# 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

近年来，越来越多的开发者为 rust 编程语言的高内存安全性和高效率性所吸引，愿意使用之以开发自己的操作系统及配套的软件生态环境。然而，操作系统开发的关键点之一在于丰富的软件生态环境，这正是新兴操作系统所欠缺的。鉴于 Linux 操作系统的生态环境相对成熟性，将 Linux 应用程序移植到生态环境尚不成熟的的新兴操作系统中成为了快速拓展新兴操作系统软件生态环境的方案之一。移植软件，首先需要理解移植对象的业务逻辑。然而，各种已有软件中广泛存在的代码异味，尤其是死代码，使得理解它们的业务逻辑变得更加困难。

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。目前，rust 语言已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，且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。

本选题提出一种基于函数调用图进行死代码探测的技术方案，帮助开发者更高效地移除死代码，在简化软件业务逻辑理解的同时，降低代码库体积，提升软件运行效率；为验证这一方案的可行性，本选题还将以实际 rust 软件项目为对象进行死代码移除，并移植到真实操作系统中，使之成为顺应 Web 3.0 时代潮流的区块链操作系统。

### 1.2. 国内外研究现状

#### 1.2.1. 死代码的概念

死代码是一种广泛存在于软件源代码中的代码异味。不同学派对死代码的定义不尽相同。Brown 等人将死代码定义为在不断变化的软件设计中始终未移除的未使用代码[1] 。Mantyla 等人认为：死代码就是过去使用过，但目前已不会再被执行的源代码[2] 。Wake 将未使用的变量、函数参数、类成员属性、类方法和类本身视为死代码[3] 。Martin 将死代码定义为从未执行过的代码（例如永假 if 内的语句块），而死函数是永远不会被调用的方法[4] 。而在程序设计语言领域，死代码是指其结果从未被使用的计算（例如，在代码中引用的变量，但在运行时实际上不使用）[5] 。虽定义不同，其本质并无二样，即死代码是在程序运行过程中永不可能被执行的部分。

虽然死代码并不会被执行，但它们对软件开发与维护仍具有负面影响，主要可以归结为以下几点[6]：

- 令代码更难理解: 开发者更难理解代码的结构和用意[7], [8] 。对于经验尚不丰富的新开发人员，他们可能会误以为死代码是有用的，从而轻则花费不必要的时间弄懂死代码、重则任其累积令代码库的质量越来越差。
- 令代码更难维护：让维护工作变得更加复杂，在日常维护或升级迭代时影响开发人员的工作效率、在降低代码质量的同时可能还会引入新的缺陷甚至错误[9] 。
- 徒增开发工时：开发者花费无用的时间维护死代码或对其进行 debug，而这部分工作对项目并无任何帮助。

- 降低运行效率：虽非总是如此，但死代码有可能降低软件运行效率或徒增内存占用。

### 1.2.2. 死代码探测与移除技术

目前，针对不同编程语言的死代码探测及移除技术不断涌现，其中较具代表性的有 JavaScript 的 Tree Shaking 技术方案，PHP 的 Web 系统的动态标记技术方案和 Java 的 DUM 技术方案。

Tree Shaking 技术方案依赖 ES2015 标准引入的 `import`、`export` 模块语法进行死代码移除。它通过分析依赖关系，确定未使用的模块并在最终生成的代码中将它们剔除。然而其简单的技术原理与思想也带来了局限性：它只能以模块为单位，而不能以模块中具体的功能为单位进行移除，导致生成的源代码体积仍然过大；另外，该过程对开发者透明，开发者无从知晓哪些模块被删除，因此无法辅助开发者主动提高代码质量。

PHP 的 Web 系统动态标记技术方案[10] 首先收集 Web 系统用到的所有文件，然后为其标注元数据，包含首次、最后使用时间，使用次数等。然后运行该 Web 系统，利用动态分析技术维护并追踪上述指标的变化[11] 。运行系统一段时间以后查阅元数据，即可得知哪个文件是冗余或不常用的。该方案在 Hostnet 工业规模中进行了测试，并安全高效地移除了 30% 的原始代码库中的死代码。虽然如此，动态分析技术的特性决定了运行该技术方案需要覆盖率足够广、持续时间足够长的测试以减少漏检；且最终结果需要人工决策，无法实现完全自动化。同时，它也和 Tree Shaking 一样，只能实现文件级的，粒度较大的死代码移除。

Java 的 DUM 技术方案提出了一种基于静态分析的技术方案，用于探测 Java 桌面软件中的不可达方法[12] 。它被设计为在 Java 字节码上工作，利用其中的信息将源程序转换为有向图表示[13] 。建图完成后，通过从一个起始节点开始遍历之来识别可达节点，其余即视为不可达节点（代表不可达方法）。与 JTombstone、Google CodePro AnalytiX 等行业已有工具比较，DUM 表现出了更高的查准率。与前两种技术方案相比，DUM 的分析粒度可达方法级别，远远细于 Tree Shaking 的模块级别和 PHP 动态标记技术的文件级别，使之能够移除尽可能多的死代码，最大地减轻开发者理解已有软件业务逻辑的心智负担，提升软件代码库质量及其运行效率。

综上所述，死代码对软件项目的负面影响使其成为有必要移除的代码异味，目前较有代表性的技术方案均有其优越性及局限性，在这其中，基于函数调用图的静态分析方法能够实现细粒度的死代码探测，其良好的效果与函数调用图在主流编程语言上的广泛性使得它成为在 rust 编程语言上实现死代码探测的理想途径。

### 1.2.3. CG-RTL 函数调用图生成工具

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。文献 [14] 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，最终绘制成一张函数调用有向图。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 [15] 和基于数据库的函数调用图生成工具 [16] 。

CG-RTL 解决了已有工具需要基础知识、产生过量冗余信息、和编译环境耦合度过高的问题。虽其仅适用于 C 语言的 RTL 中间表示分析，但其利用中间表示进行逐模块分析的方法仍然值得借鉴。

### 1.2.4. rust 的中间表示

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（Lowering），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。按层级由高到低，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 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[1]，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 API 接口用于和 MIR 交互，MIR 成为了诸多外部工具处理 rust 程序代码的不二选择，如 MIRChecker [17] 、Kani [18] 等均采用 MIR 作为其分析对象。

#### 1.2.5. Prazi 与函数调用依赖关系网络

随着托管平台投毒等安全威胁的出现，基于包依赖关系网络（Package Dependency Network，PDN）的大粒度分析已不能满足当下软件安全分析的需求。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）[19]，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外部依赖项中的函数。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。

虽然受制于 rust 编译器提供的 MIR 编程接口的不稳定性，Prazi 分析器已无法使用，但其证明了在 rust 项目上生成函数调用图并进一步分析的技术可行性；同时，传递依赖项的函数调用率锐减也反映了 rust 软件项目中死代码存在的广泛性，进一步证明了选题的意义。

#### 1.2.6. MIRAI 与 Rupta

MIRAI 由来自 Facebook 的技术团队开发，是一款工作在 rust 编程语言的中层中间表示（MIR）上的静态分析工具[2]。与追求创新性的学术用途原型工具不同，MIRAI 追求在实际工业生产环境中的实用性。其宗旨是在尽可能低的假阳率下定位存在异味的 rust 代码，并给出切实可行的修改建议。

[1] **https://blog.rust-lang.org/2016/04/19/MIR.html**
[2] **https://github.com/facebookexperimental/MIRAI**

MIRAI 的功能之一是函数调用关系生成功能（下简称为 MIRAI-CGG）。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。然而，其低假阳率的目标并不完全适用于分析死代码。经过试验发现，MIRAI 对函数调用情况存在漏报的情况，开发者若完全按照 MIRAI 的输出进行死代码移除，有概率将具有实际作用的代码移除导致软件功能缺失甚至编译不通过。因此，有必要对 MIRAI 进行必要的修改，降低其漏报率，方能用于死代码移除。

Rupta[20] 是一款上下文敏感指针分析框架，观察其代码库易知其脱胎于 MIRAI 而优于 MIRAI，在构建函数调用图（call graph）方面的能力优于 MIRAI。Rupta 采用基于调用点的上下文敏感性算法，为 Rust 中常见的静态分派、动态分派、嵌套数据结构等情况分别采用了针对性的算法方案，从而提供更精确的分析结果。与现有的两种技术 Rurta（基于快速类型分析）和 Ruscg（仅静态分派）相比，Rupta 在完整性上发现更多的调用关系，并在精度上消除了大约 70% 的虚假动态调用关系。

将 MIRAI 与 Rupta 比较，不难得出其各自的优越性：MIRAI 输出的信息格式规整，利于分析；但其低假阳率的设计初衷导致调用关系不全，存在漏报的问题；Rupta 的查全率显著更优，但其仅能输出 `.dot` 格式的有向图文件，缺少了分析死代码所需的必要信息。二者均需经过不同程度的修改，方能用于本选题的应用场景中。

#### 1.2.7. Substrate 区块链

Substrate 由以太坊项目的联合创始人 Gavin Wood 率领 Parity 团队开发，是一个基于 rust 的开源区块链框架。它对于多链结构提供了较好的支持，能够克服传统单链区块链在一条链上保有大量用户、存储大量信息，导致运行效率降低的问题，且针对多链间通信提出了一套解决方案。不仅如此，高度可定制的灵活模块化设计模式，使得 Substrate 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。

目前，官方已推出了便于开发者开发的 Substrate Node Template（下简称为 SNT）[3]。它完全基于 rust 编程语言开发，内含一个包含最基本功能的 Substrate 全节点实现。其代码库大小适中，既能体现本选题提出的死代码探测方案的泛用性，又不至于因代码量过大导致分析过分繁杂。此外，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。

##### 1.2.7.1. SNT 的代码结构

SNT 的代码库如图 Figure 1 所示，其本质是一个 Rust 工作空间（workspace），内含三个成员：node, runtime 和 pallets。这其中，pallets 成员存储 SNT 使用的所有自定义功能模块（即上文中提及的 pallet），内仅含一个模板 pallet，称之为 template；runtime 成员主要定义了 SNT 在运行时的链上状态转换逻辑，为单 `lib.rs` 文件结构；node 成员负责 P2P 网络通信、区块产生和确认（finalization）、处理外部 RPC 请求等链外事务，内含多个源代码文件，有继续细分的必要。

代码阅读的结果表明，node 成员以 8 份 rust 源代码组成，其中：

- `lib.rs` 负责将自身的 `rpc`，`chain_spec` 与 `service` 模块暴露给外界使用，而 `main.rs` 仅负责启动 `command` 模块中的 `run` 函数。它们并未为 SNT 实现更多功能，在后续代码分析时可以忽略。

[3] https://github.com/substrate-developer-hub/substrate-node-template

- `cli` 模块定义了一系列子命令，而 `command` 模块则利用前者定义的子命令结合用户传入的命令行参数进行解析，并根据之采取不同的行为。二者联合为 SNT 提供了命令行参数解析的服务，可以在逻辑上合并为一个逻辑模块。
- `chain_spec` 模块实现了初始化链时的配置选项，例如链的命名与唯一标识，链中预置的账户信息和账户余额等等。
- `rpc`
- `service`
- `benchmarking`

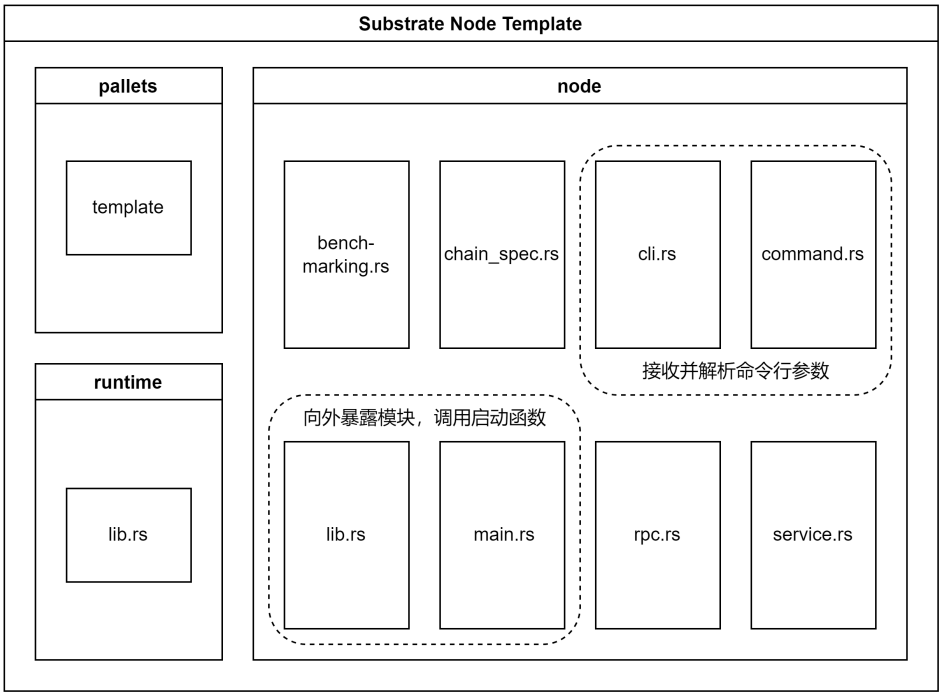


Figure 1: Substrate Node Template 代码结构

SNT 的大致依赖关系如 Figure 2 所示。结合上文对各模块功能的描述，可给出一拓扑排序，指示令 SNT 自底向上摆脱具体操作系统依赖的实现顺序。

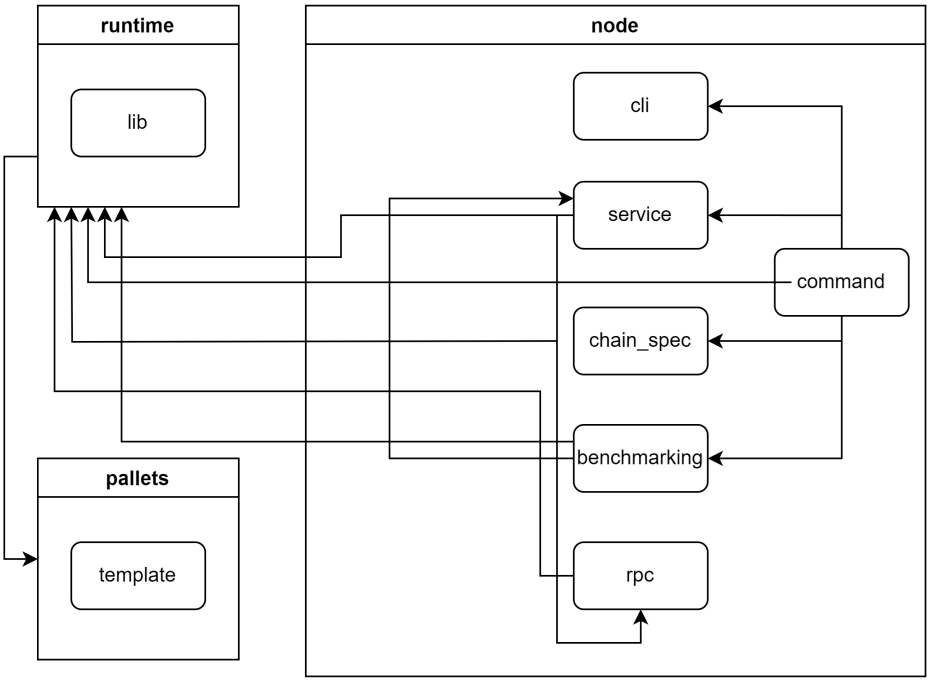


Figure 2: Substrate Node Template 大致依赖关系

**1.2.8.** 软件移植研究现状

代码复用是提升软件产品开发效率的一种常用且有效的方法。常见的代码复用手段主要包含软件模块化设计和软件移植。前者将大型软件拆分成小的可复用的模块，供其他开发者选用并成为他们所开发的软件的一部分；后者将大型软件在不同的操作系统甚至不同的硬件平台上进行移植，以拓宽同样功能代码的泛用性。

软件移植技术是关心后者的，即将软件从一个操作系统或硬件平台移植到另一个的技术。于开发商而言，软件移植以小于重新开发的代价令已有软件在新平台上得以运行，有益于拓宽市场，优化用户使用体验。然而，跨操作系统甚至硬件平台的移植并非易事，因为开发者在这一过程中可能会遇到来自硬件和软件方方面面的阻碍，例如硬件上的大小端存储、软件上的目标平台的系统调用实现不全、编程语言本身存在未定义行为导致在不同编译器实现中表现出不同的行为等问题。因此，评估软件移植难度，并设法降低这个难度便成为了众多学者与开发者追求的共同目标 [21] 。

在 Wolberg 等人的研究 [22] 中，代码库尺寸（即代码行数）与移植难度呈现指数级正相关。文献 [23] 在前者的基础上进一步将阻碍移植因素、移植代价因素纳入考量，获得了一个评估量化移植难度的模型。在构建模型时，作者团队发现移植效率的差异本质上取决于移植工程师的经验和技巧的差异，以及开发和测试环境的差异。因此，他们也呼吁从移植辅助工具、移植指南手册和软件设计准则三方面入手降低移植工作的难度。

上述两项工作的共同结论之一是，代码行数的增加会令移植工作更具挑战性。这从侧面证明了本选题所采取技术手段的合理性：通过移除不必要的代码，减少代码行数，从而降低移植工作难度，提升移植效率。

#### 1.2.9. rust 与 Linux 内核开发

##### 1.2.9.1. rust 与 C 的互操作性

作为系统级编程语言，rust 亦响应社区呼吁，推出了对 C 语言的兼容性支持。换言之，在 Rust 中编写的函数，通过链接后可以在 C 语言代码中调用。

首先创建一个 Rust 的 lib 项目：

```
cargo new example_lib --lib
```

然后编写如下示例代码（略去了为该项目添加 fastrand 依赖的 Cargo 指令）：

```
#[no_mangle] // 阻止编译器对函数名进行重命名，必须
#[export_name = "get_random_u32"] // 进一步规定导出的函数的名字，必须
pub extern "C" fn get_random_u32() -> u32 {
    let mut rng = Rng::new();
    rng.u32(1..=100)
}
```

C 代码如下编写即可编写：

```
#include <stdio.h>

extern int inc(int x);
extern int get_random_u32();

int main() {
    int x = get_random_u32();
    printf("%d\n", x);
    return 0;
}
```

最后混合编译并运行：

```
cargo build --release
gcc test.c -L ./target/release/ -l example_lib -o main
LD_LIBRARY_PATH=./target/release/ ./main
```

经过测试发现 get_random_u32 可以正常调用，并且返回一个随机数。这也证明混合编程时在 Rust 库中包含依赖项（例如该例中的 fastrand）是完全可行的。

## 2. 研究内容

本选题提出一种基于函数调用图进行死代码探测的技术方案，帮助开发者更高效地移除死代码；为验证这一方案的可行性，本选题还将以 Substrate 区块链项目为对象进行死代码移除，并移植到 rCore 操作系统中。工作大致可分为两部分：

- 改进并利用 MIRAI 获取 rust 软件项目的函数调用图，采用基于函数调用图的方法探测死代码并标记之，以在简化软件业务逻辑理解的同时，降低代码库体积，提升软件运行效率
- 将经过代码裁剪的 Substrate 移植到教学操作系统 rCore 中，使之成为顺应 Web 3.0 时代潮流的区块链操作系统。

# 3. 研究方案

## 3.1. 死代码探测实施方案

目前，市面上尚无针对 rust 编程语言进行死代码探测的工具。故本选题计划以 MIRAI 静态分析为基底，以死代码探测为目标进行修改，去除不必要的功能并完善函数调用图生成功能，以构造基于函数调用图的死代码探测技术方案。为验证该方案的可行性，取 Substrate 全节点客户端作为待分析项目，探测并移除其死代码，获得精简的 Substrate 全节点客户端。

### 3.1.1. 函数调用图的构造

所谓 rust 项目的函数调用图，乃是一张有向图。图中，每个节点均代表一个该项目中出现的函数；若函数 A 调用了函数 B，则从代表函数 A 的节点引一条指向代表函数 B 的节点的有向边。通过遍历该有向图，能够区分可达节点和不可达节点，进而确定不可达函数，从而提示开发者保留可达函数，移除不可达函数，达到死代码移除的目的。

MIRAI 和 Rupta 均工作在 rust 编程语言的 MIR 中间表示上，其最小分析粒度为函数。由 rust 中间表示的特性可知，自 THIR 开始，rustc 就抛弃了诸如 `struct`、`impl` 块等内容，而将所有的方法均展开成为了普通的函数定义。在 THIR 及之后的中间表示中，函数定义始终以 `body` 的形式存在，且每个函数均以被称为 `DefId` 的编号唯一标识。这种行为非常符合选题构造函数调用图的需求。

对 Substrate 全节点客户端分别调用 MIRAI 和 Rupta 进行函数调用关系分析发现，Rupta 输出的调用关系文件体积远大于 MIRAI 输出的关系文件，为后者的 250 倍。浏览其中的内容亦发现，Rupta 发现了实际存在、且 MIRAI 未发现的函数调用关系。因此，本选题决定在基于 MIRAI 而优于 MIRAI 的 Rupta 上进行二次开发，令其能够输出更多、更结构化、更便于分析的信息，以供死代码探测之用。

正如前文所言，rustc 为 MIR 提供了一组不稳定的 API 供开发者调用，其中提供的 `rustc_driver::Callbacks` 特征（`trait`）允许开发者介入 rustc 编译 rust 项目的不同阶段，获取编译信息并执行自定义的回调函数进行分析等操作。

MIRAI 实现了该特征中的 `after_analysis` 回调方法，该回调方法在 rustc 编译获得 MIR 之后，继续降层为 LLVM IR 之前调用，可获得编译器在将源代码编译到 MIR 中间表示中收集的所有信息。通过该方法，MIRAI 进行了自顶向下，由最大范围的 rust 包（crate）级分析到最小范围的函数调用级分析的过程，其调用链大致如 Figure 3 所示。

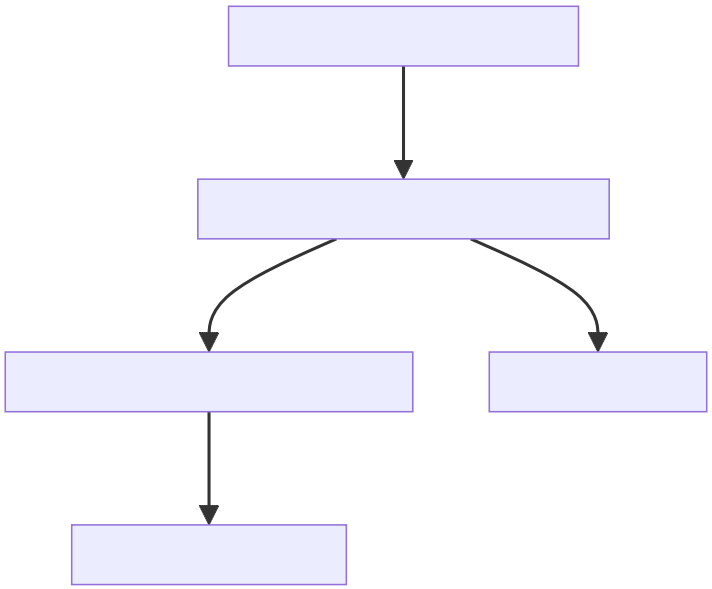

Figure 3: MIRAI 的调用链

现阶段，MIRAI 的输出存在函数调用信息不全导致查全率低、函数所属 Crate 相关信息缺失导致死代码探测工作困难等问题。因此，需要通过在调用链中的合适层级修改代码，以满足死代码探测必须的信息需求。同时，重构或另外实现一套信息输出机制，以合理的、有利于标识死代码的结构展示分析结果。预期的分析结果条目大致如下：

- crate 信息
  - crate 索引号，从 0 开始自增
  - 包含版本号的 crate 名称
  - crate 所在路径，以其 Cargo.toml 所在路径指代
- 函数信息
  - 函数索引号，从 0 开始自增
  - 函数所属 crate 编号
  - 函数所在源代码的路径
  - 函数在源代码中的行号
- 调用信息
  - 调用者索引号与被调者索引号构成的二元组

整体内容和 MIRAI 现阶段的分析结果输出相比结构和内容均有不同，主要增加了 crate 的完整名称及位置信息，方便开发者区分同一 crate 的不同版本，更有针对性地进行死代码探测及移除。

## 3.2. 跨操作系统移植实施方案

考虑到 rCore 操作系统同样以 rust 写成，且结构简单利于分析，故以 rCore 为移植目标平台，对精简的 Substrate 全节点客户端开展移植工作。这包含替换或重写不适配 rCore 运行环境的依赖项，并为 rCore 实现必须的系统调用。最终，将 rCore 打造成包含部分区块链特性的区块链操作系统。

### 3.2.1. rust-std 标准库

为方便开发者在主流操作系统上进行开发，降低开发人员负担，rust 为其在主流操作系统上的实现配备了标准库 rust-std。下至内存分配、缓冲区管理，上至网络通信等功能，rust-std

均提供了支持，封装了与操作系统交互的内容，使得开发者能够专注于开发软件业务逻辑上。rCore 作为新兴的教学操作系统，rust 并未为其提供标准库实现，故想将应用移植到 rCore 上，有两个必须进行的步骤：

1. 替换或改写需要标准库支持的依赖项
2. 实现应用需要的系统调用

随着嵌入式开发在 rust 社区中的兴起，在无标准库支持的设备上进行 rust 开发的需求水涨船高，一个名为 no-std 特性的概念也应运而生。若一个 crate 声称自己对 no-std 环境兼容，则该 crate 能够在无标准库支持的环境下提供至少一部分功能供应用使用。在诸如 crates.io 和 lib.rs 这些 rust crate 介绍平台上，支持此类特性的 crate 往往会为自己标注 no-std 的 tag。在进行现有 crate 替换时，应尽可能采用这些支持 no-std 特性的 crate，以降低改写工作量。

## 4. 工作时间安排

- 2024 年 3 月～2024 年 5 月：文献阅读。
- 2024 年 6 月：开题，方案设计。
- 2024 年 7 月～2024 年 11 月：死代码探测技术方案实现、开展移植工作。
- 2024 年 12 月～2025 年 2 月：毕业论文撰写。
- 2025 年 3 月～2025 年 6 月：毕业论文完善和毕业答辩。

## 5. 预期研究成果

预期硕士学位论文一篇，死代码移除技术方案一套，带有部分区块链功能特性的操作系统软件一套。

## 6. 本课题创新点

本选题的创新点有：一是为 rust 编程语言实现了一种基于函数调用图的死代码探测技术；二是将经过死代码移除处理后的软件移植到另一操作系统中，以证明该技术对于移植工作的便利性贡献度。

## 参考文献


[1] W. H. Brown, R. C. Malveau, H. W. ". McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. USA: John Wiley & Sons, Inc., 1998.

[2] M. Mantyla, J. Vanhanen, and C. Lassenius, “A taxonomy and an initial empirical study of bad smells in code,” in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 2003, pp. 381–384. doi: **10.1109/ICSM.2003.1235447**.

[3] W. C. Wake, *Refactoring Workbook*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[4] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. USA: Prentice Hall PTR, 2008.

[5] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, “Compiler techniques for code compaction,” *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000, doi: **10.1145/349214.349233**.

[6] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, “A Multi-Study Investigation into Dead Code,” *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 71–99, 2020, doi: **10.1109/TSE.2018.2842781**.

[7] Godfrey and Q. Tu, “Evolution in open source software: a case study,” in *Proceedings 2000 International Conference on Software Maintenance*, 2000, pp. 131–142. doi: **10.1109/ICSM.2000.883030**.

[8] P. Oman and J. Hagemeister, “Metrics for assessing a software system's maintainability,” in *Proceedings Conference on Software Maintenance 1992*, 1992, pp. 337–344. doi: **10.1109/ICSM.1992.242525**.

[9] D. L. Parnas, “Software aging,” in *Proceedings of the 16th International Conference on Software Engineering*, in ICSE '94. Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 279–287.

[10] H. Boomsma, B. V. Hostnet, and H.-G. Gross, “Dead code elimination for web systems written in PHP: Lessons learned from an industry case,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 511–515. doi: **10.1109/ICSM.2012.6405314**.

[11] T. Ball, “The concept of dynamic analysis,” *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 216–234, Oct. 1999, doi: **10.1145/318774.318944**.

[12] S. Romano, G. Scanniello, C. Sartiani, and M. Risi, “A graph-based approach to detect unreachable methods in Java software,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, in SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, pp. 1538–1541. doi: **10.1145/2851613.2851968**.

[13] F. Tip and J. Palsberg, “Scalable propagation-based call graph construction algorithms,” *SIGPLAN Not.*, vol. 35, no. 10, pp. 281–293, Oct. 2000, doi: **10.1145/354222.353190**.

[14] 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.

[15] 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.

[16] 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.

[17] 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**.

[18] 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**.

[19] 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**.

[20] W. Li, D. He, Y. Gui, W. Chen, and J. Xue, “A Context-Sensitive Pointer Analysis Framework for Rust and Its Application to Call Graph Construction,” in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, in CC 2024. <conf-loc>, <city>Edinburgh</

city>, <country>United Kingdom</country>, </conf-loc>: Association for Computing Machinery, 2024, pp. 60–72. doi: **10.1145/3640537.3641574**.

[21] J. D. Mooney, "Developing Portable Software," in *Information Technology*, R. Reis, Ed., Boston, MA: Springer US, 2004, pp. 55–84.

[22] J. R. Wolberg, "Comparing the cost of software conversion to the cost of reprogramming," *SIGPLAN Not.*, vol. 16, no. 4, pp. 104–110, Apr. 1981, doi: **10.1145/988131.988144**.

[23] M. Hakuta and M. Ohminami, "A study of software portability evaluation," *Journal of Systems and Software*, vol. 38, no. 2, pp. 145–154, 1997, doi: **https://doi.org/10.1016/S0164-1212(96)00118-5**.