When Memory-safe Languages Become Unsafe

Mingshen Sun, Yulong Zhang, Tao Wei
Baidu X-Lab

DEF CON China
May, 2018

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
- Maintainer of **MesaLock Linux**: a memory-safe Linux distribution, **TaintART**, etc.
- `mssun @ GitHub | https://mssun.me`

Outline

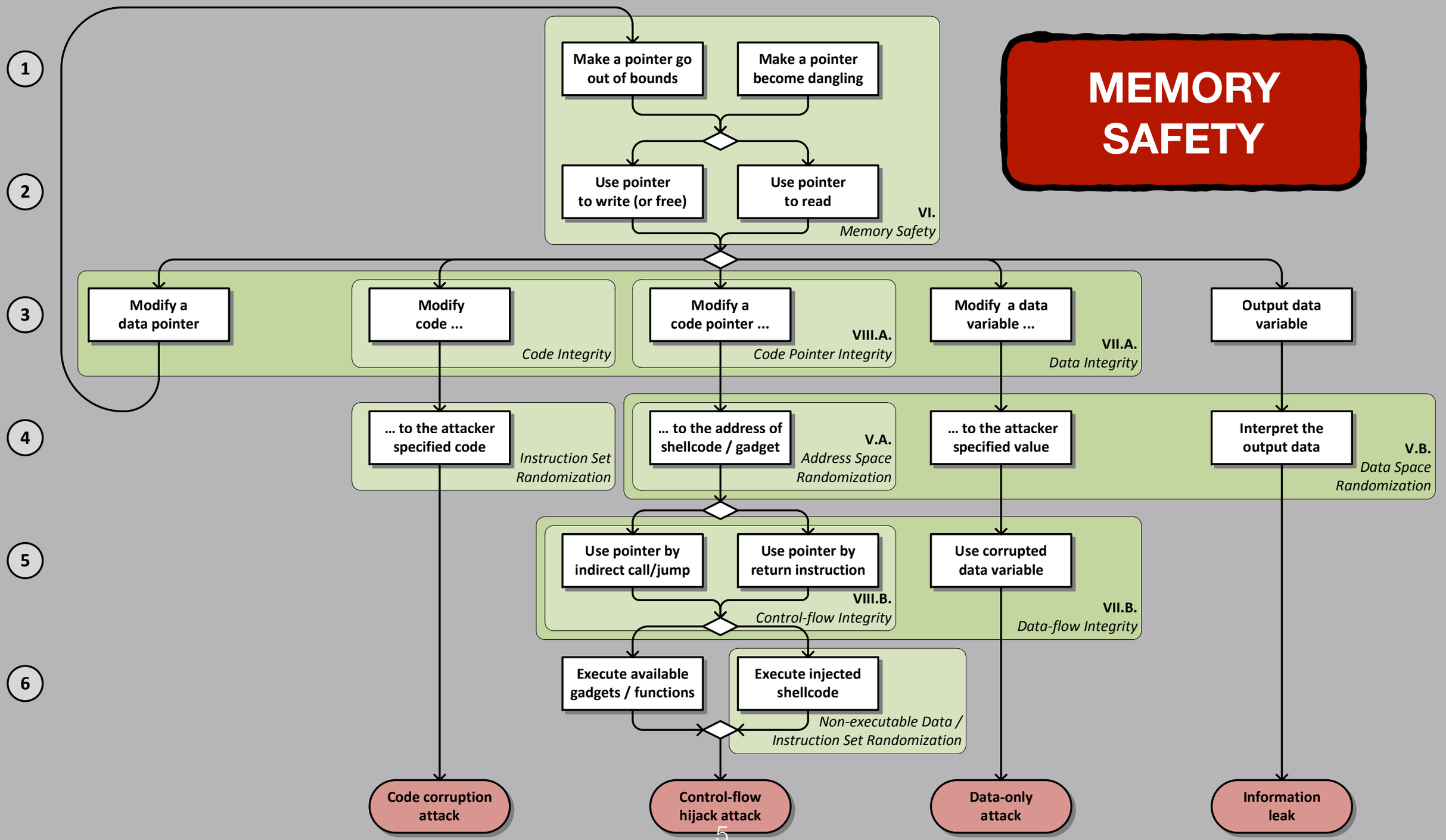
- Memory corruption and memory safety
- Memory-safe programming languages
- When memory-safe programming languages become unsafe
- Guideline of using “unsafe” code
- Conclusion

Memory Corruption and Memory Safety

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

- Code corruption attack
- Control-flow hijack attack
- Data-only attack
- Information leak

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
- **Programming languages: Rust, Go**

Memory-safe Programming Languages

- Garbage-collected memory
  - **Go** is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the **efficiency and safety** of a statically typed, compiled language.
- Ownership/Borrowing memory model
  - **Rust** is a **systems programming language** that runs blazingly **fast**, prevents segfaults, and guarantees **thread safety**.

Rust's Ownership & Borrowing Memory Model

~~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

Use After Free in C/Rust

C/C++

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

    free(mem);

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

Rust

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

Compile a UAF toy example in Rust

```
error[E0597]: `new_mem` does not live long enough
 --> src/main.rs:6:20
  |
6 |         mem_ref = &new_mem;
  |                   ^^^^^^^ borrowed value does not live long enough
7 |     }
  |     - `new_mem` dropped here while still borrowed
8 |     println!("name is {}", &mem_ref);
9 | }
  | - borrowed value needs to live until here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.
error: Could not compile `uaf`.
```

Rewrite in Rust

- **Browser**: Servo, Firefox
- **OS kernel**: Redox OS kernel, Tock OS kernel
- **Cryptocurrencies**: parity
- **System tools**: coreutils, ion shell

MesaLock Linux: a Memory-safe Linux Distribution

- Linux distribution which aims to provide a **safe** and **secure** user space environment

- **reduces attack surfaces** of an operating system exposed in the wild, leaving the remaining attack surfaces auditable and restricted

- substantially improve the security of the **Linux ecosystem**

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

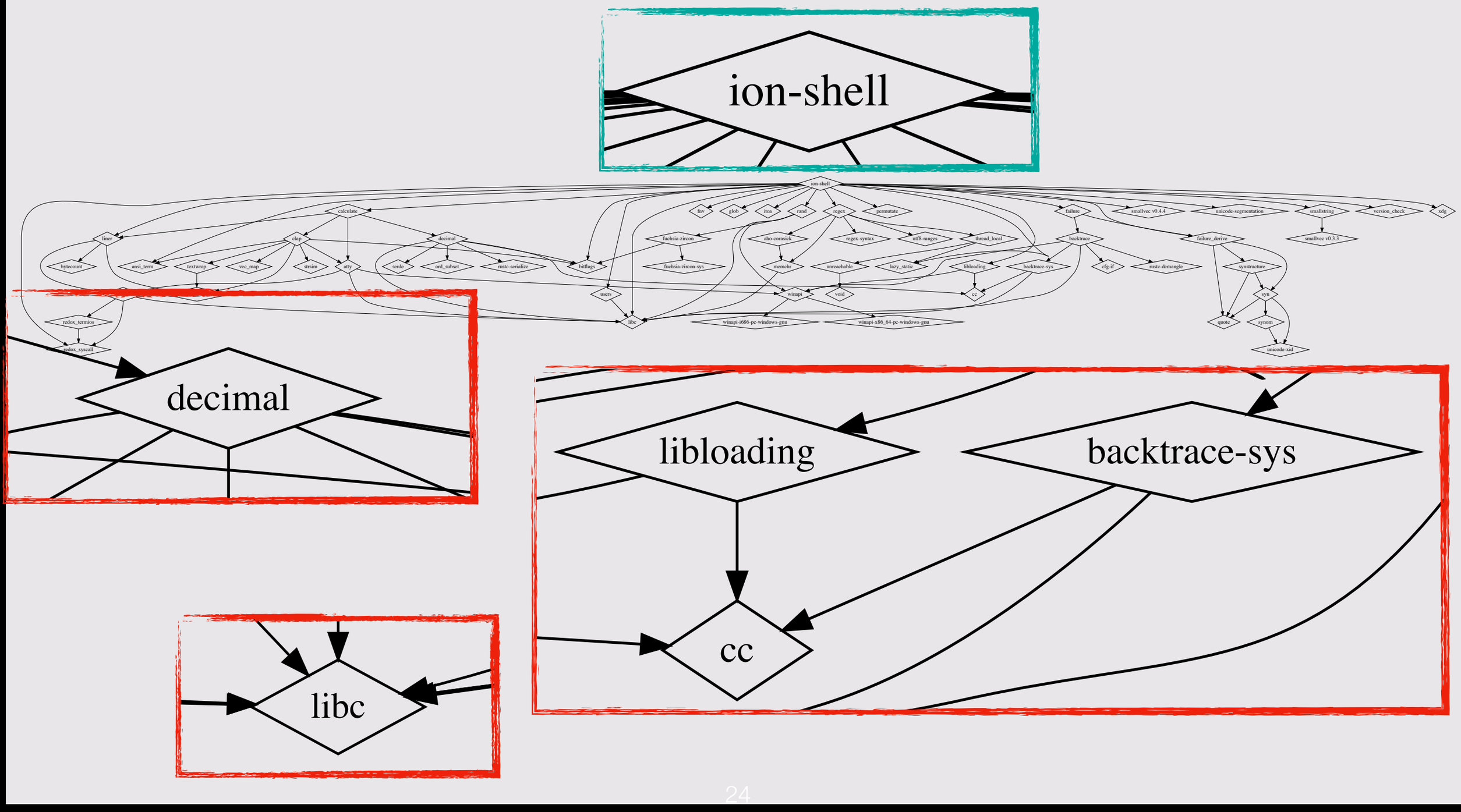
```
Library:

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.

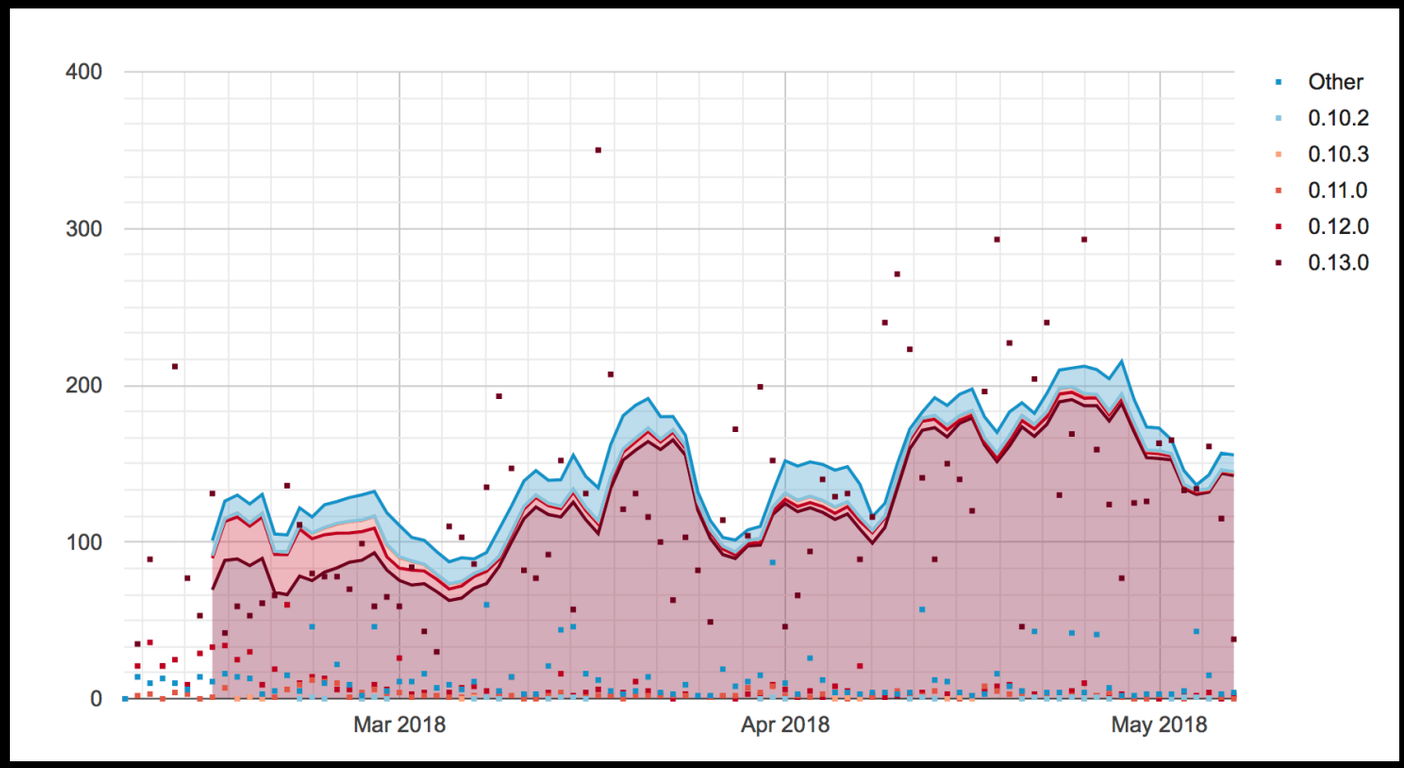
```
running: "cc" "-O0" "-ffunction-sections" "-fdata-sections"
"-fPIC" "-g" "-m64" "-I" "decNumber" "-Wall" "-Wextra" "-
DDECLITEND=1" "-o" "/Users/mssun/Repos/ion/target/debug/
build/decimal-b8ff0faecf5447ab/out/decNumber/decimal64.o" "-
c" "decNumber/decimal64.c"
```

- 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

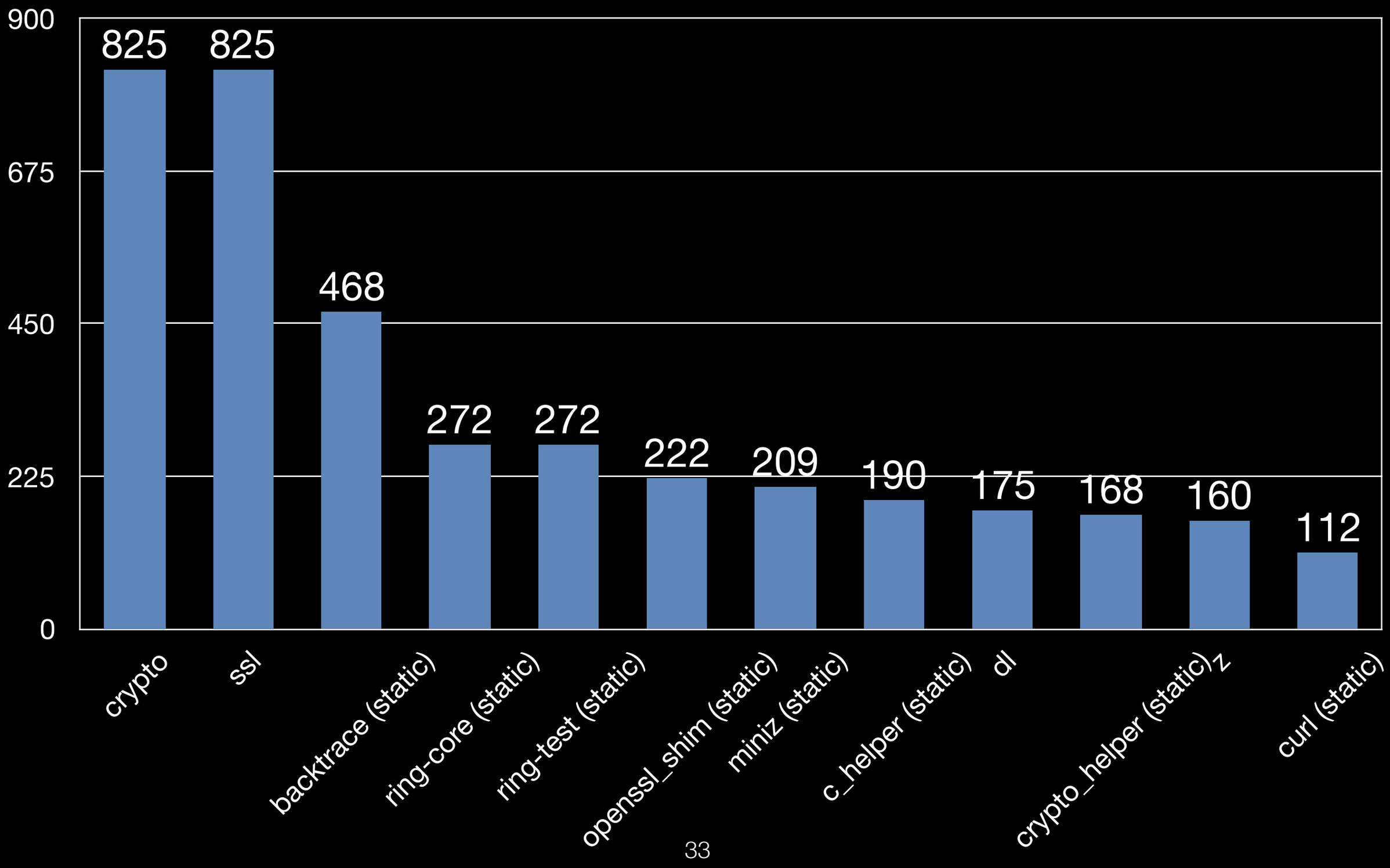
Data Collection and Study

- 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.

Lesson Learned

- Using Rust != memory-safety
- Use unsafe Rust carefully
- Don't forget to review your dependencies

Conclusion

- Memory corruption and memory safety
- Memory-safe programming languages
- When memory-safe programming languages become unsafe
  - external C/C++ libraries
  - unsafe keywords
- Guideline of using “unsafe” code

Questions?