文献综述

计算机学院 **2023** 级硕士 **9** 班 **3220231370** 傅泽

1. 前言

本文献综述旨在梳理死代码检测及移除和跨架构软件移植领域的研究现状与发展趋势，为课题《基于死代码移除的跨操作系统软件移植方案》提供较全面的理论依据。

近年来，越来越多的开发者使用 Rust 开发自己的操作系统及配套的软件生态环境，诸如 BlogOS[1]、rCore[2]和 RedoxOS[3]等基于 Rust 的操作系统内核如雨后春笋般涌现。鉴于 Linux 操作系统生态环境的成熟性，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。然而，各种已有软件中广泛存在的死代码等代码异味，不利于开发者理解软件逻辑和进行移植工作。

本文分析了自上世纪末至今关于死代码定义、检测和移除手段、跨架构软件移植等方面的多项研究成果。调查表明，不同学派对死代码定义的外延不尽相同，但其内涵极其相似；死代码检测和移除手段已在众多程序设计语言中广泛应用，其中 Java 作为编译型语言采用字节码分析和函数依赖有向图的静态分析方案，相较 JavaScript 等解释型语言采用的 Tree Shaking 方案，具有粒度更小，效率更高的优势；软件移植方面，历代学者为其建立了难度评估模型，其中代码行数指标 LoC 因其带来的显著开发心智负担占据极大权重，从侧面证明了死代码移除对简化移植流程的重要性。

本综述为后续深入探究 Rust 编程语言的死代码检测、移除以及跨架构的 Rust 软件移植方案奠定了基础，有助于研究者明确研究方向，填补相对成熟的死代码移除方案在 Rust 系统编程领域的相对不足，推动该领域进一步发展。

2. 死代码定义、检测与移除

2.1. 定义死代码

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] 。虽各学派对死代码的外延并未达成完全共识，但其内涵实则并无矛盾，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，但已有研究证明它们对软件开发与维护仍具有负面影响[6] ，其中文章[7] 将死代码的负面影响归结为了以下几点：

- 无益代码理解: 开发者更难理解代码的结构和用意[8], [9] 。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确有某种作用，从而任其累积，令代码库的质量越来越差。
- 平添维护难度：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[10] 。

[1] **https://github.com/phil-opp/blog_os**
[2] **https://rcore-os.cn/**
[3] **https://www.redox-os.org/**

- 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。

总结而言，死代码可被理解为在程序运行过程中永不可能被执行的部分。尽管去除它们是维持软件代码质量的重要一环，但目前死代码移除工作尚未在业界引起足够的重视。

2.2. 死代码探测与移除技术

死代码探测技术，即在源代码中查找符合上一章定义的死代码的技术方案。除降低软件复杂性外，死代码探测技术亦有许多用途，例如文章 [11] 提出以死代码探测评估软件质量高低的方法。目前，针对不同编程语言的死代码移除技术不断涌现，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[12] 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[13] 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[14] 。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[15] 。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[16] 。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。

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[17] 。针对一些 C 语言函数调用图生成工具存在的需要基础知识、产生过量冗余信息、和编译环境耦合度过高的问题，文献 [18] 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 [19] 和基于数据库的函数调用图生成工具 [20] 。

CG-RTL 系列工作的一大特点在于，采用编译过程的中间结果进行分析，将词法分析、语法分析、语义分析等任务交给了编译器，既避免了直接分析源代码带来的巨量工作量，又不需对编译器进行侵入式修改，从而将对编译器的耦合度维持在了较低水平。这种利用中间表示进行逐模块分析的方法非常值得借鉴。

2.4. rust 的中间表示

CG-RTL 使用寄存器传送语言作为用于分析 C 语言源代码的中间表示，DUM 使用字节码作为其生成函数调用有向图的信息来源。若想在对 Rust 的死代码检测中使用中间表示辅助分析，就必须研究 Rust 编译器的中间表示的分类和特点。

在 Rust 中，为方便从不同的角度进行全面完善的检查，编译器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。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[4]，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 [21] 、Kani [22] 等均采用 MIR 作为其分析对象。

综合上述三种中间表示的特点，可以得出结论：使用 Rust 编译器提供的，操作 MIR 的接口进行分析是最理想的方案。这种方案的优势在于：免去了人工编写中间表示分析器的过程，因为 Rust 编译器业已完成这部分工作；免去了人工从中间表示中提取信息的过程，因为 Rust 编译器已提供了一组 API 接口处理此事。同时，这种方案也存在一定的局限性，因为这组 API 接口并不稳定，意味着它在不同的 Rust 版本中均不尽相同，死代码分析器不得不和某个特定 Rust 版本强绑定，增加了一定的耦合度。

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

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

Network，CDN）[23]，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。

Prazi 诞生的最初目的并非用于死代码检测，且受制于 rust 编译器 MIR 编程接口的不稳定性，Prazi 分析器已无法使用。但其通过调查发现传递依赖项的函数调用率锐减，证明了 rust 软件项目中死代码的广泛存在，成为在 rust 项目上生成函数调用图并进行分析的技术层面可行性与必要性的又一例证。

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[5]。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。

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。

Rupta[24] 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。考虑到设计目标与测试表现，Rupta 更具有改造利用的价值。

3. 软件移植技术

3.1. 软件移植技术现状

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。文章 [25] 借用经济学中的概念，在软件工程中提出了“技术债务”的隐喻，指为节省成本而牺牲软件质量，导致后期维护成

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

本增加。技术债务的“本金”对应开发软件的工程量，而“利息”则指代维护软件的工程量。调查发现，代码复用在提高“本金”的同时，亦降低了“利息”，从而降低了技术债务的影响，有利于软件可持续性发展。

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；于开发者而言，软件移植允许在前人的基础上，针对自己的需求进行优化改良，开发个性化功能。文献 [26] 以 FreeBSD、OpenBSD 和 NetBSD 为例，针对软件移植进行了调研，发现它们的代码库中大量复用了彼此之间的代码，足以说明软件移植技术应用之广泛性。

然而，跨操作系统甚至硬件平台的移植并非易事。文章 [27] 以 LoongArch 为移植目标，将麻省理工学院教授操作系统所用的教具之一——XV6 操作系统移植到了该架构上，并对过程中遇到的阻碍及解决方案进行了详细分析记录。文章指出，移植过程中存在的技术难点主要有：内存管理方法不同、中断处理策略不同和文件系统泛用性目标不同等。不难看出，软件移植工作的机遇与挑战并存，其技术难点值得开发者设计更有效、更高效的处理方案加以应对。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 [28] 。

3.2. 软件移植难点

在 Wolberg 等人的研究 [29] 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 [30] 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。

除代码行数外，跨架构移植带来的硬件平台兼容性也是影响移植效率的重要因素。在移植过程中，不同的指令集、大小端内存布局、各异的系统调用接口等均对移植效率有显著影响。因此，为了提升移植效率，开发者需要充分利用硬件平台的特性，并在移植前对代码进行适当的优化，例如使用适合目标平台的编译器、使用标准库的替代品、使用特定平台的系统调用接口等。

4. 总结与展望

本综述首先规范了死代码的定义，并从软件移植的角度证明了死代码移除对于降低跨架构移植难度的价值。随后，调查了不同编程语言中实现死代码检测的方法，通过详细分析这些方法的实现原理和优势短板，归纳总结了函数调用有向图这一较有前景的死代码检测移除方案。围绕构造函数调用有向图这一目标，调查了 CG-RTL 及其相关工作，进一步将构造函数调用有向图的方法缩小到中间表示分析领域。在充分调查了 Rust 使用的三种中间表示各自的特点后，本综述提出了借助 Rust 编译器 MIR 接口进行中间表示分析的方案。为证明这一方案的可行性，调查了 Prazi 项目，以及其继承者 MIRAI 与 Rupta，分析这些项目的长板与局限性后，最终将目光锁定在 Rupta 项目上，并提出了 Rupta 的改造方案。

基于本综述所开展的全面梳理与分析，未来的研究工作可能聚焦多个关键方向推进。首先，在借助 Rust 编译器 MIR 接口进行中间表示分析的方案实施方面，将深入开展 Rupta 的实验研究，通过实验性分析大输入规模的测试样本集，精准评估该方案在实际应用场景中的效率与准确性。

其次，针对提出的 Rupta 改造方案，将编制计划进行方案的具体实现与优化。重点关注改造过程中的兼容性问题，确保 Rupta 在稳定运行的同时，具备输出便于分析的期望信息的能力。同时，积极探索 Rupta 与其他相关开发工具和框架的集成可能性，例如与代码版本控制系统、自动化测试工具等的有效结合，以构建一个更为完善的死代码检测与移除的开发生态链，最大程度地提升开发人员在处理跨架构软件移植时的工作效率。

最后，从理论研究角度出发，进一步深入探究死代码检测与移除算法的优化空间。引入先进的算法理论和数学模型，如机器学习中的聚类分析、神经网络优化算法等，尝试对现有函数调用有向图构建及死代码判断逻辑进行创新性改进[31]，以期突破当前技术瓶颈，实现更为精准、高效的死代码检测与移除效果，为跨架构软件移植领域提供更为强大的技术支撑与理论保障。

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. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K.-H. Prommer, “How much does unused code matter for maintenance?,” in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1102–1111. doi: **10.1109/ICSE.2012.6227109**.

[7] 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**.

[8] 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**.

[9] 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**.

[10] 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.

[11] T. Theodoridis, M. Rigger, and Z. Su, “Finding missed optimizations through the lens of dead code elimination,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, in ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 697–709. doi: **10.1145/3503222.3507764**.

[12] 杨健, “Deep Dive into Rspack & Webpack Tree Shaking,” 2024. [Online]. Available: **https://github.com/orgs/web-infra-dev/discussions/17**

[13] 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**.

[14] 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**.

[15] 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**.

[16] 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**.

[17] E. K. Ampomah, E. Mensah, and A. Gilbert, “Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages,” *Communications on Applied Electronics*, vol. 7, pp. 8–13, 2017, doi: **10.5120/cae2017652685**.

[18] 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, 2014.

[19] 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.

[20] 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.

[21] 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**.

[22] 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**.

[23] 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**.

[24] 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**.

[25] D. Feitosa, A. Ampatzoglou, A. Gkortzis, S. Bibi, and A. Chatzigeorgiou, "CODE reuse in practice: Benefiting or harming technical debt," *Journal of Systems and Software*, vol. 167, p. 110618, 2020, doi: **https://doi.org/10.1016/j.jss.2020.110618**.

[26] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, vol. 0, p. , 2012.

[27] Q. Li, W. Xu, S. Chen, and Z. Pang, "Porting and Enhancing the XV6 Kernel for LoongArch ArchitectureElevating Functionality: A Unified Approach to Kernel Porting and Enhancement," vol. 0, pp. 1–5, 2023.

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

[29] 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**.

[30] 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**.

[31] Y. Brun and M. Ernst, "Finding latent code errors via machine learning over program executions," in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 480–490. doi: **10.1109/ICSE.2004.1317470**.