开题报告

计算机学院 2023 级数硕士 9 班 3220231370 傅泽

1. 选题依据

1.1. 选题背景及意义

随着 rust 这一新兴、内存安全且高效的系统级编程语言的兴起，越来越多开发者愿意使用 rust 开发他们自己的操作系统及配套的软件生态环境。鉴于 Linux 操作系统的生态环境成熟性，将 Linux 应用程序移植到生态环境尚不成熟的、基于 rust 的操作系统中就成为了快速拓展新兴操作系统软件生态环境的方案之一。然而，由于

1.1.1. 死代码

死代码是一种随软件迭代开发过程不断累积的代码异味。不同学派对死代码的定义有所不同，但其本质均可归结为：死代码是不会出现在软件执行流中的源代码。虽然软件的功能完整性与死代码的存在无关，但死代码仍然会对软件的开发、维护工作造成负面影响。例如，死代码模糊了软件业务逻辑，使得开发维护人员难以理清软件运行的真实逻辑；开发人员可能还会尝试维护死代码，造成生产力的无谓浪费；部分死代码还会拖慢程序编译、运行效率，降低了软件代码质量、开发效率和用户的使用体验。由此观之，在软件开发维护过程中，死代码移除的工作可谓意义重大。

目前，已有不少工作针对部分主流编程语言提出了死代码移除的技术方案。然而，作为新兴系统级编程语言的 rust 编程语言却缺少类似的技术方案。rust 语言以其内存安全、高效率的特性深受软件开发者尤其是硬件底层开发者的喜爱，被广泛应用于操作系统、嵌入式设备等领域的开发工作。这种工作代码量巨大，需要进行反复的迭代和尝试，从而更容易累积死代码。然而，无论是从代码质量和可维护性的角度出发，还是从嵌入式设备上装载的、有限的存储空间的角度出发，死代码移除之于 rust 软件项目的重要性只增不减。

若能摒弃以依赖项为最小单元，转而采用以函数调用为最小单元的分析方法，则可以有效避免上述不足。由于采用函数调用为最小分析单元，原本的依赖关系可以拆分为依赖项和软件之间的函数间依赖关系。通过分析这种关系，可以识别出依赖项中未被使用的函数，进而给出裁剪建议。软件开发者可以凭借这些建议移除冗余代码。一言以蔽之，以函数调用为最小单元的分析方法在大大减少移植工作的难度、减少需要实现的系统调用数量的同时，对于帮助软件开发者理清程序逻辑，精简代码与编译所得文件之体积等方面亦有积极作用。

本课题提出一套针对 rust 项目的最佳实践工作流，其能够以函数调用作为最小单元对指定的 rust 项目（即 rust crate）进行分析，识别其中及其依赖项中的冗余代码，为开发者给出代码裁剪建议。为证明该工作流确实能有效辅助进行跨架构软件移植，本课题将以 rust 实现的 Substrate 区块链为例，使用该工作流进行分析，依据其给出的裁剪建议进行裁剪，统计裁剪前后代码的体积变化，并与未裁剪的原始代码进行比较，证明该工具的有效性。随后，尝试将裁剪完成的 Substrate 移植入 rCore 教学操作系统中，在移植过程中实现必要的系统调用，为 rCore 实现内核层级的原生区块链功能支持。

1.2. 国内外研究现状

1.2.1. Substrate 区块链

作为以太坊项目的联合创始人，Gavin Wood 开发了以太坊网络。很快，其过分追求泛用性的设计理念暴露出一系列局限性，例如高昂的 gas fee 与核心可升级性的缺乏，进而迫使开发人员使用笨拙的变通方法尝试绕开这些局限性。

为了寻求突破口，Gavin 离开了以太坊基金会，创办了 Parity，结合团队开发比特币、ZCash 和以太坊客户端的行业经验，从头重新设计区块链架构，其结果即是基于 rust 开发

的 Substrate[1]。Substrate 是一个开源、面向未来的框架，为多链、可互操作和去中心化的互联网提供动力。它被设计成一个高度可定制的灵活模块化框架，能够以功能模块（称之为 pallet）为单位进行组件增删，令开发者得以将 Substrate 打造成最契合他们需求的区块链。这种设计模式已经在许多价值数十亿美元的实时网络上证明了它们的价值。

截至目前，Substrate 上仅能在 Linux x86 上运行，故本课题选择 Substrate 作为分析移植对象。一方面，由于 Substrate 基于 rust 写成，与目标操作系统 rCore 属于同种语言，降低了移植难度；另一方面，其项目代码库较大，较有代表性，可证明本课题提出的工作流的可行性。另外，区块链技术与操作系统内核的融合领域的研究工作较少，本课题亦可借此机会探索区块链技术与操作系统融合的形式，探究此举对操作系统技术发展的意义。

1.2.2. CG-RTL 函数调用图生成工具

在阅读、分析大型软件项目的源代码时，出于更好地了解源代码结构、分析其控制流等目的，往往需要绘制该软件项目的函数调用图。然而，已有的工具或是需要使用者有一定基础知识、或是会产生大量冗余信息、或是和编译环境高度耦合。为此，文献 [1] 提出一种基于寄存器传送语言（Register Transfer Language，RTL）中间表示的分析方法，从 GCC 编译器输出的 RTL 中间表示中，利用字符串处理提取当前软件包中的函数定义、函数调用信息，与其他软件包的上述信息进行整合，最终绘制成一幅巨大的函数调用图。在此基础上，研究团队又提出了能够处理动态函数调用的 DCG-RTL [2] 和基于数据库的函数调用图生成工具 [3] 。

此工作距今已久，且 rust 编译器尚无直出寄存器传送语言的功能支持，但其思路值得借鉴，即：通过分析编译器输出的中间表示进行静态分析，进而获得函数定义信息及函数调用关系信息。受此启发，本课题找到并探究了 rust 使用的几种中间表示形式，其各自优缺点将在下文详细说明。

1.2.3. rust 的中间表示

编译型程序设计语言的编译工具链在将源代码编译为目标平台的二进制可执行文件时，往往会选择先将源代码编译为某种形式的中间表示（IR），再将中间表示编译为目标平台的机器代码。在部分语言中，为方便从不同的角度进行全面完善的检查，一些语言还将使用数种不同层级的中间表示，待上层级中间表示通过检查后，再将其编译为下一层级的中间表示，运行下一步的检查，如此逐层降低层级，直至获得最后的机器码。这些中间表示从不同角度呈现了不同的诊断信息，有利于软件开发者开发外部工具对其进行诸如静态检查等操作。

rust 程序设计语言亦采用了这种多层级中间表示的思路。截至目前，rust 一共使用了如下四种中间表示：

1. 高层级中间表示（HIR）：HIR 是 rust 编译器 rustc 中最高层级的中间表示，由对源代码进行语法解析、宏展开等处理之后的抽象语法树转换而来。其形式和 rust 源代码尚有相似之处，但将一些语法糖展开为了更易于分析的形式，例如 `for` 循环将被展开为 `loop` 循环等。但由于此时 rustc 编译器尚未进行类型检查，因此 HIR 中的类型信息较为模糊，不适合作为静态分析工具的输入。
2. 带类型的高层级中间表示（THIR）：该中间表示是由 HIR 在完成类型检查后降低层级而来，主要用于枚举穷尽性检查、不安全行为检查和下一层中间表示的构造。和 HIR 相比，THIR 最大的不同在于诸如 `struct` 和 `trait` 等结构将不会在 THIR 中出现，因为 THIR 仅保留了源代码中可执行的部分，例如定义的普通函数以及 `impl` 块中定义的关联函数、方法等。由于具有上述“仅保留可执行部分”以及结构比 HIR 更简洁的特点，THIR 非常适合用于分析 rust crate 中的函数定义信息。
3. 中层级中间表示（MIR）：这种中间表示于 RFC 1211 中初次引入[2]，用于控制流相关的安全检查，例如借用检查器。它进一步将一些语法糖展开，引入了在 rust 源代码中不可能出现的语句，同时也会执行控制流分析。由于 rustc 提供了一组不稳定的应用编程接口用于和 MIR 交互，这使得 MIR 成

[1] **https://substrate.io/**

为了诸多外部工具处理 rust 程序代码的不二选择，如 MIRChecker [4] 、Kani [5] 等均采用 MIR 作为其分析对象。需要注意的是，由于这组结构的不稳定性，这些外部工具大多是针对某一特定版本的 rust nightly 进行开发，无法保证对其他 rust nightly 版本的兼容性。

4. LLVM 中间表示（LLVM IR）：该中间表示由 MIR 降低层级而来，是 LLVM 编译工具链可以识别并加以利用的中间表示。然而，由于 LLVM IR 的表达能力有限，许多 rust 源代码中隐含的信息无法合理表示，例如 LLVM IR 中没有无符号类型，导致所有 u32 被替换为 i32，为分析工作增添了额外的负担，故如今 rust 软件开发者们几乎不再选择它作为分析工具的分析对象。

综合上述中间表示优缺点，尤其是 rust 编译器为 MIR 提供的一组（不稳定的）API 这一因素，本课题选择使用 MIRAI 采用的 MIR 中间表示作为分析对象。

1.2.4. PRAZI

为提升代码可复用性，许多现代编程语言如 JavaScript、rust 等均提供了公开可用的软件包托管平台，并辅之以清单文件来描述一个软件包依赖于其他软件包的情况。若将软件包甲依赖软件包乙的这种依赖关系视为一条有向边，那么在软件托管平台上存在的所有软件包之间即能构造出一张错综复杂的包依赖关系网络（Package Dependency Network，PDN）。通过研究包依赖关系网络，可以获得软件包级别的依赖信息，例如某个包依赖了哪些其他包的具体版本，哪些包被依赖次数最多等信息。

近年来，随着一些托管平台上出现投毒行为，所有依赖于这些软件包的软件均面临小至程序输出异常，大至重要资料被窃取、计算机系统遭到破坏等风险。为规避风险，开发者不得不舍弃这些带毒软件包转而选择替代品，甚至忍痛放弃软件的一部分功能。然而，依据软件包之间依赖关系的不同，开发者很可能根本没有使用到这些被污染的代码，却受制于包依赖关系网络的大粒度分析产生的假阳性结果作出了不必要的牺牲。从另一个方面来说，大粒度的分析也不利于安全人员分析包内实现是否具有缺陷或质量低下。

因此，Joseph Hejderup 等人提出了一种全新的依赖关系网络：函数调用依赖关系网络（Call-graph Dependency Network，CDN）[6]，并利用之进行更细粒度的分析。为获得函数调用依赖关系网络，他们开发了 Prazi 分析器。这是一款基于 MIR 中间表示、利用 Docker 的虚拟环境进行软件包编译、借助 rust-callgraphs 工具生成函数调用图的分析工具，能够针对一个 crate 生成它的函数调用依赖关系网络。

为证明该方案的可用性，作者团队为 crates.io 上的所有 crate 都使用 prazi 进行了分析，并通过分析统计数据得出了有价值的结论。在 crates.io 托管的所有软件包中，50% 的函数调用是在调用依赖项 crate 中的函数而非当前 crate 中定义的函数。不仅如此，虽然一个 crate 在其 78.8% 的直接依赖项中至少会调用一个函数，但在其传递依赖项中至少调用一个函数的概率却锐减至 40%，这表明软件包的所有传递依赖项中有一半以上可能没有被调用。若能针对这部分无用代码进行裁剪，势必能有效减小代码库体积和编译产物体积，令已有的项目运行更轻巧。

虽然 prazi 的功能强大，但正如上文所言，受制于 rust 编译器提供的 MIR 编程接口的不稳定性，该工具已无法通过编译。因此，只有另寻或专门开发新的工具，方能令实施本课题所需的以函数调用为最小单元的分析方法成为可能。经过咨询、调研，笔者认为 MIRAI 工具能够接替 rust-callgraphs 工具履行调用图生成这一职责，详见下文叙述。

1.2.5. MIRAI

MIRAI 是工作在 rust 编程语言的中层中间表示（MIR）上的分析工具[3]。开发团队发现针对 rust 程序设计语言开发的静态分析工具在 rust 软件生态中几乎是一片空白，因此启动了 MIRAI 项目用于补全这个短板。

与追求创新性的工具不同，MIRAI 的宗旨是开发一个 rust 编译器“插件”，可以方便地集成至 cargo 中调用。在运行效果方面，MIRAI 希望能在尽可能低的假阳率下定位语义模糊

[2] **https://blog.rust-lang.org/2016/04/19/MIR.html**

[3] **https://github.com/facebookexperimental/MIRAI**

的 rust 代码，并给出切实可行的修改建议；同时这个分析过程应当尽可能快。

MIRAI 包含许多功能，其中的函数调用关系生成子功能（下简称为 MIRAI-CGG）十分强大。MIRAI-CGG 支持为指定 crate 生成函数调用关系信息，这些信息不仅可以输出为`.dot`文件，供 Graphviz 工具绘制为矢量图，还能输出`JSON`格式的函数调用位置（callsite）信息，包含发生调用的源代码文件虚拟路径、行号、列号，以及调用者（called）与被调用者（callee）的信息，十分全面。

本课题将主要利用这些函数调用位置信息，进一步进行分割，获得 rust 模块（crate）内部及模块间调用情况，再分别根据此两种情况确定 crate 内函数的调用情况，进而给出裁剪建议

2. 研究内容

在进行实验室工作时发现，从 X86 向 RISC-V 的跨架构软件移植工作存在依赖关系错综复杂、代码多有冗余等情况，不利于移植工作开展。由 Prazi 通过提取函数调用 & 函数定义信息可以去除这种冗余，降低移植的难度。因此，我们的课题尝试开发一款工具来提取函数调用 & 函数定义信息，为软件开发者给出代码裁剪建议，并实地运用这个工具对 Substrate 区块链实现 Substrate Node Template 进行裁剪和部分移植以证明该工具对移植工作确实具有积极意义。

3. 研究方案

4. 预期研究成果

5. 本课题创新点

参考文献

[1] 孙卫真, 杜香燕, 向勇, 汤卫东, and 侯鸿儒, “基于 RTL 的函数调用图生成工具 CG-RTL,” 小型微型计算机系统, vol. 35, no. 3, p. 5–6, 2014.

[2] 向勇, 汤卫东, 杜香燕, and 孙卫真, “ 基于内核跟踪的动态函数调用图生成方法 ,” 计算机应用研究, vol. 32, no. 4, pp. 1095–1099, 2015.

[3] 贾荻, 向勇, 孙卫真, and 曹睿东, “ 基于数据库的在线函数调用图工具 ,” 小型微型计算机系统, vol. 37, no. 3, pp. 422–427, 2016.

[4] Z. Li, J. Wang, M. Sun, and J. C. Lui, “MirChecker: Detecting Bugs in Rust Programs via Static Analysis,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2183–2196. doi: **10.1145/3460120.3484541**.

[5] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, “Verifying dynamic trait objects in rust,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 321–330. doi: **10.1145/3510457.3513031**.

[6] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, “Präzi: from package-based to call-based dependency networks,” *Empirical Softw. Engg.*, vol. 27, no. 5, Sep. 2022, doi: **10.1007/s10664-021-10071-9**.