

# Rust 语言安全研究综述

胡 霜, 华保健, 欧阳婉容, 樊淇梁

中国科学技术大学 软件学院 合肥 中国 230027

**摘要** Rust 是为了解决系统编程领域的安全性问题, 而设计的一门面向系统编程的兼具类型安全、内存安全和并发安全的新型程序设计语言, 强调安全性和高性能, 已经在操作系统内核、Web 浏览器、网络协议栈、数据库和区块链等底层软件系统的构建中得到了越来越广泛的应用。现有研究表明, 尽管 Rust 的设计目标是保证安全性, 但其自身仍然存在许多安全问题。作为一门系统编程语言, Rust 的安全性直接影响到基于 Rust 开发的软件系统的安全性。随着 Rust 的广泛应用, 对 Rust 语言安全的研究显得尤为重要。Rust 语言安全研究正在成为研究热点, 并且在近几年已经取得了较大研究进展。本综述基于该研究领域已经公开发表的 46 篇研究论文, 对该领域的相关研究进行了系统整理、分析和总结: 首先, 研究分析了 Rust 的核心安全特性, 包括函数式编程范式、强多态类型系统、基于所有权模型的自动内存管理、对非安全代码的显式标记和隔离; 其次, 提出了 Rust 语言安全研究领域的分类学, 将已有研究分为安全实证研究、漏洞检测研究、安全增强研究和形式化验证研究四个热点方向, 并分别对这四个方向上的相关研究进行了综述、深入分析和总结, 同时分析了四个研究方向的内在联系; 最后, 指出了该研究领域的待解决的科学问题, 并对未来可能的研究方向进行了展望, 提出了四个潜在的研究方向, 以期对相关领域的研究者提供有价值的参考。

**关键词** Rust 语言; 内存安全; 并发安全; 漏洞检测与修复

**中图法分类号** TP312 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2023.11.06

## A Survey of Rust Language Security Research

HU Shuang, HUA Baojian, OUYANG Wanrong, FAN Qiliang

School of Software Engineering, University of Science and Technology of China, Hefei 230027, China

**Abstract** Rust is an emerging type-safe, memory-safe, and concurrency-safe programming language for system programming to address security issues in the system programming domain. The design of Rust emphasizes both security and high performance, thus has been increasingly used in the construction of software infrastructures such as operating system kernels, Web browsers, network protocol stacks, databases, and blockchains. However, existing research efforts have demonstrated that despite its design goal of ensuring security, Rust still suffers from many safety issues. As a system programming language, the security of Rust directly affects the security of software systems developed with Rust. Furthermore, with the widespread adoption of Rust, the study of Rust language security is becoming a hot research topic with significant research progress. In this survey, we systematically analyze and summarize the latest research progress in this field, based on a systematic analysis of 46 published research papers: first, we analyze the main security-related features of Rust, including functional programming paradigm, strong polymorphic type system, automatic ownership-based memory management model, and explicit isolation of unsafe code; second, we propose a taxonomy to classify current research efforts into four categories: empirical security study, vulnerability detection, safety enhancement, and formal verification, by reviewing and analyzing, and summarizing the relevant research not only in each of these four directions, but also in their interactions; finally, we point out open problems in this research area, and highlight the remaining challenges of Rust language security research, and propose four potential research directions. This survey serves as both a valuable reference and a starting point for future research in this field.

**Key words** Rust; memory safety; concurrency safety; vulnerability detection and rectification

### 1 引言

软件系统是现代信息基础设施中不可或缺的一部分, 已经被广泛用于通信、航空航天、医疗等安全

攸关系统中。因此, 软件系统的安全性和可靠性, 在保证信息基础设施安全方面起着重要作用。在基础软件领域, C/C++等程序设计语言因其灵活的底层硬件访问控制能力、高效的执行效率等优势, 一直是构

**通讯作者:** 华保健, 博士, 助理教授, Email: bjhua@ustc.edu.cn。

本课题得到中国科学技术大学研究生教育创新计划项目(No. 2020YJC41, No. 2021YJC34)资助。

收稿日期: 2022-03-29; 修改日期: 2022-06-15; 定稿日期: 2023-08-24

建操作系统、网络协议栈、数据库等关键基础软件系统的主流语言。但是, 由于这类语言的编程机制过于灵活且缺乏必要的安全检查机制, 导致基于这类语言构建的软件系统很容易出现安全漏洞<sup>[1-2]</sup>, 进而威胁信息基础设施的安全性和可靠性<sup>[3]</sup>。

Rust 是为了解决 C/C++ 等底层程序设计语言的安全问题, 而设计的新一代安全系统级程序设计语言<sup>[4]</sup>, 它通过提供函数式编程范式、强多态类型系统、基于所有权模型的自动内存管理等先进语言特性, 实现了兼具内存安全、并发安全和高执行效率的设计目标, 满足了构建底层基础软件的特定需求, 已经被成功用于构建一系列的基础软件, 如操作系统内核<sup>[5]</sup>、Web 浏览器<sup>[6]</sup>、文件系统<sup>[7]</sup>、云服务<sup>[8]</sup>、网络协议栈<sup>[9]</sup>、语言运行时<sup>[10]</sup>、数据库<sup>[11]</sup>和区块链<sup>[12]</sup>等。

然而, 现有研究表明 Rust 仍然存在许多安全问题<sup>[13-15]</sup>。随着 Rust 的广泛应用, 对其安全性的理解和研究也正在成为新的研究热点。Rust 语言安全研究是近十年来兴起的一个新研究领域, 并且已经产生了丰富的研究成果, 这些研究成果广泛涉及形式语义、程序验证、程序分析、模糊测试、符号执行等诸多方向, 大大加深了研究者对系统级安全程序设计语言的设计路径和核心机理的理解。但是, 鉴于该研究领域具有一定的前瞻性和新颖性, 目前尚未有研究工作对 Rust 语言安全相关研究进行系统梳理和总结, 指出目前的重要研究问题, 并讨论未来可能的研究方向。因此, 本文的研究目标是通过系统分析 Rust 语言安全研究领域的已有研究工作, 对这些研究工作进行梳理、分类、深入讨论和总结, 提炼核心研究课题, 指出重要开放问题, 并探讨未来的研究机会和方向, 以期对 Rust 语言安全乃至基础软件安全领域的研究者提供有价值的参考。

为了对该领域的研究工作进行系统地调研、分析和总结, 我们首先按照如下步骤收集并筛选自 2011 年以来(Rust 于 2011 年正式对外发布)公开发表的研究文献: 1) 使用 Google 学术搜索引擎, 以及 ACM、IEEE、Springer 和 CNKI 等论文数据库; 2) 检索关键字, 包括英文搜索引擎中的“Rust security”和中文搜索引擎中的“Rust 语言安全”等; 3) 检索从 2011 年至今的全部文献; 4) 对按照上述步骤检索得到的论文, 进行人工筛选和复核, 筛选重要的研究论文, 并总结该领域活跃的研究方向。在选择过程中, 我们重点选取了程序设计语言、软件工程、计算机与信息安全等领域的旗舰期刊和会议的顶级论文。

最终, 我们筛选得到了共计 46 篇论文, 这些论文大都来自国内外的高质量会议和期刊(按 CCF 的分类建议), 如软件工程领域的 ICSE 和 ASE 会议、程序语言领域的 PLDI 和 POPL 会议、系统领域的 SOSP 会议、信息安全领域的 CCS 和 Security 会议等。

图 1 统计了自 2011 年 Rust 首次公开发布以来, 关于 Rust 语言安全研究的已发表文献数量的变化趋势(截至本文写作时, 2022 年已发表文献 2 篇)。通过对图中数据进行分析, 可得出结论: 在 2015 年, Rust 的第一个稳定版 1.0 发布后, Rust 语言安全研究引起了研究者们的极大关注, 发表文献数量逐年递增, 2021 年的顶会论文已经达到 12 篇。文献数量的快速增长趋势说明: 尽管 Rust 语言安全研究这一方向的研究历史还不长, 但该研究方向越来越活跃, 具有研究价值。

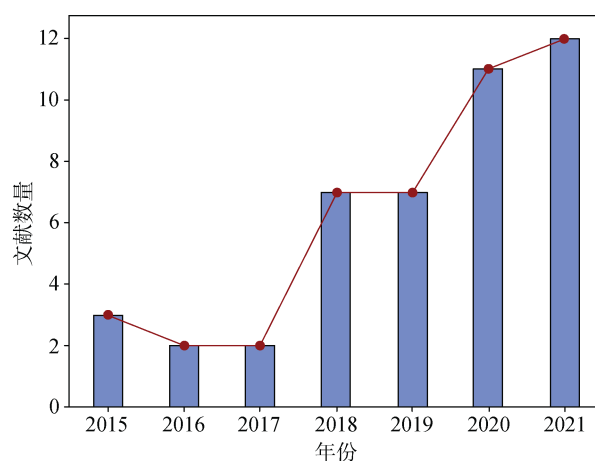


图 1 文献数量增长趋势

Figure 1 The growing trend of the number of papers

根据对文献的调研和分析结果, 本文将 Rust 语言安全研究领域总结归纳为四个重要研究方向: (1) 安全实证研究, 该方向的相关研究主要针对 Rust 安全漏洞和 unsafe 机制等方面; (2) Rust 漏洞检测研究, 已有研究主要采用了程序分析、模糊测试等技术, 研究对 Rust 程序漏洞进行静态或动态检测的有效技术; (3) Rust 安全增强研究, 相关研究工作主要使用了权限分离、程序分析等技术, 增强 Rust 程序的安全性; (4) Rust 形式化验证研究, 该方向的研究工作主要使用了形式语义、自动化程序验证等技术, 对 Rust 程序进行安全验证。本文对这四个方向的研究, 都进行了综述、深入分析和总结; 指出了重要的研究问题、现有研究的不足和开放问题; 还指出了未来的研究方向。

本文是第一个系统总结 Rust 语言安全相关研究的综述研究。总结起来, 本文的主要贡献如下:

1) 系统研究分析了 Rust 的语言安全特性, 全面调研分析了 Rust 语言安全研究领域的 46 篇已发表文献;

2) 将 Rust 语言安全研究总结为四个热点研究方向: 安全实证研究、漏洞检测研究、安全增强研究和形式化验证研究, 并对这四个方向的研究进行了深入探讨;

3) 指出了该领域待解决的重要科学问题和未来的重要研究方向。

本文余下内容安排如下: 第 2 小节系统归纳总结 Rust 的主要安全特性; 第 3 小节总结 Rust 语言安全相关的实证研究; 第 4 小节讨论 Rust 漏洞检测研究; 第 5 小节讨论 Rust 安全增强研究; 第 6 小节讨论 Rust 形式化验证研究; 最后, 第 7、8 小节分别对未来研究方向进行展望以及总结全文。

## 2 Rust 语言安全研究概述

本小节将对 Rust 的历史和核心安全特性进行系统分析和总结, 提出对现有 Rust 语言安全研究的分类方法, 并对各类研究进行概述。

### 2.1 Rust 语言历史与概况

Rust 由 Mozilla 研究院的 Graydon Hoare 从 2006 年开始设计<sup>[16]</sup>, 在 2011 年第一次公开发布, 于 2012 年对外发布了第一个有版本号的 0.1 版。Rust 在 2015 年发布了第一个稳定版 1.0, 目前最新的稳定版是 1.61.0。

图 2<sup>[13]</sup>展示了 Rust 的特性更改数量(蓝色虚线)、Rust 编译器 rustc 的代码量(红色实线)的变化趋势。经分析, 可得出结论: 自 2012 年 Rust 发布后的前五年(2012—2017 年), Rust 语言特性变化较频繁, 2017 年以后, 语言特性才逐渐趋于稳定。语言的相对稳定性给安全研究和安全软件系统的构建提供了必要前提。Rust 编译器 rustc 最初用 OCaml 实现, 在 2011 年才开始使用 Rust 进行实现, 完成了自举, 目前 Rust 编译器的规模已经非常庞大, 代码超过 70 万行。

Rust 主要面向系统编程领域, 针对系统级安全编程的科学问题, 采用基于所有权模型的自动内存管理等先进特性, 解决传统 C/C++ 语言存在的安全问题, 在保持高执行效率的同时, 保证语言安全性。由于兼顾安全和高效的优良特性, 近年来, Rust 受到了广泛欢迎, 越来越流行。根据 Stack Overflow 在 2021 年进行的一项开发者调查显示<sup>[17]</sup>: Rust 被评为“最受欢迎的程序设计语言”, 有 86.98% 的开发人员正在使用或者考虑使用 Rust; 而根据 GitHub 在 2019 年发布的统计数据<sup>[18]</sup>显示: Rust 是增长第二迅速的

编程语言。Rust 在工业界也得到了越来越广泛的使用, 微软<sup>[19]</sup>、谷歌<sup>[20]</sup>、华为<sup>[21]</sup>、甚至 Linux<sup>[22]</sup>, 都开始探索基于 Rust 来进行关键底层软件系统的研发。

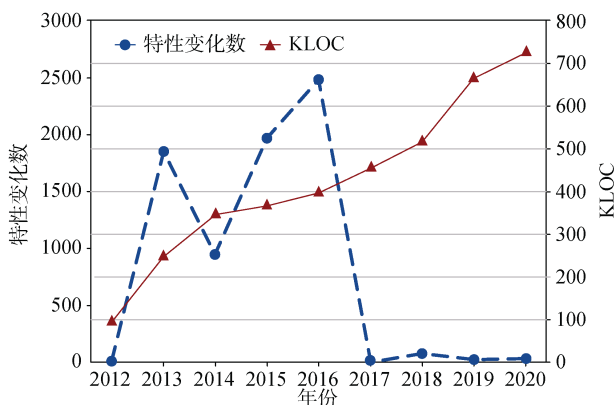


图 2 特性变化数及编译器代码量的变化趋势

Figure 2 The trend of feature change and compiler size

### 2.2 Rust 核心语言安全特性总结分析

作为一门新兴语言, 为实现保证语言安全性的设计目标, Rust 充分吸收了程序语言和软件安全领域已有的研究成果, 并设计(或从其他语言借鉴)了许多新的语言特性, 包括默认函数式编程范式、强多态类型系统、基于所有权模型的自动内存管理(也支持可选的引用计数)、同时支持消息传递和共享内存的多线程安全并发编程等。这些特性的引入, 使得 Rust 在具有可媲美 C/C++ 的高性能的同时, 实现了类型安全、内存安全和并发安全。本小节将对 Rust 的核心安全特性进行分析和归纳总结。

#### 2.2.1 函数式编程范式

为提高语言的安全性, Rust 从 SML<sup>[23]</sup>、Lisp<sup>[24]</sup>、OCaml<sup>[25]</sup>、Haskell<sup>[26]</sup>等函数式编程语言中吸收借鉴了核心函数式编程范式。第一, Rust 中的变量必须被显式初始化并且默认是不可变的, 这消除了由变量未初始化导致的安全漏洞; 并且, 在多线程访问共享变量时, 变量的不变性使得不必进行加锁等易错的同步操作, 这让安全并发程序的构建更加容易; Rust 中的可变变量必须通过关键字 `mut` 显式声明, 这样可以更精细地控制变量的读写权限。第二, Rust 引入了代数数据类型和模式匹配等语言特性, 这些特性使得异构数据结构的构建更加安全; 并且, 编译器会对所有的模式匹配分支进行强制类型检查, 避免了不安全的空分支等导致的安全漏洞。由于 Rust 提供的函数式安全编程机制非常丰富, 因此, 在目前的语言分类学中, Rust 被直接归类为函数式编程语言<sup>[27]</sup>。



### 2.2.2 强多态类型系统

Rust 使用强类型系统来保证程序的类型安全, 确保通过类型检查的程序不会出现特定的运行时错误。和其他静态检查机制类似, 尽管静态类型能够保证程序的类型安全, 但会对程序的构建形成额外约束, 并且会显著提高程序员的学习成本, 从而形成额外负担。为减轻类型约束和降低程序员负担, 并有效提高代码复用性, Rust 引入了多态类型和类型推导两种机制。

Rust 引入了两种多态机制: 基于 trait<sup>[28]</sup> 的 ad-hoc 多态和基于显式类型参数<sup>[29]</sup> 的参数多态。Rust 中的 trait 机制借鉴了 Haskell 中的 Typeclass 机制, 实现运行时动态方法绑定。Rust 引入的参数多态借鉴了 C++ 中模板和 Java 中泛型的设计理念。Rust 编译器对参数多态的实现, 也借鉴了 C++ 编译器中使用的静态单态化的编译技术<sup>[30]</sup>, 即带类型参数的多态 Rust 代码会被编译器实例化为单态程序代码, 进而编译为目标代码。尽管单态化有一些代价, 例如会增加编译时间, 且可能增大目标代码规模, 但它的主要优势是能基于具体类型对目标代码进行针对性优化, 这对提高 Rust 程序的执行效率具有积极意义。

为降低显式类型符号标注给编程带来的代价, 进而减轻程序员负担, Rust 通过 let 变量声明机制, 支持局部变量粒度的类型推导, 编译器可根据变量的初始化等语法上下文, 自动推导局部变量的类型。

尽管 Rust 引入的多态类型和类型推导比纯函数式语言的同类机制要弱(例如, Rust 不支持隐式类型参数的多态和高阶多态), 也不支持全程序范围的全局类型推导(例如, Rust 要求必须显式标注函数的返回值类型), 但 Rust 引入的多态类型和类型推导等高级特性, 仍然大大增强了该语言静态类型系统的检查能力, 为实现 Rust 的类型安全奠定了重要基础。

### 2.2.3 基于所有权模型的自动内存管理

为保证内存安全, 并消除在 C/C++ 程序中常见的悬空指针、重复释放、空指针引用等内存安全漏洞, 大部分的安全高级语言都采用了诸如自动垃圾收集等自动内存管理机制。尽管垃圾收集可通过采用并发收集等算法, 来尽量降低全局暂停时间<sup>[31]</sup>, 但垃圾收集仍会带来可观的时间开销, 因此难以适用于实时系统等有低延迟需求的场景中。

为了在实现内存安全的同时保证操作的实时性, Rust 没有采用自动垃圾收集机制, 而是引入了基于所有权模型的自动内存管理机制, 这也是 Rust 最核心的安全特性之一。Rust 中所有权模型的核心思想是: 程序中的每个值都有唯一的所有者, 该值的生

命周期等于该值的所有者的词法作用域范围, 超出对应的词法作用域后, 该值所在内存单元将被 Rust 编译器自动插入的内存回收语句自动回收释放, 该值将无法被继续访问。

图 3 给出了 Rust 基于所有权模型进行自动内存管理的典型示例代码, 第 6 和第 7 行代码分别定义了值为 22 和 33 的两个内存单元, 其所有者分别为变量  $x$  和  $y$ , 由于变量  $y$  的词法作用域在第 7 行的末尾结束, 因此第 7 行之后的语句不能再访问变量  $y$ 。

变量的所有权会随着赋值或函数调用而发生移动。例如, 图 3 的第 8 行代码调用函数  $g()$ , 这将使函数实参  $x$  的所有权转移到形参  $z$  上(第 3 行), 所有权转移也意味着变量  $x$  在第 8 行代码后将无法再被访问。因此, 如果将第 9 行代码反注释, 编译器将报错, 提示变量  $x$  的所有权已被转移, 无法再被访问。

```

1 struct Cell { data: i32, }
2
3 fn g(Cell z){}
4
5 fn f() {
6     let x = Cell { data: 22 };
7     { let y = Cell { data: 33 }; } // y 不可访问
8     g(x);
9     // x.data; // x 不可访问
10    let mut m = Cell { data: 55 };
11    let r1 = &m;
12    let r2 = &m;
13    let mut r3 = &mut m;
14    // let d = r1.data; // r1 不可访问
15 }
```

图 3 Rust 所有权模型的代码示例

Figure 3 Code example of Rust's ownership model

Rust 中基本的所有权模型要求每个值都只有唯一的所有者, 尽管这个要求提供了强安全保证, 但也对程序设计造成了额外限制。为此, Rust 对基本的所有权模型进行了扩展, 引入了“引用”的概念和基于引用的“借用”机制, 这个机制允许多个不可变引用指向同一个内存单元(称为别名), 或者唯一的可变引用指向一个内存单元。例如, 对于图 3 中的第 11 和 12 行代码, 变量  $r1$  和  $r2$  是变量  $m$  的两个不可变(只读)引用, 而第 13 行的变量  $r3$  是对变量  $m$  的可变引用。因此, 如果把第 14 行的代码反注释, 编译器将报错, 提示同时存在对变量  $m$  的可变和不可变两种引用。

所有权、移动、引用和借用等机制相互结合、共同作用, 使得 Rust 消除了 C/C++ 等语言中常见的悬空指针、二次释放、空指针引用、释放后使用等内存安全漏洞, 并且避免了线程间共享可变变量

导致的竞态条件问题, 在提供编程灵活性的同时, 保证了 Rust 语言的内存安全和并发安全。

2.2.4 对非安全代码的显式标记和隔离

语言的安全性和编程的灵活性之间存在一定的矛盾, 尽管 Rust 的强类型系统和基于所有权模型的自动内存管理提供了高安全保证, 但过强的限制也使得特定底层代码的构建变得更困难, 甚至无法实现。例如, 在底层编程中, 往往需要直接操作硬件物理地址或裸套接字, 此时必须暂时关闭编译器的静态检查。为了支持这类底层操作场景, Rust 允许程序员构建不安全代码, 并提供了关键字 `unsafe` 对不安全代码块进行显式标记和隔离。Rust 编译器不会对 `unsafe` 代码块中的 5 种不安全操作提供任何安全检查。这 5 种不安全操作包括: 解引用裸指针、调用 `unsafe` 的函数或方法(包括外部 C/C++ 函数)、实现不安全的 `trait`、访问或修改可变静态变量的值、读写 `union` 联合体中的字段。除此之外, Rust 编译器不会对 `unsafe` 代码块禁用任何安全检查。

```
1 struct Cell{ data: i32, }
2 impl Cell{
3     fn set(&self, v: i32) {
4         let p = &self.data as *const i32 as *mut i32;
5         unsafe{
6             *p = v;
7             *(p+1024) = v; //指针越界使用, 编译器不报错
8         }
9     }
```

图 4 Rust 中 unsafe 代码安全问题示例  
Figure 4 Example of unsafe code safety issue in Rust

Rust 引入 `unsafe` 机制的初衷是在保证语言功能的前提下, 提供“现实安全”的能力<sup>[32]</sup>, 但已有研究表明: 对 `unsafe` 机制的不当使用容易导致安全漏洞, 形成对 Rust 程序的新攻击面<sup>[13-14]</sup>。图 4 给出了使用 Rust 的 `unsafe` 机制的典型示例, 第 4 行代码声明了一个指针 `p`, 它指向 `Cell` 中的 `data` 字段, 程序可通过指针 `p` 对其指向的 `data` 字段进行读写。由于第 7 行代码中对指针 `p` 的解引用操作被包含在 `unsafe` 代码块中, 因此编译器不会对第 7 行代码进行任何安全检查, 示例程序可以编译通过。但是第 7 行代码的指针算术可以对任意内存进行读写操作, 从而导致安全漏洞。

2.3 研究框架

Rust 作为一门系统级编程语言, 在引入新型安全机制保证语言安全性的同时, 也引入了新的安全漏洞, 带来了新的攻击面, Rust 语言安全研究已成为近年来的研究热点。本综述基于对已有研究工作的

调研、分析和总结, 将该领域已有研究工作凝练成 4 个大的研究方向: (1) 安全实证研究; (2) 漏洞检测研究; (3) 安全增强研究; (4) 形式化验证研究。

这 4 个研究方向都围绕 Rust 语言安全, 它们是 Rust 语言安全研究的不同方面, 同时也具有非常紧密的内在联系。Rust 安全实证研究的结果对于 Rust 全生命周期安全各个阶段的研究, 包括漏洞检测、安全增强、形式化验证, 都具有重要指导意义。根据 Rust 安全实证研究的结果, 提出有效的漏洞检测技术可以发现软件系统中的安全漏洞。对于已知的安全漏洞, 相应的安全增强技术可以防止漏洞的发生, 从而增强软件系统的安全性。形式化验证技术基于严格的数学基础, 验证软件系统是否满足特定的安全属性和功能正确性, 为软件系统提供更强的安全保障。

按照上述研究方向分类, 表 1 给出了对已发表文献的分类统计。通过对图中的数据进行分析, 可得出结论: 形式化验证研究占比最高, 达到 43.47%, 这体现了形式化验证对于 Rust 语言安全的重要性; 而漏洞检测研究、安全增强研究和安全实证研究的占比分别为 19.57%、19.57%和 17.39%。在接下来的各个小节, 本文将分别对每个研究方向进行综述、深入分析和总结, 指出现有研究工作的不足, 并提出潜在的研究问题以及未来的重要研究方向。

表 1 Rust 语言安全文献的研究方向分布				
Table 1 Distribution of research directions in Rust				
研究方向	实证	漏洞检测	安全增强	形式化验证
篇数	8	9	9	20
占比(%)	17.39	19.57	19.57	43.47

3 Rust 安全实证研究

实证研究强调在观察和实验的经验事实上, 通过经验观察的数据和实验研究的手段, 来揭示一般结论, 归纳事物的本质属性和发展规律。对于 Rust 这种相对新颖的系统级安全编程语言, Rust 安全实证研究是 Rust 语言安全研究的一个重要方向, 这类研究通过理解和把握 Rust 关键安全机理, 分析漏洞形成的根因和潜在攻击面, 从而为研究有效的漏洞检测技术、安全增强技术、形式化验证技术奠定基础。

本文将现有的 Rust 安全实证研究分为三类: 对 Rust 安全漏洞的研究、对 `unsafe` 机制的研究和其他 Rust 安全相关研究。其中, 其他研究包括 Rust 高级安全特性的实际使用难度、Rust 应用于基础软件系

统的表现、Rust 与其他语言之间的自动转换等方面。本小节将讨论 Rust 安全实证研究的研究进展、现有成果、主要不足及未来的研究方向。

3.1 对安全漏洞的实证研究

Rust 的设计目标是在实现高效率的同时,通过高级类型系统和基于所有权模型的自动内存管理等机制,静态检查和保证程序的安全性。基于该设计目标,从实证研究的角度,还有三个科学问题需要回答: (1) 在基于 Rust 构建的实际软件系统中,是否还存在安全漏洞? (2) 如果存在,则有哪些根因导致这类安全漏洞的发生? (3) 这些安全漏洞会导致哪些后果? 通过安全实证研究,系统回答这些科学问题,对深入理解 Rust 安全漏洞的发生机理,以及研究有效的漏洞检测技术、安全增强技术、形式化验证技术等,有重要指导意义。

已有的 Rust 安全实证研究已经揭示: 在基于 Rust 构建的实际软件系统中, 仍然可能存在安全漏洞。基于对现有研究工作的深入分析总结, 本文将这些安全漏洞划分为内存安全漏洞和并发安全漏洞两大类, 并且按照数据集从小到大、分析从浅显到深入的原则, 对每一类漏洞的相关研究进行梳理, 对现有研究的成果进行凝练和总结, 结果如表 2 所示, 表 2 给出了漏洞类别、具体安全漏洞以及根因分析。

表 2 Rust 安全漏洞分类与根因分析  
Table 2 Rust bug classification and root cause analysis

漏洞类别	具体安全漏洞	根因分析
内存安全	缓冲区溢出	造成内存安全漏洞的主要原因有: 内存自动回收错误、使用不安全函数或外部调用和使用不安全高级特性, 同时所有内存安全漏洞都与 unsafe 机制的不当使用有关
	读取未初始化的内存	
	无效释放	
	释放后使用	
	重复释放	
并发安全	空指针解引用	造成并发安全漏洞的主要原因是 Rust 所有权机制和生命周期规则的误解, 且安全的 Rust 代码中也可能存在非死锁并发漏洞
	类型混淆	
	重复上锁(●)	
	上锁顺序冲突(●)	
	忘记解锁(●)	
	信道误用(●)	
	数据竞争(○)	
	违反原子性(○)	
	违反访问顺序(○)	

3.1.1 内存安全漏洞

Rust 严格的安全检查机制可以保证, 用 Rust 的安全语言子集(即不使用 unsafe 机制的语言子集)构建的程序不会出现内存安全漏洞。但是, 带有 unsafe 标记的函数或代码块中的五类不安全操作, 可以绕过编译器的静态检查, 导致安全漏洞。并且, 由于

unsafe 代码和安全 Rust 代码处于同一个进程地址空间中, unsafe 代码对整个进程空间有完全的访问权限, 从而导致访问内存时的脆弱点和攻击面的扩大。

Rust 中常见的内存安全漏洞包括: 缓冲区溢出、读取未初始化的内存、无效释放、释放后使用、重复释放、空指针解引用、类型混淆(Type Confusion)等。

Qin 等<sup>[13]</sup>对实际 Rust 应用程序中的 70 个内存安全漏洞进行了实证研究。该实证研究将导致内存安全漏洞的原因归纳为两类: 错误的内存访问和违反变量的生命周期规则。错误的内存访问会导致缓冲区溢出这样的典型漏洞, 即对数组进行写入操作时发生了越界; 违反变量的生命周期规则导致的典型漏洞是释放后使用, 许多内存安全漏洞都是由误用所有权和生命周期规则导致的。但是, 相较于 Xu 等<sup>[14]</sup>的工作, 该研究使用的数据集较小, 提取的内存安全漏洞的错误模式也不够全面和深刻。

为了对 Rust 中的内存安全漏洞进行更加深入的分析, Xu 等<sup>[14]</sup>对截至 2020 年 12 月 31 日已经报告的 186 个 Rust 内存安全相关的漏洞, 进行了实证分析和研究。研究表明: Rust 的类型系统能够保证 Rust 程序的安全性, 已报告的所有内存安全漏洞都和 unsafe 机制的不当使用相关。该研究工作将内存安全漏洞分成 4 类: 与 Rust 采用的所有权资源管理副作用相关的自动内存回收漏洞; 使用了不安全的函数或外部调用导致的漏洞; 使用了不安全的 Rust 高级特性, 如 trait 等, 导致的漏洞; 其他内存安全漏洞, 如算数溢出、错误的 API 使用等。该研究是对 Qin 等<sup>[13]</sup>工作的扩展, 使用了更大规模的漏洞集, 对 Rust 内存安全漏洞提出了更详细和全面的分类方法学, 因此研究结果更具代表性。

但是, 上述工作的主要不足在于它们都只是对 Rust 内存安全漏洞的初步研究, 数据集的覆盖范围有限, 而且没有讨论内存安全漏洞造成的影响。两个主流的在线漏洞数据库 CVE<sup>[33]</sup>和 RustSec<sup>[34]</sup>上报告的 Rust 内存安全漏洞已经达到 379 个, 上述工作研究的漏洞个数远小于已报告的漏洞个数。随着 Rust 的应用越来越广泛, 会有更多的内存安全漏洞暴露出来, 上述工作提出的漏洞产生原因分类可能会进一步扩展。本文认为, 未来工作采用的数据集应该覆盖已报告的重要内存安全漏洞, 从而对内存安全漏洞的产生机理进行更深刻和全面的分析, 同时还应该指出相关漏洞造成的影响, 以对语言设计和应用开发提供更有价值的建议。

3.1.2 并发安全漏洞

在以线程机制支持并发编程的语言中, 并发安

全一般指线程不会以“不可预期”的方式访问线程共享数据,这里的不可预期一般指数据竞争和死锁。Rust 以原生线程的方式支持并发,且同时支持消息传递和共享内存两种线程间的数据共享机制。为保证并发安全,Rust 采用与内存安全中相同的所有权模型来保证线程间数据的安全共享,Rust 将这一安全特性称为无畏并发(Fearless Concurrency)<sup>[35]</sup>。

基于现有研究,本文将 Rust 中的并发安全漏洞划分为死锁相关漏洞(●)和非死锁相关漏洞(○)。死锁相关漏洞主要包括重复上锁、上锁顺序冲突、忘记解锁、信道误用等;而非死锁相关漏洞主要包括数据竞争、违反原子性、违反访问顺序等。

Yu 等<sup>[15]</sup>研究了三个基于 Rust 开发的软件系统,包括 Web 浏览器 Servo、存储系统 TiKV 和随机数生成库 Rand,对其中 18 个并发安全漏洞进行了实证研究。该研究发现重复上锁和信道误用是造成死锁安全漏洞的主要原因,10 个死锁漏洞中有 7 个由重复上锁导致,3 个由信道误用导致;而数据竞争是导致非死锁安全漏洞的主要原因,8 个非死锁漏洞均由数据竞争导致,发生数据竞争的根因是使用了 unsafe 代码,或者误用了 Rust 原子操作。进一步,该研究还发现 Rust 程序中对各类并发原语的使用频率,与其他编程语言显著不同:在 Rust 程序中,使用频率最高的并发原语是信道 channel,其次才是原子变量 atomic。但是,该研究所使用数据集的覆盖范围有限,而且没有进一步分析产生这一差异的根因。

上述工作的研究规模较小,为了对 Rust 并发安全漏洞进行更全面和深入的研究,Qin 等<sup>[13]</sup>对现实 Rust 程序中的 100 个并发安全漏洞进行了实证研究。该研究发现:导致死锁漏洞的主要根因是对 Rust 的生命周期规则缺乏充分了解,而导致非死锁漏洞的主要根因是没有保护共享资源和消息传递错误。同时,该研究工作还揭示:虽然 Rust 的自动解锁机制的设计初衷是帮助开发者避免数据竞争和忘记解锁,但是,如果开发人员对变量的生命周期存在误解,这一机制反而会导致更多安全漏洞。因此,开发生命周期可视化插件和突出显示 Rust 隐式解锁位置的插件,可以帮助开发人员构建更安全的 Rust 程序。

上述两个研究工作都指出重复上锁是最主要的死锁安全漏洞,数据竞争是最主要的非死锁安全漏洞,非死锁安全漏洞不仅发生在 unsafe 代码中,还发生在安全的 Rust 代码中。本文认为,现有并发安全实证研究的局限性主要体现在两个方面:一是实证研究使用的数据集较小,容易产生过拟合问题,CVE<sup>[33]</sup>和 RustSec<sup>[34]</sup>上已报告的并发安全漏洞已经达到 125

个,而 Yu 等<sup>[15]</sup>的研究只包含了 18 个漏洞,Qin 等<sup>[13]</sup>研究的 100 个并发漏洞中只有 1 个来自 CVE 和 RustSec;二是现有研究对并发安全漏洞产生机理的分析不够深入,没有对并发漏洞提取出全面的错误模式。未来研究应该基于已报告的所有重要并发安全漏洞,提取出全面和深刻的错误模式,指出并发安全漏洞造成的影响,从而为后续的漏洞检测和修复研究奠定基础。

### 3.2 对 unsafe 机制的实证研究

Rust 引入的 unsafe 机制,提供了构建不安全代码的能力,但是对 unsafe 机制的不当使用,将导致内存安全和并发安全问题。因此,对 unsafe 机制展开实证研究,对于深入分析和理解其运行机理,从而指导语言演进,具有重要意义。

Evans 等<sup>[36]</sup>对 unsafe 机制在实际 Rust 库函数和应用程序中的使用情况,进行了大规模实证研究。研究结果表明:虽然只有不到 30%的库使用了 unsafe 机制,但由于库函数之间的调用链,有超过一半的库函数无法被 Rust 编译器静态安全检查。基于该研究结果,本文认为:尽管 unsafe 机制在实际系统中的使用有限,但 unsafe 代码会通过调用链进行传播,给 Rust 软件系统的安全性带来威胁。

上述研究只针对 unsafe 机制的使用规模,没有进一步总结 unsafe 机制的实际使用场景,为了理解开发人员使用 unsafe 机制的原因,Qin 等<sup>[13]</sup>对实际 Rust 程序中的 850 个 unsafe 代码实例进行了实证研究,总结了常见的不安全操作和使用 unsafe 代码的目的。常见的不安全操作包括:计算内存地址、解引用裸指针、通过指针访问内存、改变内存布局、内存初始化等。通过对这些不安全操作进行分析,该研究指出:尽管应该尽量减少 unsafe 代码的使用,但 Rust 应用程序中仍然大量使用了 unsafe 代码,并且大多数 unsafe 代码的使用都无法完全避免,unsafe 代码的使用目的包括重用现有代码、提高性能、与编译器交互、跨线程访问、实现特殊功能等。该研究的结果表明,由于 Rust 的安全检查非常严格,影响了代码的底层操作能力,所以 unsafe 机制对于开发人员构建底层软件系统来说不可或缺。

unsafe 代码可以绕过编译器的安全检查,因此,确保 unsafe 代码的安全性成为程序员的责任。使用 unsafe 代码需要遵循三个原则:一是应该谨慎使用 unsafe 代码;二是 unsafe 代码量应该尽可能小且易于审查;三是 unsafe 代码应该封装好,隐藏于安全抽象之后。Astrauskas<sup>[37]</sup>等通过分析在 crates.io 上公开发布的 31867 个包,对 unsafe 代码在实际项目中的使用



情况,进行了实证研究。该研究提出了对 `unsafe` 代码的使用目的的分类方法学,并评估了 `unsafe` 代码的使用遵循上述原则的情况。研究结果表明: Rust 项目使用 `unsafe` 代码的主要目的包括与其他语言交互、实现复杂的共享数据结构、使用不安全的并发特性等。进一步,研究结果还表明, `unsafe` 代码的实际使用情况部分遵循上述原则:大部分 `unsafe` 代码简单且封装良好,但 `unsafe` 代码的使用非常广泛。但是该研究没有针对如何遵循 `unsafe` 代码使用原则提出具体的技术路线。虽然 Qin 等<sup>[13]</sup>也提出了对 `unsafe` 代码的使用目的的分类方法学,但他们采用的数据集来源于 10 个基于 Rust 开发的应用和数据库,没有覆盖整个 Rust 生态系统,而 Astrauskas<sup>[37]</sup>等的数据集来源于在 crates.io 上公开发布的 31867 个包,因此,两个研究采用的数据集互为补充。

基于对已有研究的分析,本文认为,考虑到 Rust 的使用场景和设计目标,在 Rust 中引入 `unsafe` 机制是语言设计的必然选择,但对于保证 `unsafe` 代码的安全性,还有两个重要的研究方向值得进一步探索:一是保证对 `unsafe` 代码的安全使用;二是针对 `unsafe` 代码的安全检测。对于第一个研究问题,本文认为,需要进一步增强对 `unsafe` 机制的语义研究,形成并完善针对 `unsafe` 代码的最佳安全实践。例如,可以通过给 `unsafe` 代码加上前后条件和循环不变式,对代码的行为进行更严格地规定。同时,应尽可能使用 `unsafe` 函数而不是 `unsafe` 代码块,使用 `unsafe` 函数可以让程序的不变式在函数接口层面被表达和证明,从而使安全性验证更加可扩展和更加模块化。对于第二个研究问题,本文认为,需要研究更强有力的检测技术和开发更有效的检测工具,检测 `unsafe` 代码的更多静态性质。

### 3.3 其他实证研究

理解 Rust 安全规则和底层安全机制给 Rust 的应用带来的挑战,是非常重要的研究问题。Zhu 等<sup>[38]</sup>对 Stack Overflow 中的 Rust 相关问题进行了实证研究。研究结果表明:对于 Rust 开发者来说,在实践中完全理解和正确应用 Rust 的安全规则非常困难。这主要是因为同一 Rust 安全规则应用于不同代码时,其应用难度并不相同,而不同安全规则在应用于相同的代码时,也可能具有不同的难度。该研究还发现:现有安全检测工具,如 Rust 编译器等,往往并不能为开发者修复程序提供实质性帮助。

为了探索 Rust 在内存受限的嵌入式系统中的应用能力和限制,Balakrishnan 等<sup>[39]</sup>在资源密集型应用程序中,同时运行 Rust 程序和 C++程序,并从执行

时间、内存消耗两个方面对 Rust 和 C++进行对比研究。研究结果表明: Rust 的内存消耗几乎只有 C++的一半,这说明 Rust 比 C++具有更好的内存管理,但 Rust 的执行时间更长。由于该研究使用的测试集只包含快排算法,因此结论的可信度和可扩展性不高。

目前将 C 程序自动翻译为 Rust 程序的工具,只能将 C 程序翻译为等价的 `unsafe` Rust 程序,无法自动进行安全性提升。Emre 等<sup>[40]</sup>对由 C 程序自动翻译得到的 Rust 程序进行实证研究,分析了这些程序中潜在的不安全原因,以及修复这些不安全原因的相对影响,最后还提出了一种自动消除不安全性的技术,该技术通过将部分裸指针转换为安全引用来提高 Rust 程序的安全性。但是,该研究提出的安全性提升技术只能针对裸指针使用这一类不安全原因进行部分提升,无法将全部裸指针使用转换为安全的引用。提出有效的将 C 程序自动转换为安全的 Rust 程序的技术是未来一个重要但极具挑战性的研究方向。

### 3.4 本节小结

实证研究可以揭示 Rust 程序中安全问题的产生机理,理解 Rust 的安全规则和 `unsafe` 机制对程序的影响。实证研究的结果对 Rust 语言安全的其他研究方向具有重要的指导意义。例如,可以基于实证研究的结果,研究漏洞检测技术、安全增强技术、自动化程序验证技术等。但是,随着 Rust 的应用越来越广泛,针对 Rust 语言安全的实证研究应该使用包括全部重要安全漏洞、覆盖整个 Rust 生态系统的数据集,同时对安全性与语言机制的联系进行深入研究,提出对语言机制使用的最佳安全实践。

## 4 漏洞检测研究

漏洞检测针对给定的程序和需要检验的安全性质,来检测程序可能的行为和状态,是否都满足待检验的安全性质。漏洞检测是提高软件安全性、减少安全漏洞的基本手段和重要方法。

Rust 引入了许多新型语言机制,这对 Rust 漏洞检测研究提出了新的挑战,漏洞检测研究已经成为 Rust 语言安全研究的重要方向之一。本文按照检测技术将现有的 Rust 漏洞检测研究分为基于程序分析和基于模糊测试两大类,并且对已有的研究工作进行了分析、总结和对比,结果如表 3 所示,其中○示该工作没有提及具体的误报率,●表示误报率较低,●表示误报率较高;数据结构包括 Rust 编译器提供的 3 种中间表示:抽象语法树(Abstract Syntax Tree, AST)、高级中间表示(High-Level Intermediate Rep



表 3 Rust 漏洞检测研究工作总结分析

Table 3 Summary and Analysis of Rust Bug Detection Research Work

检测方法	研究工作	主要技术	辅助技术	数据结构	漏洞类型	误报率	研究进展总结
程序分析	UnsafeFencer	模式匹配	动态检测	AST	内存	○	程序分析是目前最主流的 Rust 漏洞检测技术, 但是大多数现有工作只能检测内存安全漏洞; 将程序分析和其他技术融合也是一个研究趋势。
	SafeDrop	数据流分析	污点分析	MIR	内存	●	
	Rupair	数据流分析	-	AST、MIR	内存	●	
	safeIPC	数据流分析	运行时检查	MIR	内存	○	
	MirChecker	抽象解释	约束求解	MIR	内存	●	
	Rudra	数据流分析	-	MIR、HIR	内存	●	
	Stuck-me-not	数据流分析	-	MIR	并发	●	
模糊测试	RUSTY	属性测试	混合执行测试	源代码	内存	○	模糊测试可以用于 Rust 漏洞检测, 其主要挑战是自动生成有效的测试用例。
	RULF	模糊目标生成	程序合成	源代码	-	○	
	Dewey	约束逻辑	-	源代码	-	○	

resentation, HIR)、中级中间表示(M-id-level Intermediate Representation, MIR)。

4.1 程序分析

程序分析技术通过对程序的内部运作流程进行自动化分析处理, 来获取程序的特征和属性, 以进一步确定程序的安全性或正确性等属性。程序分析是 Rust 漏洞检测研究中最常用的技术。现有基于程序分析的 Rust 漏洞检测研究主要使用数据流分析、抽象解释、模式匹配等分析技术, 且主要针对内存安全漏洞。本小节将首先按照漏洞检测的全面程度梳理针对内存安全漏洞的相关工作, 然后介绍目前唯一的并发安全漏洞检测研究。

Rust 中不正确的裸指针解引用操作, 极易导致内存安全问题, Huang 等<sup>[41]</sup>详细研究了 Rust 程序中与裸指针解引用相关的安全问题, 并分析了由裸指针解引用导致的多种安全漏洞, 包括多个可变引用、修改不可变值和访问自由值等。他们还提出了一种实用的、可移植的方法来检测不安全的裸指针解引用操作, 该方法基于模式匹配来静态识别可能生成非法多重可变引用的函数, 并在程序运行时检查内存错误。该研究还基于此方法实现了一个针对不安全裸指针解引用的漏洞检测工具 UnsafeFencer, 该工具只平均增加了 4.13% 的运行时开销。但是该方法缺乏通用性, 实验数据表明, 在收集到的 8843 个测试用例中, 由于环境配置问题, 只有 5703 个可以使用 ([42]) 该方法进行漏洞检测。

Cui 等<sup>[42]</sup>研究了 Rust 中错误的内存释放问题, 并提出了一种路径敏感的静态数据流分析方法 SafeDrop, 来检测 Rust 程序中的内存释放错误。该方法首先在 MIR 上进行数据流分析, 以提取有价值的 ([43]) 路径, 并采用一种基于 Tarjan 算法的新方法来合并冗余路径; 然后使用过程间分析收集每条数据流路

径的变量别名; 最后基于建立的别名集, 搜索可能出现错误的内存释放的代码模式, 生成错误报告。该研究在 Rust CVEs 和 Rust crates 上对 SafeDrop 进行了全面评测, 检测出了 15 个新的错误内存释放漏洞, 而且只增加了少量的编译时间开销。但是, SafeDrop 不适用于大规模程序, 对于大规模程序, SafeDrop 的误报率会高达 94%~97%, 同时, 由于算法的局限性, SafeDrop 无法收集到所有别名关系, 从而产生漏报。

Rust 缓冲区溢出漏洞占 Rust 内存安全漏洞的 52.5%, 是最严重的内存安全漏洞之一, 其中, 由整数运算溢出导致的缓冲区溢出漏洞, 被称为 IO2BO 漏洞。Rupair<sup>[43]</sup>是针对 IO2BO 漏洞的漏洞检测与修复研究。Rupair 通过在 AST 和 MIR 上进行活跃分析, 计算 unsafe 块中的活跃变量, 再对每个活跃的向量类型的变量, 通过到达定义分析, 计算出该变量潜在的定义点。当该变量的定义点是在安全的 Rust 代码中且满足溢出检测模式时, 将其添加到潜在漏洞集中。为降低误报率, Rupair 还通过约束生成和约束求解对潜在漏洞 ([44]) 集进行筛选, 确定真实漏洞。最后, Rupair 采用参数调整和插入数据保护来进行漏洞自动修复。Rupair 是目前唯一涉及 Rust 漏洞自动修复的工作, 为自动修复其他类型的安全漏洞提供了有价值的参考。

在进程间通信中, 共享数据的类型信息不匹配会导致类型混淆漏洞。Switzer 等<sup>[44]</sup>研究了 Rust 中的类型混淆漏洞, 并提出了一个类型混淆漏洞检测算法。该算法通过在 MIR ([45]) ([46]) ([47]) ([48]) ([49]) ([50]) ([51]) ([52]) ([53]) ([54]) ([55]) ([56]) ([57]) ([58]) ([59]) ([60]) ([61]) ([62]) ([63]) ([64]) ([65]) ([66]) ([67]) ([68]) ([69]) ([70]) ([71]) ([72]) ([73]) ([74]) ([75]) ([76]) ([77]) ([78]) ([79]) ([80]) ([81]) ([82]) ([83]) ([84]) ([85]) ([86]) ([87]) ([88]) ([89]) ([90]) ([91]) ([92]) ([93]) ([94]) ([95]) ([96]) ([97]) ([98]) ([99]) ([100]) ([101]) ([102]) ([103]) ([104]) ([105]) ([106]) ([107]) ([108]) ([109]) ([110]) ([111]) ([112]) ([113]) ([114]) ([115]) ([116]) ([117]) ([118]) ([119]) ([120]) ([121]) ([122]) ([123]) ([124]) ([125]) ([126]) ([127]) ([128]) ([129]) ([130]) ([131]) ([132]) ([133]) ([134]) ([135]) ([136]) ([137]) ([138]) ([139]) ([140]) ([141]) ([142]) ([143]) ([144]) ([145]) ([146]) ([147]) ([148]) ([149]) ([150]) ([151]) ([152]) ([153]) ([154]) ([155]) ([156]) ([157]) ([158]) ([159]) ([160]) ([161]) ([162]) ([163]) ([164]) ([165]) ([166]) ([167]) ([168]) ([169]) ([170]) ([171]) ([172]) ([173]) ([174]) ([175]) ([176]) ([177]) ([178]) ([179]) ([180]) ([181]) ([182]) ([183]) ([184]) ([185]) ([186]) ([187]) ([188]) ([189]) ([190]) ([191]) ([192]) ([193]) ([194]) ([195]) ([196]) ([197]) ([198]) ([199]) ([200]) ([201]) ([202]) ([203]) ([204]) ([205]) ([206]) ([207]) ([208]) ([209]) ([210]) ([211]) ([212]) ([213]) ([214]) ([215]) ([216]) ([217]) ([218]) ([219]) ([220]) ([221]) ([222]) ([223]) ([224]) ([225]) ([226]) ([227]) ([228]) ([229]) ([230]) ([231]) ([232]) ([233]) ([234]) ([235]) ([236]) ([237]) ([238]) ([239]) ([240]) ([241]) ([242]) ([243]) ([244]) ([245]) ([246]) ([247]) ([248]) ([249]) ([250]) ([251]) ([252]) ([253]) ([254]) ([255]) ([256]) ([257]) ([258]) ([259]) ([260]) ([261]) ([262]) ([263]) ([264]) ([265]) ([266]) ([267]) ([268]) ([269]) ([270]) ([271]) ([272]) ([273]) ([274]) ([275]) ([276]) ([277]) ([278]) ([279]) ([280]) ([281]) ([282]) ([283]) ([284]) ([285]) ([286]) ([287]) ([288]) ([289]) ([290]) ([291]) ([292]) ([293]) ([294]) ([295]) ([296]) ([297]) ([298]) ([299]) ([300]) ([301]) ([302]) ([303]) ([304]) ([305]) ([306]) ([307]) ([308]) ([309]) ([310]) ([311]) ([312]) ([313]) ([314]) ([315]) ([316]) ([317]) ([318]) ([319]) ([320]) ([321]) ([322]) ([323]) ([324]) ([325]) ([326]) ([327]) ([328]) ([329]) ([330]) ([331]) ([332]) ([333]) ([334]) ([335]) ([336]) ([337]) ([338]) ([339]) ([340]) ([341]) ([342]) ([343]) ([344]) ([345]) ([346]) ([347]) ([348]) ([349]) ([350]) ([351]) ([352]) ([353]) ([354]) ([355]) ([356]) ([357]) ([358]) ([359]) ([360]) ([361]) ([362]) ([363]) ([364]) ([365]) ([366]) ([367]) ([368]) ([369]) ([370]) ([371]) ([372]) ([373]) ([374]) ([375]) ([376]) ([377]) ([378]) ([379]) ([380]) ([381]) ([382]) ([383]) ([384]) ([385]) ([386]) ([387]) ([388]) ([389]) ([390]) ([391]) ([392]) ([393]) ([394]) ([395]) ([396]) ([397]) ([398]) ([399]) ([400]) ([401]) ([402]) ([403]) ([404]) ([405]) ([406]) ([407]) ([408]) ([409]) ([410]) ([411]) ([412]) ([413]) ([414]) ([415]) ([416]) ([417]) ([418]) ([419]) ([420]) ([421]) ([422]) ([423]) ([424]) ([425]) ([426]) ([427]) ([428]) ([429]) ([430]) ([431]) ([432]) ([433]) ([434]) ([435]) ([436]) ([437]) ([438]) ([439]) ([440]) ([441]) ([442]) ([443]) ([444]) ([445]) ([446]) ([447]) ([448]) ([449]) ([450]) ([451]) ([452]) ([453]) ([454]) ([455]) ([456]) ([457]) ([458]) ([459]) ([460]) ([461]) ([462]) ([463]) ([464]) ([465]) ([466]) ([467]) ([468]) ([469]) ([470]) ([471]) ([472]) ([473]) ([474]) ([475]) ([476]) ([477]) ([478]) ([479]) ([480]) ([481]) ([482]) ([483]) ([484]) ([485]) ([486]) ([487]) ([488]) ([489]) ([490]) ([491]) ([492]) ([493]) ([494]) ([495]) ([496]) ([497]) ([498]) ([499]) ([500]) ([501]) ([502]) ([503]) ([504]) ([505]) ([506]) ([507]) ([508]) ([509]) ([510]) ([511]) ([512]) ([513]) ([514]) ([515]) ([516]) ([517]) ([518]) ([519]) ([520]) ([521]) ([522]) ([523]) ([524]) ([525]) ([526]) ([527]) ([528]) ([529]) ([530]) ([531]) ([532]) ([533]) ([534]) ([535]) ([536]) ([537]) ([538]) ([539]) ([540]) ([541]) ([542]) ([543]) ([544]) ([545]) ([546]) ([547]) ([548]) ([549]) ([550]) ([551]) ([552]) ([553]) ([554]) ([555]) ([556]) ([557]) ([558]) ([559]) ([560]) ([561]) ([562]) ([563]) ([564]) ([565]) ([566]) ([567]) ([568]) ([569]) ([570]) ([571]) ([572]) ([573]) ([574]) ([575]) ([576]) ([577]) ([578]) ([579]) ([580]) ([581]) ([582]) ([583]) ([584]) ([585]) ([586]) ([587]) ([588]) ([589]) ([590]) ([591]) ([592]) ([593]) ([594]) ([595]) ([596]) ([597]) ([598]) ([599]) ([600]) ([601]) ([602]) ([603]) ([604]) ([605]) ([606]) ([607]) ([608]) ([609]) ([610]) ([611]) ([612]) ([613]) ([614]) ([615]) ([616]) ([617]) ([618]) ([619]) ([620]) ([621]) ([622]) ([623]) ([624]) ([625]) ([626]) ([627]) ([628]) ([629]) ([630]) ([631]) ([632]) ([633]) ([634]) ([635]) ([636]) ([637]) ([638]) ([639]) ([640]) ([641]) ([642]) ([643]) ([644]) ([645]) ([646]) ([647]) ([648]) ([649]) ([650]) ([651]) ([652]) ([653]) ([654]) ([655]) ([656]) ([657]) ([658]) ([659]) ([660]) ([661]) ([662]) ([663]) ([664]) ([665]) ([666]) ([667]) ([668]) ([669]) ([670]) ([671]) ([672]) ([673]) ([674]) ([675]) ([676]) ([677]) ([678]) ([679]) ([680]) ([681]) ([682]) ([683]) ([684]) ([685]) ([686]) ([687]) ([688]) ([689]) ([690]) ([691]) ([692]) ([693]) ([694]) ([695]) ([696]) ([697]) ([698]) ([699]) ([700]) ([701]) ([702]) ([703]) ([704]) ([705]) ([706]) ([707]) ([708]) ([709]) ([710]) ([711]) ([712]) ([713]) ([714]) ([715]) ([716]) ([717]) ([718]) ([719]) ([720]) ([721]) ([722]) ([723]) ([724]) ([725]) ([726]) ([727]) ([728]) ([729]) ([730]) ([731]) ([732]) ([733]) ([734]) ([735]) ([736]) ([737]) ([738]) ([739]) ([740]) ([741]) ([742]) ([743]) ([744]) ([745]) ([746]) ([747]) ([748]) ([749]) ([750]) ([751]) ([752]) ([753]) ([754]) ([755]) ([756]) ([757]) ([758]) ([759]) ([760]) ([761]) ([762]) ([763]) ([764]) ([765]) ([766]) ([767]) ([768]) ([769]) ([770]) ([771]) ([772]) ([773]) ([774]) ([775]) ([776]) ([777]) ([778]) ([779]) ([780]) ([781]) ([782]) ([783]) ([784]) ([785]) ([786]) ([787]) ([788]) ([789]) ([790]) ([791]) ([792]) ([793]) ([794]) ([795]) ([796]) ([797]) ([798]) ([799]) ([800]) ([801]) ([802]) ([803]) ([804]) ([805]) ([806]) ([807]) ([808]) ([809]) ([810]) ([811]) ([812]) ([813]) ([814]) ([815]) ([816]) ([817]) ([818]) ([819]) ([820]) ([821]) ([822]) ([823]) ([824]) ([825]) ([826]) ([827]) ([828]) ([829]) ([830]) ([831]) ([832]) ([833]) ([834]) ([835]) ([836]) ([837]) ([838]) ([839]) ([840]) ([841]) ([842]) ([843]) ([844]) ([845]) ([846]) ([847]) ([848]) ([849]) ([850]) ([851]) ([852]) ([853]) ([854]) ([855]) ([856]) ([857]) ([858]) ([859]) ([860]) ([861]) ([862]) ([863]) ([864]) ([865]) ([866]) ([867]) ([868]) ([869]) ([870]) ([871]) ([872]) ([873]) ([874]) ([875]) ([876]) ([877]) ([878]) ([879]) ([880]) ([881]) ([882]) ([883]) ([884]) ([885]) ([886]) ([887]) ([888]) ([889]) ([890]) ([891]) ([892]) ([893]) ([894]) ([895]) ([896]) ([897]) ([898]) ([899]) ([900]) ([901]) ([902]) ([903]) ([904]) ([905]) ([906]) ([907]) ([908]) ([909]) ([910]) ([911]) ([912]) ([913]) ([914]) ([915]) ([916]) ([917]) ([918]) ([919]) ([920]) ([921]) ([922]) ([923]) ([924]) ([925]) ([926]) ([927]) ([928]) ([929]) ([930]) ([931]) ([932]) ([933]) ([934]) ([935]) ([936]) ([937]) ([938]) ([939]) ([940]) ([941]) ([942]) ([943]) ([944]) ([945]) ([946]) ([947]) ([948]) ([949]) ([950]) ([951]) ([952]) ([953]) ([954]) ([955]) ([956]) ([957]) ([958]) ([959]) ([960]) ([961]) ([962]) ([963]) ([964]) ([965]) ([966]) ([967]) ([968]) ([969]) ([970]) ([971]) ([972]) ([973]) ([974]) ([975]) ([976]) ([977]) ([978]) ([979]) ([980]) ([981]) ([982]) ([983]) ([984]) ([985]) ([986]) ([987]) ([988]) ([989]) ([990]) ([991]) ([992]) ([993]) ([994]) ([995]) ([996]) ([997]) ([998]) ([999]) ([1000]) ([1001]) ([1002]) ([1003]) ([1004]) ([1005]) ([1006]) ([1007]) ([1008]) ([1009]) ([1010]) ([1011]) ([1012]) ([1013]) ([1014]) ([1015]) ([1016]) ([1017]) ([1018]) ([1019]) ([1020]) ([1021]) ([1022]) ([1023]) ([1024]) ([1025]) ([1026]) ([1027]) ([1028]) ([1029]) ([1030]) ([1031]) ([1032]) ([1033]) ([1034]) ([1035]) ([1036]) ([1037]) ([1038]) ([1039]) ([1040]) ([1041]) ([1042]) ([1043]) ([1044]) ([1045]) ([1046]) ([1047]) ([1048]) ([1049]) ([1050]) ([1051]) ([1052]) ([1053]) ([1054]) ([1055]) ([1056]) ([1057]) ([1058]) ([1059]) ([1060]) ([1061]) ([1062]) ([1063]) ([1064]) ([1065]) ([1066]) ([1067]) ([1068]) ([1069]) ([1070]) ([1071]) ([1072]) ([1073]) ([1074]) ([1075]) ([1076]) ([1077]) ([1078]) ([1079]) ([1080]) ([1081]) ([1082]) ([1083]) ([1084]) ([1085]) ([1086]) ([1087]) ([1088]) ([1089]) ([1090]) ([1091]) ([1092]) ([1093]) ([1094]) ([1095]) ([1096]) ([1097]) ([1098]) ([1099]) ([1100]) ([1101]) ([1102]) ([1103]) ([1104]) ([1105]) ([1106]) ([1107]) ([1108]) ([1109]) ([1110]) ([1111]) ([1112]) ([1113]) ([1114]) ([1115]) ([1116]) ([1117]) ([1118]) ([1119]) ([1120]) ([1121]) ([1122]) ([1123]) ([1124]) ([1125]) ([1126]) ([1127]) ([1128]) ([1129]) ([1130]) ([1131]) ([1132]) ([1133]) ([1134]) ([1135]) ([1136]) ([1137]) ([1138]) ([1139]) ([1140]) ([1141]) ([1142]) ([1143]) ([1144]) ([1145]) ([1146]) ([1147]) ([1148]) ([1149]) ([1150]) ([1151]) ([1152]) ([1153]) ([1154]) ([1155]) ([1156]) ([1157]) ([1158]) ([1159]) ([1160]) ([1161]) ([1162]) ([1163]) ([1164]) ([1165]) ([1166]) ([1167]) ([1168]) ([1169]) ([1170]) ([1171]) ([1172]) ([1173]) ([1174]) ([1175]) ([1176]) ([1177]) ([1178]) ([1179]) ([1180]) ([1181]) ([1182]) ([1183]) ([1184]) ([1185]) ([1186]) ([1187]) ([1188]) ([1189]) ([1190]) ([1191]) ([1192]) ([1193]) ([1194]) ([1195]) ([1196]) ([1197]) ([1198]) ([1199]) ([1200]) ([1201]) ([1202]) ([1203]) ([1204]) ([1205]) ([1206]) ([1207]) ([1208]) ([1209]) ([1210]) ([1211]) ([1212]) ([1213]) ([1214]) ([1215]) ([1216]) ([1217]) ([1218]) ([1219]) ([1220]) ([1221]) ([1222]) ([1223]) ([1224]) ([1225]) ([1226]) ([1227]) ([1228]) ([1229]) ([1230]) ([1231]) ([1232]) ([1233]) ([1234]) ([1235]) ([1236]) ([1237]) ([1238]) ([1239]) ([1240]) ([1241]) ([1242]) ([1243]) ([1244]) ([1245]) ([1246]) ([1247]) ([1248]) ([1249]) ([1250]) ([1251]) ([1252]) ([1253]) ([1254]) ([1255]) ([1256]) ([1257]) ([1258]) ([1259]) ([1260]) ([1261]) ([1262]) ([1263]) ([1264]) ([1265]) ([1266]) ([1267]) ([1268]) ([1269]) ([1270]) ([1271]) ([1272]) ([1273]) ([1274]) ([1275]) ([1276]) ([1277]) ([1278]) ([1279]) ([1280]) ([1281]) ([1282]) ([1283]) ([1284]) ([1285]) ([1286]) ([1287]) ([1288]) ([1289]) ([1290]) ([1291]) ([1292]) ([1293]) ([1294]) ([1295]) ([1296]) ([1297]) ([1298]) ([1299]) ([1300]) ([1301]) ([1302]) ([1303]) ([1304]) ([1305]) ([1306]) ([1307]) ([1308]) ([1309]) ([1310]) ([1311]) ([1312]) ([1313]) ([1314]) ([1315]) ([1316]) ([1317]) ([1318]) ([1319]) ([1320]) ([1321]) ([1322]) ([1323]) ([1324]) ([1325]) ([1326]) ([1327]) ([1328]) ([1329]) ([1330]) ([1331]) ([1332]) ([1333]) ([1334]) ([1335]) ([1336]) ([1337]) ([1338]) ([1339]) ([1340]) ([1341]) ([1342]) ([1343]) ([1344]) ([1345]) ([1346]) ([1347]) ([1348]) ([1349]) ([1350]) ([1351]) ([1352]) ([1353]) ([1354]) ([1355]) ([1356]) ([1357]) ([1358]) ([1359]) ([1360]) ([1361]) ([1362]) ([1363]) ([1364]) ([1365]) ([1366]) ([1367]) ([1368]) ([1369]) ([1370]) ([1371]) ([1372]) ([1373]) ([1374]) ([1375]) ([1376]) ([1377]) ([1378]) ([1379]) ([1380]) ([1381]) ([1382]) ([1383]) ([1384]) ([1385]) ([1386]) ([1387]) ([1388]) ([1389]) ([1390]) ([1391]) ([1392]) ([1393]) ([1394]) ([1395]) ([1396]) ([1397]) ([1398]) ([1399]) ([1400]) ([1401]) ([1402]) ([1403]) ([1404]) ([1405]) ([1406]) ([1407]) ([1408]) ([1409]) ([1410]) ([1411]) ([1412]) ([1413]) ([1414]) ([1415]) ([1416]) ([1417]) ([1418]) ([1419]) ([1420]) ([1421]) ([1422]) ([1423]) ([1424]) ([1425]) ([1426]) ([1427]) ([1428]) ([1429]) ([1430]) ([1431]) ([1432]) ([1433]) ([1434]) ([1435]) ([1436]) ([1437]) ([1438]) ([1439]) ([1440]) ([1441]) ([1442]) ([1443]) ([1444]) ([1445]) ([1446]) ([1447]) ([1448]) ([1449]) ([1450]) ([1451]) ([1452]) ([1453]) ([1454]) ([1455]) ([1456]) ([1457]) ([1458]) ([1459]) ([1460]) ([1461]) ([1462]) ([1463]) ([1464]) ([1465]) ([1466]) ([1467]) ([1468]) ([1469]) ([1470]) ([1471]) ([1472]) ([1473]) ([1474]) ([1475]) ([1476]) ([1477]) ([1478]) ([1479]) ([1480]) ([1481]) ([1482]) ([1483]) ([1484]) ([1485]) ([1486]) ([1487]) ([1488]) ([1489]) ([1490]) ([1491]) ([1492]) ([1493]) ([1494]) ([1495]) ([1496]) ([1497]) ([1498]) ([1499]) ([1500]) ([1501]) ([1502]) ([1503]) ([1504]) ([1505]) ([1506]) ([1507]) ([1508]) ([1509]) ([1510]) ([1511]) ([1512]) ([1513]) ([1514]) ([1515]) ([1516]) ([1517]) ([1518]) ([1519]) ([1520]) ([1521]) ([1522]) ([1523]) ([1524]) ([1525]) ([1526]) ([1527]) ([1528]) ([1529]) ([1530]) ([1531]) ([1532]) ([1533]) ([1534]) ([1535]) ([1536]) ([1537]) ([1538]) ([1539]) ([1540]) ([1541]) ([1542]) ([1543]) ([1544]) ([1545]) ([1546]) ([1547]) ([1548]) ([1549]) ([1550]) ([1551]) ([1552]) ([1553]) ([1554]) ([1555]) ([1556]) ([1557]) ([1558]) ([1559]) ([1560]) ([1561]) ([1562]) ([1563]) ([1564]) ([1565]) ([1566]) ([1567]) ([1568]) ([1569]) ([1570]) ([1571]) ([1572]) ([1573]) ([1574]) ([1575]) ([1576]) ([1577]) ([1578]) ([1579]) ([1580]) ([1581]) ([1582]) ([1583]) ([1584]) ([1585]) ([1586]) ([1587]) ([1588]) ([1589]) ([1590]) ([1591]) ([1592]) ([1593]) ([1594]) ([1595]) ([1596]) ([1597]) ([1598]) ([1599]) ([1600]) ([1601]) ([1602]) ([1603]) ([1604]) ([1605]) ([1606]) ([1607]) ([1608]) ([1609]) ([1610]) ([1611]) ([1612]) ([1613]) ([1614]) ([1615]) ([1616]) ([1617]) ([1618]) ([1619]) ([1620]) ([1621]) ([1622]) ([1623]) ([1624]) ([1625]) ([1626]) ([1627]) ([1628]) ([1629]) ([1630]) ([1631]) ([1632]) ([1633]) ([1634]) ([1635]) ([1636]) ([1637]) ([1638]) ([1639]) ([1640]) ([1641]) ([1642]) ([1643]) ([1644]) ([1645]) ([1646]) ([1647]) ([1648]) ([1649]) ([1650]) ([1651]) ([1652]) ([1653]) ([1654]) ([1655]) ([1656]) ([1657]) ([1658]) ([1659]) ([1660]) ([1661]) ([1662]) ([1663]) ([1664]) ([1665]) ([1666]) ([1667]) ([1668]) ([1669]) ([1670]) ([1671]) ([1672]) ([1673]) ([1674]) ([1675]) ([1676]) ([1677]) ([1678]) ([1679]) ([1680]) ([1681]) ([1682]) ([1683]) ([1684]) ([1685]) ([1686]) ([1687]) ([1688]) ([1689]) ([1690]) ([1691]) ([1692]) ([1693]) ([1694]) ([1695]) ([1696]) ([1697]) ([1698]) ([1699]) ([1700]) ([1701]) ([1702]) ([1703]) ([1704]) ([1705]) ([1706]) ([1707]) ([1708]) ([1709]) ([1710]) ([1711]) ([1712]) ([1713]) ([1714]) ([1715]) ([1716]) ([1717]) ([1718]) ([1719]) ([1720]) ([1721]) ([1722]) ([1723]) ([1724]) ([1725]) ([1726]) ([1727]) ([1728]) ([1729]) ([1730]) ([1731]) ([1732]) ([1733]) ([1734]) ([1735]) ([1736]) ([1737]) ([1738]) ([1739]) ([1740]) ([1741]) ([1742]) ([1743]) ([1744]) ([1745]) ([1746]) ([1747]) ([1748]) ([1749]) ([1750]) ([1751]) ([1752]) ([1753]) ([1754]) ([1755]) ([1756]) ([1757]) ([1758]) ([1759]) ([1760]) ([1761]) ([1762]) ([1763]) ([1764]) ([1765]) ([1766]) ([1767]) ([1768]) ([1769]) ([1770]) ([1771]) ([1772]) ([1773]) ([1774]) ([1775]) ([1776]) ([1777]) ([1778]) ([1779]) ([1780]) ([1781]) ([1782]) ([1783]) ([1784]) ([1785]) ([1786]) ([1787]) ([1788]) ([1789]) ([1790]) ([1791]) ([1792]) ([1793]) ([1794]) ([1795]) ([1796]) ([1797]) ([1798]) ([1799]) ([1800]) ([1801]) ([1802]) ([1803]) ([1804]) ([1805]) ([1806]) ([1807]) ([1808]) ([1809]) ([1810]) ([1811]) ([1812]) ([1813]) ([1814]) ([1815]) ([1816]) ([1817]) ([1818]) ([1819]) ([1820]) ([1821]) ([1822]) ([1823]) ([1824]) ([1825]) ([1826]) ([1827]) ([1828]) ([1829]) ([1830]) ([1831]) ([1832]) ([1833]) ([1834]) ([1835]) ([1836]) ([1837]) ([1838]) ([1839]) ([1840]) ([1841]) ([1842]) ([1843]) ([1844]) ([1845]) ([1846]) ([1847]) ([1848]) ([1849]) ([1850]) ([1851]) ([1852]) ([1853]) ([1854]) ([1855]) ([1856]) ([1857]) ([1858]) ([1859]) ([1860]) ([1861]) ([1862]) ([1863]) ([1864]) ([1865]) ([1866]) ([1867]) ([1868]) ([1869]) ([1870]) ([1871]) ([1872]) ([1873]) ([1874]) ([1875]) ([1876]) ([1877]) ([1878]) ([1879]) ([1880]) ([1881]) ([1882]) ([1883]) ([1884]) ([1885]) ([1886]) ([1887]) ([1888]) ([1889]) ([1890]) ([1891]) ([1892]) ([1893]) ([1894]) ([1895]) ([1896]) ([1897]) ([1898]) ([1899]) ([1900]) ([1901]) ([1902]) ([1903]) ([1904]) ([1905]) ([1906]) ([1907]) ([1908]) ([1909]) ([1910]) ([1911]) ([1912]) ([1913]) ([1914]) ([1915]) ([1916]) ([1917]) ([1918]) ([1919]) ([1920]) ([1921]) ([1922]) ([1923]) ([1924]) ([1925]) ([1926]) ([1927]) ([1928]) ([1929]) ([1930]) ([1931]) ([1932]) ([1933]) ([1934]) ([1935]) ([1936]) ([1937]) ([1938]) ([1939]) ([1940]) ([19

间分别平均增加了 106%和 126%。但是, 该研究目前只能检测小型程序, 扩展到实际大型软件仍然是有挑战性的研究课题。

上述研究都只针对具体的某一类 Rust 内存安全漏洞。MirChecker<sup>[45]</sup>是一个可以同时检测运行时崩溃和由悬空指针引起的内存安全漏洞的漏洞检测框架。MirChecker 首先基于 Rust 类型系统提供的信息, 利用抽象解释理论和形式化语义, 为数据流分析提供基础, 然后使用约束求解技术, 来检测潜在的运行时崩溃和内存安全漏洞。MirChecker 在真实的代码库中发现了 33 个未知漏洞, 其中 16 个是内存安全漏洞。但是, 由于 MirChecker 的内存模型是轻量级和语法驱动的, 所以它无法处理所有内存操作, 而且不支持许多 Rust 高级特性, 导致了高达 95.1%的误报率。

Rudra<sup>[46]</sup>是目前关于 Rust 内存安全漏洞检测的最新也是最全面的研究。该研究提出并实现了一个静态分析器 Rudra, 它通过数据流分析算法和发送/同步差异检查算法, 来检测 unsafe 代码中的与 panic、高阶不变量、发送/同步相关的内存安全漏洞。实验结果表明: Rudra 在 6.5h 内扫描了 Rust crates 上的所有包, 发现了 264 个新的内存安全漏洞。但 Rudra 的分析算法不能进行过程间分析, 不能对复杂程序的语义进行建模, 因此无法检测出由过程间交互引起的内存安全漏洞, 从而导致漏报; 同时, 由于对生命周期的建模不精确, 类型定义和基于 API 签名的推理受限, Rudra 还会产生误报。因此, 本文认为, Rudra 还可以通过更深层次的语义建模和使用更复杂的分析算法, 提高漏洞检测的精度。

上述研究都只针对 Rust 内存安全漏洞, 为了提出针对并发安全漏洞的检测技术, Ning 等<sup>[47]</sup>研究了基于 Rust 开发的区块链系统中常见的死锁安全漏洞, 并提出了一个检测重复上锁漏洞的工作表算法。该算法通过在 MIR 上进行数据流分析, 来追踪 lockguard 的生命周期, 从而静态检测重复上锁漏洞。基于该算法实现的工具 Stuck-me-not 在 11 个区块链系统上报告了 29 个未知漏洞, 只有一个误报; 并且, Stuck-me-not 只增加了 1.79%的编译时间。本文认为, 该研究只针对重复上锁漏洞, 是对 Rust 并发漏洞检测的初步探索, 在未来可以从两方面进行进一步研究: 一是对更多并发漏洞的检测研究; 二是将漏洞检测算法集成到 IDE 中, 在开发阶段就定位到程序中的并发安全漏洞。

本文认为, 在基于程序分析技术的 Rust 漏洞检测研究领域, 在未来有三个方向值得进一步探索:

一是对 Rust 的并发安全漏洞检测技术的研究; 二是对同时检测多种类型漏洞的研究; 三是对漏洞的自动修复理论和技术的研究。

## 4.2 模糊测试

模糊测试是一种重要的漏洞检测技术, 其核心思想是为程序提供大量测试用例, 在程序执行过程中监控程序的异常行为, 以发现程序漏洞。模糊测试技术也被广泛用于 Rust 安全漏洞的检测。本文将基于模糊测试的 Rust 漏洞检测研究总结为针对 Rust 程序、Rust 库、Rust 编译器三类, 并按此分类对现有工作进行梳理和对比, 深入分析其局限性。

RUSTY<sup>[48]</sup>使用属性测试(Property-based Testing)来对 Rust 程序进行模糊测试, 然后利用基于 SMT 求解器的混合执行引擎(Concolic Execution Engine), 来自动生成漏洞利用。同时, RUSTY 使用基于值的变异生成测试用例, 从而更具适应性。RUSTY 在从 GitHub 收集的 Rust 项目上进行实验, 实验结果表明, 它可以检测出内存泄漏、缓冲区溢出等内存安全漏洞。本文认为, 该研究存在局限性: 首先, 属性测试要求测试人员熟悉被测试程序, 而不能以黑盒的方式进行; 其次, 程序需要满足的属性必须手工构建, 而不能自动化完成。

对 Rust 库 API 的不正确调用, 可能会导致严重安全漏洞。要对 Rust 库进行模糊测试, 一般需要生成一组 Rust 库 API 的调用序列, 作为被测试的目标程序。RULF<sup>[49]</sup>是一个程序自动生成算法, 该算法首先使用带剪枝的宽度优先算法, 来确定一组 API 序列; 然后提炼 API 序列集, 获得 API 覆盖率最小的子集作为目标程序集; 最后利用程序合成技术来生成可以通过编译器检查的程序。该研究在 11 个 Rust 官方库和 3 个 GitHub 上流行的库中检测到了 30 个未知的漏洞。但是, 该算法依然具有局限性: 首先, 它只支持基本类型上的依赖关系推导, 不支持多态 API; 其次, 该算法只支持函数和方法的 API, 不支持宏 API 和异步 API; 最后, RULF 生成的目标程序只支持普通的模糊测试工具, 不支持 LibFuzzer<sup>[50]</sup>这类进程内模糊测试工具。

语言模糊测试是一种有效的对编译器或解释器进行模糊测试的技术。Dewey<sup>[51]</sup>等提出了一种对 Rust 的类型检查器进行模糊测试的方法, 该方法利用约束逻辑编程<sup>[52]</sup>, 来生成类型良好的 Rust 程序, 可用于检测类型检查器中的精度错误、健全性错误和一致性错误。该研究实现的原型系统在超过六百个小时的时间里, 生成了近九亿个 Rust 程序, 并将这些程序作为测试用例对 Rust 的类型检查器进行模糊测

试, 成功发现了 18 个漏洞。但是, 本文认为, 该方法也存在一定的不足: 首先, 该方法的时间开销远高于静态程序分析; 其次, 对于报告的错误, 还需要人工复核, 检测效率较低; 最后, 该方法只针对 Rust 编译器中的类型检查器, 要对 Rust 编译器这样一个多模块的复杂软件进行语言模糊测试, 需要全面考虑其各模块的具体功能, 利用程序合成技术<sup>[53]</sup>来自动生成大量的有效 Rust 程序作为测试用例, 这是一项非常具有挑战性的工作。

本文认为, 模糊测试技术已经成功用于对 Rust 程序、Rust 库和 Rust 编译器进行漏洞检测。但是相关工作都存在检测效率低、需要大量计算资源等不足。在针对 Rust 的模糊测试研究中, 未来有两个可能的研究方向: 一是研究利用程序合成技术, 来自动生成更有效的测试用例, 从而提高路径覆盖率, 减少测试时间, 获得更好的测试效果; 二是研究更好的自动生成漏洞利用的理论和技術, 减少手动分析程序异常来定位程序漏洞的成本。

4.3 本节小结

漏洞检测是软件安全领域的重要研究方向, 也是 Rust 语言安全研究中非常活跃的研究方向。程序分析是 Rust 漏洞检测的最主流技术, 通常通过在

Rust 程序的中间表示上进行数据流分析, 来收集程序的属性和特征, 还经常结合动态检测和约束求解等辅助技术进行漏洞检测, 具有时间开销少, 检测效率高的优点, 但一般存在一定的误报率。模糊测试则依赖于被测试目标的运行时信息, 其发现的漏洞一定是可达的, 因此检测结果更加精确, 但一般需要运行大量的测试用例, 因此时间开销大, 检测效率低。

5 安全增强研究

安全增强是安全研究的重要组成部分, 对于程序中的安全漏洞, 安全增强研究利用相关实证研究和漏洞检测研究的结果, 通过在软件或系统中引入特定安全机制, 增强软件系统的防御功能, 防止或阻断漏洞的发生。

本文按照增强技术将现有的 Rust 安全增强研究分为基于权限分离和基于程序分析两大类, 其中, 基于权限分离的安全增强研究又主要分为内存隔离和代码沙箱两个方向。本小节将对 Rust 安全增强相关的各类研究进行梳理和总结, 并从主要技术、能防止的漏洞类型、是否实现自动化等方面进行深入对比, 结果如表 4 所示。

表 4 Rust 安全增强研究工作总结分析

Table 4 Summary and analysis of Rust safety enhancement research work

增强技术	主要技术	研究工作	辅助技术	漏洞类型	自动化	研究进展总结
权限分离	内存隔离	Fidelius Charm	-	内存	×	基于权限分离的 Rust 安全增强技术主要分为两类, 一类是将内存划分为不同的区域, 并分别进行访问控制, 另一类是利用代码沙箱技术来隔离不安全的代码或数据。现有的相关研究都主要防止内存安全漏洞的发生, 不涉及对并发安全漏洞的防护。
		XRust	静态分析	内存	✓	
		Galeed	伪指针	内存	✓	
	代码沙箱	Sandcrust	-	内存	✓	
		RUSBOX	静态分析	内存	✓	
程序分析	生命周期分析	Dominik	-	内存、并发	×	基于程序分析的安全增强研究通过分析 Rust 编译器提供的中间表示, 获得程序中变量的生命周期、变量的所有权、调用栈等信息, 从而防止产生内存和并发安全漏洞。
		VRLifeTime	-	内存、并发	✓	
	调用栈分析	Lindgren	-	内存	✓	

5.1 权限分离

权限分离技术通常通过将软硬件或各种计算实体和资源, 划分成不同的分组, 并赋予不同的权限, 从而分隔可能包含漏洞的代码和实现安全性增强。相关研究采用的技术路线主要分为两种: 一是通过将内存空间划分为不同区域, 并对每个区域进行访问控制, 来实现内存隔离; 二是利用代码沙箱技术来隔离代码或数据。

5.1.1 内存隔离

第一个利用内存隔离技术来对 Rust 程序进行安全增强的研究是由 Almohri 等<sup>[54]</sup>提出的 Fidelius

Charm 系统(简称 FC), 该系统增加了一组 FC 调用接口, 来控制内存区域的访问权限。FC 系统将整个内存地址空间划分成私有内存、安全只读内存和不安全内存三个部分: 私有内存存放不允许 unsafe 代码访问的数据; 安全只读内存存放会被 unsafe 代码只读访问的数据; 而不安全内存存放可能被 unsafe 代码任意读写访问的数据。为了禁止不安全代码修改内存页面的访问权限, FC 系统还对 mprotect 系统调用进行了监控。

本文认为, 尽管 FC 系统对数据所在页面做了明确的权限区分, 但其本质上还是一个粗粒度的防护

策略。FC 系统存在的问题主要有两个: 首先, 为了保护特定的程序数据, 程序员需要手动加入对 FC 接口的调用, 没有实现自动化, 对于大型程序而言, 这增加了系统的实现难度和程序员的编程负担; 其次, FC 对安全数据和不安全数据进行了绝对区分, 而在很多 Rust 程序中安全和不安全数据没有明确的边界, 这将导致 FC 的防护失效。例如, `unsafe` 代码块对程序数据进行了修改(如改变向量类型 `vec` 的值中的元数据), 但回到安全代码中才会对数据进行访问(如访问向量 `vec` 的元素), FC 无法阻止这类内存错误。

为了实现自动化和更有效的防护, Liu 等<sup>[55]</sup>提出了 XRust 技术, XRust 的核心是一个新型的内存分配器, 该分配器将内存划分为两个不相交的区域: 安全内存区和不安全内存区; XRust 保证不安全代码只能在不安全内存区内进行内存分配, 且其访问的内存不能超过不安全内存区边界; 为此, XRust 对不安全代码的访存操作进行显式地插桩和检查。XRust 使用了全局的静态程序分析, 能够跨安全和不安全代码进行全程序级检查。XRust 在真实漏洞测试代码上进行了实验, 实验结果表明, XRust 能够防止对已知安全漏洞的攻击, 且具有较高的执行性能(平均执行性能只比 `ptmalloc2` 分配器慢了不到 5%)。

本文认为, 相比于 FC 系统, XRust 更加有效但实现更加复杂, 它需要对 Rust 的生态系统做如下三方面的改变: 一是修改 Rust 编译器的源代码, 加入对新的内存分配接口的显式调用; 二是需要增加新的内存分配器; 三是需要修改应用程序的源代码, 从而进行程序插桩。同时, FC 系统只能通过系统调用将数据和不安全的外部函数隔离, 而 XRust 还可以处理安全代码中的共享不安全对象。但 XRust 也存在局限性: 由于 XRust 依赖于静态程序分析, 所以不能处理动态代码生成。针对这一局限性的一个潜在解决方案是跟踪动态生成的代码, 并进行运行时分析。

在内存隔离这一方向上的研究还有 Rivera 等<sup>[56]</sup>提出的 Galeed 技术, 该技术使用英特尔 CPU 上最新的内存保护密钥(Memory Protection Keys, MPK)特性<sup>[57]</sup>, 对内存进行权限隔离。Galeed 将内存划分为 Rust 内存和本地方法内存, 当进行外部 C 函数调用时, Galeed 关闭 Rust 内存的访问权限, 使得本地方法代码无法访问或破坏本地方法内存。而为了允许对 Rust 内存的访问, Galeed 使用了伪指针技术。本文认为, Galeed 的主要不足在于它缺乏程序的数据信息, 不能保护被 `unsafe` 代码访问的数据。

### 5.1.2 代码沙箱

Sandcrust 框架<sup>[58]</sup>是第一个使用沙箱技术, 对不

安全代码的内存访问进行隔离的研究。Sandcrust 将外部不安全函数调用进行了封装, 转换成了对沙箱进程中运行的函数的远程方法调用。本文认为, 尽管 Sandcrust 框架的防护策略能够通过沙箱和进程隔离, 有效隔离可能的内存安全漏洞, 但相比于进程内方法调用, 远程方法调用有更高的执行代价, 对于大量使用了本地代码的 Rust 程序, 需要进一步研究探讨该策略对其执行性能的影响。

另一个使用代码沙箱进行安全增强的重要研究是 RUSBOX<sup>[59]</sup>, RUSBOX 采用将静态程序分析与代码沙箱技术相结合的方法, 有效又高效地防止 Rust 程序中出现缓冲区溢出漏洞。由于 Rust 使用单进程执行模型, `unsafe` 代码中发生的缓冲区溢出, 可能会影响由安全代码分配的内存对象, 最终破坏整个地址空间中的数据。为了保证 Rust 中的数据完整性, RUSBOX 使用沙箱来隔离易受缓冲区溢出攻击的数据。RUSBOX 通过分析控制流图和调用图识别出所有潜在的缓冲区溢出漏洞, 然后利用 `tarpc` 框架来对可能存在漏洞的代码进行沙箱隔离。本文认为, 尽管 RUSBOX 采用了更精细的保护策略, 但由于其采用远程方法调用, 因此仍然具有客观的运行时间开销。

## 5.2 程序分析

程序分析也是 Rust 安全增强研究中的一个常用重要技术, 它通过分析 Rust 编译器中的 MIR、LLVM IR 等中间表示, 得到关于变量的生命周期、变量的所有权、程序的调用图等重要信息, 然后利用这些信息来发现潜在的安全威胁, 以排除安全漏洞, 从而构建更安全的软件系统。

生命周期是 Rust 的一个核心特性, 它是 Rust 编译器进行安全检查、自动化内存管理和安全并发编程的关键。然而, Rust 开发人员往往无法正确推断变量的生命周期, 这可能会导致严重的并发和内存安全漏洞。Dominik 等<sup>[60]</sup>研究了从 Rust 编译器, 尤其是借用检查器中进行信息提取的机制, 来获得变量生命周期错误相关的重要信息, 并将生命周期、生命周期之间的约束等, 通过图的形式表示出来, 以期尽早定位并修复代码中的漏洞。但本文认为, 该机制导出的信息通常非常复杂, 难以理解, 因此难以应用到实际大型软件系统的防护中。

Blaser 等<sup>[61]</sup>改进了 Dominik 等<sup>[60]</sup>的工作, 提出了一种更好的机制来创建变量生命周期图, 并开发了一个工具 Rust Life Assistant 来将生命周期信息可视化。但 Rust Life Assistant 只能将生命周期相关信息可视化, 不能直接指出可能存在的安全漏洞。进一步, Zhang 等<sup>[62]</sup>提出了 VRLifeTime, 该研究基于 Rust 编



译器中的 MIR 进行程序分析, 计算出变量的生命周期, 然后将变量的生命周期范围可视化, 并且指出潜在的内存和并发安全漏洞。

上述研究都是利用 Rust 程序的生命周期信息, 而 Lindgren 等<sup>[63]</sup>提出了针对 Rust 程序的静态调用栈分析方法, 该方法在编译时基于 LLVM IR 进行调用栈分析, 从而保证内存安全。该研究在 ARM 的 Cortex-M 系列微处理器上的实验结果表明: 它既可以提供堆栈使用估计, 又不会对开发人员造成限制。但是, 在将 Rust 源代码翻译到 LLVM IR 的过程中, 函数调用图重构会造成信息丢失, 这将导致调用图的精度降低。

### 5.3 本节小结

安全增强通过使用权限分离、程序分析等技术, 引入特定安全机制, 防止或修复安全漏洞, 增强程序安全性。Rust 中的所有权和生命周期等新型语言机制, 给 Rust 的安全增强研究提供了新的研究方向和机会。本文认为, Rust 安全增强研究在未来存在两个潜在研究方向: 一是研究和建立针对 unsafe 代码的安全增强技术; 二是研究和实现实际可用的安全缺陷可视化、安全漏洞自动定位与修复工具等, 来实现程序的自动安全增强。

## 6 形式化验证研究

形式化验证是基于严格数学基础, 验证程序是否符合预期的设计属性和安全规范的技术。形式化验证在形式语义和形式规约的基础上, 将程序的分析 and 验证问题转化为逻辑推理问题或形式模型的判定问题, 利用定理证明器、约束求解器或者原型工具来进行验证。

形式化验证研究是 Rust 语言安全研究的重要部分, 其相关文献数量的占比在 Rust 语言安全的四个方向中最高, 达到 43.47%。形式化验证与 Rust 语言安全研究的其他方向具有紧密联系。对于经过漏洞检测和安全增强的软件系统, 还需要利用形式化验证技术来严格证明程序的安全性和功能正确性, 同时, 形式化验证技术还可以用来验证安全增强技术的有效性。

使用形式化验证技术验证 Rust 程序的安全性, 是 Rust 语言安全研究领域的热点方向。本文将该方向上的研究归纳为两类: 一是对 Rust 形式语义的研究; 二是自动化程序验证研究。对于 Rust 形式语义研究, 本小节将按照时间顺序、相关研究之间的承接关系, 梳理和总结相关工作及其发展脉络。对于 Rust 自动化程序验证研究, 本文按照验证技术将现有研

究进一步分为模型检测、符号执行、定理证明三类, 并进行深入分析、对比和总结。

### 6.1 Rust 形式语义研究

形式语义学利用数学工具, 精确地定义编程语言的语义, 为语言表达能力、完备性和可靠性的研究提供数学支撑。Rust 形式语义研究是 Rust 程序验证的重要基础, 目前已有许多针对 Rust 形式语义的相关工作。

Patina<sup>[64]</sup>是第一个针对 Rust 的形式语义研究, 该研究提出了针对一个 Rust 子集的形式化语义, 该子集主要刻画了 Rust 的引用和借用特性, 用于描述 Rust 的内存管理。该研究还形式化了 Rust 早期借用检查器(Borrow Checker)的操作语义, 并基于该操作语义, 给出了 Rust 子集的可靠性证明。本文认为, Patina 的主要不足是其借用检查不包括生命周期推理, 而且没有形式化数组的借用检查, 因此难以扩展到大型程序上。

为了进一步扩展 Patina<sup>[64]</sup>, LAMQADEM 等<sup>[65]</sup>对一个更大的 Rust 子集定义了形式语义, 该语义不仅涵盖了生命周期推理, 还显式定义了借用检查器的特殊语义, 并证明了 Rust 内存管理代码的安全性。但是, 该工作没有考虑 Rust 的 unsafe 特性, 因此难以证明大量使用了 unsafe 特性的 Rust 库函数。

上述研究都只关注了 Rust 的内存相关特性, 为了提出更加通用的 Rust 形式化语义, Wang 等<sup>[66]</sup>基于描述编程语言语义的形式化框架 K<sup>[67]</sup>, 提出了第一个针对 Rust 的形式化可执行语义 KRust。KRust 涵盖了 Rust 中的基本类型、基本运算、复合数据类型、基本的控制流、函数、模式匹配等较为全面的语法特征。KRust 实现的语义规则包含 Rust 最重要的三个特性, 即所有权、借用和生命周期。该研究使用 Rust 官方编译器提供的测试集和针对语义规则写的 Rust 代码, 进行一致性对比测试, 成功发现并定位了 Rust 编译器中的一个安全漏洞。但是, 该研究的主要不足是其实现的语义规则仍然不完备, 不包含引用、unsafe 机制等重要特性, 因此无法应用于实际的大型 Rust 程序。

上述工作都不提供对 unsafe 机制的支持。Rustbelt 项目<sup>[68]</sup>对 Rust 的一个核心子集  $\lambda$ Rust, 给出了一个形式化的且机器可自动检查的证明。和其他类似工作相比, Rustbelt 的主要技术优势在于它是可扩展的, 即对于使用了 unsafe 机制的 Rust 库, 该框架仍然能够证明 Rust 库满足安全验证条件。本文认为, 该工作的不足之处主要有三点: 一是 Rust 子集仅包括 Rust 的所有权、生命周期等语法, 未包括 trait 等

高级特性;二是在通过机器安全证明  $\lambda$ Rust 之前,需要手工将 Rust 源代码转写成  $\lambda$ Rust,这将引入大量费时易错的人工操作;三是 Rustbelt 使用的操作语义与部分编译器优化不兼容。

后续, Dang 等<sup>[69]</sup>对 RustBelt<sup>[68]</sup>项目进行了扩展,加入了对 Rust 库中广泛使用的弱内存模型的支持。该研究针对弱内存模型下的内存回收问题,提出了一个称为同步影子状态(Synchronized Ghost State)的结构,并在 Rust 的 Arc 库中发现了一个新的数据竞争安全漏洞。但是,该工作的主要局限性是仍然需要对 Rust 程序源代码进行手工转写。

RustBelt<sup>[68]</sup>同时提供了生命周期和所有权的形式语义,不同于 RustBelt, Weiss 等<sup>[70]</sup>提出了一个抽象的 Rust 形式语义,该形式语义可以在没有生命周期细节的情况下,获取变量的所有权和借用信息;然后通过对所有权的语义进行建模,获取接近源代码级别的 Rust 语义;进一步,该研究通过将标准库中的函数接口建模为原语,支持对标准库中 unsafe 代码的验证。

为了更深入地研究 unsafe 机制的语义, Jung 等<sup>[71]</sup>提出了一种全新的描述 Rust 访存行为的形式化操作语义 Stacked Borrows,该语义形式化了 Rust 中的别名规则,并且定义了程序的未定义行为。该工作使用 Coq 定理证明器<sup>[72]</sup>,证明了使用过程内优化的一大类程序的正确性。同时,该工作基于其提出的操作语义,实现了一个解释器原型系统,对实际的 Rust 代码和标准库进行了验证,结果表明该框架可以处理大部分实际的 unsafe 代码。但是,该研究的主要不足是其提出的操作语义不能进行过程间分析和优化,因此难以用于实际的 Rust 编译器代码。

本文认为,对 Rust 形式语义的已有研究,尽管使用了不同的方法和工具,但都采用了类似的技术路线,即首先对 Rust 的一个子集定义形式语义,再根据该语义对 Rust 程序进行显式建模,最后利用定理证明器或程序验证器完成对程序的验证。本文认为,这类验证方法还存在两点不足:一是已有工作的语义模型只考虑了 Rust 的部分子集,没有包含 Rust 的全部特性;二是由于缺乏通用的自动验证工具,现有研究都需要大量的人工程序转写。因此,研究和建立对 Rust 完整语言的形式化语义和模型,是一个有重要意义但极具挑战性的未来研究方向(见本文第 7 小节)。

## 6.2 Rust 自动化程序验证研究

自动化程序验证研究的主要目标是基于程序验证相关理论(如霍尔逻辑等),利用自动化验证工具,证明程序的部分或完全正确性,即程序在运行时满足特定规范。需要注意,自动化程序验证和第 6.1 小节讨论的 Rust 形式语义的一个主要不同在于,自动化程序验证的主要研究目标是实现验证过程的自动性,从而有效降低程序验证所需的代价并扩展到实际的大型程序,但是仍然可能需要手工标注程序属性。

Rust 自动化程序验证研究是 Rust 语言安全研究的重要方向之一,本文通过对已有的研究进行系统分析和总结,按照验证技术将现有研究分为模型检测、符号执行和定理证明三类。如表 5 所示,本文对 Rust 自动化程序验证方向上的代表性工作,从验证工具、中间语言、验证技术、数据结构、验证性质、支持 unsafe? 和用户标注等几个方面进行了详细对比分析。

表 5 Rust 自动化程序验证研究工作总结对比

Table 5 Summary and Comparison of Rust Automation Program Verification Research Work

研究工作	验证工具	中间语言	验证技术	数据结构	验证性质	支持 unsafe?	用户标注
CRUST	CBMC	C	模型检测	AST	内存安全	✓	过滤器
RSMC	SMACK	Boogie	模型检测	LLVM IR	内存、并发安全	✓	-
Baranowski	SMACK	Boogie	模型检测	LLVM IR	功能正确性	✓	规范
RustHorn	SeaHorn	CHC	模型检测、抽象解释	MIR	功能正确性	×	-
Lindner	KLEE	LLVM IR	符号执行	LLVM IR	内存安全	✓	-
Rust2Viper	Viper	Silver	符号执行	HIR	功能正确性	✓	规范
Prusti	Viper	Viper	符号执行	MIR	功能正确性	×	规范
Merigoux	F*	F*、low*	定理证明	AST	内存、功能正确性	✓	规范
Ullrich	Lean	Lean	定理证明	MIR	功能正确性	×	规范
CREUSOT	WHY3	WHY3	定理证明	MIR	功能正确性	×	规范

### 6.2.1 模型检测

模型检测的基本思想是通过自动遍历软件系统模型的有穷状态空间, 来验证系统的语义模型和其性质规范之间的满足关系。模型检测是 Rust 自动化程序验证最常用的验证技术。许多现有验证工具都提供了基于模型检测的验证后端, 如 CBMC<sup>[73]</sup>、SMACK<sup>[74]</sup>、SeaHorn<sup>[75]</sup>等。

CRUST<sup>[76]</sup>是第一个基于模型检测的 Rust 自动化程序验证研究, 该研究采用穷举测试例生成和有界模型检测技术, 来验证 unsafe 库代码的内存安全, 以及判断 Rust 指针别名使用是否违反了不变量。图 5 给出了 CRUST 的架构。CRUST 接受过滤器和库代码作为输入; 将 Rust 库代码和驱动程序翻译为 C 代码; 最后使用针对 C 语言的有界模型检测器 CBMC<sup>[73]</sup>, 来验证每个被检测函数的内存安全属性。该研究通过向 Rust 标准库的三个模块中手动引入两个内存错误, 来测试 CRUST 的有效性, 实验结果表明, CRUST 能够发现引入的两个内存错误。本文认为, CRUST 的优点是实现了程序验证的自动化, 不需要手工标注需要验证的程序属性; 主要局限性是多个驱动会触发同一个错误, 导致报告的错误远多于实际的错误, 实验中, CRUST 共报告了 73 个错误, 但都是由引入的两个内存错误导致。

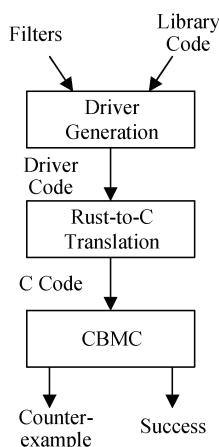


图 5 CRUST 架构

Figure 5 CRUST Architecture

CRUST<sup>[76]</sup>只能验证内存安全属性, 虽然 Rust 相比于其他语言降低了并发编程的风险, 但它并不能完全保证不发生并发错误。因此, YAN 等<sup>[77]</sup>提出了一个基于 SMACK<sup>[74]</sup>验证工具的 Rust 程序验证框架 RSMC, 来验证 Rust 程序中的并发安全和内存安全。SMACK 是一个针对 C 语言的验证框架, 接受 LLVM IR 作为输入, 使用模型检测技术进行程序验证。RSMC 结合了并发原语模型检测和内存边界模型检

测技术, 通过断言生成器自动生成安全属性断言, 并插入到 Rust 程序中, 不需要用户输入程序标注。该研究的主要局限性是只关注内存访问并发问题, 并未考虑网络访问并发和文件访问并发。

Baranowski 等<sup>[78]</sup>也提出了一个基于 SMACK 的自动化程序验证方法, 但该方法采用了不同于上述研究的技术路线。该方法通过扩展 SMACK 和对 Rust 库进行建模, 来对 Rust 程序进行验证, 并在实际的 Rust 项目上发现了 3 个安全漏洞。本文认为, 该研究存在的不足主要有三点: 一是只对少量标准库代码进行了建模, 无法检查 Rust 的裸指针; 二是转换之前需要手动改写 Rust 程序, 没有实现完全自动化; 三是无法验证并发安全性质。

基于受限霍恩子句(Constrained Horn Clauses, CHC)的可满足性问题判定, 是自动化程序验证的常用方法。SeaHorn<sup>[75]</sup>是一个程序验证框架, 它可以接受 CHC 作为输入, 然后利用后端的模型检测验证器进行验证。Matsushita 等<sup>[79]</sup>提出了一种基于受限霍恩子句的程序验证方法, 该方法利用 Rust 的所有权机制来处理指针操作, 从而将包含指针操作的 Rust 子集翻译为受限霍恩子句表示, 然后使用验证框架 SeaHorn 进行自动验证。该工作形式化描述了这个 Rust 子集和翻译过程, 证明了翻译的正确性; 实现了一个用于 Rust 程序验证的原型系统 RustHorn; 并在一组小型程序测试集上验证了方法的有效性。但是, 该研究不能用于复杂程序的验证, 缺乏通用性和可扩展性。

### 6.2.2 符号执行

符号执行通过符号分析, 系统性地遍历程序路径空间, 符号执行中的程序路径精确刻画了路径上的程序信息, 可以基于路径上的程序信息进行程序验证。

Lindner 等<sup>[80]</sup>针对 Rust 的内存安全和程序错误问题, 提出了第一个基于符号执行的程序验证框架, 该框架还可以通过移除程序中冗余的动态检查, 来提高程序的执行性能。该研究基于 KLEE 符号执行引擎<sup>[81]</sup>实现了一个原型系统, 并在 Rust 程序和库编译生成的 LLVM IR 上进行验证。随后, Lindner 等<sup>[82]</sup>又扩展了上述 LLVM-KLEE 符号执行验证框架, 使其可以支持基于断言来验证 Rust 安全函数(即不包括 unsafe 块的函数), 并讨论了符号执行中的路径状态爆炸问题。但是, 该方法的主要局限是在于: 使用的符号执行技术具有较大的内存开销, 实验数据表明, 验证一个简单函数的内存开销可以超过 130M。

Viper<sup>[83]</sup>是一个通用验证框架, 它提供了一种中

间语言 Silver, 以及基于符号执行的后端验证器。Hahn 等<sup>[84]</sup>提出了第一个基于 Viper 验证框架的 Rust 程序验证方法 Rust2Viper。Rust2Viper 将 Rust 编译器提供的 HIR 中间表示翻译为 Sliver 代码, 然后使用 Viper 的符号执行后端进行程序验证。Rust2Viper 支持的 Rust 子集包括递归函数、典型数据结构、简单循环以及一部分 unsafe 程序。但是, Rust2Viper 无法验证 trait 和闭包等特性。

Astrauskas<sup>[85]</sup>也基于 Viper 验证框架, 提出了一个程序验证技术, 该技术首先收集 MIR 提供的类型信息, 利用分离逻辑<sup>[86]</sup>自动合成证明程序, 然后利用 Viper 的符号执行验证器进行验证。该研究还基于此技术, 实现了一个程序验证原型系统 Prusti, 并验证了典型 Rust 程序的功能正确性。随后, Schilling 等<sup>[87]</sup>对 Prusti 进行了扩展, 使其支持对 Rust 内置的序列类型和 trait 机制的验证。本文认为, 上述两个研究仍然存在一些局限性: 一是不支持闭包、迭代器、生命周期等高级特性, 也不支持 unsafe 特性, 缺乏通用性; 二是验证过程依赖 Java 虚拟机, 时间开销较大。

### 6.2.3 定理证明

基于定理证明进行形式化验证的基本思想是, 将“程序满足其规范”这一论断作为逻辑命题, 通过一组推理规则, 将逻辑命题的证明问题转化为一组数学命题的证明。现有基于定理证明的 Rust 自动化程序验证研究都是先将 Rust 翻译为函数式编程语言, 然后利用验证工具进行验证。目前已经有许多基于定理证明的程序验证工具, 如 F\*<sup>[88]</sup>、Lean<sup>[89]</sup>、WHY3<sup>[90]</sup>等。

由于 Rust 包括大量函数式特性, 直接基于这些特性进行验证是一个值得探索的方向。Merigoux 等<sup>[91]</sup>提出了一个新的程序验证框架 hacspect, hacspect 是一个纯函数式 Rust 子集。该框架首先使用 hacspect 在 Rust 程序中书写可执行的验证规范; 然后将规范翻译为 F\*<sup>[88]</sup>, 再结合使用 low\*<sup>[92]</sup>实现的安全组件进行验证; 验证成功后生成 C 程序或汇编程序作为目标代码。但是, hacspect 的纯函数特征导致其无法验证可变状态和借用, 也不支持结构体、枚举类型等代数数据结构。

Merigoux 等<sup>[91]</sup>的研究只针对 Rust 的函数式子集, 而将 Rust 中的可变引用、指针别名等特性翻译到函数式编程语言是一个挑战。Ullrich 等<sup>[93]</sup>提出了第一个通用的基于定理证明的 Rust 程序验证框架, 该框架先将 Rust 自动翻译为定理证明器语言 Lean<sup>[89]</sup>, 再进行程序验证。该框架支持可变引用的翻译, 并且成功证明了二分查找算法的正确性和渐进运行时间复

杂度, 以及 FixedBitSet 数据结构的正确性。但是, 本文认为, 该工作还存在一些局限性: 首先, 它不支持多态、生命周期、unsafe 等特性的验证; 其次, 在翻译为 Lean 之后, 还需要用户手动加上待验证的程序属性。

为了提出一个更加通用的验证技术, Denis 等<sup>[94]</sup>基于 RustHorn<sup>[79]</sup>的研究, 形式化定义了 Rust 的一个包括借用和生命周期的子集 MiniMir, 然后将 MiniMir 自动翻译为函数式编程语言 anyML, 并且证明了这种翻译的正确性。接着, 他们又进一步扩展了该研究工作<sup>[95]</sup>, 扩大了可验证的 Rust 子集, 支持对多态和 trait 机制的验证, 并实现了一个 Rust 程序验证原型系统 CREUSOT, CREUSOT 通过将 Rust 程序翻译为中间表示 WHY3<sup>[90]</sup>来进行验证。但是, 这两个研究工作的局限都是不支持对 unsafe 机制的验证。

Rust 自动化程序验证相关研究都采用了类似的技术路线, 即都基于 Rust 编译器提供的中间表示, 将 Rust 程序翻译为现有验证工具支持的中间表示, 再使用现有验证工具完成验证。本文认为, 在这个研究方向上存在较大的技术挑战: 首先, Rust 作为多编程范式的语言, 同时支持函数式、命令式和面向对象特性, 将 Rust 的所有特性都编译到一种中间表示上的技术挑战较大; 其次, 已有研究工作使用多样的验证工具和中间语言, 而随着 Rust 的演化, 不断更新和维护多种验证工具的成本较高。上述这类工具在发布后, 大多没有得到进一步的扩展和维护。

本文认为, 在 Rust 自动化程序验证研究领域, 未来有两个可能的研究方向: 一是研究面向 Rust 的通用验证技术和工具, 降低同时更新和维护多个验证工具的成本; 二是在现有验证技术的基础上, 通过多技术的深度融合提出新的 Rust 自动化程序验证框架, 例如将插值技术和 SMT 结合、将模型检测和抽象解释结合等, 在现有验证技术的基础上提出可配置的验证框架, 在框架中集成多种验证技术和求解器, 来对 Rust 软件系统进行验证, 从而提高验证框架的性能和可扩展性。

## 6.3 本节小结

形式化验证是严格证明 Rust 的安全性, 或证明基于 Rust 构建的软件系统满足特定安全规范的重要路径。本文认为, 这个方向上的已有研究工作可以总结为两类: 一是对 Rust 形式语义的研究, 这一类工作已经在 Rust 的实际子集上取得了较好成果; 二是构建 Rust 自动化程序验证框架, 目前这个方向上的工作取得了初步成果, 但受限于目前定理证明器或约束求解器的实际证明能力<sup>[96]</sup>, 这个方向上的研究



还存在很多技术挑战, 是未来的重要工作方向。

## 7 未来研究方向展望

随着 Rust 的持续演化, 以及 Rust 应用领域的持续扩大, Rust 语言安全研究仍将是未来的重要研究方向。通过系统梳理、分析并总结已有研究工作, 本文认为, 该研究领域还有 4 个重要的研究问题: 一是缺少对 Rust 中常见漏洞的自动修复研究; 二是缺少通用的 Rust 中间表示和程序分析框架; 三是缺少 Rust 全语言的形式化验证模型; 四是缺少从 C 到 Rust 的自动转换研究。针对这 4 个研究问题, 本文提出了 4 个未来的研究方向。

(1) Rust 常见漏洞的自动修复理论与技术研究。研究表明<sup>[97]</sup>, 在现代软件开发中, 漏洞修复成本占开发过程所有成本的 50%~70%。因此, 自动化漏洞修复研究, 对于保证软件质量有重要意义。虽然目前已有大量针对 Java<sup>[98]</sup>或 C<sup>[99]</sup>的程序漏洞自动修复的理论和技术的研究, 但是这些研究提出的修复理论和技术无法直接应用于 Rust 程序: 一是因为 Rust 引入的所有权和显式生命周期等新语言特性, 给程序漏洞自动修复提出了挑战; 二是因为 Rust 程序漏洞的自动修复, 会涉及 unsafe 代码和安全代码的交互。因此, 针对 Rust 程序漏洞的自动修复研究, 是一个全新且具有挑战性的研究方向。目前, Rupair<sup>[43]</sup>已经研究了针对 Rust 中 IO2BO 漏洞的自动修复理论和技术。未来, 研究针对更多 Rust 安全漏洞的自动修复理论和技术, 有助于提高漏洞修复的效率并有效降低漏洞修复成本。

(2) 通用 Rust 中间表示和程序分析基础框架的研究与构建。在 Rust 语言安全研究中, 通用程序中间表示和程序分析基础框架起到关键作用, 它们不但为漏洞检测相关工作提供基础, 为程序的静态和动态分析的构建提供核心能力, 还为研究成果的持续积累和演进提供必要前提。现有研究工作都是直接使用 LLVM IR、AST 或 MIR 作为程序分析的基础架构, 尚未建立统一的中间表示和程序分析基础设施。最近, Rudra<sup>[46]</sup>提出了一种融合 AST 和 MIR 的中间表示基础设施, 实验结果表明其能够扩展到生态系统级的程序表示和分析, 这给程序分析基础框架的研究提供了新的可行思路。但是, 研究和建立面向 Rust 的通用程序中间表示和程序分析基础设施, 仍是一个非常具有挑战性但亟待解决的研究问题。

(3) Rust 全语言形式化验证模型的研究和构建。研究和构建 Rust 全语言的形式化模型, 将有助于对整个 Rust 语言进行形式化建模, 并对语言的所有安

全特性进行形式化验证。这对于从根本上保证语言的安全性, 并持续推动 Rust 在安全领域内的应用, 有重要意义。目前, 已有研究工作都只完成了对某个 Rust 子集的形式化验证, 尚未完成对整个语言的形式化建模。现有研究<sup>[100]</sup>表明, 对完整的函数式语言进行形式化建模是可行的。本文认为, 这个研究方向的主要研究机会在于两个方面: 首先, 作为一门相对年轻的语言, Rust 提供了诸如所有权模型、生命周期等许多先进语言机制, 并仍在持续演化中, 研究这些新语言机制的语义理论和形式化模型是一个重要方向; 其次, Rust 是一个集成了命令式、函数式、面向对象和泛型编程等多种编程范式的程序设计语言, 在一个统一框架内, 对这些编程范式进行统一形式化建模是一个重要研究方向。

(4) 从 C 到 Rust 的自动转换原理及实现技术研究。Rust 是为了解决 C/C++ 中的安全问题, 而设计的新一代安全的系统级程序设计语言。通过将现有的大量基于 C 语言开发的软件系统移植到 Rust 上, 可以解决其最突出的内存安全问题, 从而提高软件系统的可靠性和安全性。但是, 手动将这些软件系统移植到 Rust 上是一项非常艰巨的任务。因此, 研究从 C 到 Rust 的自动转换原理及实现技术对于保证软件系统安全有重要意义。但是, 现有转换工具<sup>[101]</sup>只能将 C 转换为不安全的 Rust, 没有达到 Rust 保证内存安全的设计目标, 无法直接用于软件系统的构建。因此, 现有工具只是为自动转换提供了一个起点, 仍需要开发人员手动推理程序的安全属性和重写代码。对于将 C 自动转换为安全的 Rust, 目前还没有全面的研究和可用的工具。因此, 从 C 到安全 Rust 的自动转换原理及实现技术研究是一项亟待展开的重要课题。

## 8 结语

系统级程序设计要求实现语言兼具高性能、编程灵活性和安全性, Rust 是面向这一应用场景, 而提出的一种新型系统级安全编程语言。它通过提供一系列全新的语言机制和特性, 实现了类型安全、内存安全和并发安全的设计目标, 正在成为下一代系统级程序设计的首选语言。本文通过分析 Rust 的核心安全特性, 并系统梳理、分析和总结关于 Rust 语言安全的已有研究, 提出了对已有研究的分类方法: 安全实证研究、漏洞检测研究、安全增强研究和形式化验证研究, 并对这四个研究方向进行了综述、深入分析、对比和总结。最后, 本文对 Rust 语言安全研究的潜在方向和发展趋势进行了展望, 提出了自动化漏洞修复理论和技术研究、统一中间表示和程

序分析基础框架的研究与构建、Rust 全语言的形式化验证模型的研究与构建、从 C 到 Rust 的自动转换原理及实现技术研究这四个潜在的研究方向和机会, 以期对相关领域的研究者提供有价值的参考。

## 参考文献

- [1] Yaswinski M R, Chowdhury M M, Jochen M. Linux Security: A Survey[C]. *2019 IEEE International Conference on Electro Information Technology*, 2019: 357-362.
- [2] Zhang W X, Dong R, Wang S H, et al. Research on Defect Classification and Analysis of Spacecraft Ground Software[C]. *The International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery*, 2022: 1008-1017.
- [3] Ghafoor I, Jattala I, Durrani S, et al. Analysis of OpenSSL Heartbleed Vulnerability for Embedded Systems[C]. *17th IEEE International Multi Topic Conference*, 2015: 314-319.
- [4] The Rust Programming Language. <https://doc.rust-lang.org/stable/book/>. 2018.
- [5] Tock. Tock Embedded Operating System. <https://www.tockos.org/>. 2019.
- [6] The Servo Browser Engine. <https://servo.org/>. 2019.
- [7] Next-generation file system:TFS. <https://github.com/redoxos/tfs>. 2020.
- [8] TTstack. <https://github.com/rustcc/TTstack>. 2020.
- [9] A standalone, event-driven TCP/IP stack:smoltcp. <https://github.com/smoltcp-rs/smoltcp>. 2022.
- [10] Tokio is an asynchronous runtime for the Rust programming language. <https://tokio-cn.github.io/>. 2022.
- [11] TiKV is an open-source, distributed, and transactional key-value database. <https://github.com/tikv/tikv>. 2022.
- [12] Parity is the fastest and most advanced ethereum client. <https://github.com/paritytech/parity-ethereum>. 2020.
- [13] Qin B Q, Chen Y L, Yu Z M, et al. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs[C]. *The 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020: 763-779.
- [14] Xu H, Chen Z B, Sun M S, et al. Memory-Safety Challenge Considered Solved? an In-Depth Study with all Rust CVEs[J]. *ACM Transactions on Software Engineering and Methodology*, 2022, 31(1): 1-25.
- [15] Yu Z M, Song L H, Zhang Y Y. Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software[EB/OL]. 2019: arXiv: 1902.01906. <https://arxiv.org/abs/1902.01906>.
- [16] The history of Rust.[https://en.wikipedia.org/wiki/Rust\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language)).
- [17] Technology-most-loved-and-wanted-languages. <https://insights.stackoverflow.com/survey/2021>. 2021.
- [18] The State of the Octoverse. <https://octoverse.github.com/>.
- [19] Rust/WinRT Public Preview. <https://blogs.windows.com/windowsdeveloper/2020/04/30/rust-wintr-public-preview/>. 2020.
- [20] Rust in the Android platform. <https://security.googleblog.com/2021/04/rust-in-android-platform.html>. 2021.
- [21] StratoVirt is an enterprise-level virtualization platform for cloud data centers. <https://github.com/openeuler-mirror/stratovirt>. 2022.
- [22] Linux kernel modules in safe Rust. <https://github.com/fishinabarrel/linux-kernel-module-rust>. 2021.
- [23] The Standard ML language. <https://smlfamily.github.io/>.
- [24] The Lisp language. <https://lisp-lang.org/>.
- [25] The OCaml programming language. <https://ocaml.org/>.
- [26] The Haskell programming language. <https://www.haskell.org/>.
- [27] Category: Functional languages. [https://en.wikipedia.org/wiki/Category:Functional\\_languages](https://en.wikipedia.org/wiki/Category:Functional_languages).
- [28] Traits: Defining Shared Behavior. <https://doc.rust-lang.org/book/ch10-02-traits.html>.
- [29] Generics in Rust. <https://doc.rust-lang.org/book/ch10-00-generic-s.html>.
- [30] The Rust Compilation Model Calamity. <https://en.pingcap.com/blog/rust-compilation-model-calamity>.
- [31] Wu M, Zhao Z, Yang Y, et al. Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services[C]. *2020 USENIX Annual Technical Conference*, 2020: 159-172.
- [32] Patrick Gaydon. Project Servo, Technology from the past come to save the future from itself. <http://venge.net/graydon/talks/intro-talk-2.pdf>. 2010.
- [33] CVE. Common Vulnerabilities and Exposures. <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust>.
- [34] RustSec. Security advisory database for Rust crates. <https://rust-sec.org>.
- [35] Fearless Concurrency. <https://doc.rust-lang.org/book/ch16-00-concurrency.html>.
- [36] Evans A N, Campbell B, Soffa M L. Is Rust Used Safely by Software Developers? [C]. *The ACM/IEEE 42nd International Conference on Software Engineering*, 2020: 246-257.
- [37] Astrauskas V, Matheja C, Poli F, et al. How do programmers use unsafe rust?[J]. *Proceedings of the ACM on Programming Languages*, 2020, 4(OOPSLA): 1-27.
- [38] Zhu S F, Zhang Z Y, Qin B Q, et al. Learning and Programming Challenges of Rust: A Mixed-Methods Study[C]. *The 44th International Conference on Software Engineering*, 2022: 1269-1281.
- [39] Balakrishnan A K, Nattanmai Ganesh G. Modern C++ and Rust in embedded memory-constrained systems[D]. Chalmers University of Technology, 2022.
- [40] Emre M, Schroeder R, Dewey K, et al. Translating C to Safer Rust-Extended Version[C]. *Object-Oriented Programming, Systems, Languages, and Applications*, 2021:1-29.
- [41] Huang Z J, Wang Y J, Liu J. Detecting Unsafe Raw Pointer Dereferencing Behavior in Rust[J]. *IEICE Transactions on Information and Systems*, 2018, E101.D(8): 2150-2153.
- [42] Cui M H, Chen C J, Xu H, et al. SafeDrop: Detecting Memory De-allocation Bugs of Rust Programs via Static Data-Flow Analysis[EB/OL]. 2021: arXiv: 2103.15420. <https://arxiv.org/abs/2103.15420>.
- [43] Hua B J, Ouyang W R, Jiang C M, et al. Rupa: Towards Automatic Buffer Overflow Detection and Rectification for Rust[C]. *ACSAC '21: Annual Computer Security Applications Conference*,

- 2021: 812-823.
- [44] Switzer J F. Preventing IPC-facilitated type confusion in Rust[D]. Massachusetts Institute of Technology, 2020.
- [45] Li Z H, Wang J C, Sun M S, et al. MirChecker: Detecting Bugs in Rust Programs via Static Analysis[C]. *The 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021: 2183-2196.
- [46] Bae Y C, Kim Y, Askar A, et al. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale[C]. *The ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021: 84-99.
- [47] Ning P X, Qin B Q. Stuck-me-Not: A Deadlock Detector on Blockchain Software in Rust[J]. *Procedia Computer Science*, 2020, 177: 599-604.
- [48] RUSTY: A Fuzzing Tool for Rust. <https://www.acsac.org/2020/program/final/p27.html>. 2020.
- [49] Jiang J F, Xu H, Zhou Y F. RULF: Rust Library Fuzzing via API Dependency Graph Traversal[C]. *2021 36th IEEE/ACM International Conference on Automated Software Engineering*, 2022: 581-592.
- [50] libFuzzer—a library for coverage-guided fuzz testing. <https://lvm.org/docs/LibFuzzer.html>.
- [51] Dewey K, Roesch J, Hardekopf B. Fuzzing the Rust Typechecker Using CLP (T)[C]. *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, 2016: 482-493.
- [52] Jaffar J, -L Lassez J. Constraint Logic Programming[C]. *The 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987: 111-119.
- [53] Gulwani S, Polozov O, Singh R. Program Synthesis[J]. *Foundations and Trends® in Programming Languages*, 2017, 4(1/2): 1-119.
- [54] Almhori H M J, Evans D. Fidelius Charm: Isolating Unsafe Rust Code[C]. *The Eighth ACM Conference on Data and Application Security and Privacy*, 2018: 248-255.
- [55] Liu P M, Zhao G, Huang J. Securing UnSafe Rust Programs with XRust[C]. *2020 IEEE/ACM 42nd International Conference on Software Engineering*, 2020: 234-245.
- [56] Rivera E E. Preserving Memory Safety in Safe Rust during Interactions with Unsafe Languages[D]. Massachusetts Institute of Technology, 2021.
- [57] Park S, Lee S, Xu W, et al. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)[C]. *The 2019 USENIX Conference on Usenix Annual Technical Conference*, 2019: 241-254.
- [58] Lamowski B, Weinhold C, Lackorzynski A, et al. Sanderust: Automatic Sandboxing of Unsafe Components in Rust[C]. *The 9th Workshop on Programming Languages and Operating Systems*, 2017: 51-57.
- [59] Ouyang W R, Hua B J. RusBox: Towards Efficient and Adaptive Sandboxing for Rust[C]. *2021 IEEE International Symposium on Software Reliability Engineering Workshops*, 2022: 1-2.
- [60] Dominik D. Visualization of Lifetime Constraints in Rust[D]. ETH Zurich. 2018.
- [61] Blaser D. Simple Explanation of Complex Lifetime Errors in Rust[D]. ETH Zurich. 2019.
- [62] Zhang Z Y, Qin B Q, Chen Y L, et al. VRLifeTime — an IDE Tool to Avoid Concurrency and Memory Bugs in Rust[C]. *The 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020: 2085-2087.
- [63] Lindgren P, Fitinghoff N, Aparicio J. Cargo-Call-Stack Static Call-Stack Analysis for Rust[C]. *2019 IEEE 17th International Conference on Industrial Informatics*, 2020: 1169-1176.
- [64] Reed E. Patina: A formalization of the Rust programming language[D]. University of Washington, 2015.
- [65] Amin Ait Lamqadem. A Formalization of the Static Semantics of Rust[D]. Corso di Laurea Magistrale in Informatica, 2019.
- [66] Wang F, Zhang J. KRust: Formal Executable Semantics of Rust[J]. *Journal of Frontiers of Computer Science and Technology*, 2019, 13(12): 2008-2014.  
(王丰, 张俊. KRust: Rust 形式化可执行语义[J]. *计算机科学与探索*, 2019, 13(12): 2008-2014.)
- [67] Roşu G, Şerbănuţă T F. An Overview of the K Semantic Framework[J]. *The Journal of Logic and Algebraic Programming*, 2010, 79(6): 397-434.
- [68] Jung R, Jourdan J H, Krebbers R, et al. RustBelt: Securing the foundations of the Rust programming language[J]. *Proceedings of the ACM on Programming Languages*, 2017, 2(POPL): 1-34.
- [69] Dang H H, Jourdan J H, Kaiser J O, et al. RustBelt meets relaxed memory[J]. *Proceedings of the ACM on Programming Languages*, 2019, 4(POPL): 1-29.
- [70] Weiss A, Patterson D, Ahmed A. Rust Distilled: An Expressive Tower of Languages[EB/OL]. 2018: arXiv: 1806.02693. <https://arxiv.org/abs/1806.02693>.
- [71] Jung R, Dang H H, Kang J, et al. Stacked borrows: an aliasing model for Rust[J]. *Proceedings of the ACM on Programming Languages*, 2019, 4(POPL): 1-32.
- [72] The Coq proof assistant. <https://coq.inria.fr/>.
- [73] CBMC is a Bounded Model Checker for C and C++. <https://www.cprover.org/cbmc/>.
- [74] SMACK is both a modular software verification toolchain and a self-contained software verifier. <https://smackers.github.io/>.
- [75] Gurfinkel A, Kahsai T, Komuravelli A, et al. The SeaHorn Verification Framework[C]. *International Conference on Computer Aided Verification*, 2015: 343-361.
- [76] Toman J, Pernsteiner S, Torlak E. Crust: A Bounded Verifier for Rust (N)[C]. *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, 2016: 75-80.
- [77] Yan F, Wang Q Z, Zhang L Q, et al. RSMC: A Safety Model Checker for Concurrency and Memory Safety of Rust[J]. *Wuhan University Journal of Natural Sciences*, 2020, 25(2): 129-138.
- [78] Baranowski M, He S, Rakamarić Z. Verifying Rust Programs with SMACK[C]. *International Symposium on Automated Technology for Verification and Analysis*, 2018: 528-535.
- [79] Matsushita Y, Tsukada T, Kobayashi N. RustHorn: CHC-Based Verification for Rust Programs[C]. *European Symposium on Programming*, 2020: 484-514.
- [80] Lindner M, Aparicius J, Lindgren P. No Panic! Verification of Rust Programs by Symbolic Execution[C]. *2018 IEEE 16th International Conference on Industrial Informatics*, 2018: 108-114.
- [81] KLEE Symbolic Execution Engine. <https://klee.github.io/>.

- [82] Lindner M, Fitinghoff N, Eriksson J, et al. Verification of Safety Functions Implemented in Rust - a Symbolic Execution Based Approach[C]. *2019 IEEE 17th International Conference on Industrial Informatics*, 2020: 432-439.
- [83] Viper is a language and suite of tools developed at ETH Zurich. <https://www.pm.inf.ethz.ch/research/viper.html>.
- [84] Hahn F. Rust2Viper: Building a static verifier for Rust[D]. ETH Zürich, 2016.
- [85] Astrauskas V, Müller P, Poli F, et al. Leveraging Rust types for modular specification and verification[J]. *Proceedings of the ACM on Programming Languages*, 2019, 3(OOPSLA): 1-30.
- [86] Reynolds J C. Separation Logic: A Logic for Shared Mutable Data Structures[C]. *17th Annual IEEE Symposium on Logic in Computer Science*, 2002: 55-74.
- [87] Schilling J, Bílý A, Poli F, et al. Specifying and Verifying Sequences and Array Algorithms in a Rust Verifier[D]. FAU Erlangen-Nürnberg, 2021.
- [88] F\* is a general-purpose functional programming language. <https://www.fstar-lang.org/>.
- [89] Lean is a functional programming language and an interactive theorem prover. <https://leanprover.github.io/>.
- [90] Why3 is a platform for deductive program verification. <http://why3.lri.fr/>.
- [91] Merigoux D, Kiefer F, Bhargavan K. Hacspecc: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Technical report. Inria, 2021.
- [92] Low\* is a subset of F\*. <https://fstarlang.github.io/lowstar/html/>.
- [93] Ullrich S. Simple verification of rust programs via functional purification[D]. Karlsruher Institut für Technologie (KIT), 2016.
- [94] Denis X. Mastering Program Verification using Possession and Prophecies[C]. *32<sup>ème</sup> Journées Francophones des Langages Appliqués*. 2021: 174-189.
- [95] Denis X, Jourdan J H, Marché C. The Creusot Environment for the Deductive Verification of Rust Programs[D]. Inria Saclay Île de France, 2021.
- [96] Z3 is a theorem prover from Microsoft Research. <https://github.com/Z3Prover/z3>.
- [97] Zhang Y, Liu J K, Xia X, et al. Research Progress on Software Bug Localization Technology Based on Information Retrieval[J]. *Journal of Software*, 2020, 31(8): 2432-2452.  
(张芸, 刘佳琨, 夏鑫, 等. 基于信息检索的软件缺陷定位技术研究进展[J]. *软件学报*, 2020, 31(8): 2432-2452.)
- [98] Sinha S, Shah H, Görg C, et al. Fault Localization and Repair for Java Runtime Exceptions[C]. *The eighteenth international symposium on Software testing and analysis*, 2009: 153-164.
- [99] Zhang C, Wang T L, Wei T, et al. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time[C]. *European Symposium on Research in Computer Security*, 2010: 71-86.
- [100] Lee D K, Crary K, Harper R. Towards a Mechanized Metatheory of Standard ML[C]. *The 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007: 173-184.
- [101] C2Rust. C2Rust can migrate C99-compliant code to Rust. <https://c2rust.com/>.



胡霜 于 2019 年在中国科学技术大学信息安全专业获得学士学位。现在中国科学技术大学软件工程专业攻读硕士学位。研究领域为程序语言、计算机安全。CCF 学生会员。Email: guangan@mail.ustc.edu.cn



华保健 于 2008 年在中国科学技术大学计算机专业与理论专业获得博士学位。现任中国科学技术大学软件学院助理教授、硕士研究生导师。研究领域为程序语言与编译器、计算机安全。研究兴趣包括: 基于语言理论的安全技术。Email: bjhua@ustc.edu.cn



欧阳婉容 于 2020 年在武汉理工大学软件工程专业获得学士学位。现在中国科学技术大学电子信息专业攻读硕士学位。研究领域为网络与信息安全。Email: oywr@mail.ustc.edu.cn



樊淇梁 于 2016 年在中国科学技术大学软件工程专业获得硕士学位。现任中国科学技术大学软件学院高级工程师。研究领域为 Android 应用安全、智能合约安全、Rust 语言安全。研究兴趣包括: 编译器、语言安全等。Email: sa613162@mail.ustc.edu.cn