CCF-华为胡杨林基金

软件工程

2023 年度申报课题及说明

CCF-华为胡杨林基金技术委员会——软件工程领域组

二零二三 年 四 月

第一条 课题

本专题中课题主要为了持续提升相关软件工程技术的能力上界，针对业界典型问题和痛点，解决关键技术瓶颈或形成技术断裂点，大幅提升产品/平台/解决方案竞争力，创造产业价值，搭建产学研创新机制与平台，加速各研究项目的落地与转化 。

第二条 基金池

2023 年度软件工程专题基金池共计 560-650 万人民币。

第三条 产业课题

单课题原则上资助额度为 40-60 万人民币，为期一年。提交成果中原则上需要包含源代码。

2023 年度拟资助 9 个课题：

1. 基于大模型的软件研发技术项目群
    1) 子方向 1-代码大模型的提示工程技术
    2) 子方向 2-代码表征与生成场景的预训练模型微调技术
    3) 子方向 3-工具增强的代码大模型后处理技术
    4) 子方向 4-面向可信代码生成的人机协同自然交互编程语言
    5) 子方向 5-代码分析与大模型结合的单元测试用例生成和演化技术
    6) 子方向 6-基于大模型的系统集成测试生成和优化技术
    7) 子方向 7-基于大模型的研发生产线智能支持技术
    8) 子方向 8-基于大模型的代码精准重构技术
2. 全链路压测演练场景下智能辅助性能工程
3. 基于 eBPF 的业务透明的应用观测技术
4. 未知开源漏洞的精准发现与验证
5. 基于流量的 API 安全分析与治理
6. 基于源码分析的恶意代码识别
7. Fuzzing 测试中利用机器学习技术增强深层代码覆盖能力项目
8. 污点分析误报漏报场景优化
9. C、C++二进制代码的成分分析技术

课题一：基于大模型的软件研发技术项目群

随着预训练语言大模型技术 LLM 的发展与成熟（如 Codex，GPT4，LLaMa，PaLM），越来越多的 LLM 技术被用于支持软件研发领域的各个环节（如代码生成与解释、单元测试用例生成、问题定位、测试脚本生成、智能运维等），从而提升各个环节的智能化水平和交付效率。为了更加系统性地探索大模型技术在软件工程领域的能力边界并充分利用 LLM 技术全面提升研发场景智能化水平，**特设立“基于大模型的智能软件研发技术项目群”**，鼓励广大高校研究团队紧跟时代最新技术，围绕该领域积极投入共同提升研发工具体系智能化水平。

该项目群内部包括 8 个项目子方向（列表参考如下），感兴趣研究团队可根据团队兴趣、技术积累等，**自由选择其中 1 个或多个子方向进行项目提案申报**（如选择多个子方向，请为每个子方向，单独提交申报材料）。所有申报材料收集完毕后会集中组织评审，根据接收到的提案材料质量情况，预计**围绕该项目群最终评审出 4 个左右项目提案**来推进后续胡杨林基金课题正式合作流程。

子方向列表（具体研究内容和项目目标，请参考该目录下的对应同名文件说明）：

1) 子方向 1-代码大模型的提示工程技术
2) 子方向 2-代码表征与生成场景的预训练模型微调技术
3) 子方向 3-工具增强的代码大模型后处理技术
4) 子方向 4-面向可信代码生成的人机协同自然交互编程语言
5) 子方向 5-代码分析与大模型结合的单元测试用例生成和演化技术
6) 子方向 6-基于大模型的系统集成测试生成和优化技术
7) 子方向 7-基于大模型的智能支持技术
8) 子方向 8-基于大模型的代码精准重构技术

子方向 1-代码大模型的提示工程技术

项目背景

随着预训练大模型技术的发展，越来越多的代码上的大模型如 Codex，GPT4，LLaMa，PaLM 等被提出。虽然这些大模型在一些代码上的任务取得了良好的零样本推理效果，但是面对复杂的任务需求如用户自定义类的类型推断、复杂代码的调试优化等场景仍表现不佳。因此，需要探索基于代码大模型的提示工程技术，充分挖掘代码大模型的潜力，提升其在实际复杂使用场景中的效果。

研究内容

面向软件工程中研发工具智能化，主要针对 Python、Java、C、C++、JS 和 Go 等常用编程语言，利用提示工程技术提升代码大模型的表现。在众多研发智能化任务中，选取开发阶段的代码生成、代码调试与优化任务进行研究。具体可选的研究方向有：

方向一：支持上下文关联的代码生成：探索在代码生成任务中，通过提示工程提高大模型对超长上下文的理解能力。针对任务选择高质量的上下文输入输出样例以及任务指令，利用代码切片、代码简化的代码长度压缩、串联上下文等方式为大模型提供理解超长上下文的提示，

在无需对模型进行精调的情况下，提升代码生成性能。
方向二：提高可执行性的代码调试与优化：探索在代码调试与优化任务中，通过提示工程提高输出代码的可执行性。通过语法分析、类型分析、数据流分析、控制流分析、依赖分析等方式提取代码结构化信息，通过提示工程和任务指令构建可辅助大模型学习代码中的可执行性特征的提示，在无需对模型进行精调的情况下，提升结果输出代码的可执行性。
方向三：代码生成与代码调试的提示工程数据集：探索代码智能化领域的提示工程数据集，在代码生成、代码调试与优化等任务中构建高质量的提示工程数据，提升模型的泛化能力和零样本学习能力。

项目目标

方向一：在上下文关联的代码生成任务中，面对各主流编程语言，函数级生成各语言准确率（Pass@1）相比于提示工程前平均提升 30%， snippet level 代码生成任务上的中各语言平均 BLEU score 达到 35，函数级代码生成任务中各语言平均 BLEU score 达到 30。
方向二：在代码调试与优化任务中，面对各主流编程语言，函数级生成可执行性相比于提示工程前平均提升 30%。
方向三：构建代码智能化 2 个任务中总计不少于 50000 条提示工程数据集，在采集到的数据集中对不同参数的大模型进行提示工程，使得模型的在代码生成、代码调试与优化等任务的零样本生成的表现相比于提示工程前提升至少 30%。在百亿以下模型量级中函数级代码生成 Pass@1 达到 28，Pass@10 达到 45。

子方向 2-代码表征与生成场景的预训练模型微调技术

项目背景

随着预训练大模型技术的发展，越来越多的代码上的大模型如 Codex，GPT4，LLama，PaLM 等被提出。虽然这些大模型在一些代码上的任务取得了良好的零样本推理效果，但是在一些需要特定的安全性场合/私有化模型的情形下，较大参数的模型面临着难以部署/微调的困难。因此，考虑到大模型的推理资源的有限性，以及下游场景对模型迭代的需求，此课题的目的在与如何在有限的硬件上，实现模型有效微调。

研究内容

面向软件工程中代码智能化的多个任务，利用低成本微调技术提升预训练模型在代码表征与生成任务上的表现，包括但不限于如代码补全、用例生成、代码修复、代码翻译等常见下游任务。本项目拟探索通用的黑盒微调、PEFT（parameter efficient fine tune），完成相关下游任务的微调与表现提升。在研究阶段，具体可选的研究方向有：
方向一：黑盒微调框架/技术的探与研究，在给定特定下游场景数据集的情况下，在不更新原始大模型权重的情况下，完成代码表征与生成场景下游任务的指标提升。现利用模型 API 优化若干特定场景下的代码下游任务生成能力，包括但不限于采用 soft prompt，后处理等技术实现一套在特定场景具有广泛普适性的黑盒微调框架/技术。
方向二：PEFT（parameter efficient fine tune）技术的探索与研究，在给定预训练 Checkpoint 的条件下，微调少量参数（相比预训练模型），利用较少的资源实现模型在代码表征与生成任务场景的性能提升。

方向三：通过 finetune/unlikelihood training 等技术，解决预训练模型存在的 degeneration 问题。当模型规模不是特别大的时候，语言模型的“复读“问题会较为明显，虽然有后处理技术可以进行一定程度上的规避，如何微调模型参数解决该问题，是本方向探究的重点

项目目标

方向一：在代码表征与生成下游任务生成场景中，利用黑盒微调达到或超越 FullTuning 模型微调性能。包括但不限于：特定的代码生成任务，代码翻译任务，注释生成任务等；若采用外部模型，参数量应尽可能的小。
方向二：利用 PETF 得到的微调模型与原始的 Full Tuning 任务在代码表征与生成下游任务中，达到或超越 FullTuning 模型微调性能。
方向三：在特定评测数据集上生成结果的质量，在 greedy、beam 等解码策略下，保障生成结果的连贯性，多样性，正确性，要求生成的通过率不低于预训练模型，并明显降低模型的 degeneration 现象。

子方向 3-工具增强的代码大模型后处理技术

项目背景

随着大规模语言模型（LLM）技术的发展，越来越多的大模型具有了处理代码的能力，可以进行代码生成、解释、翻译、问题发现与解决等任务。随着数据量和模型规模的增长，模型的代码生成能力虽然也在提升，但相比于模型在其他自然语言任务上的提升幅度并不显著。即使是目前最强的 GPT-4，在 LeetCode 和 Codeforces 评测中的表现也并不突出。大规模语言模型在编程上的瓶颈可能缘于其顺序的建模方式和自回归的训练方式，这与代码的强结构性和非线性等特性存在不对齐的情况。一方面，单纯基于大模型实现的 AI 编程在实际开发中的效果仍有提升空间，存在跨文件上下文感知不足、长距离依赖遗忘、语义错误和漂移等情况；另一方面，开发者每天都在使用的成熟工具尚未与大模型形成联动。因此，有必要参考 ChatGPT Plugins 的方式，研究如何使用各类编码辅助工具增强代码大模型的代码生成效果。

研究内容

本研究计划将大模型与传统的开发工具、程序分析技术等恰当结合，将专业的部分交给专业的工具来执行，利用现有工具的成熟能力和所提供的精确信息，增强现有大模型的代码生成能力，从而进一步提高代码大模型在实际工业级开发中的实用价值。
具体研究内容为：
首先，通过 IDE/Language Server/程序静态分析等技术或工具自动检测代码生成中的错误：对模型生成的代码进行检查和处理，从而自动发现并尽可能修复所生成代码中存在的语法和语义错误，特别是引用与定义不一致、引用未定义标识符、节点或数据类型错误等典型问题。
随后，结合程序自动修复工具和大模型编辑技术修改所生成代码，从而消除错误：结合现有的成熟 APR 工具，研发和增强类似 Codex-E（Editor）类似具有代码编辑能力的模型，依据错误信息、定位、上下文依赖等信息，对错误涉及的代码进行修改或重新生成（可并行生成多个候选修复方案并从中筛选），从而尽量修复错误，使生成的代码在用户接受后可正常编译和执行。

项目目标

1. 在 Java 语言代码生成场景中，通过后处理自动修复 80%以上的变量/方法/类型/常量引用错误。
2. 在 CoderEval 上，将方法级生成任务的编译通过率提升 50%（相对值），将 Pass@1 指标提升 5~10%（绝对值），将语句级生成的准确率（以编辑相似度衡量）提升 50%（相对值）。

子方向 4-面向可信代码生成的人机协同自然交互编程语言

项目背景

随着大规模语言模型（LLM）技术的发展，越来越多的 AI 具有了辅助开发的能力，可以进行需求分析、知识问答、代码生成、代码解释、错误调试等任务。在需求分析、代码解释、知识问答等以自然语言为主要输出形式的阶段，大模型带来的增益大部分是纯正向的。但是，在以程序语言为主要输出形式的代码生成、错误调试等阶段，由于大模型在确定性、稳定性、精准性等方面的固有缺陷，仍需要人类开发者对模型所生成代码进行深层次的理解、全面的测试、细粒度的定制化修改、系统性的重构等工作，这些工作依然对背景知识、技术水平、精力成本等由较高的要求和投入。因此，如何进一步降低人与 AI 协作进行软件开发的沟通和交互成本，是本课题关注的主要问题。

研究内容

本课题计划研究**一种面向可信代码生成的人机协同自然交互编程语言**，使全民开发者可以以更自然的交互方式与大模型协作生成可信代码，而不要求对现有某种高级编程语言及相关技术有较高的掌握程度。

这种语言定位为一种“甚高级程序语言”，应介于自然语言与编程语言之间，尽量贴近自然语言（从而可以利用模型已有的强大自然语言能力和知识基座），但需要一定程度上克服自然语言的歧义性等问题，并且需要进一步增强其易理解性、交互性、可修改性等方面；另一方面，这种语言的实现途径可以通过自动转换为现有高级编程语言、或直接解释执行，从而保障了代码质量、降低了人工成本。该语言可以是面向特定领域的、纳入领域术语的非通用性语言，其最终目的是为了降低与 AI 协作进行软件开发和演化的门槛和成本，一切设计和取舍都是为了这一目的服务。一方面，该语言不限于文字语言，图形化语言也算一种语言；另一方面，该语言支撑下的人机交互方式也不限于线性思维的对话形式，也可以考虑更高效、更发散、并行化的交互方式。

项目目标

1. 设计此“甚高级程序语言”的语法规则，需尽量满足如下两个性质： a. 相比高级编程语言，该语言更容易让人类开发者进行理解（可理解性高）；b. 支持自动转换为现有高级编程语言、或直接解释执行的途径；c.适合大模型根据用户需求自动化地生成该“甚高级编程语言”描述的代码。从而使用户通过这种语言与 AI 进行交互。
2. 针对某一特定领域，说明该语言的价值、增益、必要性，并给出使用该语言实现一个该领域完整应用的实例。

子方向 5-代码分析与大模型结合的单元测试用例生成和演化技术

项目背景

随着现代软件系统规模与复杂度的持续增加，如何有效保障软件系统正确性日益严峻。现有研究表明，软件系统开发阶段发现缺陷越早，对应付出的测试和维护成本就越低。单元测试是一种在系统开发过程中帮助开发者尽早发现缺陷并降低后期系统维护成本的有效手段。然而，单元测试用例目前严重依赖人工编写和维护，费时费力，导致开发者编写单元测试用例的意愿普遍不高。

为了减少开发者编写和维护单元测试用例的人工成本，研究人员已提出较多基于代码分析技术（如启发式搜索、符号执行、Fuzz 等）来实现测试用例的自动生成。然而， 现有技术仍然存在诸多不足之处，包括：1）自动生成技术执行时间长或在给定时间内生成测试用例覆盖率低（特别是在程序复杂的场景下）；2）生成代码可读性差，不利于开发者理解和维护；3）生成代码风格与已有测试用例代码不一致等。此外，被测代码发生变更后，已有的对应测试用例可能存在断言失效或覆盖率降低等问题。针对此场景需探索单元测试用例的自动化演化技术以持续保证单元测试用例的正确性与稳定性。

近期大模型技术在基于自然语言描述的代码生成任务上已表现出较好的效果，围绕测试场景，也已经有较多学者开始关注布局[1,2,3,4,5,6,7,8,9]。不过基于大模型的技术同样存在不足之处，包括：1）大模型技术是基于概率模型的方法，直觉上难以生成适用于特定业务逻辑的测试数据，进而影响生成的测试用例的覆盖率；2）大模型难以获取程序运行时的信息，生成回归测试预言能力仍然可以存在改进空间；3）大模型生成单元测试用例可能存在可编译和可执行性较差的问题；4）大模型可能无法处理生成测试用例时依赖的大量上下文信息。

综上所述，为了加强围绕大模型技术在单元测试用例场景中的能力孵化，本课题希望聚焦在探索如何结合代码分析和大模型技术各自的优势来生成和演化出更好的单元测试用例。例如针对代码分析方法中生成的测试用例可读性差的问题，可以先通过大模型方法生成一组初始测试用例集，再通过代码分析方法进行扩充。或者针对大模型生成代码可能存在的可编译性问题，通过代码分析技术进行后处理优化。

研究内容

面向 C/C++/Java/Python 等常用的一种或多种编程语言以及常用的一种或多种测试框架（如 gtest，JUnit，pytest 等），综合利用代码分析和大模型技术，探索以下任意一个或多个研究方向：

方向 1：探索并交付代码分析与大模型结合的单元测试用例自动生成技术

该技术的输入为被测代码文件，输出结果为可读性高、可编译性强、可执行性强、覆盖率高的单元测试用例。具体结果目标如下：

1) 支持 C/C++/Java/Python 的一种或多种变成语言和常用的一种或多种测试框架（gtest，JUnit，pytest 等）
2) 单文件（500 行内）生成时间（包括推断和后处理）在 1 分钟以内
3) 在约定的基准集上，相较已有 SOTA 工作，生成的单元测试用例的编译执行率提升 20%
4) 在约定的基准集上，相比已有 SOTA 工作，生成的单元测试用例的分支覆盖率提升 20%

方向 2：探索并交付代码分析与大模型结合的单元测试用例自动演化技术

该技术的输入为被测代码变更和变更前单元测试用例，输出结果为能够适配被测代码变更的单元测试用例。具体结果目标如下：

1) 支持 C/C++/Java/Python 的一种或多种变成语言和常用的一种或多种测试框架（gtest，JUnit，pytest 等）

2) 单文件（500 行内）生成时间（包括推断和后处理）在 1 分钟内以内
3) 在约定的基准集上，相较已有 SOTA 工作，生成的单元测试用例的编译执行率提升 20%

参考文献

[1] Michele Tufano, et al. Unit test case generation with transformers and focal context. *arXiv preprint, arXiv:2009.05617*, 2020.

[2] Elizabeth Dinella, et al. TOGA: a neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 2130-2141, 2022.

[3] Michele Tufano, et al. Methods2Test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 299-303, 2022.

[4] Max Schäfer, et al. Adaptive Test Generation Using a Large Language Model. *arXiv preprint, arXiv:2302.06527*, 2023.

[5] Sungmin Kang, Yoon Juyeon, and Yoo Shin. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *Proceedings of 45th International Conference on Software Engineering*, 2023.

[6] Caroline Lemieux, et al. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *Proceedings of 45th International Conference on Software Engineering*, 2023.

[7] Zhe Liu, et al. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. In *Proceedings of the 44th International Conference on Software Engineering*, 2023.

[8] Baptiste Roziere, et al. Leveraging automated unit tests for unsupervised code translation. In *Proceedings of Tenth International Conference on Learning Representations*, 2022

[9] Bei Chen, et al. Codet: Code generation with generated tests. *arXiv preprint, arXiv:2207.10397*, 2022.

子方向 6-基于大模型的系统集成测试生成和优化技术

项目内容：

在软件研发过程中，系统集成验证是守护产品和服务质量生命线的至关重要环节。在系统集成验证阶段，不仅要端到端地看护被测系统的特性功能，功能交互，还要看护其特性级、系统级性能、可靠可用性、韧性、安全性、兼容性、可服务性等质量属性。因此，往往需要耗费大量测试专家和测试自动化工程师的人力设计和维护测试用例，开发测试自动化脚本，覆盖被测系统白盒代码，灰盒架构元素和黑盒需求、用户场景等，保障测试的充分性，通过执行测试用例发现被测系统缺陷，防止缺陷遗漏到生产环境引起重大现网事故。

随着软件规模和复杂度的增加，需要持续补充、更新、维护的测试文本用例、测试用例脚本等数量也急剧增加。我们期望基于大模型技术， 喂以产品服务领域特定测试上下文语料，构建测试大模型， 探索测试文本用例（承载具体的预置条件、测试步骤、测试数据和预期结果自然语言描述）的生成、检查与补全修订技术，以及测试自动化脚本的生成与补全技术，以及如何快速地进行模型迁移。该课题旨在充分利用大模型的自然语言上下文理解能力和 text-to-code 转化能力，通过人机 pair 交互方式提升测试用例设计效率、用例规范检查及修订补全效率，测试自动化脚本开发和补全效率，同时提升测试用例文本和测试脚本用例的规范性与编写质量。

研究内容

本项目拟探索基于大模型的测试文本用例和测试脚本生成、检查、优化技术，具体可选的研究方向有：

方向一：探索测试文本用例的生成、检查与修订技术。

1) 基于大模型（华为自研的 LLM 或开源语言大模型）结合特定产品领域测试需求/场景/功能描述-测试用例构造指令训练，依据软件需求和场景上下文描述及验证点自动推荐生成测试文本用例。
2) 基于语言大模型（华为自研的 LLM 或开源语言大模型）对存量和新增测试文本用例进行检查，通过结合检查规则 prompt 工程，判断测试用例是否满足精确性、简洁性等规范，是否有步骤和检查点遗漏。进一步，对于不符合规范的文本用例，通过优化元 prompt 模板，引导大模型给出补全步骤，优化步骤描述，以提高测试文本用例的质量，防止测试用例二义性和遗漏引起的测试结果不可重复，不可信。

方向二：探索测试脚本的生成与修订技术。基于大模型（华为自研代码大模型）结合特定产品领域历史测试用例步骤-测试脚本片断对，以及测试方法代码（含方法 signature，方法 body 代码）进行调优

1) 推演场景一：根据测试步骤描述及其上下文自动生成测试脚本片断，包括测试操作方法和测试判定方法组件的调用，参数实例化，判定点的实例化生成
2) 推演场景二：根据测试方法注释或测试方法 signature 生成测试方法组件代码

项目目标

方向一：在测试文本用例生成任务上，采纳率不低于 30%。在测试文本用例检查&补全优化任务上，步骤遗漏检测准确率不低于 70%，补全推荐 top5 采纳率不低于 60%，模糊描述检查准确率不低于 60%，修订优化 top5 采纳率不低于 40%，复杂句检查准确率不低于 60%。

方向二：在测试脚本生成任务上测试方法调用 top3 采纳率不低于 60%。

方向三：构建多任务的指令微调集，测试领域模型训练加速 30%。

子方向 7-基于大模型的研发生产线智能支持技术

项目背景

DevOps 作为一种软件开发方法越来越受到更多关注，其核心理念是通过高度集成的开发和运维团队，实现快速交付、持续改进和高质量的软件产品。使用大模型智能助手来协助开发和运维团队进行项目管理和优化，可以极大地提高工作效率，降低成本，同时保证软件产品的质量。

本项目期望能开发一个智能聊天工具，它能与开发人员定义的 API 进行交互，执行相应的操作，实现辅助用户完成相关任务。

研究内容

本课题的主要研究可选方向一为：① 通过对话方式与用户交互，理解用户意图后生成对应的知识回答进而可以生产相应执行指令，②通过用户输入识别用户需要配置、查询、接入相关的工具所需必选参数和可选参数。③通过 Instruction tuning、RLHF 等技术提升语言模

型对用户意图的理解能力及指令生成的准确度。④使模型具备多轮对话及提取指令必要输入的能力， 从而使智能更加易用，功能更加强大。
可选的方向二：多模态能力支持，目前语料及用户问答有很多多模态的情况比如图文问答，如何准确识别图片中的信息，高效率地理解用户意图是能够回答用户问题的关键 ①图片意图识别：准确识别图片信息和高效理解用户意图对回答问题至关重要。 ②多模态融合：将图像特征与问题文本特征进行融合，构建一个统一的多模态表示，以便模型更好地理解问题的上下文信息。③模型选择与训练：选取模型，对模型进行训练或微调。④答案排序与选择：对答案进行排序并选择最佳答案 ⑤后处理：提供更加准确且自然的回答

项目目标

方向一的项目目标为：
目标 1：zero shot 的问题：对于不同用户不同问法，模型可正确识别意图，提供高泛化能力和适应性模型，同时对于具有错别字、顺序错乱等问题可以进行纠正。
目标 2：模型迭代，数据生成：第一版通过大量人工标注，可以提供高质量的语料和模型，后续知识更新、模型迭代自动生成高质量语料以提升标注速度及标注数量。
目标 3：多轮对话能力：上下文信息提取、理解、一致性回答，信息补全引导问题，例如完成某一功能需要 5 个步骤，可以通过问答识别出来缺失步骤，引导用户进行步骤补全。
目标 4：意图理解及指令生成：语言模型 connect 到实际的操作流 flow 的链路建立及探索，及链路更新迭代模式探索。

方向二的项目目标为：
目标 1：多模态意图理解：通过图片理解用户问题场景，可以理解图片意图，从而实现用户问题的回答
目标 2:多模态融合：将图像特征与问题文本特征进行融合，构建一个统一的多模态表示，以便模型更好地理解问题的上下文信息。
目标 3：模型调优、加速：选取模型，对模型进行训练或微调，并且有相应的处理可以对答案进行排序并选择最佳答案，提供准确且自然的回答

子方向 8-基于大模型的代码精准重构技术

项目背景

软件的迭代更新会对现有的软件代码结构提出新的诉求，解决相关诉求会涉及到代码的优化增强，代码重构是一种重要的代码优化增强手段，经过重构以后的代码，应该达到规模减小，可读性增加，便于后续的代码扩展和维护的目的。但是，当前 IDE（如 Intellij IDEA）仅提供了简单的重构操作辅助选项（如重命名，方法提取等），虽然支持便捷操作降低部分重构工作量，但开发者需要根据自己的知识储备来判断何时何地使用它们。如何自动识别代码中的重构机会，以及对需要重构的代码实施哪些精准重构操作是十分重要。
最新的研究工作提供了重构检测工具 RefDiff[1] 可以用于从 OSS 项目中收集重构样例。研究工作[2]可以识别 Move Method 这种类型的重构时机。

研究内容

本课题拟综合利用代码分析、数据挖掘和机器学习等技术，从代码变更历史数据中挖掘提炼各类重构模式，探索基于大模型的代码精准重构技术，并在某几种特定的代码坏味道及其对应的重构类型上进行验证。具体坏味道类型不限于依恋情结、重复代码、数据泥团、被拒绝的馈赠、发散式变化等。具体可选合作方向有：

方向一：自动识别重构时机，检测程序中是否存在不良设计的代码坏味道，将这些具有坏味道的代码识别为需要进行重构的时机。

方向二：基于已发现的代码重构时机（代码位置），生成对应的精准重构操作推荐给开发者。

项目目标

方向一：重构时机识别

重构时机识别算法的准确率不低于 90%，召回率不做强制要求。针对特定的坏味道类型，从开源工程中挖掘出不小于 50 个有效的坏味道特征。

方向二：生成重构操作

定义一种重构模式描述语言，并开发相应的重构操作执行引擎，能够自动匹配代码上下文并进行代码重构，准确率不低于 80%。

[1] D. Silva, J. Silva, G. Santos, R. Terra, M. Valente, “RefDiff 2.0: A Multi-Language Refactoring Detection Tool,” IEEE Trans. Software Eng, vol. 47, pp. 2786-2802, 2021.

[2] D. Cui, S. Wang, Y. Luo, X. Li, J. Dai, L. Wang, Q. Li, “RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code,” ICSME 281-292, 2022.

课题二：全链路压测演练场景下智能辅助性能工程

项目背景

万物互联的时代，软件的规模化、复杂化程度不断提升，现网性能问题愈发严峻，尤其遇到大促、抢红包、抢票、抢课等特殊业务场景，由于预估不足，流量冲击大导致服务中断的例子屡见不鲜，一旦故障发生，问题定位需协同服务上下游多方专家团队搜集大量日志共同诊断问题，寻找根因，问题定位效率低下。因此加强预发态和生产环境常态化全链路压测演练，借助压测演练辅助实现重大事件提前容量预估、检验告警及时性、问题定位及恢复影响时长、版本变更后性能 benchmark 对比，性能调优等性能工程，成为业界趋势。但是，当前在性能压测演练过程中，虽然已经能实现性能模型的设计和自动下发，但针对异常告警的性能瓶颈定位、根因分析、基线对比、及性能调优等大多依赖人工收集各指标 metrics 数据、trace 数据、log 数据、和服务系统配置数据等根据经验对比分析，分析效率低下，由于配置参数多，调优效率更是低下。

研究内容

本项目旨在研究一种轻量级的技术，用于获取代码与测试用例之间的关联关系。具体研究方向：

1. 性能基线对比和异变分析

1）版本间或版本与基线间单指标性能恶化/增益分析：含应用业务 KPI(throughput, 响应时长等)和资源类指标（CPU、内存占用率等）

2）多指标关联关系分析及跨版本关联异变分析：包括指标间绝对值关联和异变分析和指标变化趋势关联和异变分析

3）异变根因推断：依据性能测试执行机日志、调用链日志、产品日志等分析异变的根因

2. 性能瓶颈辅助诊断

压测过程中，及时检测异常点，结合多态监控数据定界到具体服务组件、定位到具体事件类型。

3. 自适性容量预估

探索实现自适性梯度 load 模型，基于 KPI 时序数据的准实时局部稳态检测和过载极限检测，达成

1）爬坡等待时间自适应

2）爬坡增压梯度自适应

实现自适应无人值守性能摸高即极限容量预估。

4. 探索性能调优

1）建立服务系统配置参数与性能指标的关联，识别敏感参数

2）敏感参数多指标目标综合调优

项目目标

1. 性能回归恶化检测：对比两个版本在相同负载模型下性能指标时序数据，自动检测单指标恶化、指标间关联关系异变，检查准确率挑战 85%；对比两个版本在不同负载模型下性能指标时序数据，自动检测单指标恶化、指标间关联关系异变，检查准确率挑战 60%。
2. 性能异常诊断：基于 KPI 指标、Trace 调用链和服务日志关联分析实现性能指标异常检测和性能瓶颈智能诊断，异常检测准确率 80%，根因事件推断（top5）准确率 70%
3. 自适应容量预估模型：实现性能容量自适应摸高爬坡模型，性能摸高耗时耗资源相对于人工配置固定爬坡等待时间减少 30%以上。

课题三：基于 eBPF 的业务透明的应用观测技术

项目背景

随着云原生时代到来，分布式系统得到了广泛的应用，在分布式系统中，故障排查和性能优化是非常重要的任务，而分布式全链路追踪是解决这些问题的关键手段。目前分布式全链路追踪已经发展成为一种成熟的技术，并得到广泛应用，目前主流的追踪技术/系统包括 OpenTracing、OpenTelemetry 和 SkyWalking 等，然而现有技术仍然存在一些不足，例如对源码有侵入性、无法实现跨语言、系统开销较大等问题，如何实现透明的调用链跟踪技术成为新的研究课题。

研究内容

针对云原生场景下的分布式系统，提供全栈、全链路、高性能的应用性能观测能力，需要主要解决两个问题：

1) 传统分布式 Tracing 依赖 SDK、字节码注入、侵入式修改埋点等手段，存在**难维护、绑定语言、全栈跟踪能力不足**等问题；
2) 全栈观测云原生应用的每个 TCP 链接、Request 访问时，数据是由分布式部署的 Agent 产生，**数据来源多样化，易产生数据孤岛、全栈观测的数据相关性不足**；

针对上述两个问题，本项目拟探索无侵入、跨语言、低开销的大规模分布式系统 Tracing 技术，当前可探索的技术路线是利用 eBPF 技术实现服务内部和外部完全透明的调用链追踪。具体合作方向有：

方向一：服务内外部透明的大规模应用分布式跟踪；

方向二：端侧业务透明的多模态数据采集与融合技术；

项目目标

方向一：服务内外部透明的大规模应用分布式跟踪

1) 业务无侵入，支持跨语言，低开销且支持高频采样；
2) 支持线程池、异步等复杂场景，且协议无关；
3) 支持面向大规模分布式系统的请求全链路跟踪；

方向二：端侧业务透明的多模态数据采集与融合技术；

1) 实现用户透明的日志、调用链、指标等运维数据的生成、采集和分析，在内核侧利用 eBPF 形成统一的可观测性解决方案；
2) 在采集的同时通过无业务侵入的方式实现日志、调用链、指标等运维数据的关联与融合，辅助根因定位；

课题四：未知开源漏洞的精准发现与验证

项目背景

根据 Sonatype 发布的《2022 年软件供应链状况报告》，2022 年针对开源存储库的已知攻击同比增长了 633%，自 2019 年以来，平均年增长高达 742%。另外，据统计 2022 年漏洞 CVE 新公开的漏洞数量为 33034 个，达到新高。漏洞是软件供应链安全中关注的首要风险，因此如何高效的发现、分析、验证漏洞已经成为了业界的研究热点。

围绕漏洞发现目前已经有很多的研究，研究者结合静态分析、动态分析、深度学习等多种技术在特定数据集上已经取得了很好的效果，但在实际的应用中，当前的算法仍然存在误报偏高的问题。针对漏洞分析以及验证，目前业界主要还是依赖专家经验，需要耗费大量的时间。针对这种现状，本课题希望探索一种更精准的漏洞发现算法，降低漏洞发现的误报，并能对漏洞分析及验证进行自动生成，提升漏洞的分析与验证的效率。

研究内容

基于海量的开源社区代码、漏洞分析语料、POC 样例等数据，结合静态分析、Fuzzing、大模型等多种技术，构建开源漏洞挖掘算法，实现更高效的未知开源漏洞挖掘；智能生成漏洞的分析报告以及漏洞的验证程序，辅助安全专家进行分析。

具体可选合作方向有：

方向一：未知开源漏洞的精准发现

方向二：漏洞分析报告的自动生成

方向三：漏洞验证程序的自动生成

项目目标

方向一：未知开源漏洞的精准发现

1. 在约定的基准集上，准确率>0.7，召回率>0.7
2. 在实际的项目中发现 20+个未知漏洞

方向二：漏洞分析报告的自动生成

1. 基于漏洞函数的上下文，准确的预测漏洞的类型、漏洞的利用原理等漏洞关键信息，生成漏洞分析报告
2. 支持 token 级别的漏洞片段解释，辅助程序员对漏洞进行理解

方向三：漏洞验证程序的自动生成

1. 基于漏洞的分析报告、漏洞代码的上下文等信息，端到端的自动生成漏洞验证程序
2. 生成的漏洞验证程序能触发漏洞点，进行自动验证

课题五：基于流量的 API 安全分析与治理

项目背景

在数据分析时代，API 已经成为众多企业技术和经济模式的竞争力来源，API 不仅适用于云服务、物联网和大等数字化场景，还可以帮助企业挖掘并维护客户、构筑全新的业务生态。随着 API 巨大价值的体现，API 攻击也呈现出指数级的增长趋势。

Garter 报告指出：明确地将 API 安全确立为保证 PaaS 安全的一个重要类别，将 API 安全安全范畴独立于 WAF 和 API 网关。API 将成为网络攻击者利用最频繁的载体，而通过攻击 API 可以非授权使用企业的应用数据。仅通过一般应用安全解决方案保护 API 已被证明是无效的，每一个 API 都是一个全新的可能攻入系统的攻击向量。典型的 API 面临的问题挑战如下。

大规模 API 接口难以管控：Gartner22 年报告预测，到 2025 年，由于 API 的爆炸式增长超过了 API 管理工具的功能，将管理不到 50%的企业 API。到 2025 年，应用程序中使用的第三方 API 的百分比平均将从 2021 年的不到 10%上升到 30%，这将使依赖关系管理复杂化。API 安全问题十分突出。Akamai 的报告指出“API 请求已占所有应用请求的 83%，预计 2024 年 API 请求命中数将达到 42 万亿次”。

授权未最小化导致敏感数据泄露：OWASP TOP1/API Security TOP 1 问题均为访问控制不当或授权不当导致敏感数据泄露。

研究内容

本议题的核心是 API 安全问题发现与 API 安全问题治理。

整个议题是围绕 API 安全，输入可以是 SAAS 服务流量，通过针对流量的分析识别各类 API 安全问题。基于流量识别 API 较为成熟，仅仅是识别的第一步，后续基于各类模型、调用关系、行为、画像等做进一步分析，识别出 API 的各类安全问题、API 攻击流量等，并基于分析进行告警或阻断。

针对 API 安全问题类型的说明：API 安全问题类型有很多，具体可参考 OWASP API Security TOP 10，模型和行为特点也是不一样的，此处不限定安全问题的具体类型，也可以给更多老师参与该议题的空间。

项目目标

识别流量中的 API：基于 SaaS 服务的流量进行分析，识别出流量中的 API，识别和消除影子/僵尸 API，解决大规模 API 误用、难以管控问题。

API 安全治理：基于 SaaS 服务的流量和典型行为等进行分析，识别出敏感信息防止通过 API 交互的敏感数据泄露问题、API 调用链中的非法调用关系、API 各类安全威胁、识别并拦截攻击流量、识别 API 越权、令牌过度，授权等安全问题。也可专注识别 API 某类特定安全问题并治理。

参考：

https://salt.security/

https://nonamesecurity.com/

https://github.com/OWASP/API-Security/tree/master/2023/en/src

课题六：基于源码分析的恶意代码识别

项目背景

随着针对源代码攻击呈几何级增长，恶意代码注入成为了互联网安全的一个主要威胁。一些代码会在为用户服务过程中会从事一些可疑的危险行为，如：预留特殊口令或密钥，擅自连接网络等后门行为，监控屏幕、键盘，隐藏进程等敏感行为，这些行为可能破坏计算机系统可用性，为用户带来了一系列损失和风险，更使得企业生产、传播和信息安全受到严峻挑战。在案例调查中代码提交阶段注入恶意代码的投毒行为最为普遍研究发现；建立代码审查及人员审核机制可以很大程度地降低风险[1]。

当前规则扫描现状：人工定义恶意规则模式，检测性能较好，但未深入到代码分析层面，存在着显著的检测精度不理想问题。

当前恶意行为分析现状：NDSS2021 文章《Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages》介绍了对解释语言软件包的恶意行为检测方法，成功识别 300+恶意软件包。另外，部分研究分析不同语言单文件的恶意行为，可一定程度提升规则扫描的检测精度。

研究内容

基于静态行为分析的恶意源码的高效精准识别：现有以二进制为主的病毒扫描引擎无法很好检测源码上的恶意行为，尽管相关研究逐渐变多，但由于语言差异导致检测技术和效果各不相同。单文件源码和全量源码的静态行为分析存在着漏报或效率低下问题，无法支撑数量众多的软件的安全扫描需求。

项目目标

方向一：技术通用性

洞察各语言源码的恶意行为全貌，提供多语言通用的检测技术，支持多种恶意行为模式检测，包括但不限于后门、信息泄露、反弹 shell、加密恶意代码等。

方向二：检测性能

支持恶意行为相关的部分文件自动筛选分析，使支持跨文件的恶意行为检测特性，又提升代码分析效率。

[1] Zimmermann M, Staicu C A, Tenny C, et al. Small world with high risks: A study of security threats in the npm ecosystem[C]//28th USENIX Security Symposium (USENIX Security 19). 2019: 995-1010.

课题七：Fuzzing 测试中利用机器学习技术增强深层代码覆盖能力

项目背景

Fuzzing 测试技术已被广泛应用于开发者测试、安全测试等多个安全防护阶段。然而，由于其轻量级的策略与随机性，往往只能覆盖被测程序的浅层代码与状态，难以稳定测试深层代码与状态，从而导致其缺乏充分性；一些白盒 Fuzzing 测试技术借助污点追踪、符号执行等策略定向探索深层代码，但其开销较大，且在实际测试中面临状态爆炸、控制流不完整等问题，难以实际应用

近年来，随着机器学习的快速发展与效果提升，借助机器学习技术增强 Fuzzing 测试技术覆盖深层代码的能力，逐渐受到学术界及工业界的关注，并出现了一些尝试。然而，具体如何使用机器学习，从本质上解决这一问题，目前仍不明确。

目前已有一些技术如 FuzzGuard、MEUZZ、DeepFuzz 等技术借助机器学习技术优化输入生成与选择，但对 Fuzzing 测试效果提升仍有限，且未定向针对难以覆盖的深层代码进行技术优化。

研究内容

如何使用机器学习技术增强/替代 Fuzzing 测试中的程序分析：针对 Fuzzing 测试中难以覆盖的深层代码特性，如何借助机器学习技术代替难以实际应用的程序分析技术，提供 fuzzing 测试覆盖这些代码所需的信息，或辅助程序分析技术（提升其效果与效率）提供这些信息。

如何使用机器学习技术增强 Fuzzing 测试中输入的生成与变异：如何使用机器学习技术优化初始种子输入生成、种子输入变异（包括输入选择、字节选择、变异方式选择等），加速 Fuzzing 测试技术覆盖深层代码。

项目目标

方向一：理论研究方法

探索利用机器学习技术增强深层代码覆盖能力，提出一种或多种可行的方法，讨论其可行性与优劣。

方向二：实际应用

依据理论研究的成果，基于公司常见业务场景研发原型系统 ，保证常见业务场景下 Fuzzing 测试中，由高效覆盖深层代码带来的高代码覆盖率与缺陷检出率。

[1] https://anchore.com/software-supply-chain-security-report-2022/

[2] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17).

课题八： 污点分析误报漏报场景优化

项目背景

污点分析在华为公司代码质量看护场景中被广泛使用。当前污点分析存在严重的误报/漏报问题，针对华为公司高度关注的三种典型污点分析场景有先进方法的诉求，包括 (1) 不同抽象层次的嵌套体内存建模对污点分析结果影响巨大，全域抽象精度过低，单变量传播开销过高，需要更合理的内存建模方式；(2) 缓冲区大小敏感和 (3) 路径约束敏感。

研究内容

嵌套体内存建模：用户原始标记的结构体成员污点，如何在跨函数数据流传递过程中，发生多次结构体类型转换、偏移后（含转成 char *、void *、或其他类型的结构体等），依然能够准确还原出原始被标记的污点，如 Case 1 与 Case 4。

缓冲区大小计算：类型转换/循环等代码语句严重影响缓冲区大小的精确计算，如 Case 2。

路径约束计算：计算路径约束依然缺乏高效、通用的解决方案，如 Case 3。

Case 1：类型转换导致的缓冲区大小难以精确计算：acDate是长度为4的char型数组，而实际当作变长数组使用

```
1  uint32_t Case_1(Handle *handle, char**msgBuf, uint32_t *msgLen)
2  {
3      uint32_t bufLen = CacuLen(handle->inputLen, handle->outputLen);
4      char *buffer = (char *)malloc(bufLen);
5      if(buffer == NULL)
6          retunt -1;
7
8      Head *msgHead = (Head *)buffer;
9      msgHead->uiId = handle->id;
10     msgHead->flag = handle->flag;
11     ......
12
13     uint32_t ret = MSG_Input(msgHead, handle);
14     ......
15 }
16
17 uint32_t MSG_Input(Head *msgHead, Handle *handle){
18     ......
19     rc = memcpy_s(msgHead->date, 8, &data, 8);
20     ......
21 }
```

分配一大块内存给buffer，其中一部分预留给date[4]使用

预留的内存足够给date[4]使用，实际使用不会发生溢出

```
23 typedef struct{
24     uint32_t uiId;
25     uint32_t flag;
26     ...
27     ... 缓冲区定义大小4字节
28     char date[4];
29 }Head;
```

告警点位于第19行， Buffer mismatch Size, arg 0 is Buffer,length is 4,arg 1 is Size,value is 8

Case 2：循环导致的缓冲区大小难以精确计算

```
1  uint32_t Case_2(const Info1* pInfo, Info2* outInfo){
2      struct {
3          char *dest;
4          size_t destSize;
5          const char *src;
6      } map[] = {
7          {outInfo->id, sizeof(outInfo->id), pInfo->id },
8          {outInfo->name, sizeof(outInfo->name), pInfo->name },
9          {outInfo->id2, sizeof(outInfo->id2), pInfo->id2 },
10     };
11
12     for(uint32_t i=0; i < (sizeof(map)/sizeof(map[0])); i++){
13         int rc = strcpy_s(map[i].dest, map[i].destSize, map[i].src);
14
15         if(rc != 0){
16             return 1;
17         }
18     }
19
20     return 0;
21 }
```

结构体定义

结构体初始化

dest与destSize一一对应，不存在溢出风险

告警点位于第13行， Buffer mismatch Size,arg 0 is Buffer,length is 12,arg 1 is Size,value is 28

Case 3：路径约束不敏感导致的误报问题

```
1  void Case_3(){
2      ......
3      uint32_t msgSize = headSize + sizeof(MsgHead);
4
5      if((sizeof(buffer) < msgSize) || (sizeof(buffer) - msgSize < dataLen)){
6          return;
7      }
8
9      MsgHead *msgHead = (MsgHead *)(buffer + headSize);
10
11     if(dataLen > 0){
12         int rc = memcpy_s(buffer + msgSize, sizeof(buffer) - msgSize, data, dataLen);
13         if(rc != 0)
14             return;
15     }
16     ......
17 }
```

条件检查保证msgSize <= sizeof(buffer)

条件检查保证msgSize < sizeof(buffer)

经过上面判断，可以保证buffer + msgSize不溢出

告警点位于第12行， Index value maybe overrun, array() elements is 1500, index (ctrlMsgSize) maybe 1500 (0x5DC)

误报原因：由于ctrlMsgSize < sizeof(buffer), 因此ctrlMsgSize的值小于1500，因此不存在指针偏移越界风险

Case 4：类型转换导致的污点标记丢失

```
01、typedef struct tagPacket{
02、    unsigned char* data;
03、    unsigned short length; // taint
04、    unsigned short size;
05、}Packet;
06、typedef struct tagMSG{
07、    unsigned char ucType;
08、    unsigned char ucPort;
09、    Packet* packet;
10、    unsigned char ucNum;
11、    unsigned char ucState;
12、}MSG;
```

```
14、void packet_proc(char *p) {
15、    Packet *pkt;
16、    pkt = (Packet *)p; // 类型还原
18、    char dest[100];
19、    memcpy(dest, pkt->data, pkt->length) // 如何知道pkt->length是污点数据?
20、}
21、// 入口点: entry_1
22、// 污  点: pMsg->packet->length
23、// 污点标记语法：0.2.1，表示第1个参数的第3个成员的第2个成员是污点
24、void entry_1(MSG *pMsg) {
25、    char *pkt = (char *)pMsg->packet; // 类型转换，污点标记丢失；也有可能时转换为void*
26、    packet_proc(pkt);
```

项目目标

方向一：

功能：基于 LLVM IR 分析，能够精确地计算缓冲区大小与路径约束(比范围分析精度提高 50%)

性能：方案在实际项目中不产生性能瓶颈(overhead 不高于 50%)。

方向二：

功能：基于 LLVM IR 分析，静态分析中嵌套结构体的内存建模，精度相较于最粗抽象粒度(全域污染)提升 50%；

性能：污点还原方案在实际项目中不产生性能瓶颈(overhead 不高于 50%)。

课题九：C、C++二进制代码的成分分析技术

项目背景

现在越来越多的开发者会利用开源的第三方组件或者源码信息去构建属于自己的软件制品。随着产品功能越来越多样化和丰富，第三方组件的成分比例也越来越大。复用开源组件的好处就是能够加快产品开发，防止重复造轮子。开源第三方组件也成为了软件供应链中最重要的一环。然而，广泛的使用开源组件也是一把双刃剑，复用的这些开源组件制品当中一旦含有一些高危漏洞，其影响范围也会成倍增加。现今，第三方的组件安全问题已经成为了 top3 的安全威胁了[1]。

C/C++作为一个通用程序设计语言，已经被广泛地使用了很多年。像很多重要的操作系统的内核都是用 C/C++语言来实现的。但是 C/C++不像一些高级的程序语言，例如 Java，JavaScript，Python，GO 有自己的包管理器，很容易可以识别出程序依赖的成分信息。C/C++的复用相比这些语言要复杂的多，即可随意的使用不同的源码构建自己的软件，也可以直接调用一些编译好的可执行二进制文件。多样化的调用粒度和缺乏包管理机制使得 C/C++的二进制成分分析增加了难度。此外，二进制码是机器相关的，不像 Java 编译好的文件是 bytecode，这种特殊的“binary code”是机器无关的，加上 C/C++语言在编译过程中有众多的配置选项，不同的配置选项可以使得相同的源码编译之后以及反编译之后的二进制千差万别，诸如此类的原因使得 C/C++二进制成分分析变得难上加难。

目前已有一些技术如 OSSPoLice[2]，LibDX[3]等技术都是提取了字符串常量和一些操作码去识别 C/C++的二进制代码的成分信息，但是这些工具的效果都不是很好。我们需要一个精确的 C/C++二进制成分分析工具来帮助我们识别成分信息和漏洞检测等下游任务。

研究内容

如何使用软件分析技术精确地识别一个二进制可执行的 C/C++制品中的开源成分信息：在 C、C++中一个微小的源码改变都会导致编译出来的可执行二进制软件和之前的二进制发生较大的差异。即使完全相同的 C/C++源码，由于 C/C++的编译配置选项很丰富，采用不同的编译器，而且这类编程语言是机器相关的，所以生成的二进制可执行文件的差异也是非常的大。现有的很多 C/C++二进制制品都可以被加固或者是混淆，也加剧了精准软件分析的难度。现有的反编译工具也只能部分反编译一部分二进制代码。

项目目标

方向一：成分识别

黑盒的二进制可执行的 C/C++制品软件，能够精确地识别出里面的开源组件的成分信息。

方向二：漏洞识别

构建一个完善的 C/C++漏洞数据库，里面有详细的漏洞补丁信息，漏洞代码片段信息，可以判别这个开源组件的漏洞是不是真的影响这个待检测软件，或者某个软件成品是否已经还有漏洞的补丁代码，确认这个软件已经被修复了。

方向三：合规分析

分析到文件级别的软件许可证的使用情况。

[1] https://anchore.com/software-supply-chain-security-report-2022/

[2] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17).

[3] W. Tang, P. Luo, J. Fu and D. Zhang, "LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code," *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada, 2020, pp. 104-115, doi: 10.1109/SANER48275.2020.9054845.