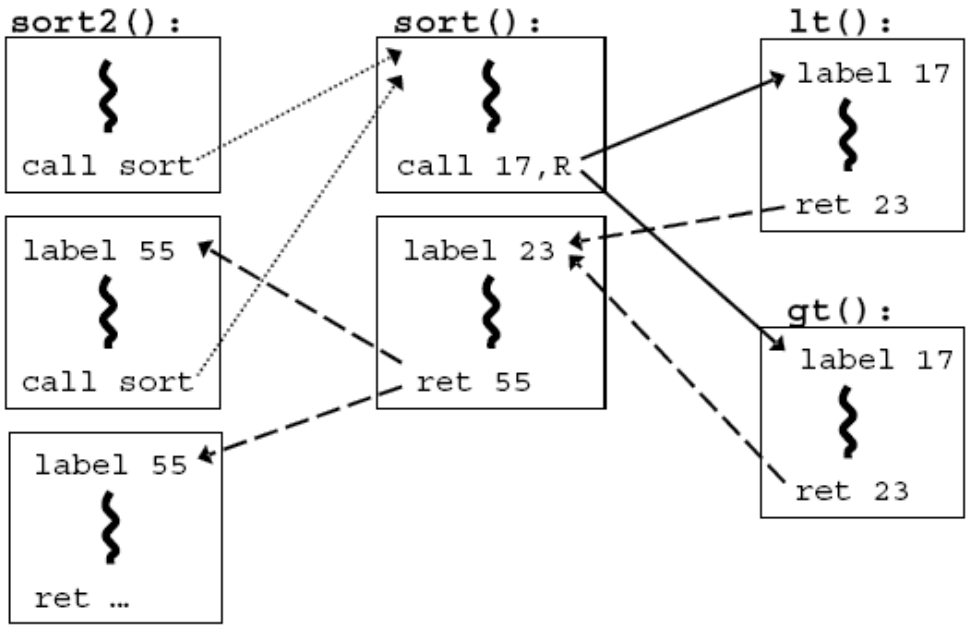
```
int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // ... elided code ...
    if( (s->cmp == strcmp) || (s->cmp == stricmp) ) {
        return s->cmp( s->buff, "file://foobar" );
    } else {
        return report_memory_corruption_error();
    }
}
```

General CFI

- In general, similar sanity checks can be done on any *computed control flow transfer*
  - Mainly: calls through function pointers, and returns
- The challenge is to do this
  - Efficiently
  - And precisely
- The original CFI determined a Control Flow Graph of the program, and then inserted *label-based checks*

Example CFI with labels

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Overview

- Countermeasures of the future:
  - Protected module architectures
    - Fine grained isolation at machine code level
    - Supported in the most recent Intel Skylake processors under the name Intel Software Guard eXtentions (Intel SGX)
  - Safe systems programming languages
    - Compiled languages with low-level control over memory, but with strong safety assurance
    - Supported in the Rust and Go programming languages
  - Advanced compiler based countermeasures
    - Control-flow integrity (CFI)
    - Pointer-based checking

Pointer-based checking for C

- Challenging, because:
  - For compatibility reasons, you should not change the size of a pointer (so no **fat** pointers)
  - Performance overhead should be low
- The most promising approach uses metadata about pointers maintained in a disjoint metadata space
- For a detailed discussion, see:
  - Santosh Nagarakatte, Milo M. K. Martin, Steve Zdancewic: Everything You Want to Know About Pointer-Based Checking. SNAPL 2015

How does it work?

- For each pointer (i.e. each memory address), we maintain **metadata** at run-time in a separate area of memory, e.g.:
  - **Base** and **bound** information: what is the size of the memory blob that this pointer is valid for?
  - **Lock** and **key** information to detect temporal safety issues
- Intel Memory Protection Extensions (Intel MPX) provides hardware support for maintaining such metadata
  - Currently only base and bound

(a) Memory Allocations

```
p = malloc(size);
```

```
p_key = next_key++;
p_lock = allocate_lock();
*(p_lock) = p_key;
p_ base = p;
p_bound = p != 0 ? p+size: 0;
```

(b) Memory Deallocations

```
free(p);
```

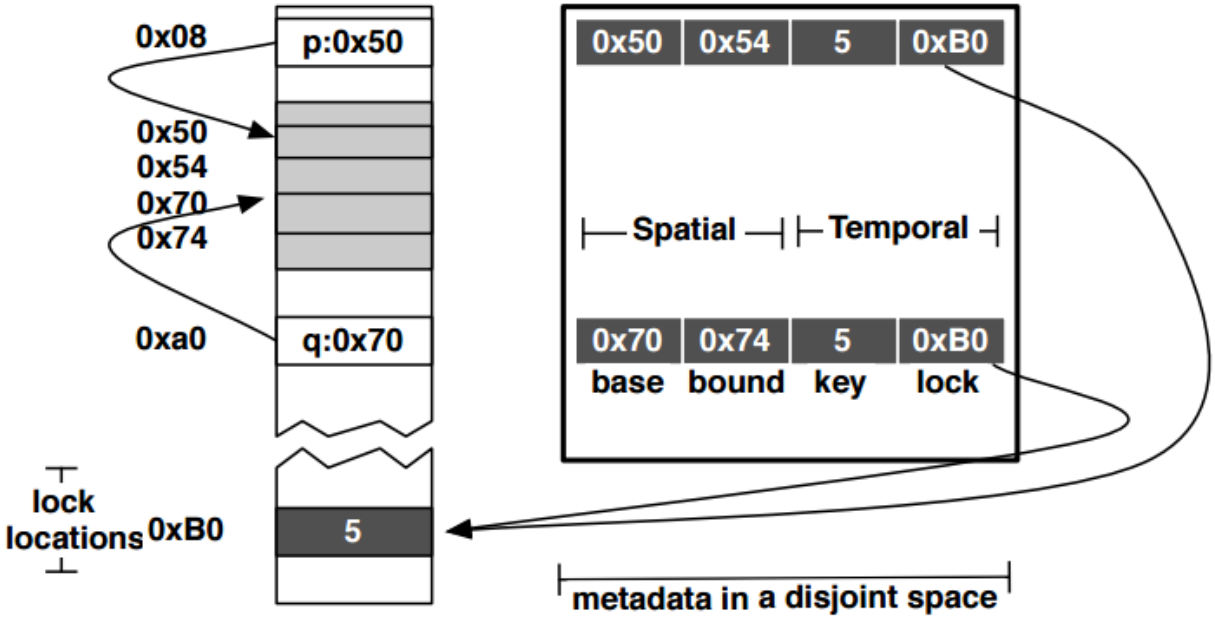
```
*(p_lock) = INVALID;
deallocate_lock(p_lock);
```

(c) Temporal Check

```
tcheck(p_key, p_lock) {
  if (p_key != *(p_lock))
     raise exception();
}
```

(d) Spatial Check

```
scheck(p, p_base, p_bound, size) {
  if (p < p_base ||
       p + size >= p_bound)
     raise exception();
}
```



iMinds
CONNECT.INNOVATE.CREATE

Performance costs

- Software-only implementations:
  - From a few percent up to 250% execution time overhead
- Hardware-supported implementations:
  - Approximately 20% execution time overhead

Conclusions

- Vulnerabilities in infrastructural systems software (operating systems, servers, middleware) have been an important concern for security for decades
- Memory safety related vulnerabilities are one of the most important categories of vulnerabilities in systems software
- Decades of research are resulting in some interesting new approaches to:
  - Protect application modules from infrastructural software
  - Prevent memory safety vulnerabilities through safe systems programming language
  - Comprehensively detect triggering of memory safety vulnerabilities at run-time for C with reasonable performance
- If you are interested in following these developments more closely, come talk to me about possible collaborations!

References

- J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, F. Piessens, **Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base**, USENIX Security 2013
- P. Agten, R. Strackx, B. Jacobs, F. Piessens, Secure compilation to modern processors, IEEE 25th Computer Security Foundations Symposium (CSF 2012),
- https://software.intel.com/en-us/isa-extensions/intel-sgx
- https://www.rust-lang.org/
- https://golang.org/
- U. Erlingsson, Y. Younan, F. Piessens, Low-level software security by example, Handbook of Information and Communication Security, 2010
- S. Nagarakatte, M. M. K. Martin, S. Zdancewic: Everything You Want to Know About Pointer-Based Checking. SNAPL 2015