

CCF系统软件专委会

Rust程序分析平台：问题、现状和未来

徐辉

复旦大学计算机科学技术学院



2024年8月24日

事件1: BatBadBug碰瓷Rust

关于Rust命令注入漏洞(CVE-2024-24576)的安全预警-东南大...

2024年4月11日 Rust标准库中存在命令注入漏洞(CVE-2024-24576,被称为BatBadBut),该漏洞的CVSS评分为10.0,可能在Windows系统上导致命令注入攻击,目前该漏洞的细节已公开。Rus...

东南大学网络与信息中心

别用Rust了?Win7/8/10系统中发现高危漏洞

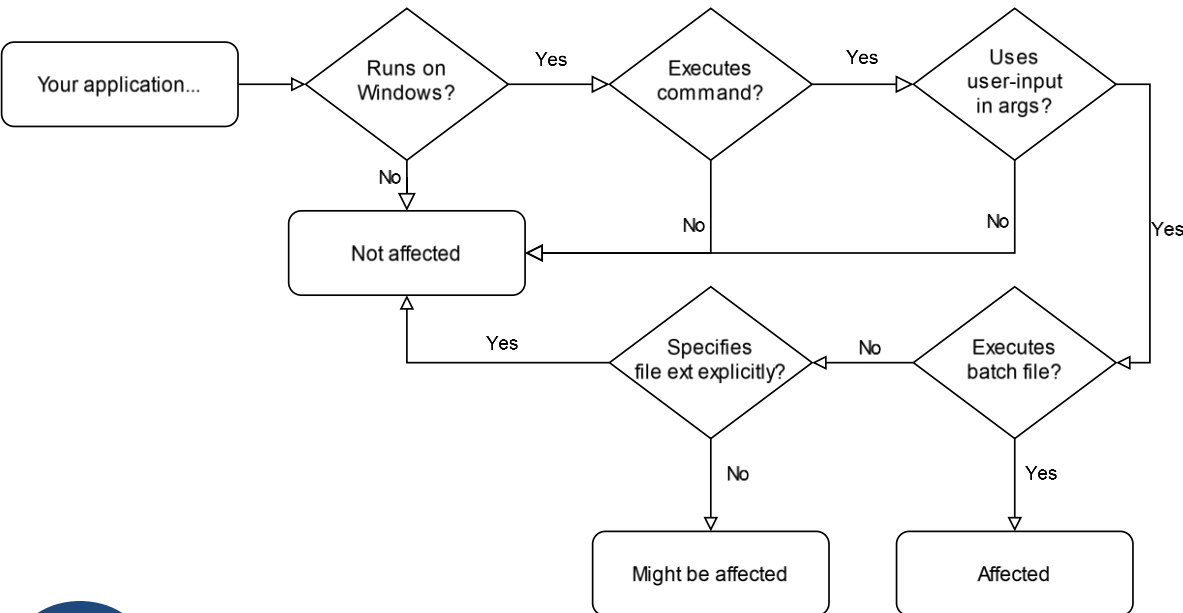


2024年4月10日 近日,安全专家发现了一个名为CVE-2024-24576的漏洞。这个漏洞存在于使用Rust编程语言开发的软件中,允许攻击者对Windows系统进行命令注入攻击。该漏洞是由于操作系统命令和参数注...

中关村在线



真相: 所有语言都面临的Windows的问题



```
use std::process::Command;

Command::new("cmd.exe")
    .args(["escape letter", "&calc.exe"])
    .spawn()
    .expect("command failed to start");
```

用户可能通过escape letters注入新的命令

事件2: Rust碰瓷CrowdStrike蓝屏



微软IT故障提醒:RUST比C/C++更好



2024年7月24日 微软Azure 的 CTOMark Russinovich表示, 开发人员应该逐步弃用 C/C++, 转而使用内存安全的 Rust 语言, 以减少系统崩溃和蓝屏死机。当然, 这条推文与 CrowdStrike 的错误更新没有直...

云云众生云原生

用Rust重写Windows,能阻止150亿美元的蓝屏惨案吗?|CrowdSt...



2024年7月27日 因此, 仅仅依靠Rust的内存安全机制, 并不能完全避免类似事件的发生。更重要的是, 此次事件的根本原因在于CrowdStrike的配置变更发布流程存在严重缺陷。根据SRE原则, 配置变更应该分阶...

新浪网

程序员激辩用Rust能否改写CrowdStrike引发的最大IT故障的...

2024年7月25日 CrowdStrike 没有对部署进行任何验证。但根据 CrowdStrike 最新更新的事后分析来看, 他们确实没有进行任何...

deepin官方论坛

避免再次全球蓝屏宕机!微软计划用Rust重构Win11内核



避免再次全球蓝屏宕机! 微软计划用Rust重构Win11内核 快科技7月30日消息, 在经历了CrowdStrike驱动程序故障引发的全球性Windows电脑蓝屏死机事件后, 微软正计划对其Windows 11内核安全性进行重大...

快科技

...受灾者仅获赔10美元引热议,程序员激辩用Rust能否改写史...



2024年7月25日 如果CrowdStrike 是用 Rust 编写的, 那确实可以降低发生故障的可能性, 但它并不能解决导致故障发生的根本原因。所以看到许多人说 Rust 是解决这次事故的唯一答案, 我就感到非常恼火...

36kr

```
EXCEPTION_RECORD: fffffb0d18d3ec28 -- (.cxr 0xfffffb0d18d3ec28)
ExceptionAddress: fffff8021df335a1 (csagent+0x00000000000e35a1)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 0000000000000000
Parameter[1]: 000000000000009c
```

```
Attempt to read from address 000000000000009c
```

```
CONTEXT: fffffb0d18d3e460 -- (.cxr 0xfffffb0d18d3e460)
rax=fffffb0d18d3f2b0 rbx=0000000000000000 rcx=0000000000000003
rdx=fffffb0d18d3f280 rsi=ffff9a81b596f9a4 rdi=ffff9a81b596605c
rip=fffff8021df335a1 rsp=fffffb0d18d3ee60 rbp=fffffb0d18d3ef60
r8=000000000000009c r9=0000000000000000 r10=0000000000000000
r11=0000000000000014 r12=fffffb0d18d3ef28 r13=fffffb0d18d3f0d0
r14=000000000000001a r15=0000000000000004
```

```
iopl=0          nv up ei pl nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00050206
```

```
csagent+0xe35a1:
```

```
fffff8021df335a1 458b08
```

```
Resetting default scope
```

```
mov     r9d,dword ptr [r8] ds:002b:00000000`0000009c=????????
```

```
BLACKBOXBSD: 1 (!blackboxbsd)
```

```
BLACKBOXNTFS: 1 (!blackboxntfs)
```

```
BLACKBOXPNP: 1 (!blackboxpnp)
```

```
BLACKBOXWINLOGON: 1
```

```
PROCESS_NAME: System
```

```
READ_ADDRESS: 000000000000009c
```

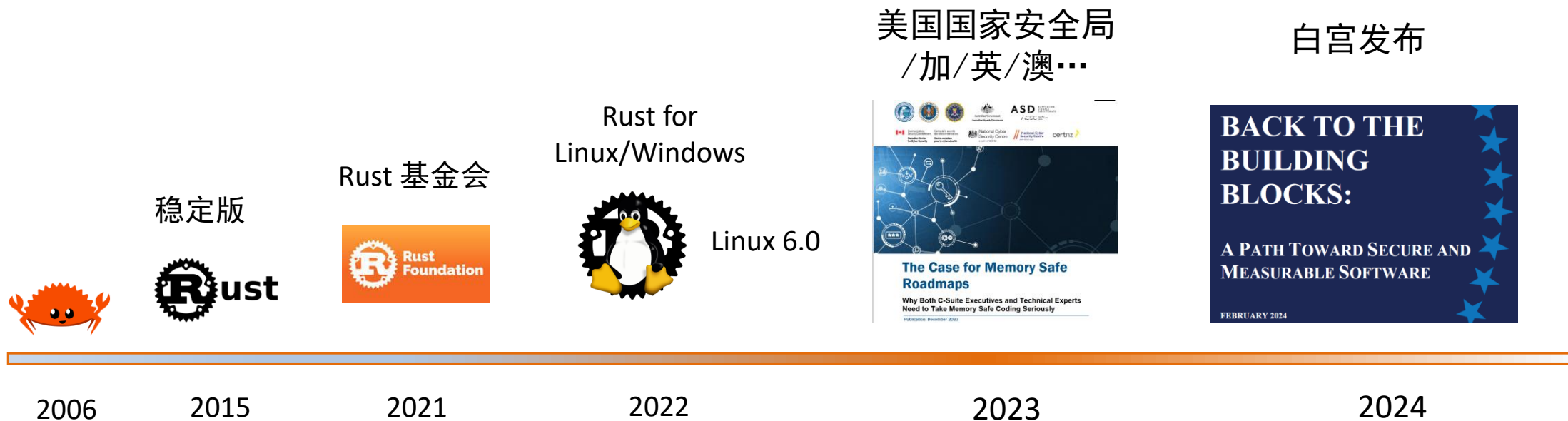
大纲

- 一、问题：背景
- 二、现状：Rust程序分析生态
- 三、未来：RAP研究思路
- 四、总结

大纲

- 一、问题：背景
- 二、现状：Rust程序分析生态
- 三、未来：RAP研究思路
- 四、总结

Rust很强大，但并未消除内存安全问题



问题关键: **Unsafe Rust!!!**

Rust带来了什么：语言设计赋能程序分析

□ 传统程序分析算法具有理论局限性

□ 无法同时保证精确性、无漏报、可终止

□ 通过限制语言表达能力，降低程序分析问题难度

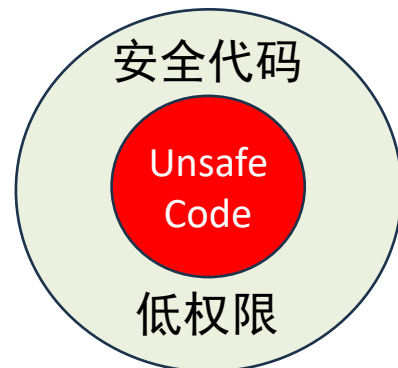
□ Rust所有权 => 避免复杂指针分析问题

□ Rust安全域思想 + 函数生命周期标注 => 分治（基于摘要的过程间分析）

$$S[[P]] = lfp \stackrel{\sqsubseteq}{=} F[[P]]$$

↓ 基于安全域实现分治

$$S[[P]] = lfp \stackrel{\sqsubseteq}{=} F[[P]] [lfp \stackrel{\sqsubseteq}{=} {}^1 F[[P_1]], \dots, lfp \stackrel{\sqsubseteq}{=} {}^n F[[P_n]]]$$




Rust特性举例： 函数标注

□ 生命周期约束

- 避免跨函数分析
- 编译器检查

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



□ 安全性约束

- 避免unsafe语义分析
- 开发者保障安全
- Interior Unsafe

```
impl<T> Vec<T> {  
    pub fn push(&mut self, value: T) {  
        if self.len == self.buf.capacity() {  
            self.buf.reserve_for_push(self.len);  
        }  
        unsafe {  
            let end = self.as_mut_ptr().add(self.len);  
            ptr::write(end, value);  
            self.len += 1;  
        }  
    }  
}
```


Rust特性举例：唯一可变引用原则（XOR Mutability）

□ 所有权 + 借用检查 => 避免UAF/DF缺陷

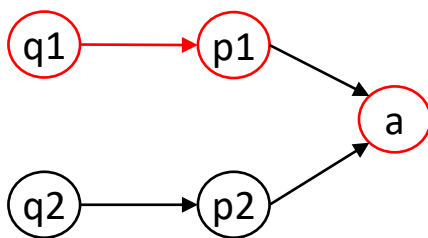
```
fn main(){
  let mut alice = 1;
  let bob = &mut alice;
  println!("alice:{}", alice);
  println!("bob:{}", bob);
}
```

bob

alice

□ 无需追踪多级可变指针

```
let mut a = 1;
let mut p1 = &a;
let p2 = &a;
let mut q1 = &mut p1;
let q2 = &p2;
```



- 可变值
- 不可变值
- 不可变借用
- 可变借用

做Rust程序分析研究的意义

- 提供Rust编译器无法保障的安全检查功能
 - 允许误报
- 发现好的语言设计，降低unsafe的影响
 - 语言设计赋能程序分析

大纲

- 一、问题：背景
- 二、现状：Rust程序分析生态
- 三、未来：RAP研究思路
- 四、总结

当前主要的Rust程序分析工具

□形式以Cargo插件为主，发布在crates.io

	Dynamic Checker (面向用例)	Static Analyzer (面向模式)	Model Checker (面向属性)
官方工具	Miri	Clippy	
第三方工具		Rudra lockbud MIRAI SafeDrop rCanary	Kani Verus Prusti

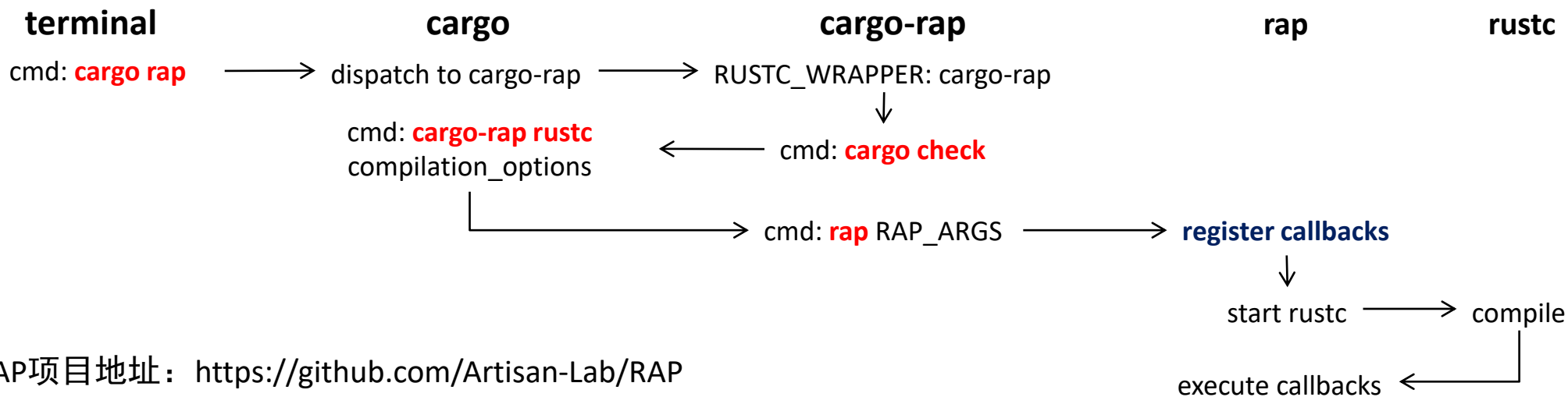
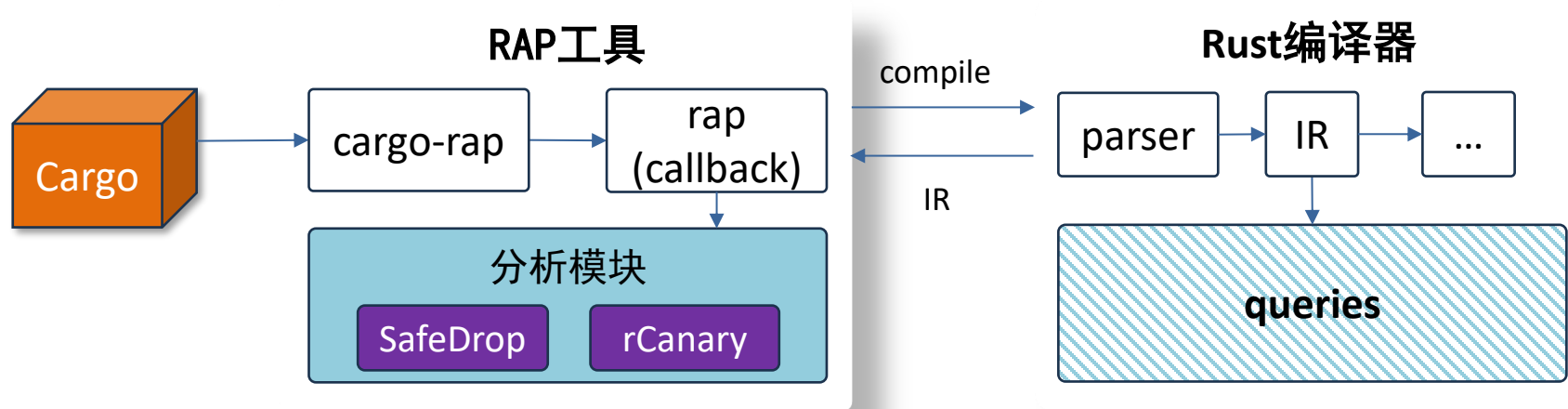
```
use std::slice;
fn foo<T>(a: &mut [T]){
    // require 4-byte alignment
    let p = a.as_mut_ptr() as *mut u32;
    println!("{:?}", p);
    unsafe {
        let s = slice::from_raw_parts_mut(p, 2);
        let _x = s[0];
    }
}

fn main(){
    let mut x = [0u8;16];
    foo(&mut x[0..16]);
}
~
~
~
~
~
~
~
~
~
~
```

```
aisr@aisr:~/test/testmiri$
```

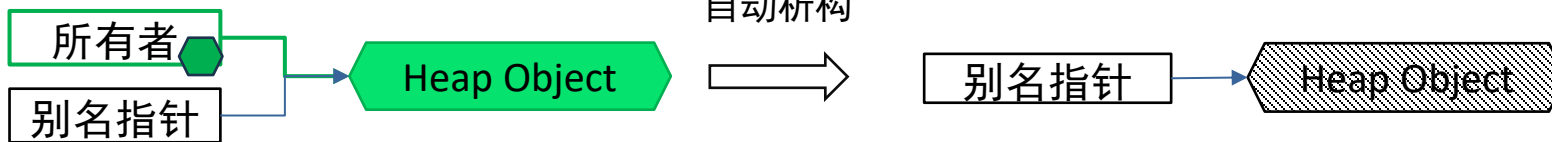
SafeDrop + rCanary => Rust程序分析平台

□ Cargo插件：无需修改编译器



SafeDrop: 面向Rust所有权副作用的悬空指针检测

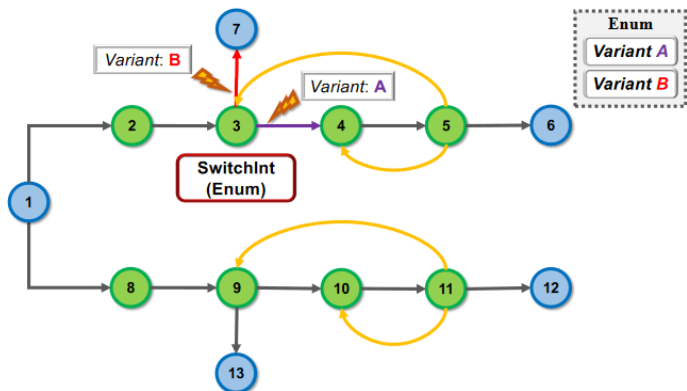
3. 模式识别



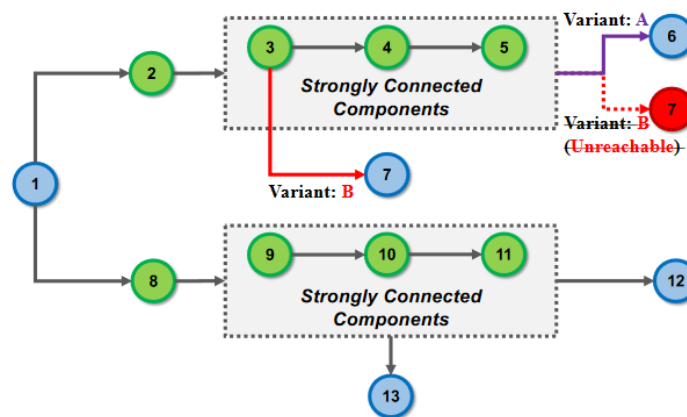
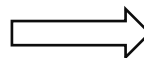
2. 别名分析

```
Statement 1:  _2 = &_1;           // alias set: {_1, _2}
Statement 2:  _1 = move _4;       // alias sets: {_1, _4}, {_2}
Statement 3:  _3 = &_1;           // alias sets: {_1, _3, _4}, {_2}
```

1. 路径提取



控制流图



生成树

SafeDrop

```
use std::env;

#[derive(Debug)]
struct MyRef<'a> { a: &'a str, }

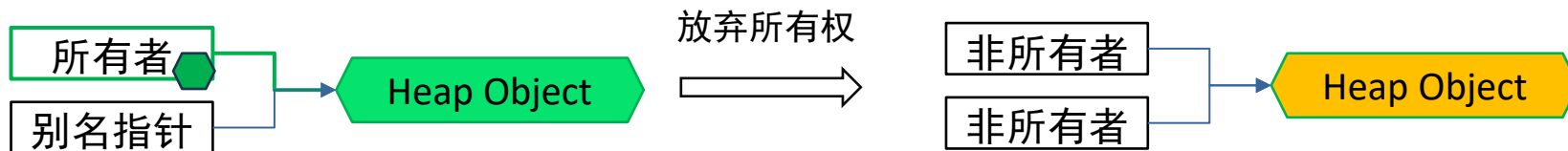
unsafe fn f<'a>(myref: MyRef<'a>) -> MyRef<'static> {
    unsafe {
        std::mem::transmute(myref)
    }
}

fn main() {
    let string = "Hello World!".to_string();
    let args: Vec<String> = env::args().collect();
    let my_ref = unsafe { f(MyRef { a: &string }) };
    if args.len() > 2 {
        drop(string);
    }
    println!("{:?}", my_ref.a);
}
~
~
~
```

```
aisr@aisr:~/RAP/test cases/uaf_drop2$
```

rCanary: 面向Rust内存泄露的缺陷检测

3. 约束求解

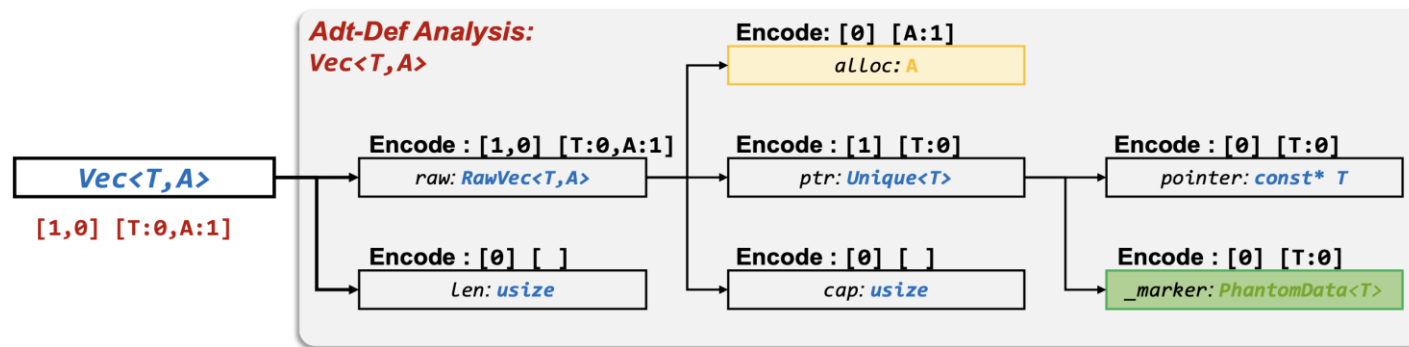


$$\frac{M \vdash x: \vec{\sigma}_x, y: \vec{\sigma}_y \quad \vec{\sigma}_x', \vec{\sigma}_y' \text{ new} \quad C' = C \wedge \{\vec{\sigma}_y = \vec{0}\} \wedge \{\vec{\sigma}_x' = \vec{0}\} \wedge \{\vec{\sigma}_y' = \vec{\sigma}_x\}}{O; M; C \vdash y = \text{move } x \Rightarrow M[y \mapsto \vec{\sigma}_y', x \mapsto \vec{\sigma}_x']; C'}$$

2. 约束提取

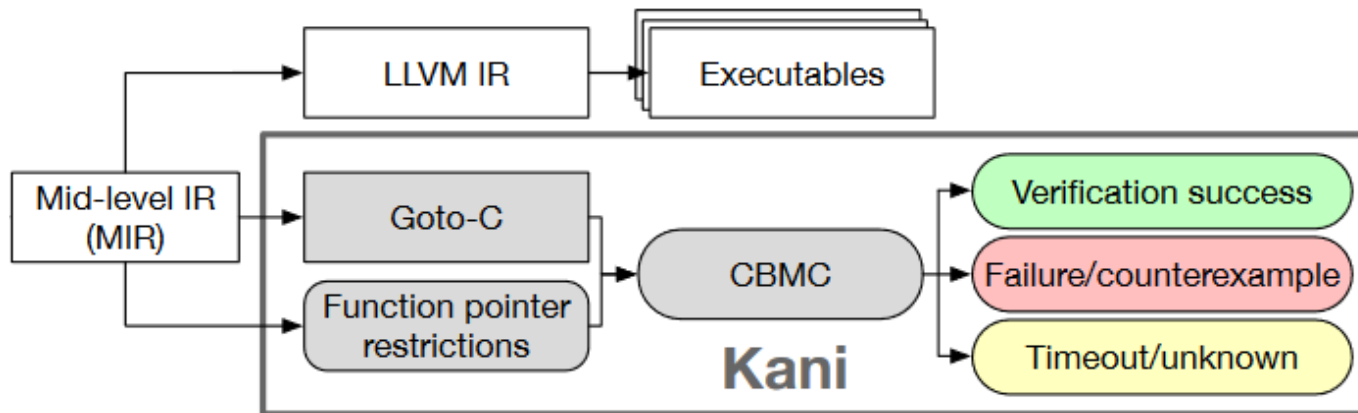


1. 类型分析




```
aisr@aisr:~/RAP/test cases/leak_proxy$
```

Kani: 模型检查



□ 受制于符号执行的局限性：

- 路径爆炸、循环处理
- 约束建模准确性：外部函数调用...

□ 有限的安全属性支持

```
fn foo(x: u32, y: u32) -> u32 {
    if x>y {
        x
    } else if x == 1234567 {
        x + y
    } else {
        y
    }
}
#[cfg(kani)]
#[kani::proof]
fn main() {
    let x: u32 = kani::any();
    let y: u32 = kani::any();
    foo(x, y);
}
```

```
~
~
~
~
~
~
~
```

```
aisr@aisr:~/test/testkani$
```


大纲

- 一、问题：背景
- 二、现状：Rust程序分析生态
- 三、未来：RAP研究思路
- 四、总结

聚焦重要问题：面向Unsafe Rust的安全检查

- ❑ 如何验证interior unsafe代码的soundness?
- ❑ Interior unsafe是Rust系统软件开发中常用且必要的设计模式

repo:asterinas/asterinas path:*.rs "unsafe {"

Asterinas

76 files (101 ms) in asterinas/asterinas X

repo:microsoft/windows-rs path:*.rs "unsafe {"

windows-rs

608 files (140 ms) in microsoft/windows-rs X

org:Rust-for-Linux path:*.rs "unsafe {"

1.5k files (198 ms) in Rust-for-Linux X

1.5k

0

42

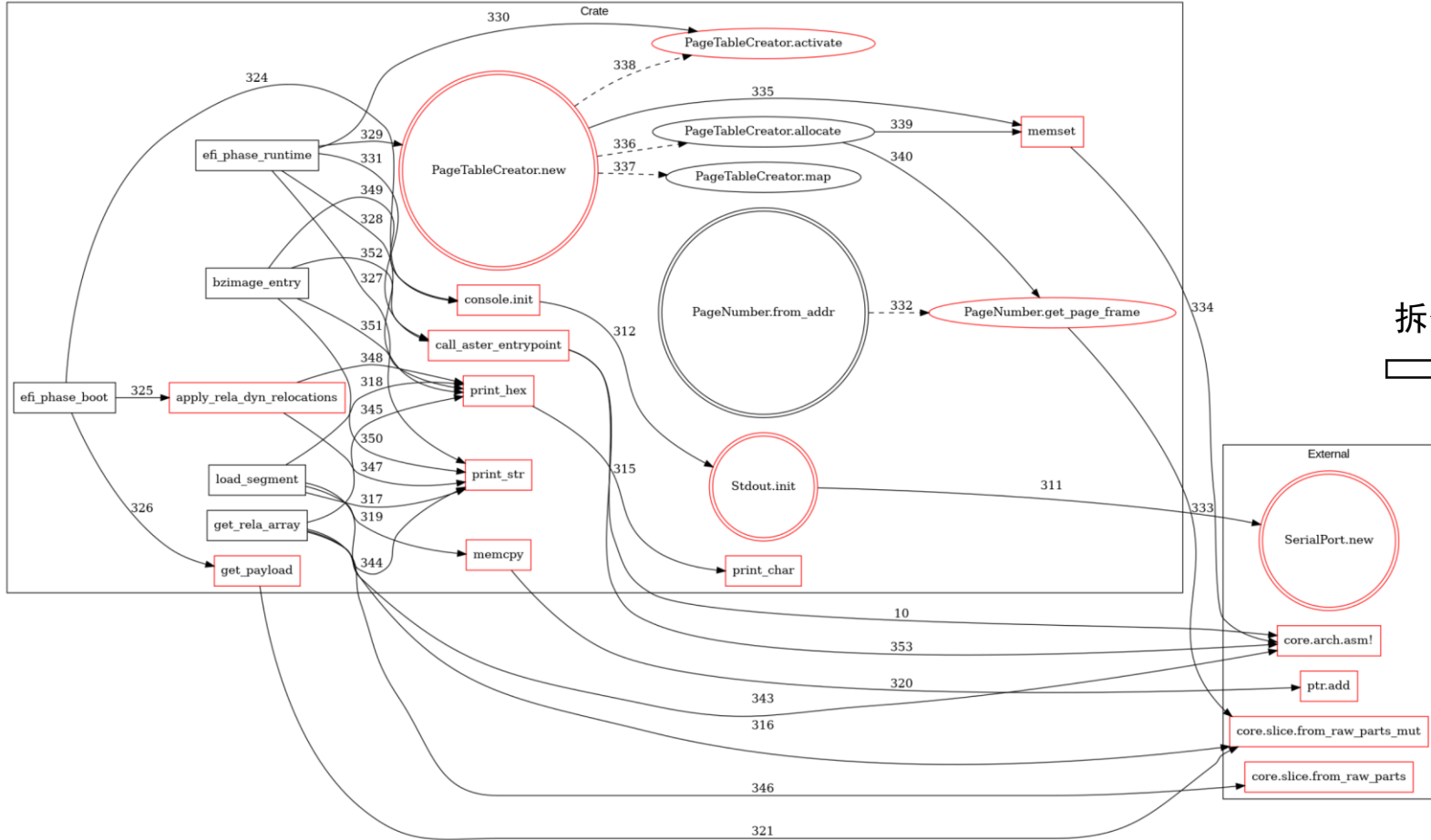
▼ Rust-for-Linux/linux · rust/kernel/lib.rs

```
111 // SAFETY: FFI call.  
112 unsafe { bindings::BUG() };  
113 }
```

研究思路: Unsafety传导分析 => 轻量级形式化验证

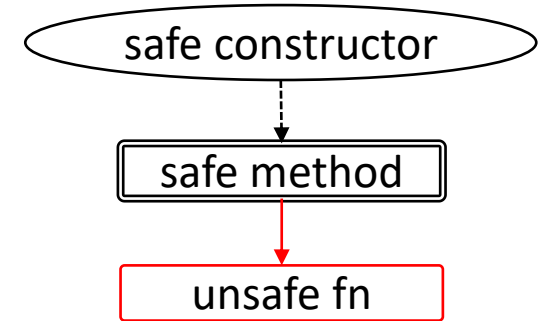
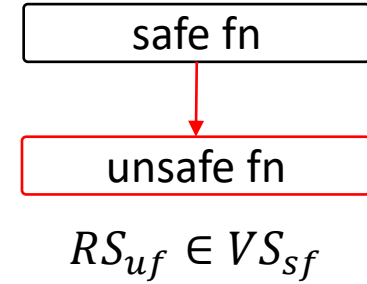


Unsafety传导图举例 (以Asterinas项目为例)



拆分子图

验证子图



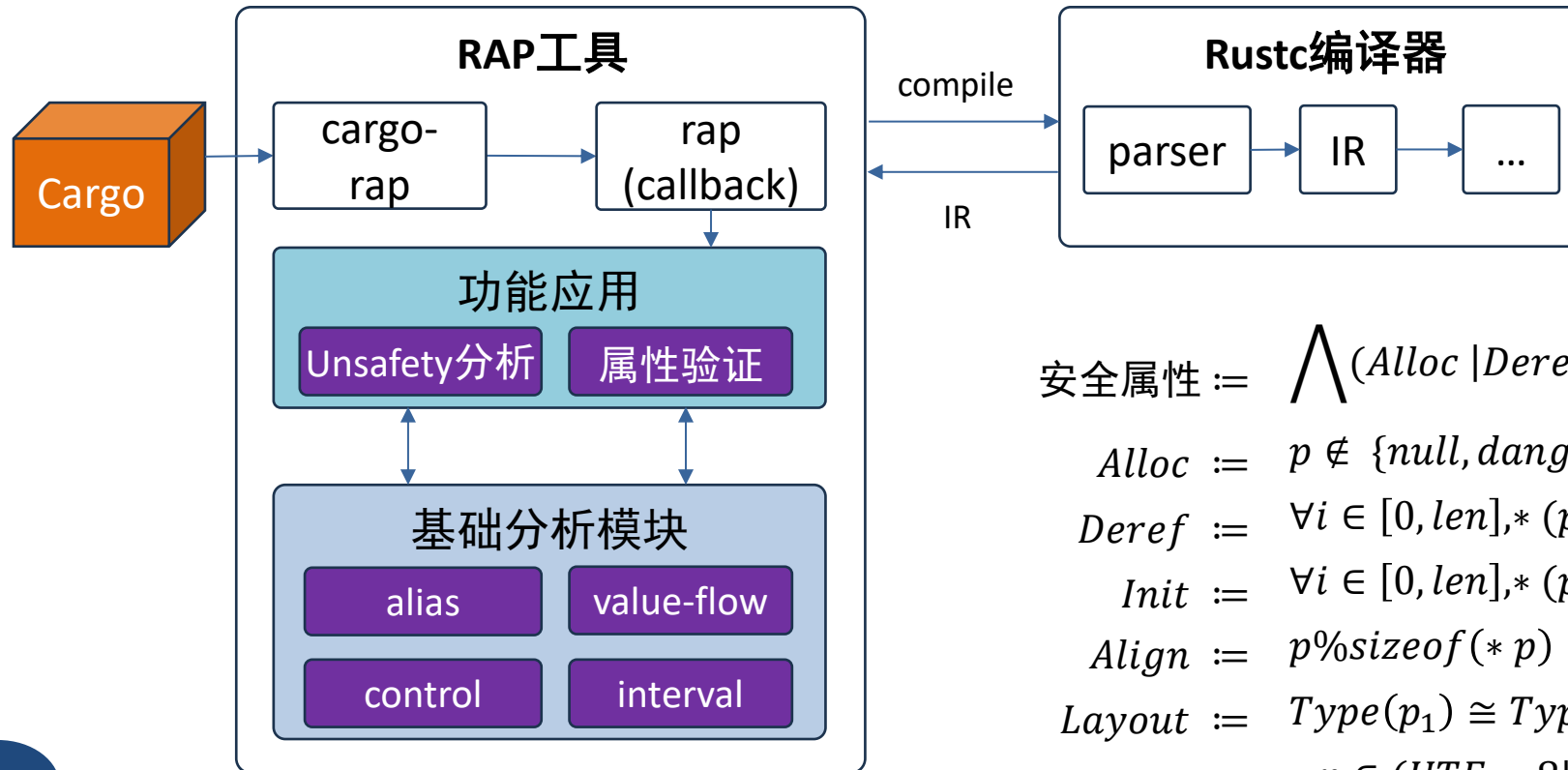
$$RS_{uf} \in VS_{sm} + VS_{sc}$$

RS: Required Safety Property
 VS: Verified Safety Property

Rust程序分析平台：面向模式 => 面向属性

□ 未来尝试的方向：

- 分层框架：基础程序分析和属性验证功能分离
- 结果可靠性：误报/漏报概率分析，提升实用价值



安全属性 := $\bigwedge (Alloc | Deref | Init | Align | Layout | Encoding)$

$Alloc := p \notin \{null, dangling\}$

$Deref := \forall i \in [0, len], * (p + i) \in valid$

$Init := \forall i \in [0, len], * (p + i) \in Obj(* p)$

$Align := p \% sizeof(* p) = 0 \mid p_2 \% sizeof(* p_1) = 0$

$Layout := Type(p_1) \cong Type(p_2)$

$Encoding := * p \in (UTF - 8 | CString)$

大纲

- 一、问题：背景
- 二、现状：Rust程序分析生态
- 三、未来：RAP研究思路
- 四、总结

总结

□ Rust程序分析研究的关键意义在于面向unsafe代码：

- 提供Rust编译器无法保障的安全检查功能
- 发现好的语言设计，进一步赋能程序分析

□ Rust程序分析工具生态：

- 以Cargo插件形式为主，无需修改编译器
- 主要工具：Miri动态未定义行为检测、静态分析RAP、模型检查Kani/Verus

□ 目前尚缺少统治级别的Rust程序分析工具和框架

- 期待更多研究人员加入和资源投入

谢谢! Q&A



RAP项目地址: <https://github.com/Artisan-Lab/RAP>