Building Safe and Secure Systems in Rust: Challenges, Lessons Learned, and Open Questions

Mingshen Sun | Baidu X-Lab, USA

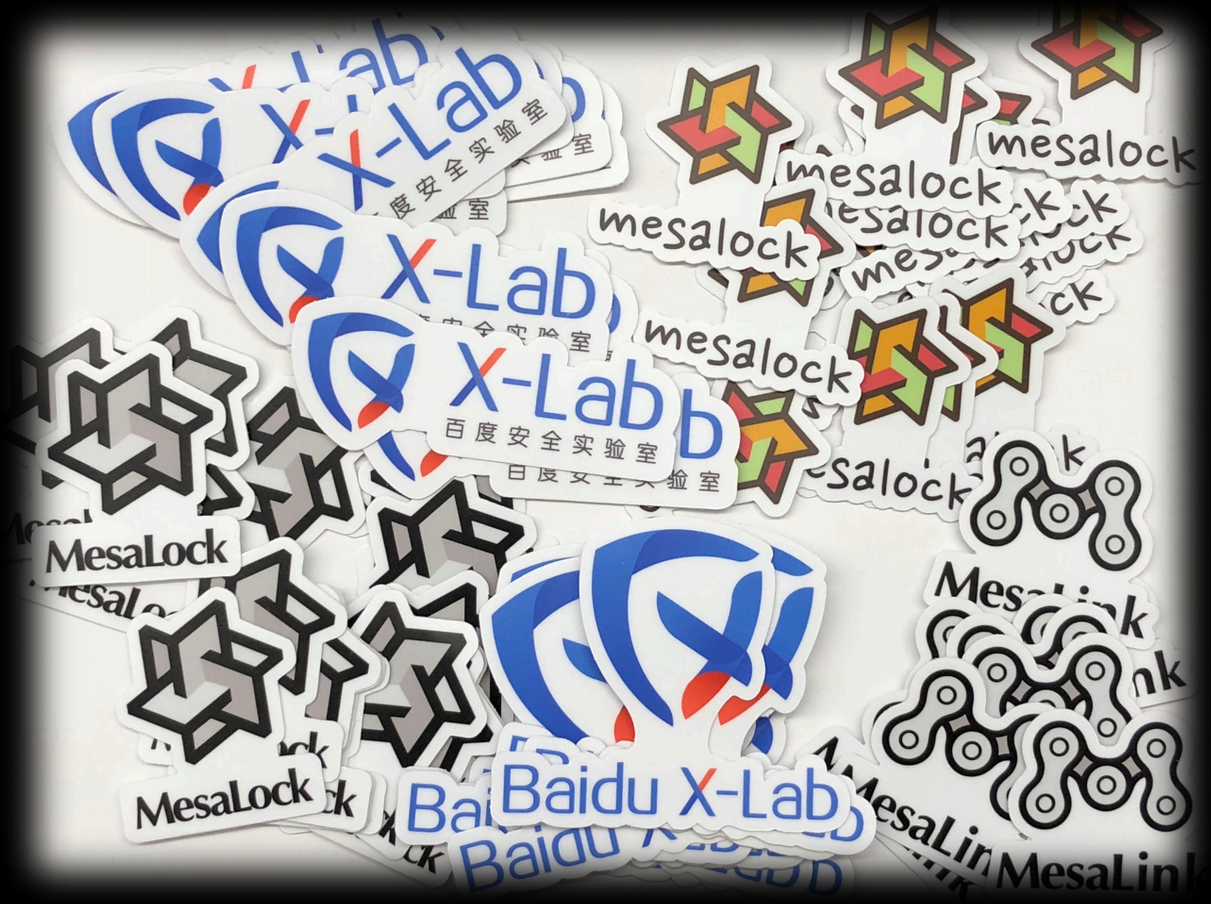
October 2018

Northeastern University, Boston

About Me

- Senior Security Researcher in **Baidu X-Lab**, USA
- PhD, The Chinese University of Hong Kong
- System security, mobile security, IoT security, and car hacking
- Maintaining open-source projects: MesaLock Linux, MesaPy, 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

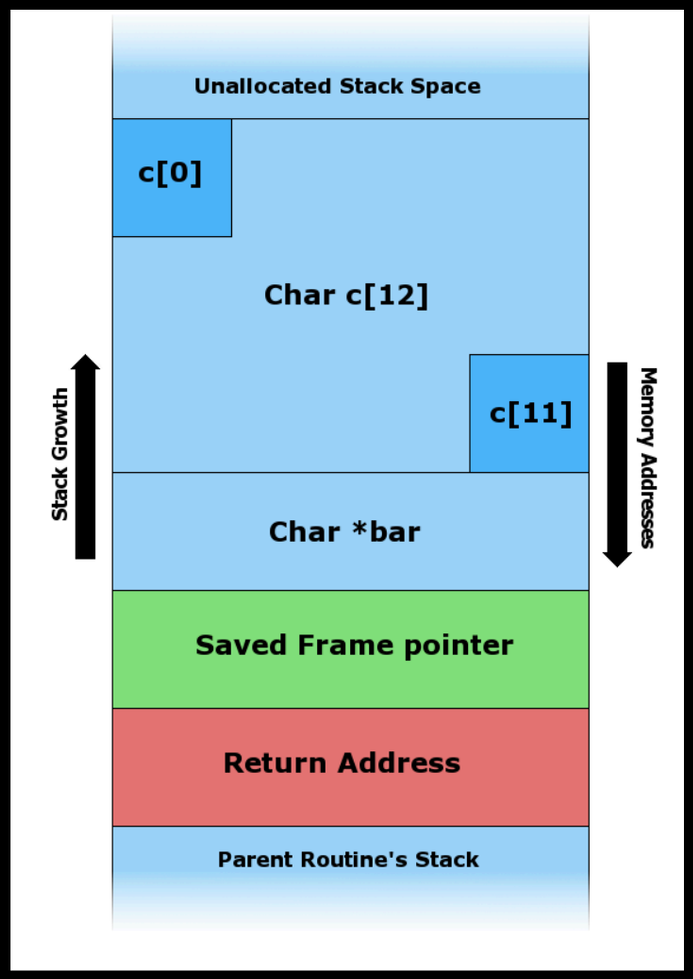
Outline

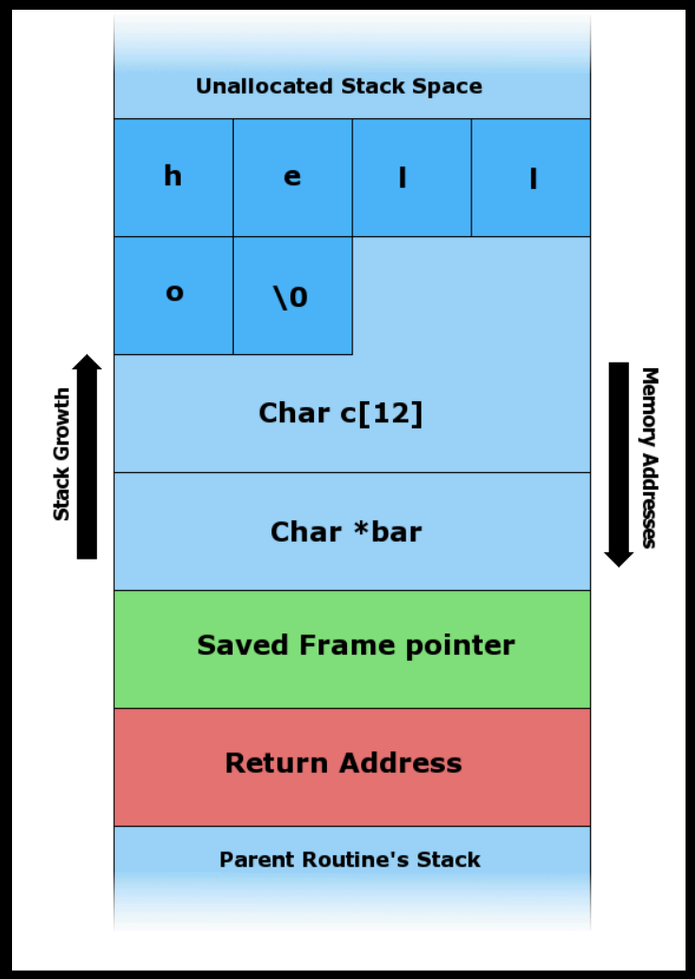
- Building **safe** and **secure** systems in Rust
- **Challenges**, **lessons** learned, and open **questions**

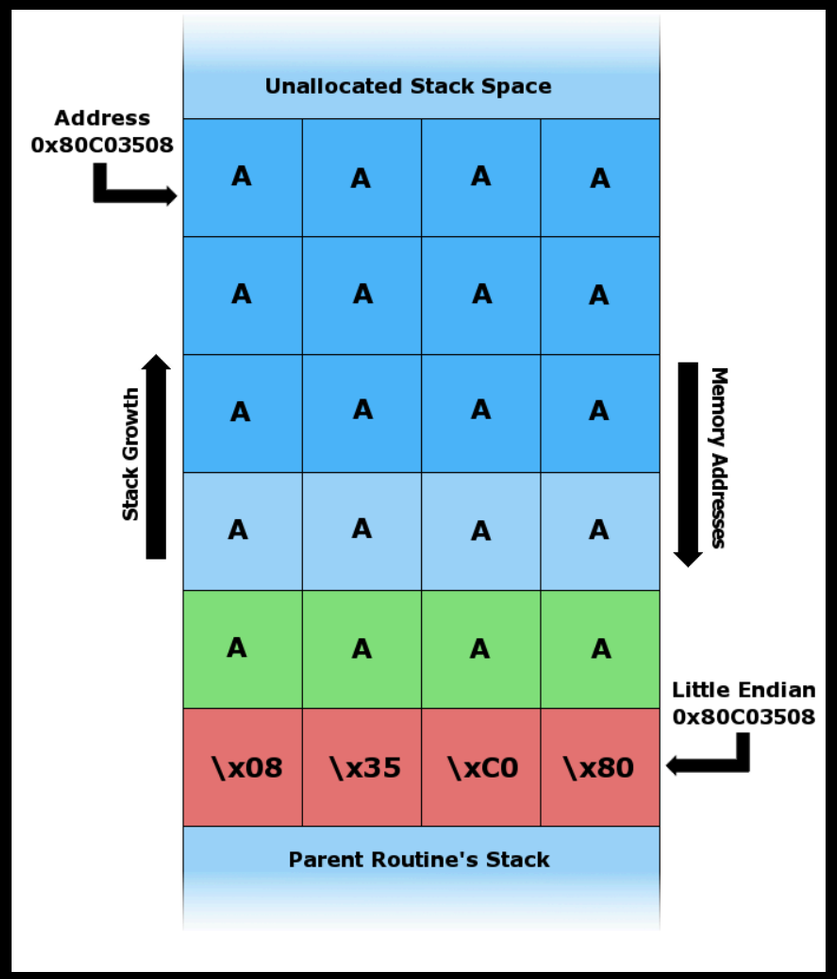
Background

- Memory-safety
- "Safe" programming language: Rust

Basic: Buffer Overflow







https://youtu.be/T03idxny9jE

Memory corruption

- "**Unsafe**" language can lead to "**undefined behavior**".

"Safe" and "unsafe", in this context, mean how the language handle memory allocation and deallocation.

Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may **execute incorrectly** (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.

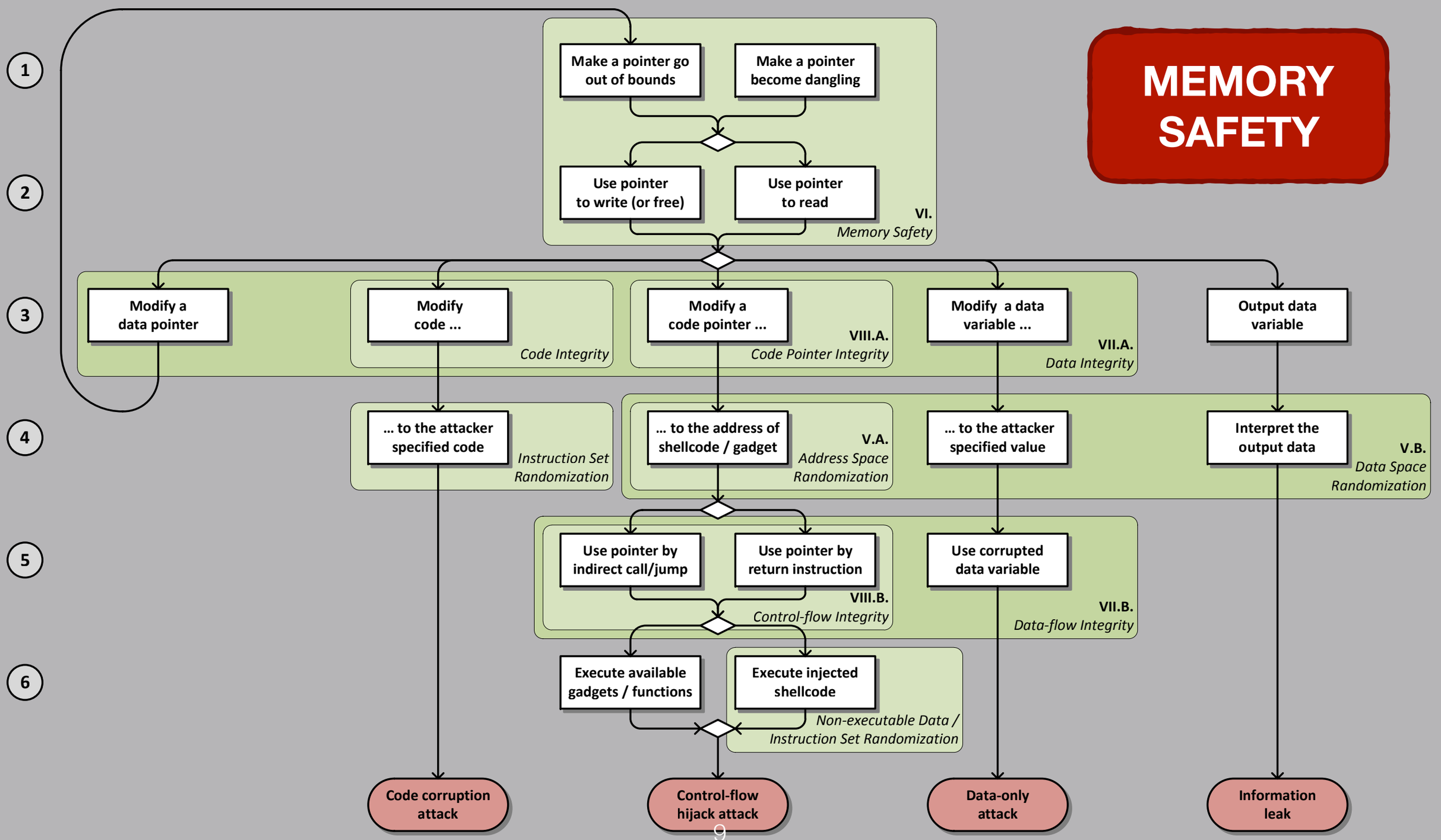
Memory corruption

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

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**~~
- **"Safe" programming languages: Rust, Go, etc**

Memory-Safe Programming Language

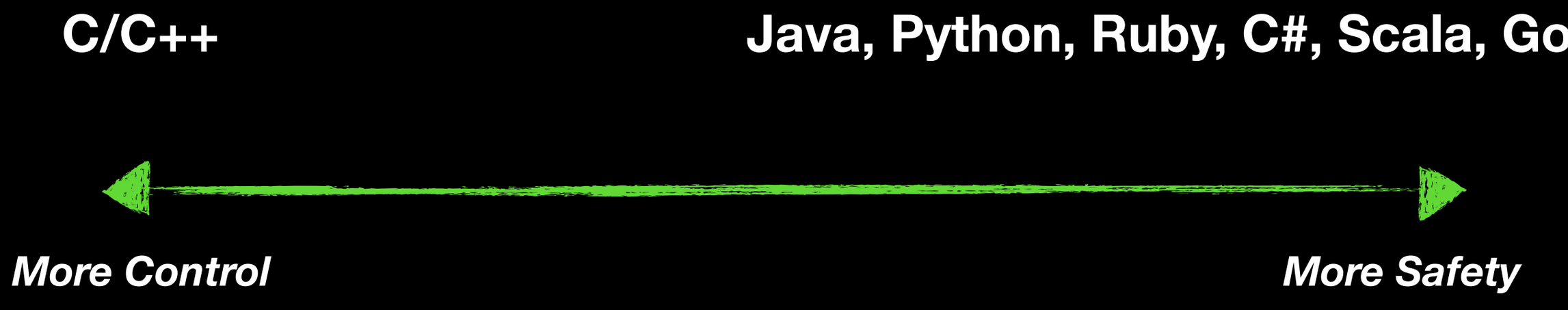
E.g., garbage collector (GC) in managed languages

- provides a **mechanism** that handles the memory deallocation which prevents **incorrect memory access**

- stack/heap overflow
- use-after-free
- double frees
- data races
- etc.

System programming

- Must be **fast** and have **minimal runtime overhead**
- Should be support **direct memory access**, but be **memory-safe**



System programming

- ✓ Restrict direct access to memory
- ✓ Run-time management of memory via periodic GC
- ✓ No explicit malloc and free, no memory corruption issues

- ✗ Overhead of tracking object references
- ✗ Program behavior unpredictable due to GC (bad for real-time systems)
- ✗ Limited concurrency (GIL typical)
- ✗ Larger code size
- ✗ VM must often be included
- ✗ Needs more memory and CPU power (i.e. not bare-metal)

C/C++

Java, Python, Ruby, C#, Scala, Go

More Control

More Safety

Observation

C++

```
void example() {
    vector<string> vector;
    …
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
}
```

use-after-free

Observation

C++

```
void example() {
    vector<string> vector;
    …
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
}
```

use-after-free

Aliasing + ***Mutation***

Hide dependencies

Cause memory to be freed.

Rust's Type System

- 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 (T)

Alice


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

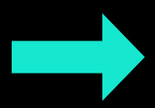
Ownership (T)

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)

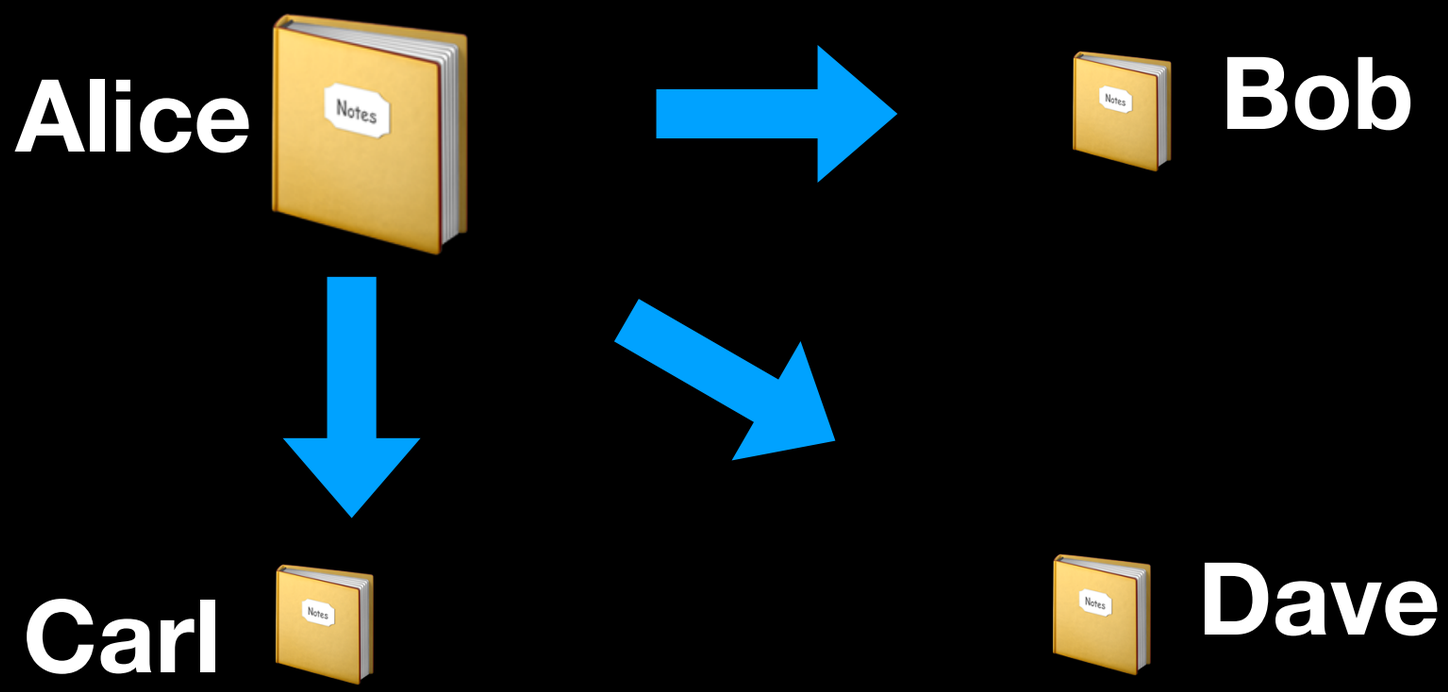
Alice

```
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"
Shared borrow	`&T`	"read-only" "shared access"
Mutable borrow	`&mut T`	"mutable" "exclusive access"

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

Building Safe and Secure Systems in Rust

- **Safe**: safe memory access, safe concurrency
- **Secure**: less vulnerabilities, reduce attack surfaces

Building Safe and Secure Systems in Rust

- **operating system**: TockOS, RedoxOS
- **compiler**: Rust
- **network service**: DNS, TLS, web server, etc.
- **database**: TiKV
- **browser**: Servo, CSS engine, etc.

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
- **MesaPy**: secure and fast Python based on PyPy
- **Rust SGX SDK**: provides the ability to write Intel SGX applications in Rust
- and many more ...

Challenges, Lessons Learned, and Open Questions

Challenges

- Rust language and ecosystem
- Unsafe Rust
- Foreign Function Interface (FFI)
- Challenges in hybrid memory model

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

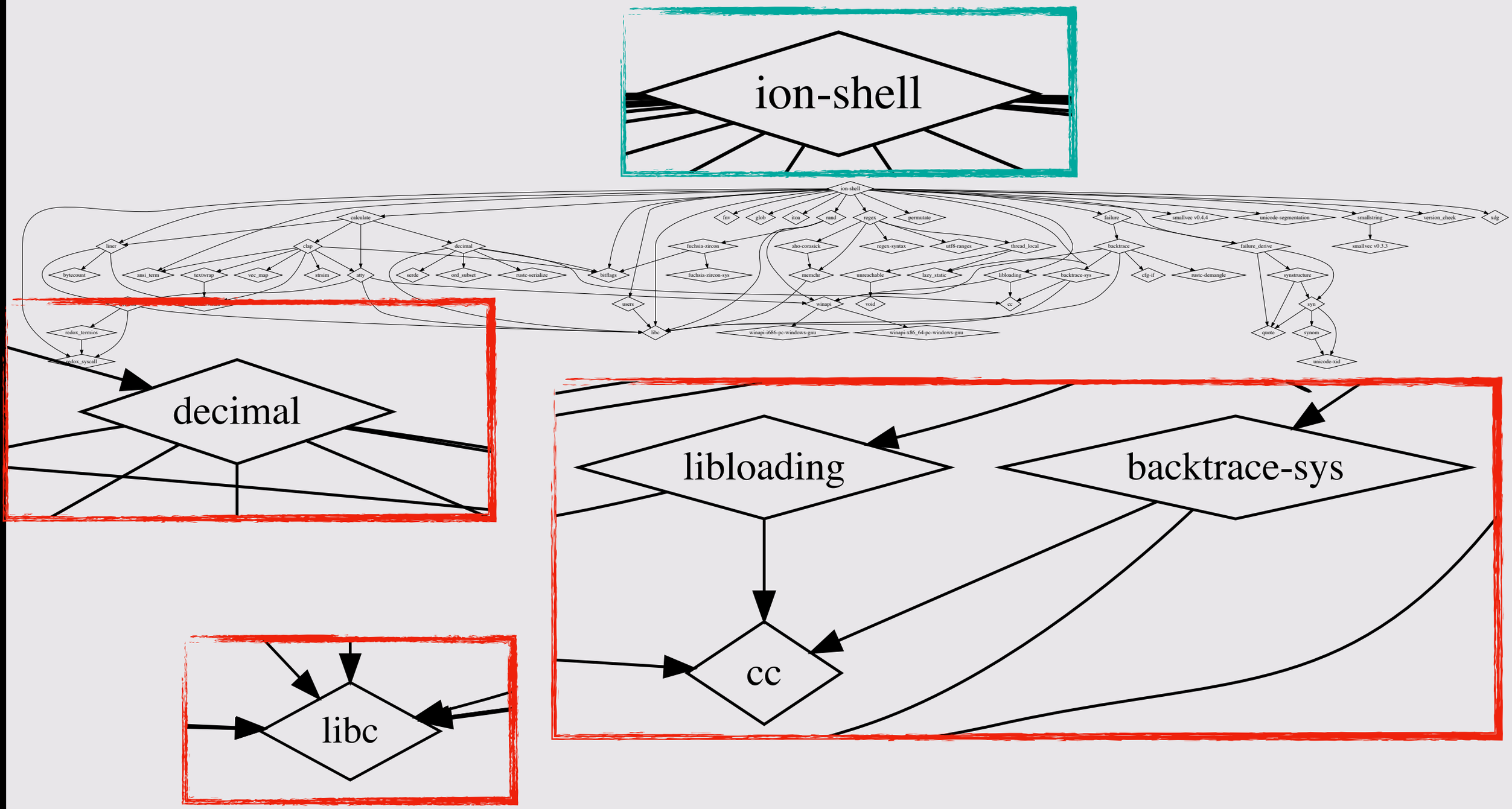
```
Library:

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.

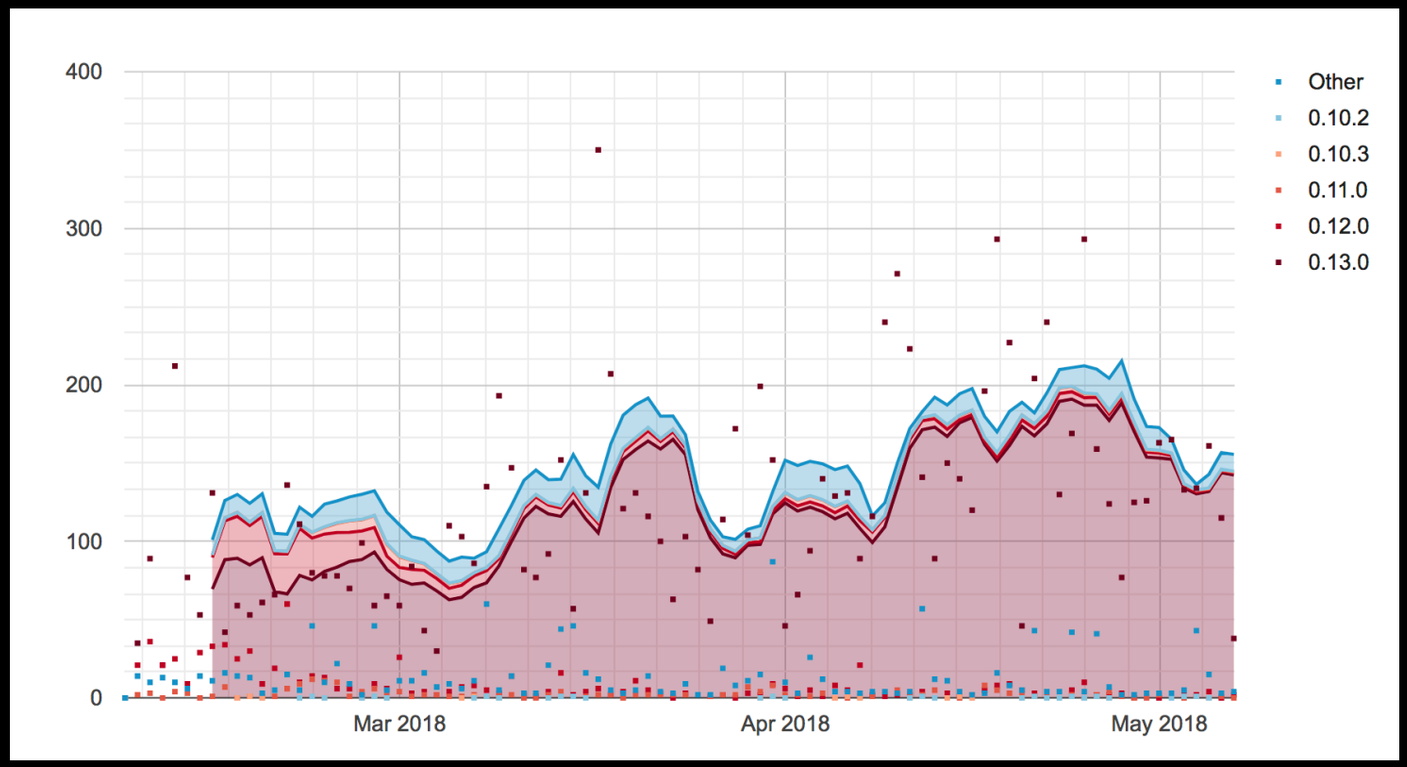
```
running: "cc" "-O0" "-ffunction-sections" "-fdata-sections"
"-fPIC" "-g" "-m64" "-I" "decNumber" "-Wall" "-Wextra" "-
DDECLITEND=1" "-o" "/Users/mssun/Repos/ion/target/debug/
build/decimal-b8ff0faecf5447ab/out/decNumber/decimal64.o" "-
c" "decNumber/decimal64.c"
```

- 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
```

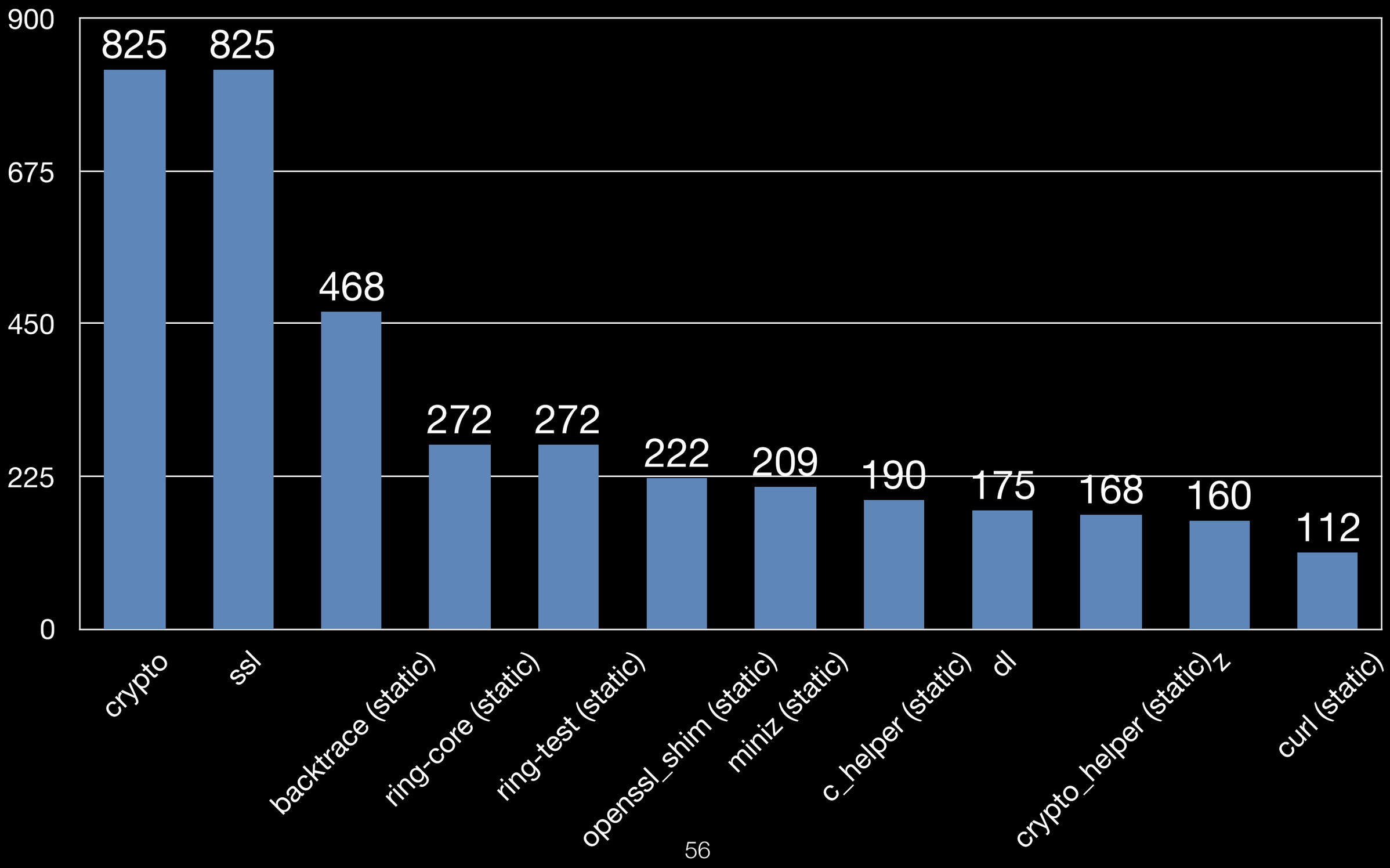
static-linked SQLite

- 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 (>= 100)



“unsafe” code

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

Lessons Learned

- Unsafe in the XML library
- CVEs in the Rust standard library

Lessons Learned

- Unsa
- CVEs

CVE-2018-1000657: a Buffer Overflow vulnerability in std::collections::vec_deque::VecDeque::reserve() function that can result in Arbitrary code execution

CVE-2018-1000810: The `str::repeat` function in the standard library allows repeating a string a fixed number of times, returning an owned version of the final string. The capacity of the final string is calculated by multiplying the length of the string being repeated by the number of copies. This calculation can overflow, and this case was not properly checked for.

The rest of the implementation of `str::repeat` **contains unsafe code that relies on a preallocated vector having the capacity calculated earlier**. On integer overflow the capacity will be less than required, and which then writes outside of the allocated buffer, leading to buffer overflow.

Lessons Learned

- Unsafe in the XML library
- CVEs in the Rust standard library
- Unsafe in actix
- FFI (Foreign Language Interface) in the miniz_oxide library
- FFI in Firefox Quantum

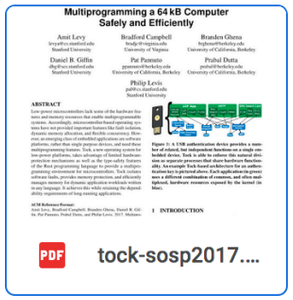
Open Questions

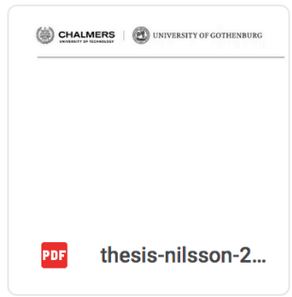
- C to Rust **translation**
- Safety and security in the Rust **compiler** and **std**
- Unsafe Rust **code analysis**
- Rust unsafe code **sandbox** and **isolation**
- **Formal verification** of Rust and its libraries
- Memory-safety across various **boundaries**

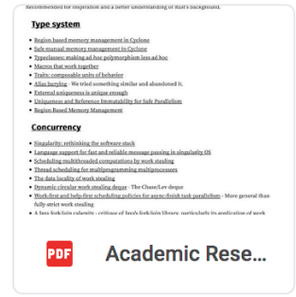
Papers

Files

Name

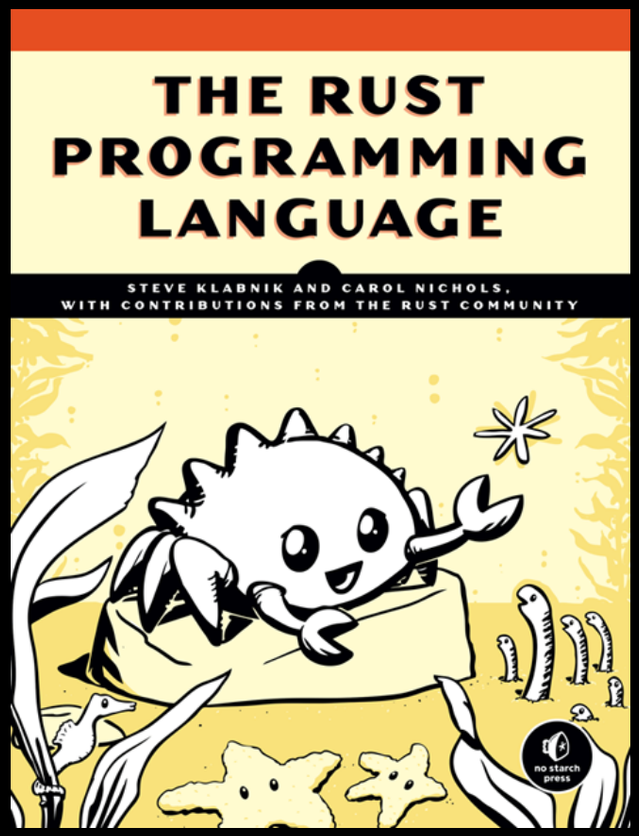


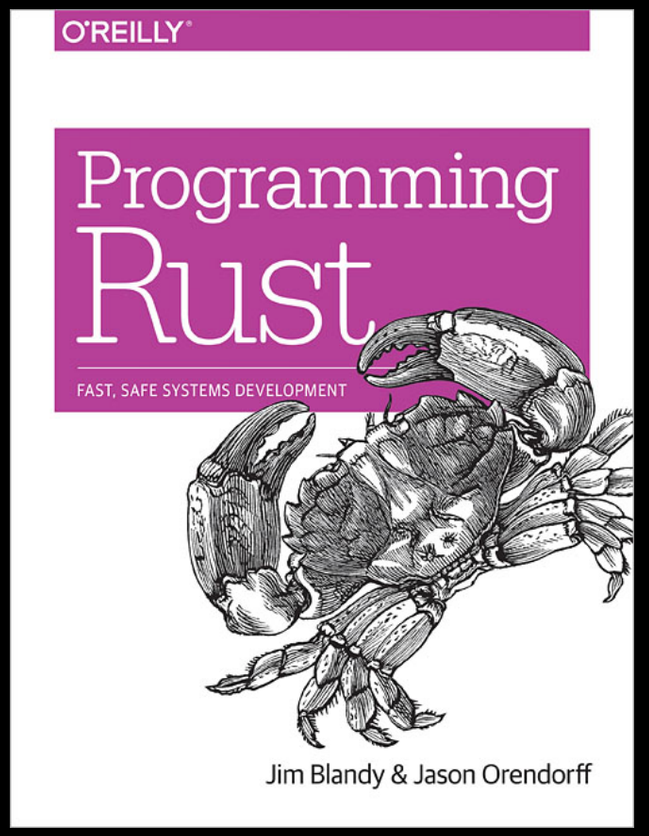




Books

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





Conclusion

- Building safe and secure systems in Rust
- Challenges, lessons learned, and open questions

Questions?