Rust, Memory-Safety, and Beyond

Mingshen Sun | Baidu X-Lab, USA

GoSSIP Summer School

July 2018

Keynote Live

https://bit.ly/2096QkX

This is O (OK).

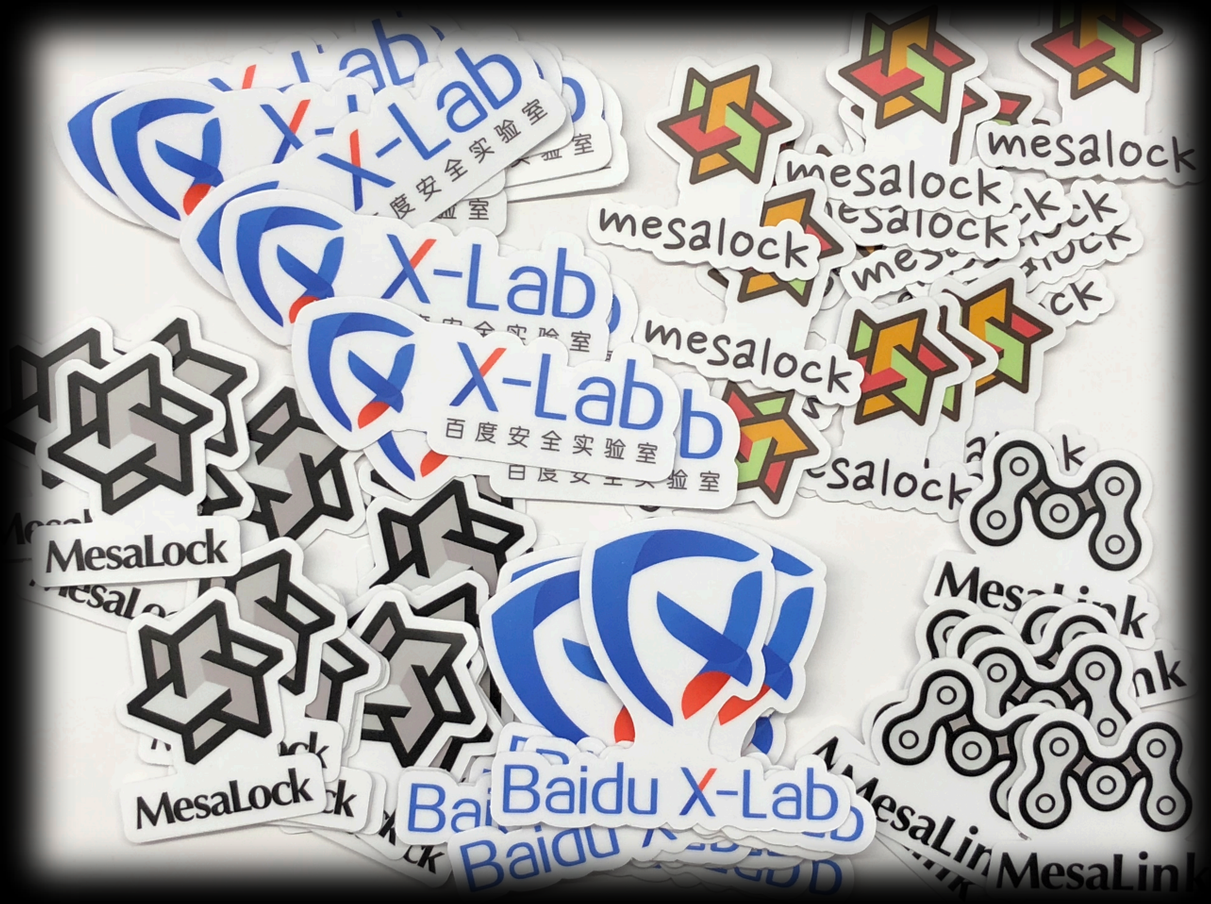
iOS/OS X
Open in Safari
Keynote

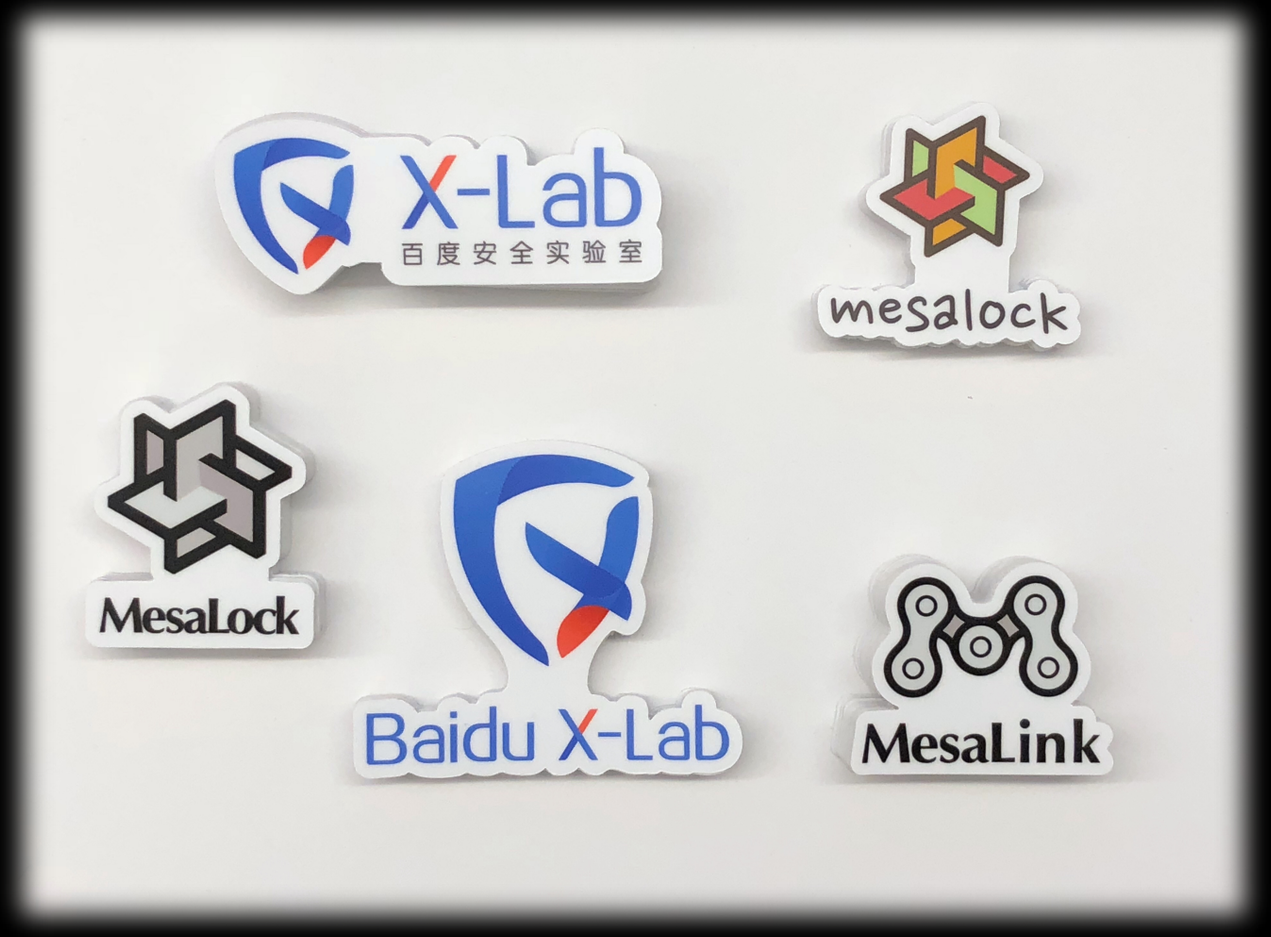
Windows/Linux
Browser

whoami

- Senior Security Research in **Baidu X-Lab**, Baidu USA
- PhD, The Chinese University of Hong Kong
- System security, mobile security, IoT security, and car hacking
- Maintaining MesaLock Linux, TaintART, Pass for iOS, etc.
- `mssun @ GitHub` | `https://mssun.me`

Baidu X-Lab


X-Lab
百度安全实验室
mesalock
MesaLock
Baidu X-Lab
MesaLink


X-Lab
百度安全实验室
mesalock
MesaLock
Baidu X-Lab
MesaLink

Baidu X-Lab ❤️ Rust

- **MesaLock Linux**: a memory-safe Linux distribution
- **MesaBox**: a collection of core system utilities written in Rust
- **MesaLink**: a memory-safe and OpenSSL-compatible TLS library
- **Rust SGX SDK**: provides the ability to write Intel SGX applications in Rust
- and many more ...

Outline

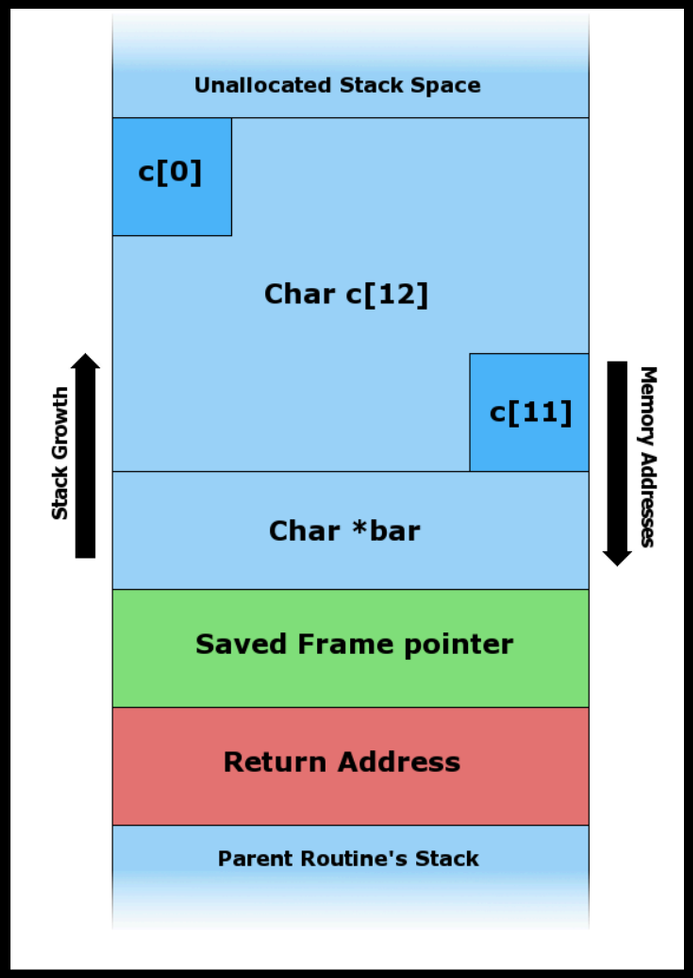
- Intro
- Memory-safety
- Rust
- Beyond

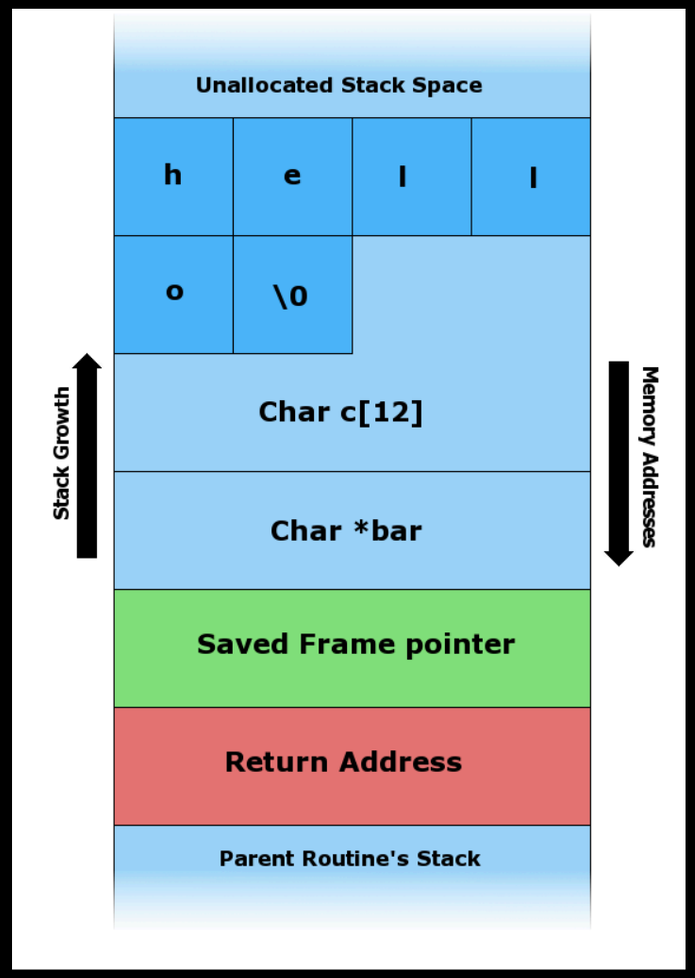
Why

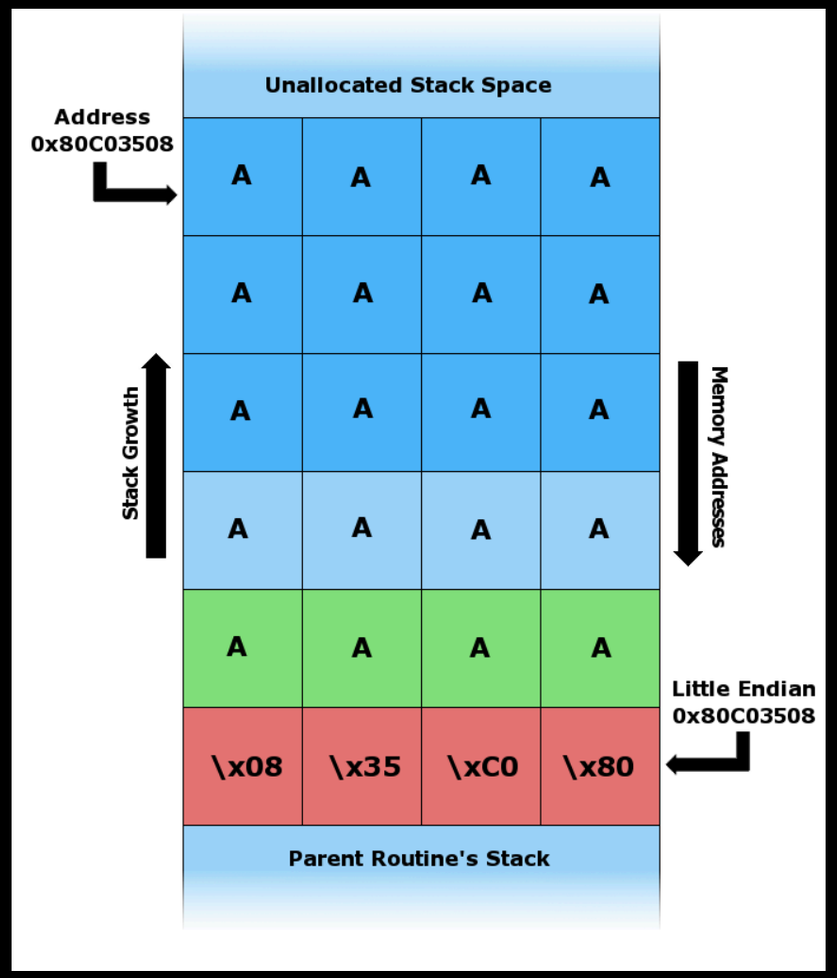
- **Memory corruption** occurs in a computer program when the contents of a memory location are **unintentionally modified**; this is termed violating memory safety.

- **Memory safety** is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as **buffer overflows and dangling pointers**.

Stack Buffer Overflow





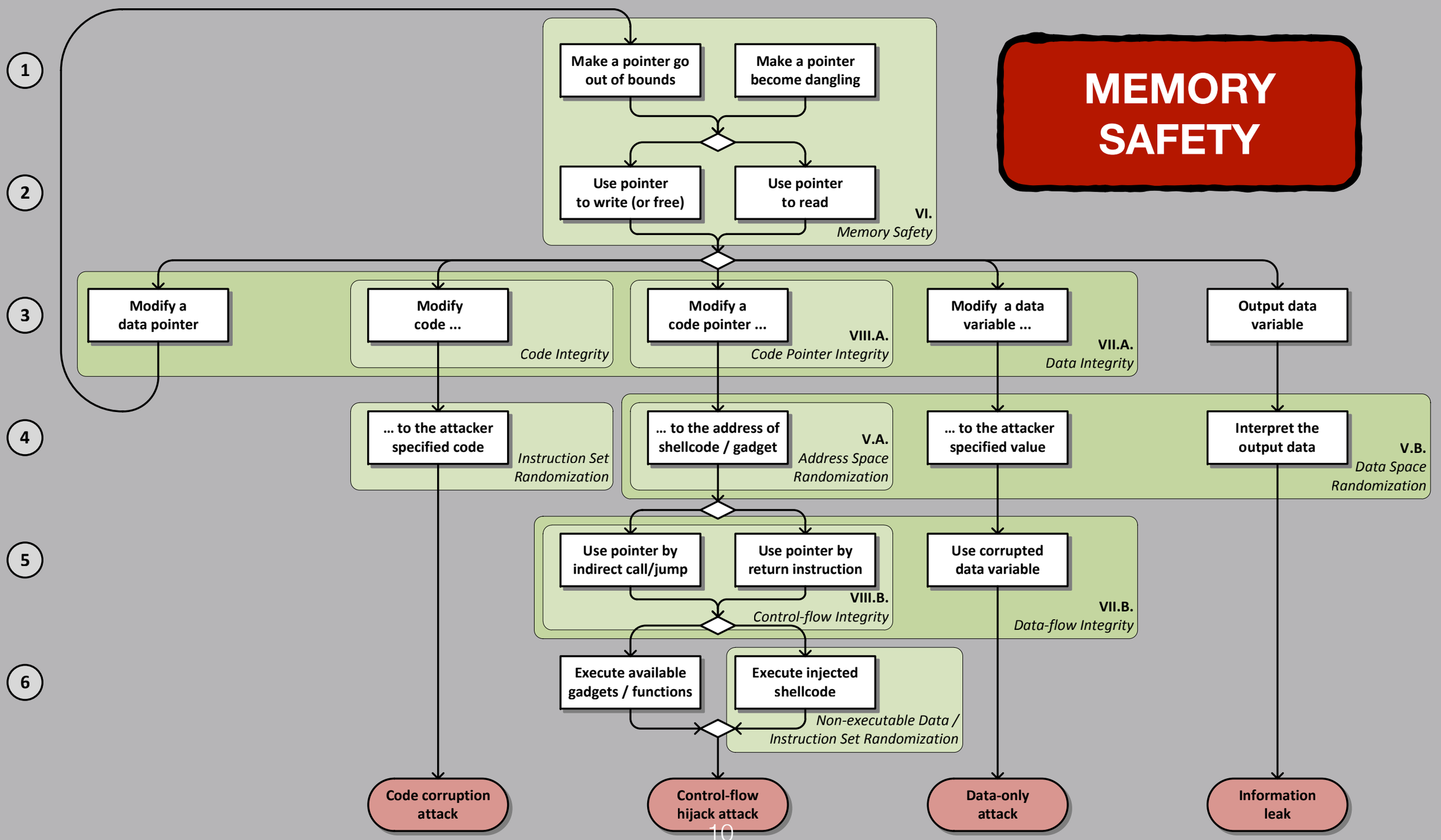


- https://youtu.be/T03idxny9jE

Types of memory errors

- Access errors
  - Buffer overflow
  - Race condition
    - Use after free
    - Segmentation fault
- Uninitialized variables
- Memory leak
  - Double free

SoK: Eternal War in Memory
Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song
Proceedings of the 2013 IEEE Symposium on Security and Privacy



Approaches to Mitigate Memory Corruption Errors

- Program analysis like symbolic execution: **KLEE**
- Memory-checking virtual machine: **Valgrind**
- Compiler instrumentation: **AddressSanitizer**
- Fuzzing: **AFL, libFuzzer**
- Formal verification: **Seahorn, Smack, Trust-in-Soft**

Approaches to Mitigate Memory Corruption Errors

- Program analysis like symbolic execution: KLEE
- Memory-checking virtual machine: Valgrind
- Compiler instrumentation: AddressSanitizer
- Fuzzing: AFL, libFuzzer
- Formal verification: Seahorn, Smack, Trust-in-Soft

Approaches to Mitigate Memory Corruption Errors

- ~~Program analysis like symbolic execution: KLEE~~
- ~~Memory-checking virtual machine: Valgrind~~
- ~~Compiler instrumentation: AddressSanitizer~~
- ~~Fuzzing: AFL, libFuzzer~~
- ~~Formal verification: Seahorn, Smack, Trust-in-Soft~~
- **Programming languages: Rust, Go**

System Programming

- Memory management
- Error handling
- Static Typing
- Compiling
- ...

Rust

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

What causes memory issues?

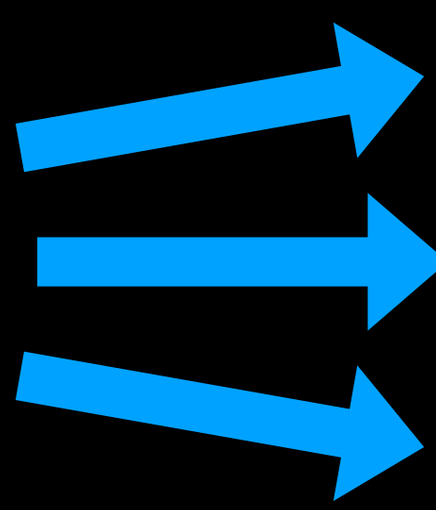
Aliasing* + *~~Mutation~~

~~Aliasing~~* + *Mutation

Aliasing* + *Mutation

How Does Rust Guarantee Memory Safety?

- Ownership
- Borrowing
- Lifetime



- No need for a runtime (C/C++)
- Memory safety (GC)
- Data-race freedom

Ownership and Borrowing

- In Rust, every value has a **single, statically-known, owning path** in the code, at any time.

- Pointers to values have limited duration, known as a "**lifetime**", that is also **statically tracked.**

- All pointers to all values are known **statically**.

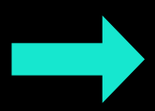
Ownership

Alice


```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

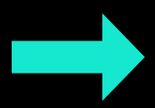
Alice **Bob**



```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

Alice **Bob**



```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

Alice


```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

Alice

```
error[E0382]: use of moved value: `alice`
 --> src/main.rs:7:27
  |
4 |         let bob = alice;
  |             --- value moved here
...
7 |     println!("alice: {}", alice[0]);
  |                           ^^^^^ value used here after move
  |
  = note: move occurs because `alice` has type
`std::vec::Vec<i32>`, which does not implement the `Copy` trait
```

```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

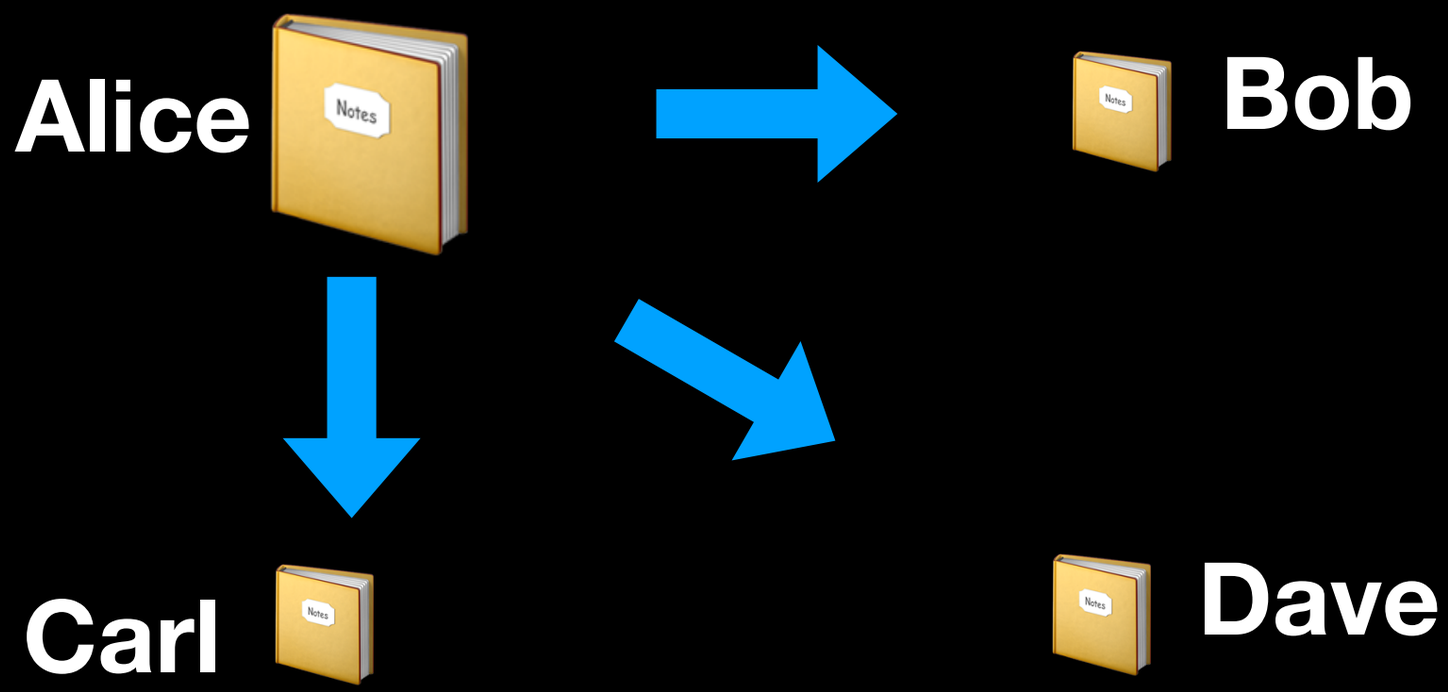
Alice

```
error[E0382]: use of moved value: `alice`
 --> src/main.rs:7:27
  |
4 |         let bob = alice;
  |             --- value moved here
...
7 |     println!("alice: {}", alice[0]);
  |                           ^^^^^ value used here after move
  |
  = note: move occurs because `alice` has type
`std::vec::Vec<i32>`, which does not implement the `Copy` trait
```

```
fn main() {
    let mut alice = vec![1, 2, 3];;
    {
        let mut bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Shared Borrow (&T)

Aliasing + ~~Mutation~~



Mutable Borrow (&mut T)

Alice


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

Alice **Bob**


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

Alice


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

Alice


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

~~Aliasing~~ + Mutation

Alice

The lifetime of a borrowed reference should end before the lifetime of the owner object does.

Rust's Ownership & Borrowing

~~Aliasing~~ + ***~~Mutation~~***

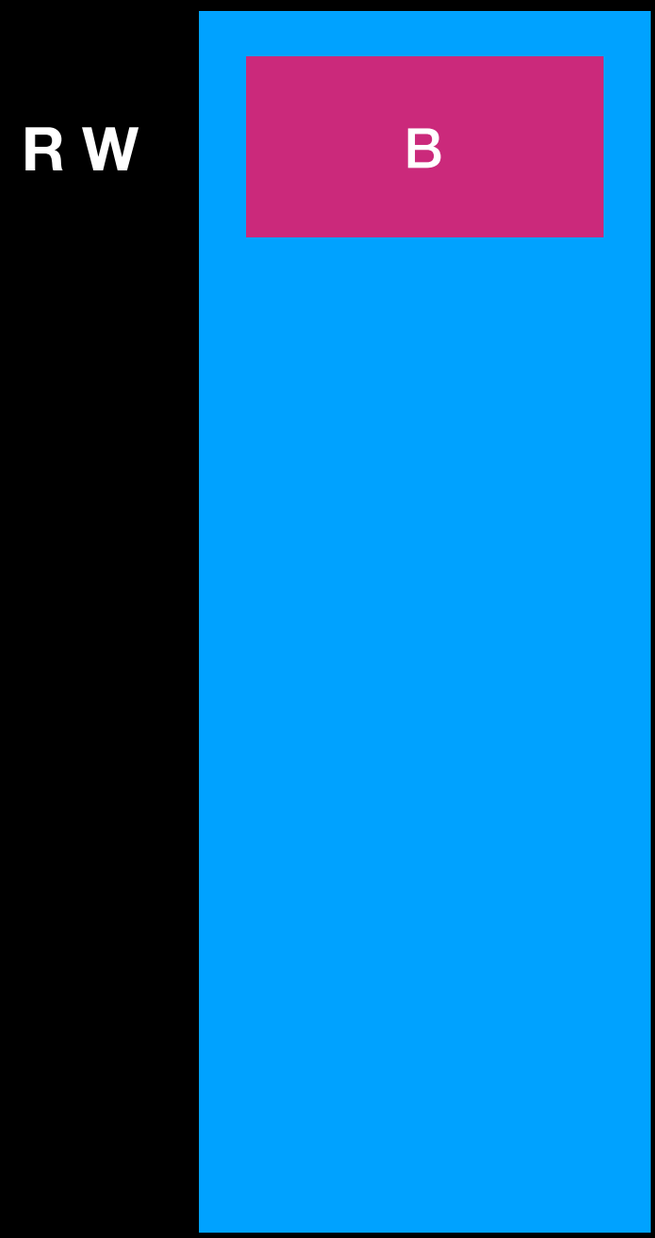
- Compiler enforced:
  - Every resource has a unique **owner**
  - Others can **borrow** the resource from its owner (e.g., create an **alias**) with restrictions
  - Owner **cannot** free or mutate its resource while it is borrowed

Ownership & Borrowing

Owership	`T`	"owned"
Exclusive access	`&mut T`	"mutable"
Shared access	`&T`	"read-only"

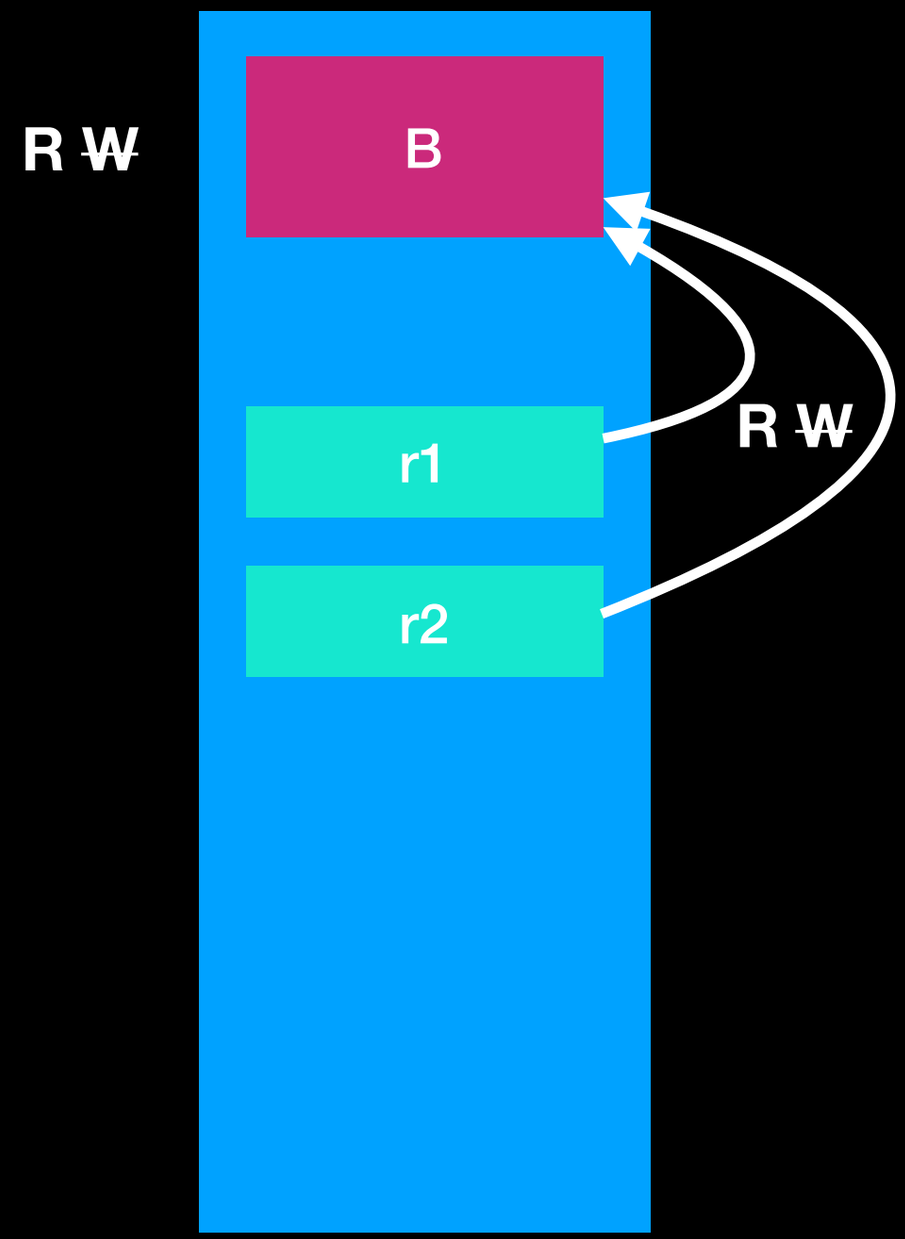
Stack allocation

```
let b = B::new();
```



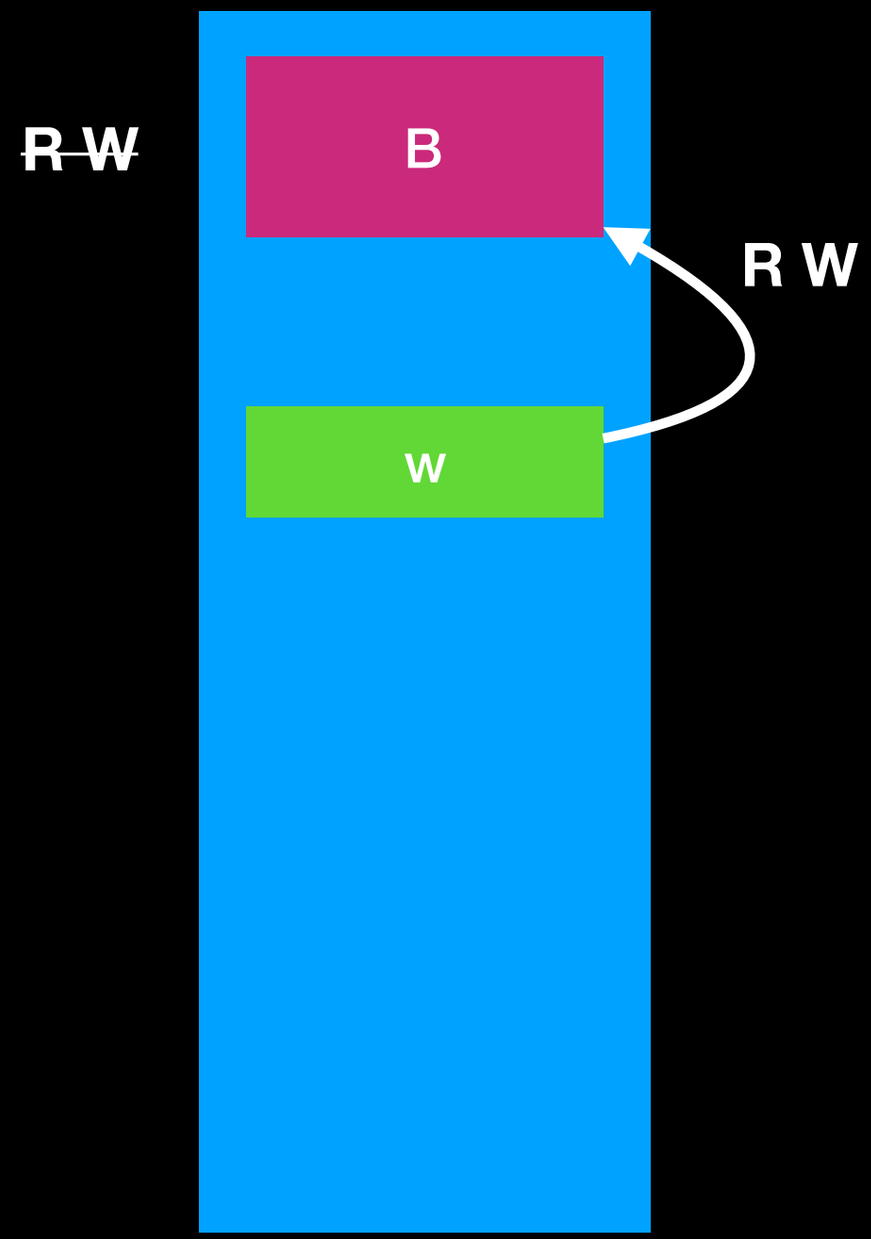
Stack allocation

```
let b = B::new();

let r1: &B = &b;
let r2: &B = &b;

// stack allocation and
// immutable borrows, b has
// lost write capability
```



Stack allocation

```
let b = B::new();

let w: &mut B = &mut b;

// stack allocation and mutable
borrows, b has temporarily lost
both read and write
capabilities
```

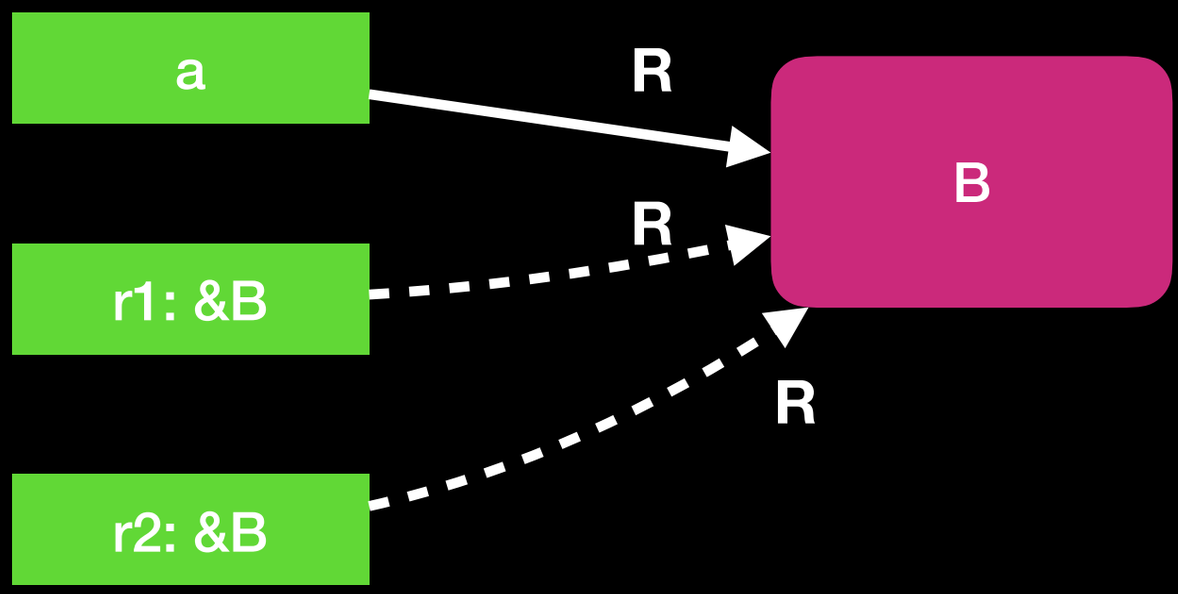


Heap allocation

```
let a = Box::new(B::new());

// Boxed B, a (as owner) has both read and write
capabilities.
```

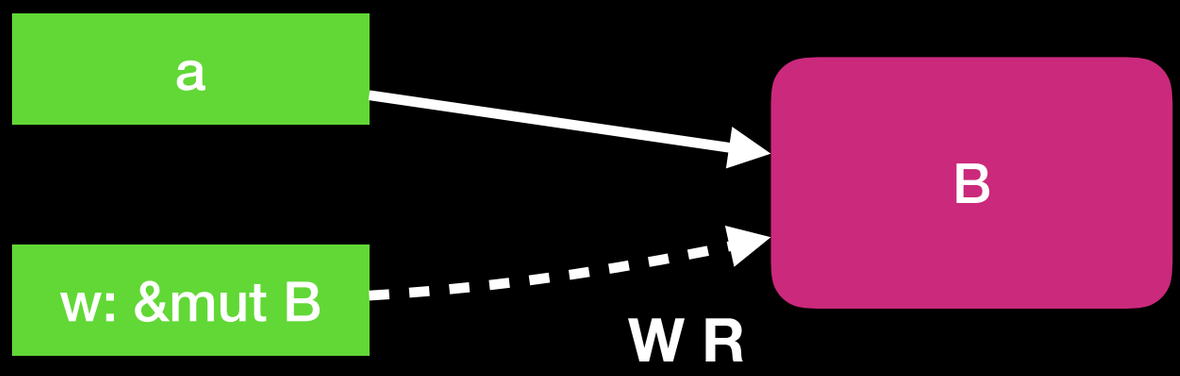
Immutably borrowing a box

```
let a = Box::new(B::new());
let r_of_box: &Box<B> = &a; // not directory a ref of B

let r1: &B = &*a;
let r2: &B = &a;  // <-- coercion!

// immutable borrows of heap-allocated B, a retains
read capabilities (has temporarily lost write)
```



Mutably borrowing a box

```
let a = Box::new(B::new());
let r_of_box: &Box<B> = &a; // not directory a ref of B

let w: &mut B = &mut a; // (again, coercion here)

// mutable borrow of heap-allocated B, a has
temporarily lost both read and write capabilities
```



Lifetime

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

Lifetime

```
error[E0597]: `x` does not live long enough
  --> src/main.rs:7:5
   |
6  |           r = &x;
   |                 - borrow occurs here
7  |       }
   |       ^ `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here
```

Borrow Checker

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

Borrow Checker

```
{
    let x = 5;                  // ----------+-- 'b
                                //           |
    let r = &x;                 // --+-- 'a  |
                                //   |       |
    println!("r: {}", r);       //   |       |
                                // --+       |
}                               // ----------+
```

Lifetime in Functions

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime in Functions

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expected lifetime
parameter
  |
  = help: this function's return type contains a
borrowed value, but the
signature does not say whether it is borrowed from `x`
or `y`
```

Lifetime in Functions

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime in Functions

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime in Functions

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}


fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime in Functions

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}


fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime in Functions

```
error[E0597]: `string2` does not live long enough
  --> src/main.rs:15:5
   |
14 |         result = longest(string1.as_str(), string2.as_str());
   |                                             ------- borrow
occurs here
15 |     }
   |     ^ `string2` dropped here while still borrowed
16 |     println!("The longest string is {}", result);
17 | }
   | - borrowed value needs to live until here
```

Use-After Free in C/Rust

C/C++

```
void func() {
    int *used_after_free = malloc(sizeof(int));

    free(used_after_free);

    printf("%d", *used_after_free);
}
```

Rust

```
fn main() {
    let name = String::from("Hello World");
    let mut name_ref = &name;
    {
        let new_name = String::from("Goodbye");
        name_ref = &new_name;
    }
    println!("name is {}", &name_ref);
}
```

Use-After Free in Rust

```
error[E0597]: `new_name` does not live long enough
 --> main.rs:7:5
  |
6 |         name_ref = &new_name;
  |                    --------- borrow occurs here
7 |     }
  |     ^ `new_name` dropped here while still borrowed
8 |     println!("name is {}", &name_ref);
9 | }
  | - borrowed value needs to live until here

error: aborting due to previous error
```

Formal Verification

- RustBelt: Securing the Foundations of the Rust Programming Language (POPL 2018)

- *In this paper, we give the first **formal (and machine-checked) safety proof** for a language representing a realistic subset of Rust.*

- https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf

Break

Rust

Rust

- Created by Graydon Hoare
- Funded by Mozilla
- Primary goal: A safer system language
- First version: 2010
- Current stable version 1.27.1 (2018-07-07)

Why use Rust?

- Open Source and Open Governance
- Top-tier performance (like C/C++ or better)
- Memory safe (No memory leak)
- No runtime, no GC (runs everywhere)
- No undefined behavior

Why use Rust?

- Zero-cost abstractions
- Ergonomic syntax
- Expressive data structures
- Targets the same use cases as C/C++ (all of them)
- Sponsored by Mozilla (makers of Firefox)
- Most loved programming language (Stack Overflow Developer Survey in 2016 and 2017)

Rust

- Rust is a **systems programming** language focused on three goals: **safety**, **speed**, and **concurrency**.

- Rust combines **low-level control** over performance with **high-level convenience** and safety guarantees.

- Rust is a language for **confident**, **productive** systems programming.

Rust Syntax

- Rust has a C-style syntax with influences from functional languages.

- Specific functionality will be covered later.

Basic

```
// Function declaration
fn add_them(first: i32, second: i32) -> i32 {
    first + second
}

fn main() {
    // Mutable variable
    let mut some_value = 1;
    // Immutable, explict type
    let explicitly_typed: i32 = 1;

    // Function call
    some_value = add_them(some_value, explicitly_typed);

    // Macro, note the !
    println!("{}", some_value)
}
```

if

```
fn main() {
    let value = 2;

    if value % 2 == 0 {
        // ...
    } else if value == 5 {
        // ...
    } else { /* ... */ }
}
```

match

```
fn main() {
    let maybe_value = Some(2);
    match maybe_value {
        Some(value) if value == 2 => {
            // ...
        }
        Some(value) => {
            // ...
        },
        None => {
            // ...
        },
    }
}
```

if let

```
fn main() {
    let maybe_value = Some(2);

    if let Some(value) = maybe_value {
        // ...
    } else { /* ... */ }
}
```

loop and while

```
fn main() {
    let mut value = 0;
    // Loop with break
    loop {
        if value >= 10 {
            break;
        }
        value += 1;
    }
    // Break on conditional
    while value <= 10 {
        value += 1;
        // ...
    }
}
```

for and while let

```
fn main() {
    // Loop over iterator
    let range = 0..10;
    for i in range {
        // ...
    }
    // while let
    let mut range = 0..10;
    while let Some(v) = range.next() {
        // ...
    }
}
```

struct, type, and enum

```
struct Empty;

struct WithFields {
    foo: i32,
    bar: Choice,
}

type Explanation = String;

enum Choice {
    Yes,
    No,
    Maybe(Explanation),
}

fn main() {}
```

impl and trait

```
trait Bar {
    // This can be overridden
    fn default_implementation(&self) -> bool {
        true
    }
    fn required_implementation(&self);
}

impl Bar for Foo {
    fn required_implementation(&self) {
        // ...
    }
}

impl Foo {
    fn new() -> Self { Foo }
}
```

Borrowing

```
// &mut denotes a mutable borrow
fn accepts_borrow(thing: &mut u32) {
    *thing += 1
}

fn main() {
    let mut value = 1;
    accepts_borrow(&mut value);
    println!("{}", value)
}
```

Lifetimes

```
fn with_lifetimes<'a>(thing: &'a str) -> &'a str {
    thing
}

fn main() {
    let foo = "foo";
    println!("{}", with_lifetimes(foo))
}
```

Scopes: Rust is block scoped. Scopes can return values.

```
fn main() {
    let foo = 1;
    let bar = {
        // Shadows earlier declaration.
        let foo = 2;
        foo
    };
    println!("{}", foo);
    println!("{}", bar);
}
```

Closures

```
fn main() {
    // Shorthand
    let value = Some(1).map(|v| v + 1);
    // With a block
    let value = Some(1).map(|v| {
        v + 1
    });
    // Explict return type
    let value = Some(1).map(|v| -> i32 {
        v + 1
    });
    // Declared
    let closure = |v| v + 1;
    let value = Some(1).map(closure);
}
```

Generics

```
// Inline syntax
fn generic_inline<S: AsRef<str>>(thing: S) -> S {
    thing
}

// Where syntax
fn generic_where<Stringish>(thing: Stringish) -> Stringish
where Stringish: AsRef<str> {
    thing
}

// Enums too!
struct GenericStruct<A> {
    value: A,
}

fn main() {
    let foo = "foo";
    generic_inline(foo);
    generic_where(foo);
}
```

use and mod

```
use foo::foo;

mod foo {
    pub fn foo() {
        // ...
    }
}

// Will try to open `./bar.rs` relative to this file.
pub mod bar;

fn main() {
    foo()
}
```

Attributes

- Rust attributes are used for a number of different things. There is a full list of attributes in the reference.

```
#[derive(Clone, Copy)]
struct Foo;

#[inline(always)]
fn bar() {}

fn main() {}
```

Attributes

- Comparison traits: Eq, PartialEq, Ord, PartialOrd
- Clone, to create T from &T via a copy.
- Copy, to give a type 'copy semantics' instead of 'move semantics'
- Hash, to compute a hash from &T.
- Default, to create an empty instance of a data type.
- Debug, to format a value using the {:?} formatter.

Error Handling

- Rust groups errors into two major categories:
  - recoverable -> `Result<T, E>`
  - unrecoverable errors -> `panic!`

Unrecoverable Errors with `panic!`

- print a failure message
- unwind and clean up the stack, and then quit
- occurs when a bug of some kind has been detected and it's not clear to the programmer how to handle the error.

```
fn main() {
    panic!("crash and burn");
}

$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25 secs
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Recoverable Errors with `Result<T, E>`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Recoverable Errors with `Result<T, E>`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn foo() -> Result<usize, std::io::Error>

match foo() {
    Ok(size) => println!("size: {}", size);
    Err(e) => panic!("panic: {:?}", e);
}
```

Recoverable Errors with `Result<T, E>`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
std::io::Stdin

pub fn read_line
    (&self, buf: &mut String) -> Result<usize>

type std::io::Result<T> =
    std::result::Result<T, std::io::Error>;
```

Recoverable Errors with `Result<T, E>`

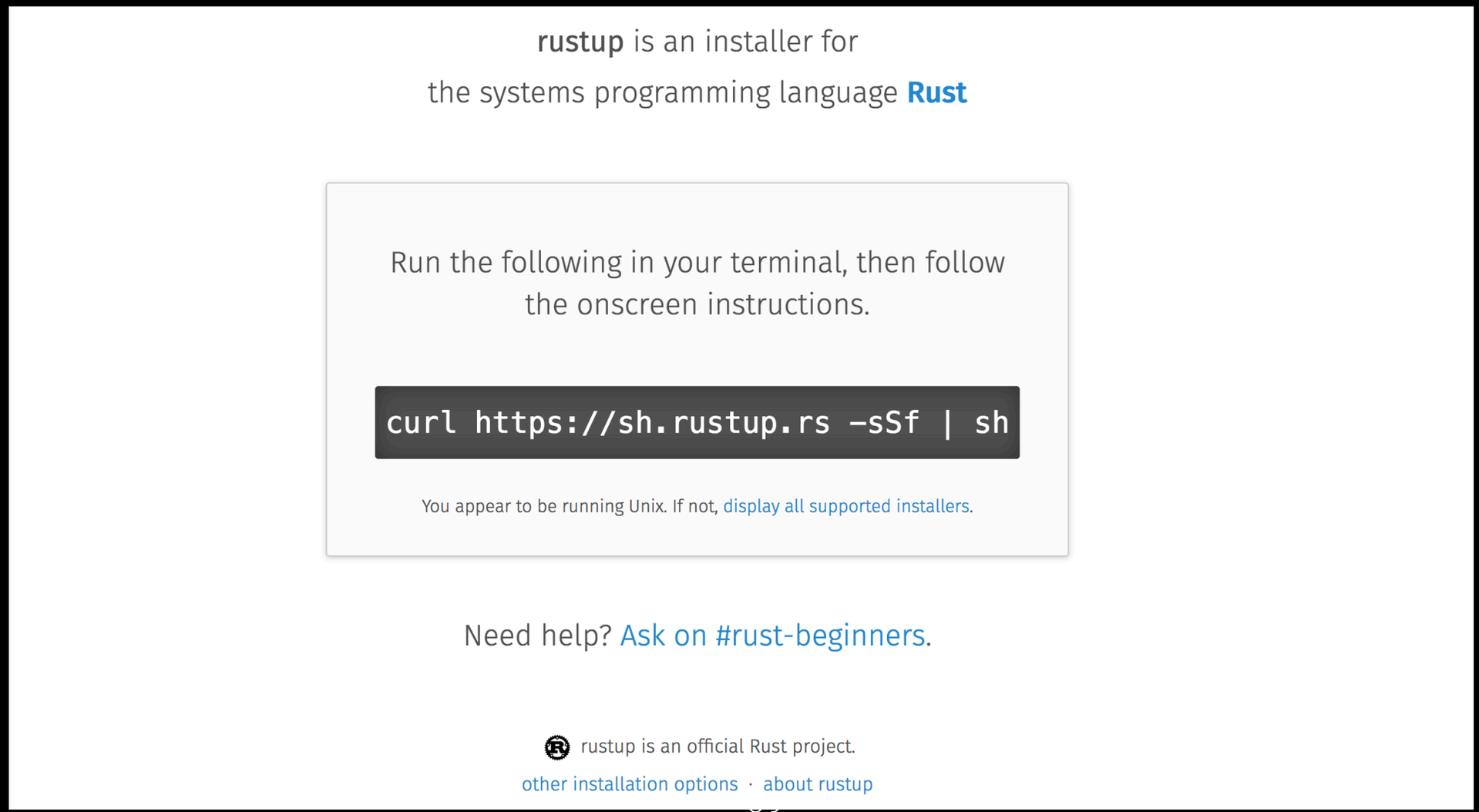
```
use std::io;

fn get_string() -> io::Result<String> {
    let mut buffer = String::new();

    match io::stdin().read_line(&mut buffer) {
        Ok(_) => {},
        Err(e) => return Err(e)
    }

    Ok(buffer)
}
```

Recoverable Errors with `Result<T, E>`

```
use std::io;

fn get_string() -> io::Result<String> {
    let mut buffer = String::new();

    io::stdin().read_line(&mut buffer)?;

    Ok(buffer)
}
```

Getting Started

- Installation: https://rustup.rs/



Hello, World!

```
fn main() {
    println!("Hello, world!");
}

$ rustc main.rs
$ ./main
Hello, world!
```

Cargo

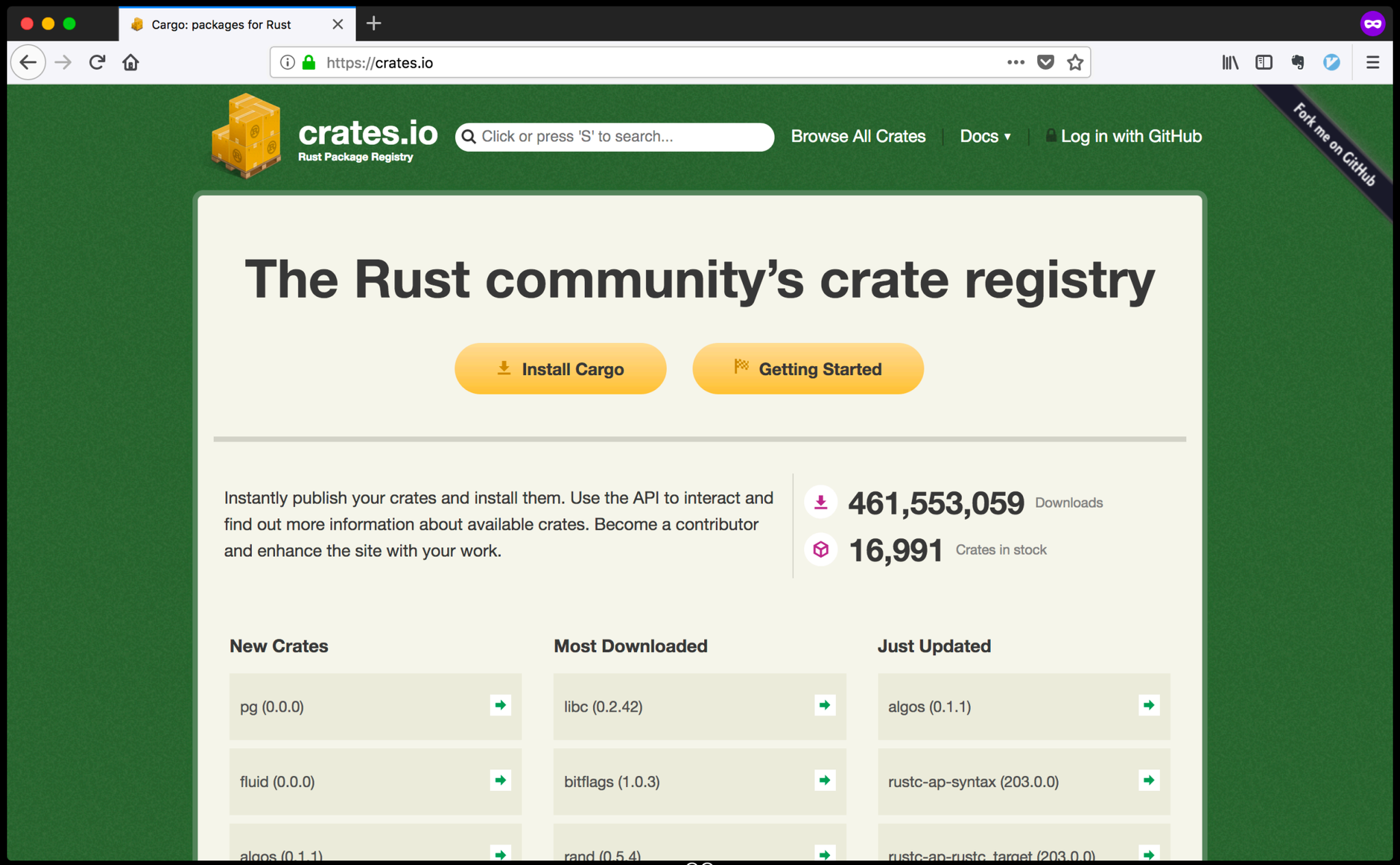
Cargo

Cargo is the Rust package manager. Cargo downloads your Rust project’s dependencies, compiles your project, makes packages, and upload them to crates.io, the Rust community’s package registry.

Cargo

- cargo new
- cargo build
- cargo run
- cargo XXX

crates.io

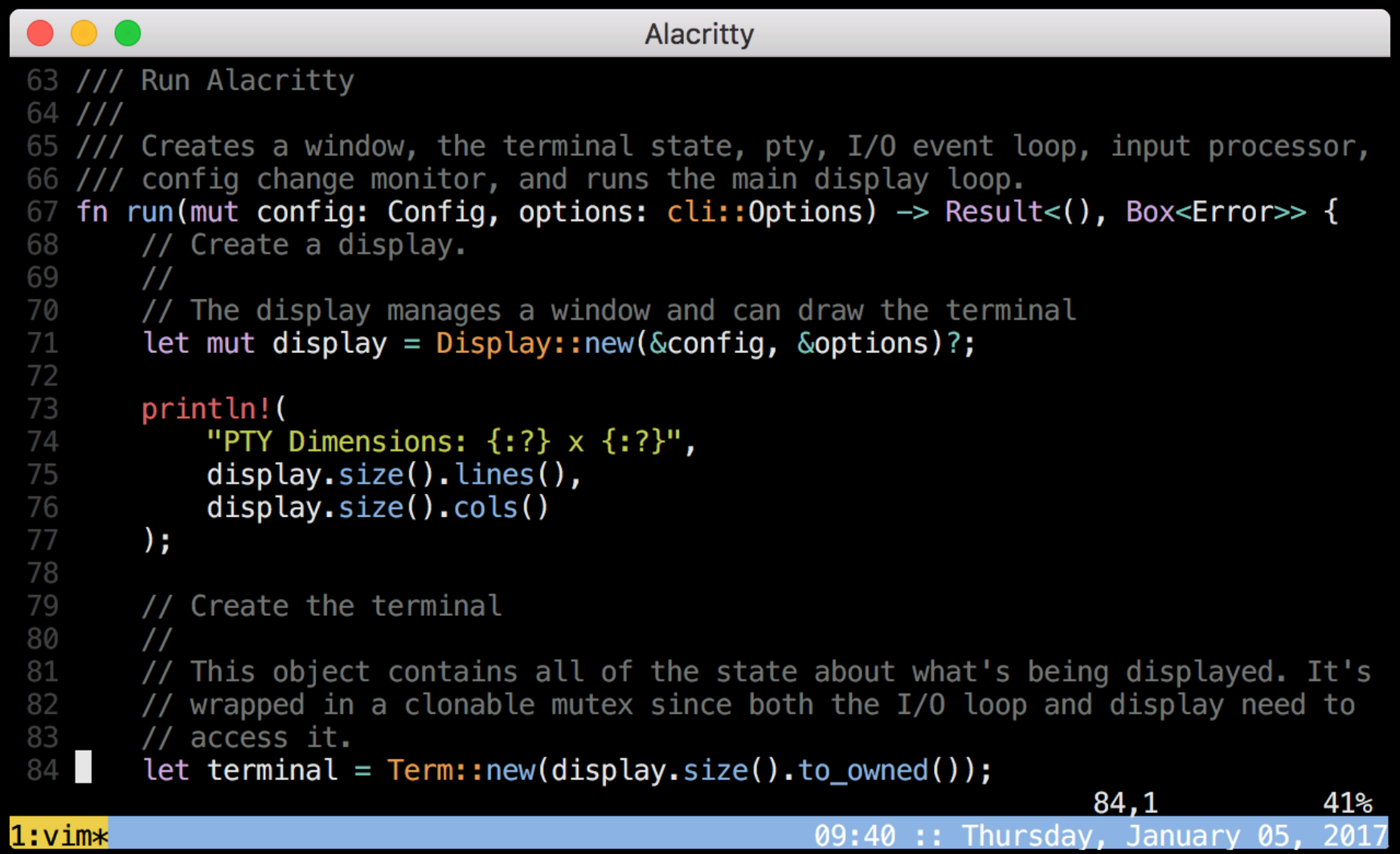

Cargo: packages for Rust
https://crates.io
crates.io
Rust Package Registry
Click or press 'S' to search...
Browse All Crates
Docs
Log in with GitHub
Fork me on GitHub
The Rust community's crate registry
Install Cargo
Getting Started
Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work.
461,553,059 Downloads
16,991 Crates in stock
New Crates
pg (0.0.0)
fluid (0.0.0)
algos (0.1.1)
Most Downloaded
libc (0.2.42)
bitflags (1.0.3)
rand (0.5.4)
Just Updated
algos (0.1.1)
rustc-ap-syntax (203.0.0)
rustc-ap-rustc_target (203.0.0)

Major projects

- Rust compiler and Cargo
- **Servo** – Mozilla's new parallel web browser engine
- **Redox OS** – a microkernel operating system
- **TockOS** – an embedded operating system
- **ripgrep** - text search provider in VS code

Alacritty: a GPU-accelerated terminal emulator

- Alacritty is the fastest terminal emulator in existence.



Xi

Xi.md

```
# xi-editor

The xi editor project is an attempt to build a high quality text
editor, using modern software engineering techniques. It is initially
built for Mac OS X, using Cocoa for the user interface. There are also
frontends for other operating systems available from third-party
developers.

Goals include:

* Incredibly high performance. All editing operations should commit and
paint in under 16ms. The editor should never make you wait for
anything.

* Beauty. The editor should fit well on a modern desktop, and not look
like a throwback from the '80s or '90s. Text drawing should be done
with the best technology available (Core Text on Mac, DirectWrite on
Windows, etc.), and support Unicode fully.

* Reliability. Crashing, hanging, or losing work should never happen.

* Developer friendliness. It should be easy to customize xi editor,
whether by adding plug-ins or hacking on the core.

## Getting started
```

Redox OS

Documentation Community News Screenshots
Donate GitLab RSoC

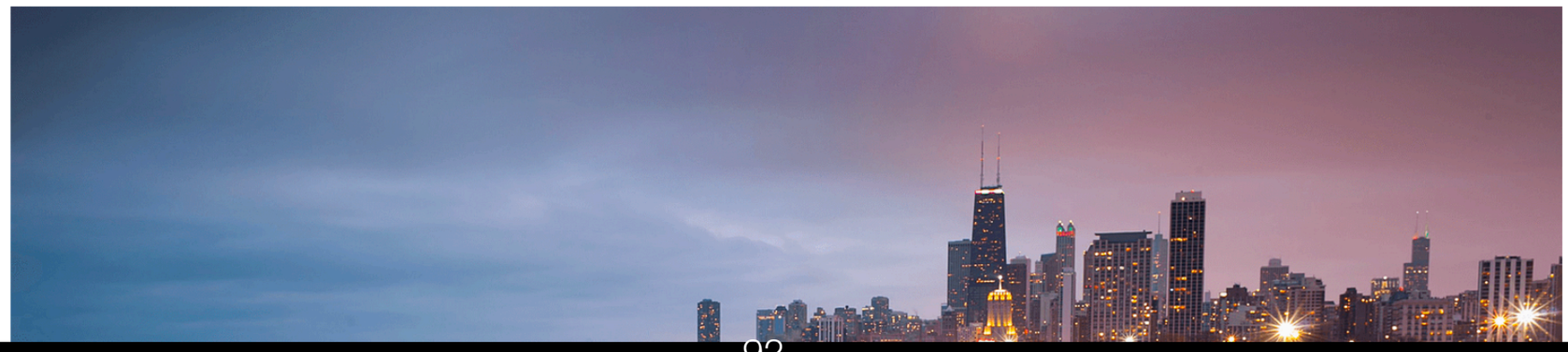
Redox is a Unix-like Operating System written in **Rust**, aiming to bring the innovations of Rust to a modern microkernel and full set of applications.

View Releases

Pull from GitLab

- Implemented in Rust
- Microkernel Design
- Includes optional GUI - Orbital
- Supports Rust Standard Library
- MIT Licensed
- Drivers run in Userspace
- Includes common Unix commands
- Newlib port for C programs

Redox running Orbital



Beyond

When building up the MesaLock Linux, I'm excited to see Rust as a programming language to fundamentally solve the memory safety issue.

- **lots of useful libraries**
- **prosperous ecosystem**
- **many useful rewrite**

But we need to have a deep understand Rust and its memory safety promise first, ...

Memory safe? Meh...

https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html

Unsafe Rust

All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time. However, Rust has a second language hiding inside of it that does not enforce these memory safety guarantees: unsafe Rust. This works just like regular Rust, but gives you extra superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler is trying to determine if code upholds the guarantees or not, it's better for it to reject some programs that are valid than accept some programs that are invalid. That inevitably means there are some times when your code might be okay, but Rust thinks it's not! In these cases, you can use unsafe code to tell the compiler, "trust me, I know what I'm doing." The downside is that you're on your own; if you get unsafe code wrong, problems due to memory unsafety, like null pointer dereferencing, can occur.

There's another reason Rust has an unsafe alter ego: the underlying hardware of computers is inherently not safe. If Rust didn't let you do unsafe operations, there would be some tasks that you simply could not do. Rust needs to allow you to do low-level systems programming like directly interacting with your operating system, or even writing your own operating system! That's one of the goals of the language. Let's see what you can do with unsafe Rust, and how to do it.

Unsafe Superpowers

To switch into unsafe Rust we use the `unsafe` keyword, and then we can start a new block that holds the unsafe code. There are four actions that you can take in unsafe Rust that you can't in safe Rust that we call "unsafe superpowers." Those superpowers are the ability to:

1. Dereference a raw pointer
2. Call an unsafe function or method
3. Access or modify a mutable static variable
4. Implement an unsafe trait

What is Unsafe Rust?

- All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time.
- However, Rust has a second language hiding inside of it that **does not enforce** these memory safety guarantees: **unsafe Rust**. This works just like regular Rust, but gives you **extra superpowers**.

Unsafe Superpowers

1. Dereference a **raw** pointer
2. Access or modify a **mutable static variable**
3. Call an unsafe function or method
4. Implement an unsafe trait

Unsafe Superpowers

1. Dereference a raw pointer

Rust

```
unsafe {
    let address = 0x012345usize;
    let r = address as *const i32;
}
```

Read/write arbitrary memory address.

Unsafe Superpowers

2. Access or modify a mutable static variable

Rust

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe { COUNTER += inc; }
}

fn main() {
    add_to_count(3);

    unsafe { println!("COUNTER: {}", COUNTER); }
}
```

Data races.

Unsafe Superpowers

3. Call an unsafe function or method

Rust

```
unsafe fn dangerous() {
    let address = 0x012345usize;
    let r = address as *const i32;
}

fn main() {
    unsafe { dangerous(); }
}
```

Call functions may cause undefined behaviors.

Unsafe Superpowers

3. Call an unsafe function or method (external)

Rust

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C:
{}", abs(-3));
    }
}
```

Call external functions may cause undefined behaviors.

"Unsafe" is agnostic

- **Rust developers**: It's OK. At least you **explicitly** type the **"unsafe" keyword** in the source code, and I know it is "unsafe" before using it.

- **Me**: Wrong. The "unsafe" code could be included in the dependent libraries. Did you review the source code of dependencies?

"Unsafe" is agnostic

Rust

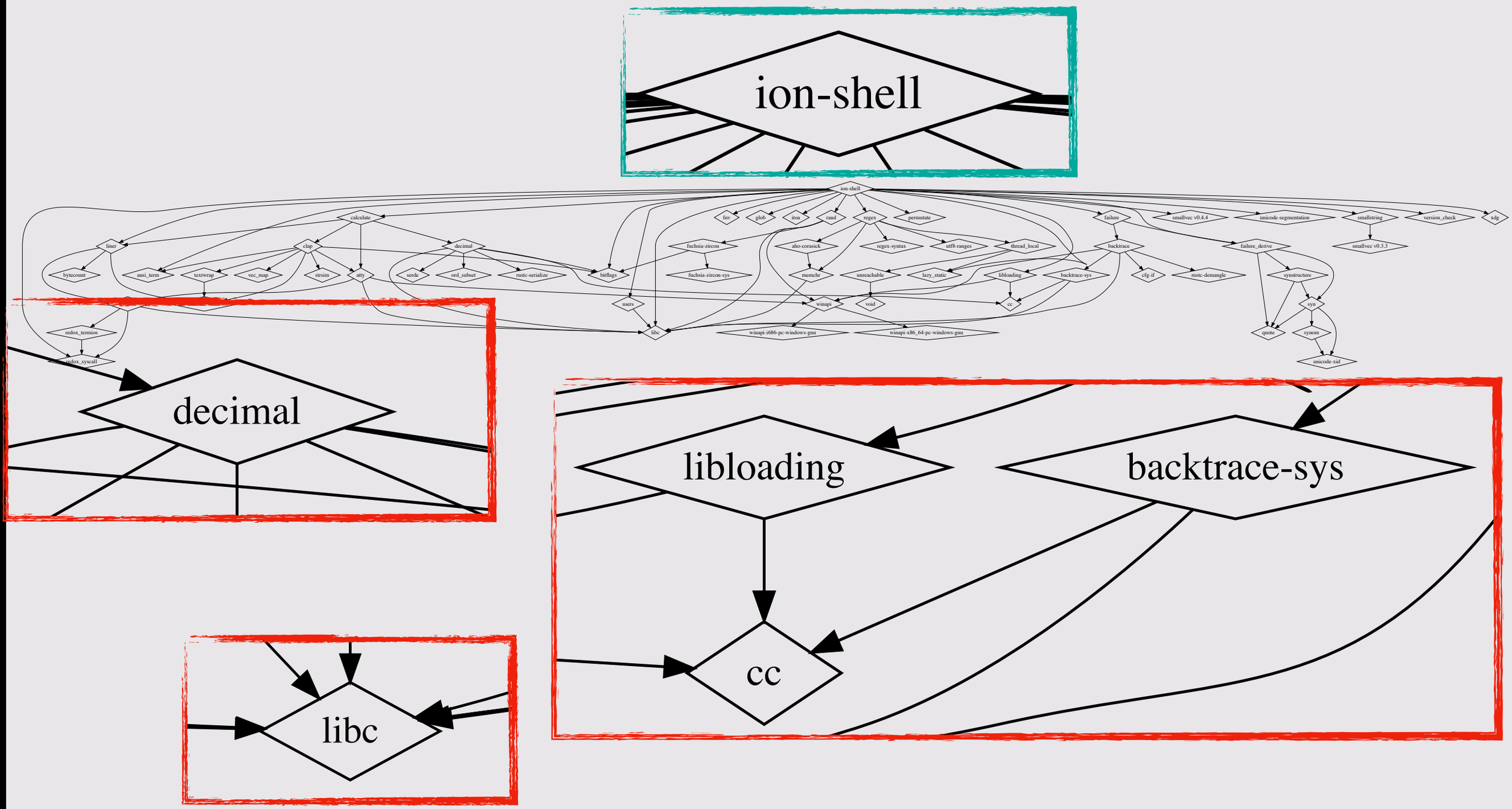
```
Library:

unsafe fn dangerous() {
    let address = 0x012345usize;
    let r = address as *const i32;
}

fn safe_function() {
    unsafe { dangerous(); }
}

Developer:

fn main {
    safe_function();
}
```

some libraries (including the std library) wrap unsafe code and re-export as "safe" functions

Case study: Ion Shell

- Ion is a modern system shell that features a simple, yet powerful, syntax. **It is written entirely in Rust, which greatly increases the overall quality and security of the shell.** It also offers a level of performance that exceeds that of Dash, when taking advantage of Ion's features. While it is developed alongside, and primarily for, RedoxOS, it is a fully capable on other *nix platforms.

Dependency graph of Ion shell



C libraries in Ion Shell

- Linked C libraries
  - glibc
  - decimal
  - libloading
  - backtrace-sys
- What is cc crate?
  - compiles C sources and (statically) links into Ion shell

cargo build -vv

- Build Ion shell again with verbose output.

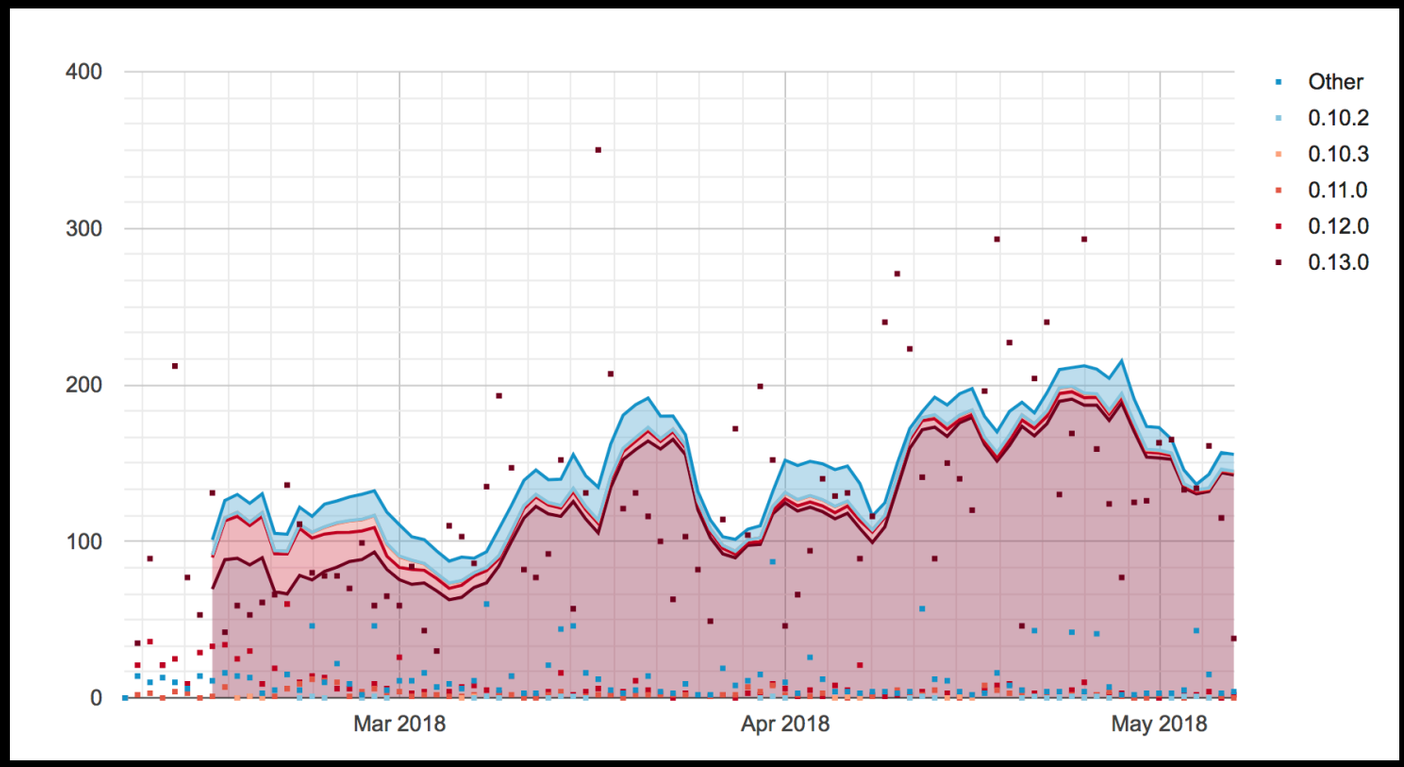
```
running: "cc" "-O0" "-ffunction-sections" "-fdata-sections"
"-fPIC" "-g" "-m64" "-I" "decNumber" "-Wall" "-Wextra" "-
DDECLITEND=1" "-o" "/Users/mssun/Repos/ion/target/debug/
build/decimal-b8ff0faecf5447ab/out/decNumber/decimal64.o" "-
c" "decNumber/decimal64.c"
```

- decimal crate: Decimal Floating Point arithmetic for rust based on the decNumber library. (http://speleotrove.com/decimal/decnumber.html)

- Ion shell depends on a decimal crate which still uses C code with potential memory safety issues.

Case study: rusqlite

- rusqlite is a Rust library providing SQLite related APIs
- an API wrapper of SQLite written in C
- 38 crates directly depend on rusqlite
- 200 downloads/day



Memory corruption in rusqlite library

- We tried a SQLite type confusion bug (CVE-2017-6991) in rusqlite library

- We can easily trigger the vulnerabilities

Many Birds, One Stone: Exploiting a Single SQLite Vulnerability Across Multiple Software, Siji Feng, Zhi Zhou, Kun Yang, BlackHat USA 17

Rust

```
extern crate rusqlite;
use rusqlite::Connection;

fn main() {
    let conn = Connection::open_in_memory().unwrap();
    match conn.execute("create virtual table a using fts3(b);", &[]) {
        // ...
    }
    match conn.execute("insert into a values(x'4141414141414141');", &[]) {
        // ...
    }
    match conn.query_row("SELECT HEX(a) FROM a", &[], |row| -> String
{ row.get(0) }) {
        // ...
    }
    match conn.query_row("SELECT optimize(b) FROM a", &[], |row| -> String
{ row.get(0) }) {
        // ...
    }
}
```

Run

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.05 secs
     Running `target/debug/rusqlite`
success: 0 rows were updated
success: 1 rows were updated
success: F0634013D87F0000
[1]    31467 segmentation fault  cargo run
```

static-linked SQLite

- sqlite3.c file is included in the Rust library
- statically linked into the binary/library using rusqlite
- did not keep track of the upstream SQLite repository

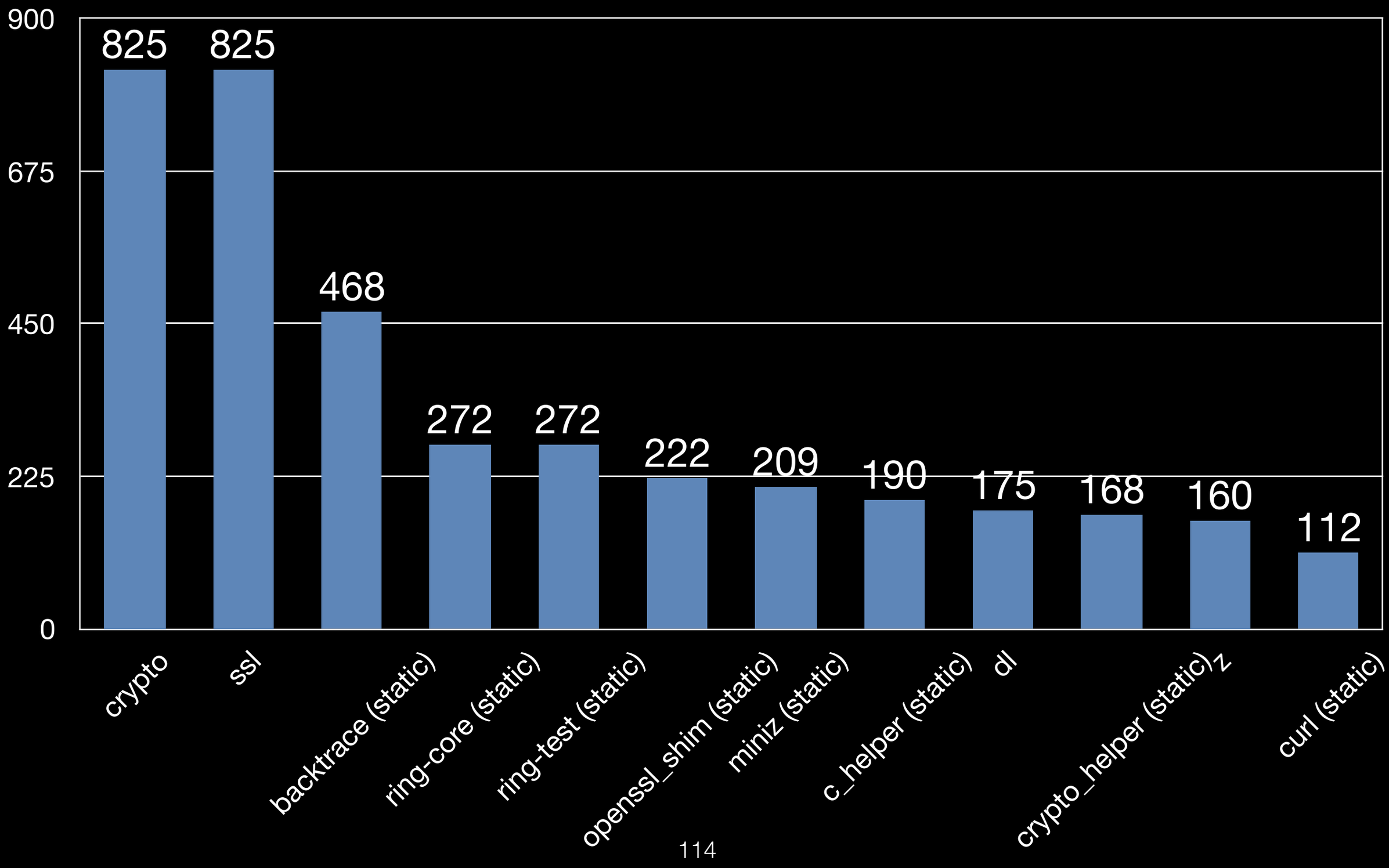
Data Collection and Study

- 10,693 Rust libraries in crates.io
- 200 million public downloads in total
- two studies
  - usage of external C/C++ libraries
  - usage of unsafe keywords

Usage of external libraries

- build.rs: a build script for Rust to compile third-party non-Rust code, for example C libraries
- We tried to build all downloaded libraries
- Analyze compiler building log
  - compile C/C++ source code using build.rs
  - static link/dynamic link built libraries or system libraries

Usage of external libraries (>= 100)



Analyze unsafe code

- Use Rust compiler to dump AST (abstract syntax tree)
- Find unsafe keyword in AST and extract corresponding code

“unsafe” code

- **3,099** out of 10,693 Rust libraries (crates) contain unsafe code
- **14,796** files in total
- **651,193** lines of code

Fuzz Rust Libraries

- cargo-fuzz
- Use after Free when parsing this XML Document (https://github.com/shepmaster/sxd-document/issues/47)
- src/string_pool.rs uses unsafe extensively, unsafe will break ownership and lifetime of a resource (data or variable)

Guideline of using “unsafe” code

Rules-of-thumb for hybrid memory-safe architecture designing proposed by the Rust SGX SDK project: https://github.com/baidu/rust-sgx-sdk/blob/master/documents/ccsp17.pdf

1. Unsafe components **must not taint** safe components, especially for public APIs and data structures.

2. Unsafe components should be **as small as possible** and **decoupled** from safe components.

3. Unsafe components should be **explicitly marked** during deployment and ready to upgrade.

More Rust?

Serde JSON

- Serde is a framework for serializing and deserializing Rust data structures efficiently and generically.

- `json!` macro to build serde_json::Value objects with very natural JSON syntax

Macros

```
let full_name = "John Doe";
let age_last_year = 42;

// The type of `john` is `serde_json::Value`
let john = json!({
  "name": full_name,
  "age": age_last_year + 1,
  "phones": [
    format!("+44 {}", random_phone())
  ]
});
```

Macros

```
let full_name = "John Doe";
let age_last_year = 42;

// The type of `john` is `serde_json::Value`
let john = json!({
  "name": full_name,
  "age": age_last_year + 1,
  "phones": [
    format!("+44 {}", random_phone())
  ]
});
```

Macros

```
let full_name = "John Doe";
let age_last_year = 42;

// The type of `john` is `serde_json::Value`
let john = json!({
  "name": full_name,
  "age": age_last_year + 1,
  "phones": [
    format!("+44 {}", random_phone())
  ]
});
```

Macros

```
let full_name = "John Doe";
let age_last_year = 42;

// The type of `john` is `serde_json::Value`
let john = json!({
  "name": full_name,
  "age": age_last_year + 1,
  "phones": [
    format!("+44 {}", random_phone())
  ]
});
```

Macros

- Macros can be super crazy
- Macros is a syntax parser
- The Little Book of Rust Macros: https://danielkeep.github.io/tlborm/book/index.html

```
fn main() {
    let _ = Ook!(
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook?  Ook? Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook?  Ook! Ook!  Ook? Ook!  Ook? Ook.
        Ook! Ook.  Ook. Ook?  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook. Ook.
        Ook! Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook! Ook.  Ook! Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook.  Ook. Ook?  Ook. Ook?
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook! Ook.
        Ook. Ook?  Ook. Ook?  Ook. Ook?  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook.
    );
}
```

This macro is an implementation of the Ook! esoteric language, which is isomorphic to the Brainfuck esoteric language.

The language itself is comprised of just three tokens: `Ook.`, `Ook?`, and `Ook!`.

These are combined in pairs to form the eight different operations:

`Ook.` `Ook?` - increment pointer.
`Ook?` `Ook.` - decrement pointer.
`Ook.` `Ook.` - increment pointed-to memory cell.
`Ook!` `Ook!` - decrement pointed-to memory cell.
`Ook!` `Ook.` - write pointed-to memory cell to standard output.
`Ook.` `Ook!` - read from standard input into pointed-to memory cell.
`Ook!` `Ook?` - begin a loop.
`Ook?` `Ook!` - jump back to start of loop if pointed-to memory cell is not zero; otherwise, continue.

Advanced Rust

- Multi-thread
- Macros, advanced trait, lifetime
- Unsafe Rust
- Async I/O: Tokio, Mio, Futures
- Rust compiler: borrow checker
- Rust + Web Assembly

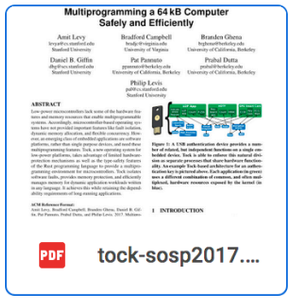
Research Problems on Rust

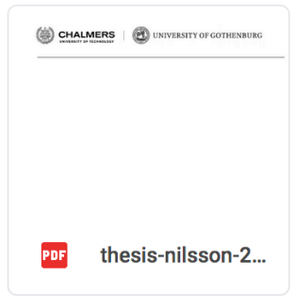
- C to Rust translation
- Unsafe code analysis
- Rust unsafe code sandbox/isolation
- Memory-safety of boundaries (FFI)
- Formal verification
- Fuzzing and symbolic execution
- Use Rust's memory model to redesign current systems
- Static analysis

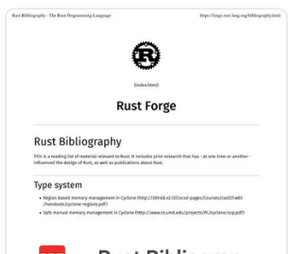
Papers

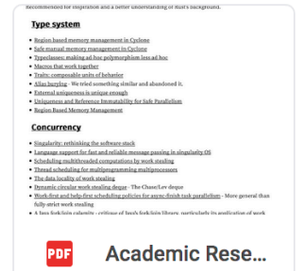
Files

Name



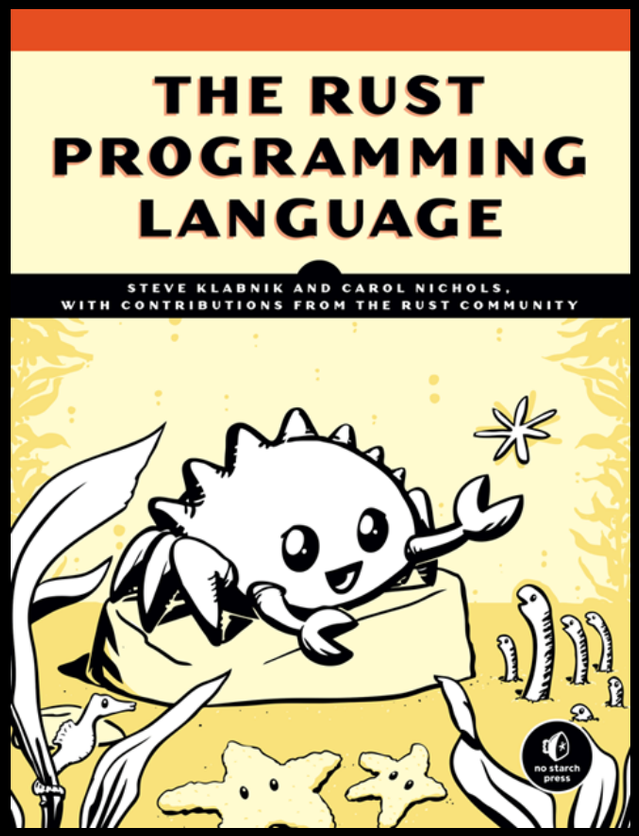


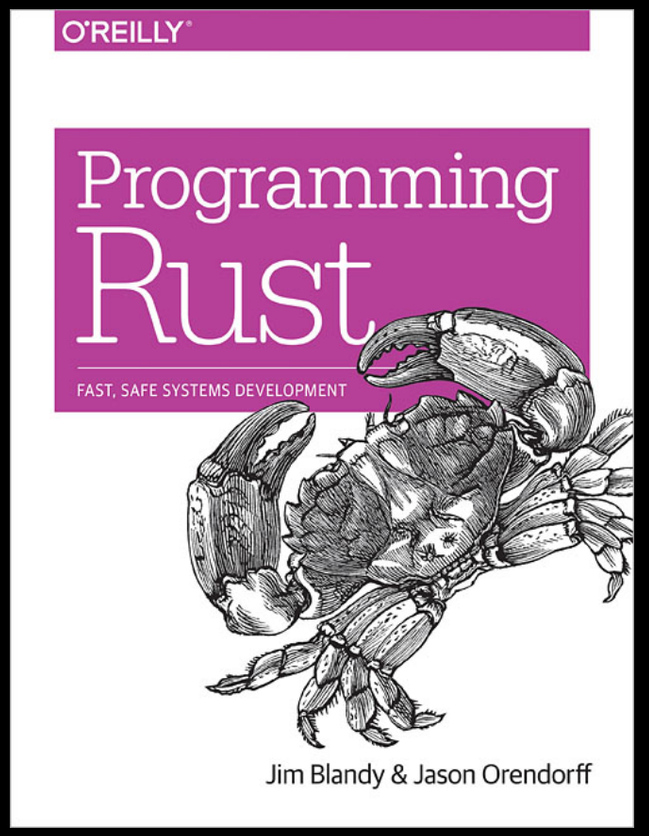




Books

- The Rust Programming Language
- Programming Rust: Fast, Safe Systems Development





RustConf and RustFest

- Rust YouTube channel: https://www.youtube.com/channel/UCaYhcUwRBNscFNUKTjgPFiA

Rust Meetup


facebook


Mononoke
A Mercurial Server in Rust
Arun Kulshreshtha
Software Engineer at Facebook
Mononoke
A Mercurial Server in Rust

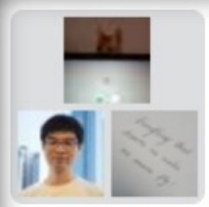
Misc

- This Week in Rust: https://this-week-in-rust.org/
- Awesome Rust: https://github.com/rust-unofficial/awesome-rust
- A Quick Intro to Rust Macros: https://danielkeep.github.io/quick-intro-to-macros.html
- rust-learning: https://github.com/ctjhoa/rust-learning

Community

- The Rust Programming Language Forum: https://users.rust-lang.org/
- Rust Internals: https://internals.rust-lang.org
- IRC, Discord

中文社区?



Rust, memory safety,
and beyond

Valid until 7/28 and will update upon joining group