当内存安全变得不再安全

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

DEF CON China
May, 2018

whoami

- 百度安全实验室，高级安全研究员
- PhD，香港中文大学
- 系统安全、移动安全、IoT 安全、车辆安全
- **MesaLock Linux**（**内存安全的 Linux 发行版**）、**TaintART**项目维护者, etc.
- mssun @ GitHub | https://mssun.me

内容概述

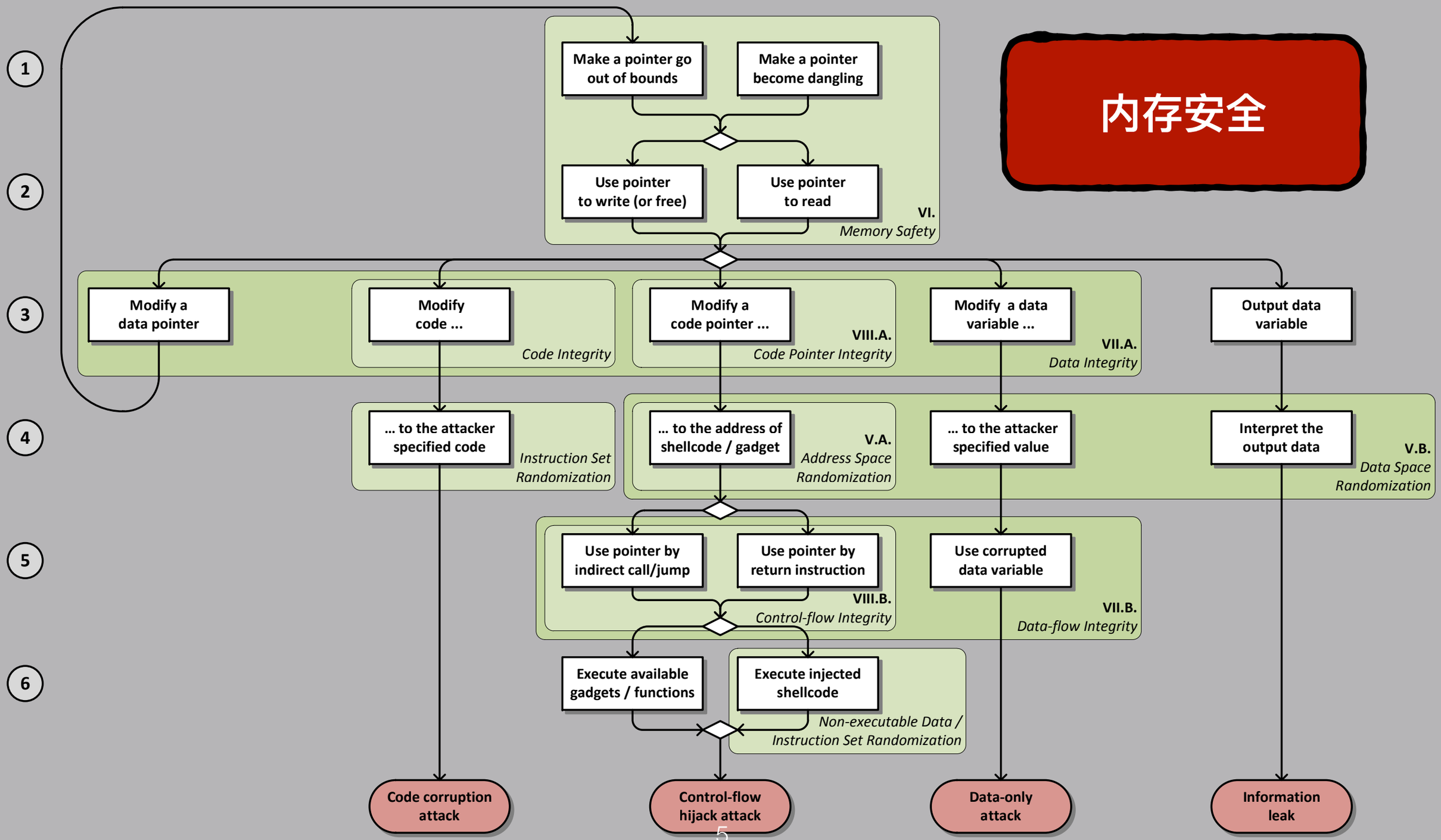
- 内存破坏以及内存安全
- 内存安全的编程语言
- 当内存安全变得不再安全
- 非内存安全的代码编写指南
- 总结

内存破坏与内存安全

- 当一段程序的内存在运行时被修改，并且是非故意的修改，我们把它**称之为内存破坏**，也称为**破坏内存安全**的行为。

- 代码破坏攻击
- 控制流劫持攻击
- 数据攻击
- 内存信息泄漏

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



缓解内存破坏的一些方法

- 程序分析例如符号执行：KLEE
- 内存检查虚拟机：Valgrind
- 编译器检查：AddressSanitizer
- Fuzzing: AFL, libFuzzer
- **编程语言: Rust, Go**

内存安全的编程语言

- 基于垃圾回收器的内存管理
  - **Go** 语言结合了解释的动态类型语言的方便性、与静态类型编译的**高效和安全性**。
- 所有权 (ownership) / 借用 (borrowing) 的内存模型
  - **Rust** 语言是一个**系统编程语言**，运行速度快，避免 segfaults 并且保证**线程安全**。

所有权 (ownership) / 借用 (borrowing) 的内存模型

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

- 编译器强制规则:
  - 每个资源有一个**拥有者**
  - 其他人只能有限制的**借用**这个资源（e.g.，创建一个**alias**）
  - 拥有者不能**释放**已经被借用的资源

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`.
```

使用 Rust 重写

- **浏览器**: Servo, Firefox
- **操作系统内核**: Redox OS kernel, Tock OS kernel
- **数字货币**: parity
- **系统工具**: coreutils, ion shell

MesaLock Linux: 一个内存安全的 Linux 发行版

- 用 Rust、Go 等内存安全语言重写用户空间应用 (user space applications)，以在用户空间中逐步消除高危的内存安全漏洞
- 在用户空间中逐步消除高危的内存安全漏洞，并且使得剩余的攻击面可审计、可收敛
- 实质性地提升 Linux 生态的安全性

我们在构建 MesaLock Linux 项目的时，使用 Rust 语言解决了内存安全的问题，除了语言层面的有点，Rust 同时提供了：

- 大量的库（crates）
- 繁荣的生态
- 对于老旧模块的重写

但是在使用之前，我们需要对 Rust 有一个深刻的理解。

内存安全？嗯...

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

什么是非内存安全的 Rust?

- 目前，我们讨论的 Rust 的代码都能保证在编译时强制内存安全
- 但是，Rust 也包括另外一个并没有强制内存安全的保证的语言：非内存安全的 Rust。它和普通的 Rust 语法相同，但是有一些额外的**超能力**。

非内存安全的超能力

1. 解引用原始指针
2. 访问或修改静态可变变量
3. 调用非内存安全的方法和函数
4. 实现非内存安全的 trait

非内存安全的超能力

1. 解引用原始指针

Rust

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

任意读写内存地址。

非内存安全的超能力

2. 访问或修改静态可变变量

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); }
}
```

数据竞争。

Unsafe Superpowers

3. 调用非内存安全的方法和函数

Rust

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

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

调用非内存安全的函数，触发未定义的行为（内存破坏）。

Unsafe Superpowers

3. 调用外部非内存安全的方法和函数

Rust

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

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

调用外部 C/C++ 函数，触发未定义的行为（内存破坏）。

"非内存安全代码" 是不容易感知

- **Rust 开发者**: 没什么太大问题。至少“非内存安全的代码”需要使用 unsafe 关键字**显式的调用**。我知道哪段代码是安全的哪段代码可能有安全问题。

- **Me**: 错。非内存安全的代码可能存在于依赖的库中。你是否有检查过依赖库的代码?

"非内存安全代码" 是不容易感知

Rust

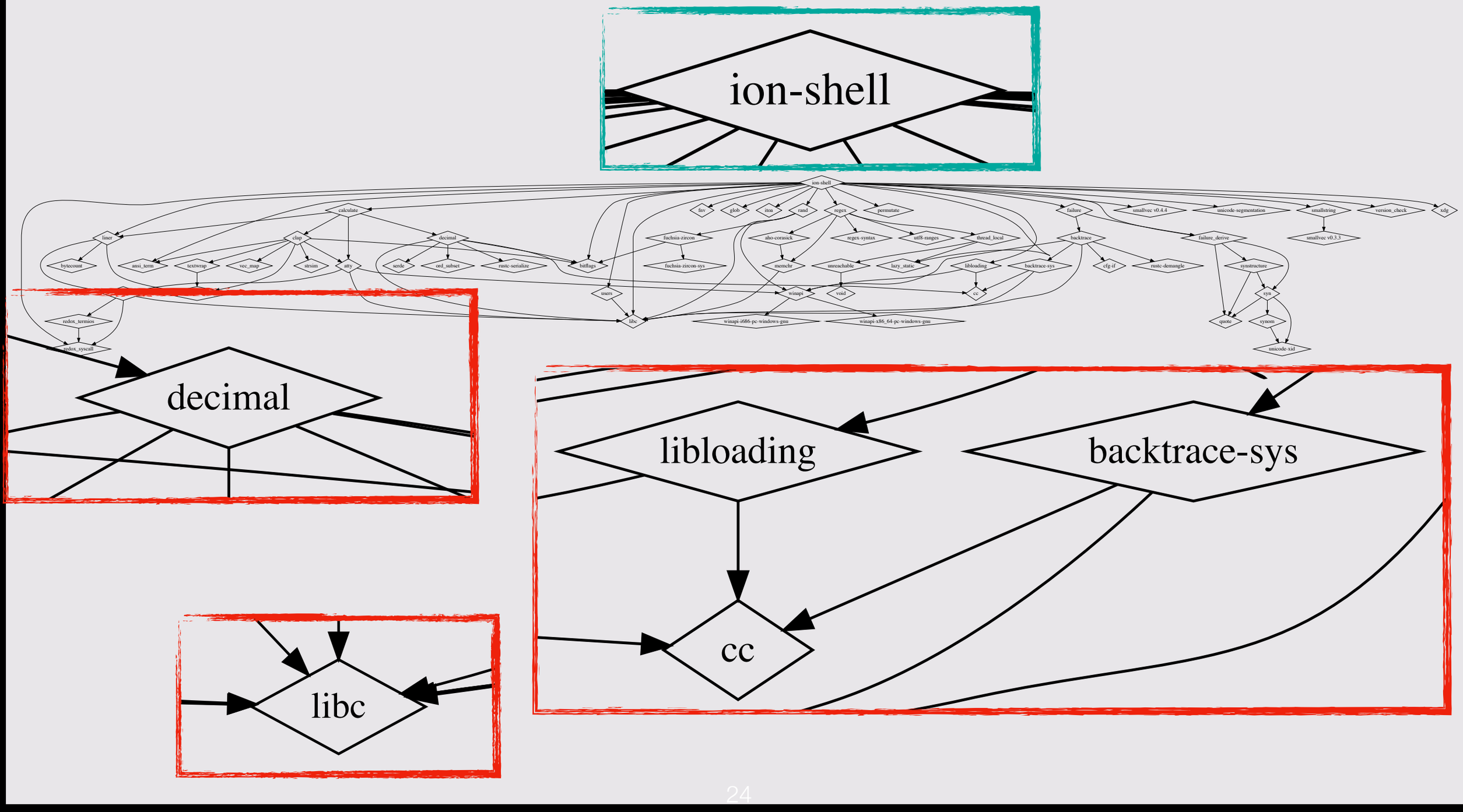
```
Library:

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

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

Developer:

fn main {
    safe_function();
}
```

一些库（包括标准库）包装了非内存安全的代码，
重写导出为安全的函数。

案例研究: Ion Shell

- Ion 是一个现代的系统 shell，简单并强大的语法。所有代码都由 Rust 重写编写，**提高了 shell 的内存安全**。除此之外， Ion shell 比 Dash 的运行速度要快。

Ion shell 依赖库的关系图



Ion Shell 中的 C 库

- 链接的 C 库
  - glibc
  - decimal
  - libloading
  - backtrace-sys
- cc crate?
  - 编译 C 的代码，链接进入 Ion shell

cargo build -vv

- 使用 verbose 重新编译 ion shell.

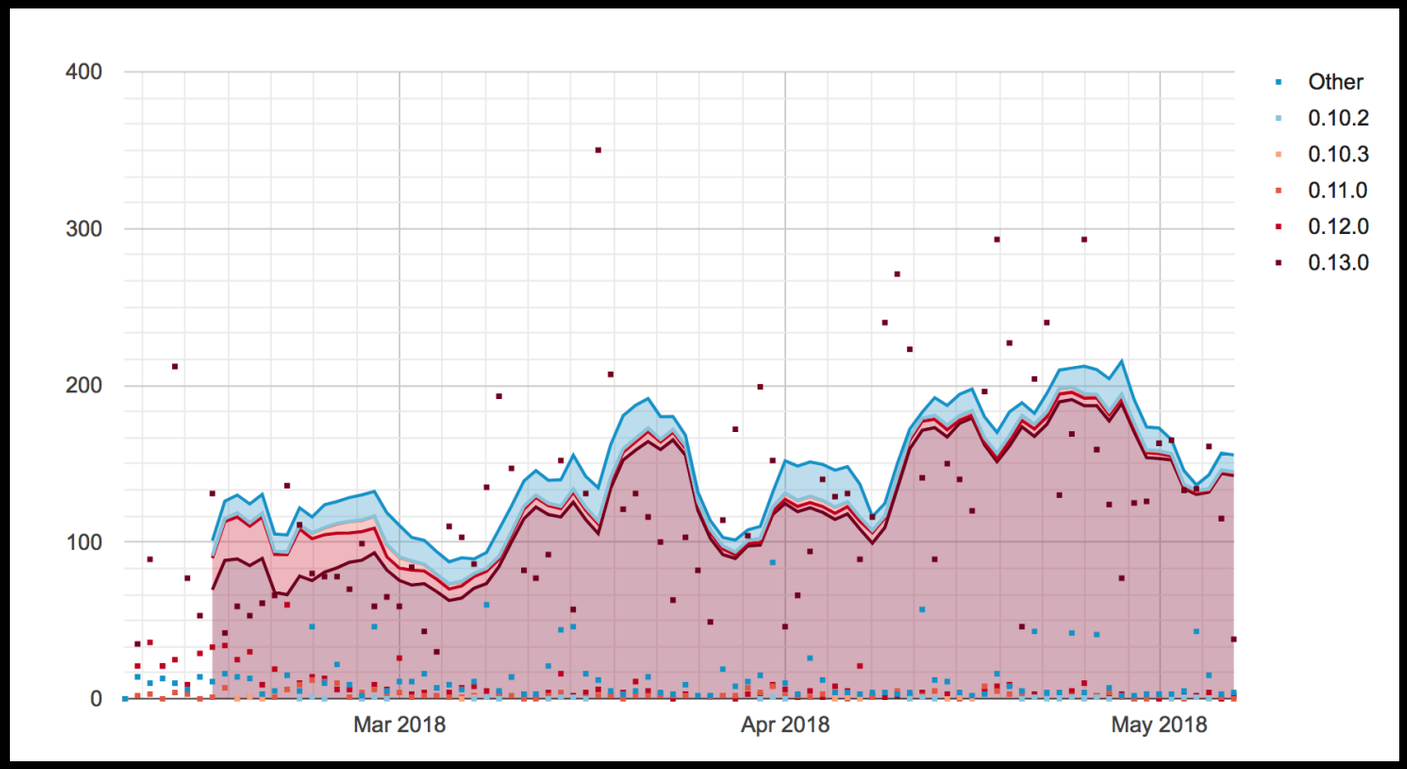
```
running: "cc" "-O0" "-ffunction-sections" "-fdata-sections"
"-fPIC" "-g" "-m64" "-I" "decNumber" "-Wall" "-Wextra" "-
DDECLITEND=1" "-o" "/Users/mssun/Repos/ion/target/debug/
build/decimal-b8ff0faecf5447ab/out/decNumber/decimal64.o" "-
c" "decNumber/decimal64.c"
```

- decimal crate: 小数浮点数数学计算库，基于 C decNumber 库。 (http://speleotrove.com/decimal/decnumber.html)

- Ion shell 基于浮点数数学计算库，仍然存在潜在的内存安全问题。

案例研究: rusqlite

- rusqlite 是一个 Rust 库提供了 SQLite 相关的 API
- 本质是一个 SQLite API 的包装库
- 共计 38 个 Rust crates 直接依赖 rusqlite
- 200 下载量/天



rusqlite 库的内存破坏问题

- 我们使用了 SQLite 的类型混淆 bug (CVE-2017-6991)
- 能够触发 Rust 库的内存破坏

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
```

静态链接 SQLite

- rusqlite 库中包含 sqlite3.c 文件
- 静态链接入使用 rusqlite 的库和工具
- 没有与上游 SQLite 代码版本保持一致

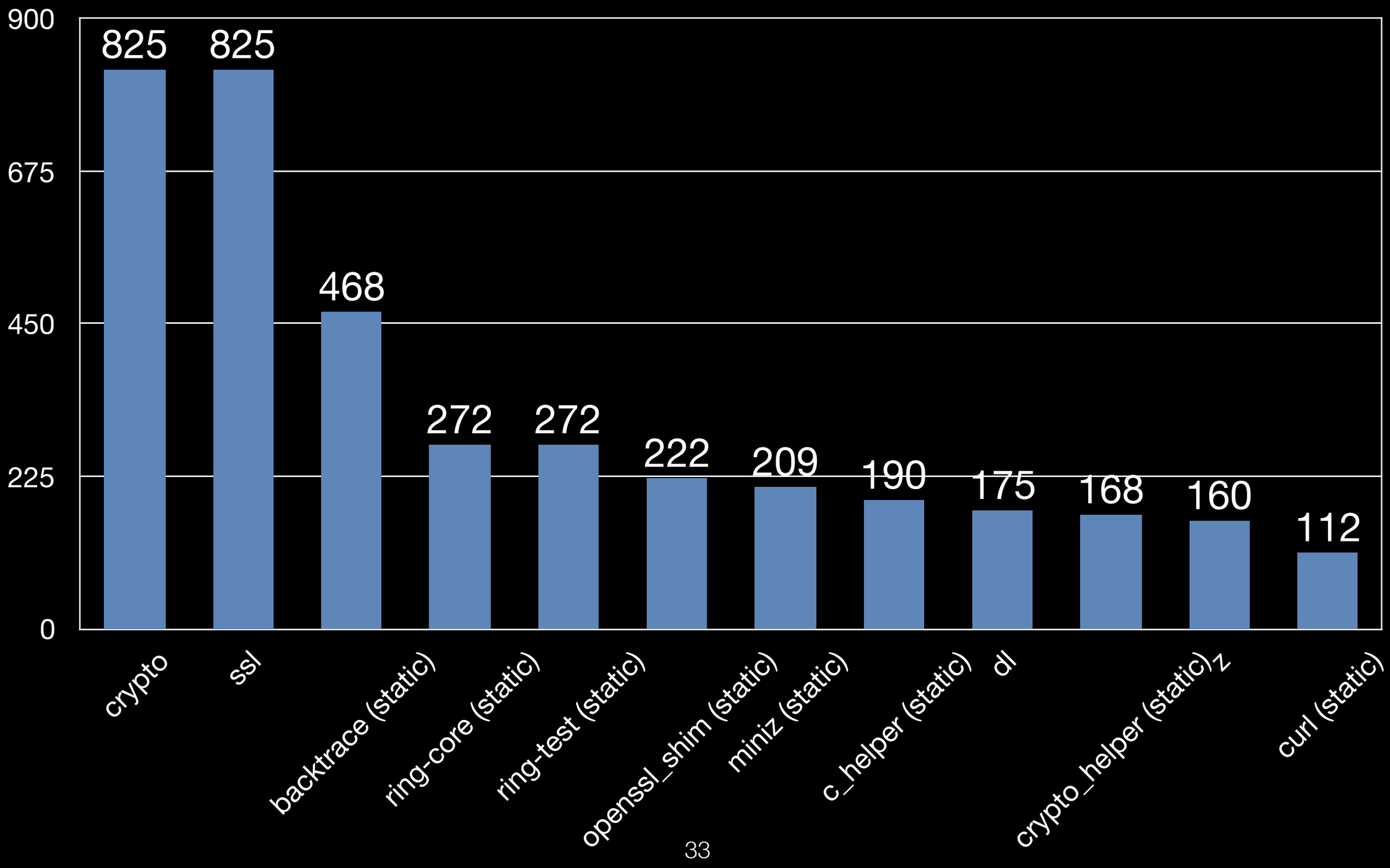
数据收集与研究

- 10,693 Rust 库（crates.io）
- 2 亿下载量
- 从两个方面分析
  - 外部 C/C++ 库的使用情况
  - usafe 关键字的使用情况

外部 C/C++ 库的使用情况

- build.rs 文件: 编译第三方由非 Rust 代码的编译脚本，如：编译外部的 C/C++ 库
- 尝试编译所有收集的 Rust 库
- 分析编译器的编译 log
  - 使用 build.rs 编译 C/C++ 源代码
  - 静态/动态链接编译好的库或者系统库

外部 C/C++ 库的使用情况 (>= 100)



分析 unsafe 代码

- 使用 Rust 编译器生成 AST (abstract syntax tree)
- 在 AST 中找到使用 unsafe 关键字的代码

“unsafe” 代码使用情况

- **3,099** / 10,693 Rust 库 (crates) 包含 unsafe 代码
- **14,796** 文件
- **651,193** 行代码

Fuzz Rust 库

- cargo-fuzz
- 通过 fuzz 发现的 Use after Free 问题：XML Document (https://github.com/shepmaster/sxd-document/issues/47)
- src/string_pool.rs 文件使用了大量的 unsafe 代码, unsafe 代码会破环函数中变量（数据/资源）的所有权和生命周期。

如何使用 “unsafe” 代码

遵循 Rust SGX SDK 项目中提出的混合代码内存安全架构三原则：：https://github.com/baidu/rust-sgx-sdk/blob/master/documents/ccsp17.pdf

1. 隔离并模块化由非内存安全代码编写的组件，并最小化其代码量。

2. 由非内存安全代码编写的组件不应减弱安全模块的安全性，尤其是公共 API 和公共数据结构。

3. 由非内存安全代码编写的组件需清晰可辨识并且易于更新。

一些经验

- 使用 Rust != 内存安全
- 小心的使用非安全 Rust
- 记得检查依赖的使用情况

总结

- 内存破坏和内存安全
- 内存安全的编程语言
- 当内存安全的编程语言变得不再安全
  - 外部 C/C++ 库的使用
  - unsafe 关键词的使用
- 如何使用 “unsafe” 代码

欢迎提问