大模型赋能软件供应链开发环节安全研究综述

刘井强[1,2], 田 星[1,2], 舒钰淇[1,2], 朱小溪[1], 刘玉岭[1,2], 刘奇旭[1,2]

[1]中国科学院信息工程研究所 北京 中国 100085
[2]中国科学院大学网络空间安全学院 北京 中国 100049

摘要 随着信息技术的快速发展, 软件的模块化与产业化趋势愈加显著, 导致软件构建的复杂性持续攀升, 从而暴露出更多的攻击面, 引发了多起软件供应链攻击事件。软件供应链安全不仅具有攻击门槛低、攻击方式多样、攻击隐蔽性强等特点, 而且能够影响软件供应链下游安全, 显著扩大了攻击范围, 成为业界广泛关注的焦点。首先, 本文介绍了软件供应链安全的背景, 以大模型及软件供应链安全的相关概念为出发点, 描述了软件供应链安全防护的发展历程。接着, 本文着重探讨了大模型在软件开发环节供应链安全防护中的应用研究, 通过系统梳理和分析现有研究成果, 分别从顶级源、依赖项、软件包构件及其构建过程四个维度, 介绍了大模型赋能软件供应链安全防护技术的研究现状。在此基础上, 本文通过对比传统软件供应链安全防护的技术与方法, 重点分析了大模型赋能软件供应链开发环节安全方面的优势和机遇。最后, 结合对当前研究现状的调研分析, 本文总结了大模型在软件供应链安全防护技术中面临的数据集构建、模型训练微调、模型稳定性以及引入新的供应链安全等问题, 并据此提出了未来可能的研究方向, 以期为推动该领域的持续发展提供有益的参考和启示。

关键词 大模型; 软件供应链安全; 软件开发; 软件安全
中图法分类号 TP393.0 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2024.09.11

A Review of Security Research in the Development Stage of Software Supply Chain Enhanced by Large Models

LIU Jingqiang[1,2], TIAN Xing[1,2], SHU Yuqi[1,2], ZHU Xiaoxi[1], LIU Yuling[1,2], LIU Qixu[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract With the rapid development of information technology, the modularization and industrialization trends of software have become increasingly prominent, leading to a continuous increase in the complexity of software construction process, thereby exposing more attack surfaces and triggering multiple software supply chain attack events. The security of software supply chain not only features low thresholds of attacks, diverse attack methods, and strong concealment of attacks, but also can affect the security of the downstream of the software supply chain, significantly expanding the scope of attacks, thus becoming a focus of widespread concern in the industry. Consequently, the issue of software supply chain security has become a focal point of attention in the industry. Firstly, this paper introduces the background of software supply chain security, taking the related concepts of large models and software supply chain security as the starting point, and describes the development process of software supply chain security protection. Then, this paper focuses on the application research of large models in the security protection of the software development supply chain. Through systematically reviewing and analyzing existing research results, it introduces the current research status of large models enabling software supply chain security protection technology from four dimensions: top-level sources, dependencies, software package artifacts, and their construction process. On this basis, this article focuses on analyzing the advantages and opportunities of leveraging large models to enhance the security of software supply chain development processes, by comparing it with the techniques and methods of traditional software supply chain security protection. Finally, based on the investigation and analysis of the current research status, this paper summarizes the challenges faced by large models in software supply chain security protection technologies, including dataset construction, model training and fine-tuning, model stability, and the introduction of new supply chain security issues, and proposes possible future research directions accordingly, aiming to provide useful references and inspirations for promoting the continuous development in this field.

Key words large language model; software supply chain security; software development; software security

通讯作者: 刘奇旭, 博士, 研究员, Email: liuqixu@iie.ac.cn。
本课题得到中国科学院青年创新促进会; 中国科学院网络测评技术重点实验室; 网络安全防护技术北京市重点实验室项目资助; 国家电网有限公司科技项目资助(No. 5700-202352606A-3-2-ZN)。
收稿日期: 2024-03-30; 修改日期: 2024-05-21; 定稿日期: 2024-07-25

1 引言

近年来, 随着软件供应链的复杂性和多样化不断增加, 软件供应链安全的风险日益突出。这些风险不仅来源于开发、交付、运行等环节中引入的安全漏洞和受到的攻击, 还与开源软件的广泛使用及其多方依赖关系有关。软件供应链中的任何一个环节受到攻击, 都可能对整个供应链的安全性和完整性造成严重影响。在此背景下, 软件供应链安全已成为网络空间攻防对抗的焦点, 备受各国政府和企业的高度关注。现阶段, 大语言模型(Large Language Model, LLM)成为推动技术革新的关键力量, 被广泛应用于自然语言处理、计算机视觉、语音识别等多个领域, 对各行各业产生了深远影响。利用大模型高效的数据处理能力, 安全人员可以对软件供应链的各个环节进行更加深入的检测和分析, 识别和响应安全威胁。大模型与软件供应链安全的结合, 为提升软件开发效率和安全能力提供了新的可能性。

在软件开发早期, 敏捷开发模式以最快的速度将代码从集成开发环境(Integrated Development Environment, IDE)或代码存储库带到生产环境, 显著提升了开发效率。然而, 当开发人员将恶意代码植入公开访问的代码库时, 使用这些开源组件的软件将面临安全问题。Synk 最新发布的报告[1]指出, 每个应用程序中未解决的关键漏洞平均数量为 5.1 个, 修复单个漏洞的平均天数是 97.8 天。开发人员频繁地重用代码, 加剧了软件对第三方组件的依赖, 导致第三方组件的安全风险也随之传递至当前软件中。根据 Sonatype 发布的报告[2], 开发人员每月会下载 12 亿易受攻击的依赖组件, 每 7 个包含漏洞的项目中大约有 6 个受到可传递依赖的影响。OSSRA 2023 的报告[3]显示, 91%的代码库包含两年内仍未更新的组件。此外, 通过对软件供应链上的薄弱环节发起攻击, 攻击者能够轻松访问上下游业务系统, 更容易扩大攻击范围。据 Gartner 的分析[4]预测, “2025 年全球 45%的组织将遭受软件供应链攻击, 比 2021 年增加三倍”, 因此亟需国家、企业、组织各个层面开展软件供应链安全防护技术研究, 提升软件供应链在安全风险发现和分析处置方面的能力。

近年来, 人工智能(Artificial Intelligence, AI)技术在网络安全领域表现出了极大的优势。2023 年 8 月, 美国白宫发起了一项为期两年的人工智能竞赛(AI Cyber Challenge, AIxCC)[5], 该竞赛由国防部高级研究计划局(Defense Advanced Research Projects Agency, DARPA)主导, 与谷歌、微软、OpenAI 等公司合作, 要求参赛者充分利用 AI 识别和修复软件漏洞, 旨在通过 AI 解决网络安全问题。特别是生成式预训练 Transformer(Generative Pre-trained Transformer, GPT)模型、Meta 大语言模型(Large Language Model Meta AI, Llama)、Claude 大模型等的出现, 为解决现有的软件安全问题提供了新的思路。得益于对复杂数据的强大分析能力, 大模型在代码生成、语义理解、代码审计、漏洞检测等方面表现出了巨大的潜力, Devin AI、Copilot、IBM Watsonx, Amazon CodeWhisperer、SWE-Agent 等基于 LLM 的智能代码助理[6-7], 有望转变现有软件的开发模式。这些大模型本质上是基于 Transformer 架构的语言模型, 通过给定的词序列, 从上下文中学习丰富的语义特征, 在处理复杂任务时, 展现出了惊人的理解和表达能力, 并且在自动化标注方面降低了基于人类反馈的强化学习(Reinforcement Learning with Human Feedback, RLHF)成本。在软件供应链安全防护方面, 大模型能够从软件和依赖关系中提取深层次的关键特征, 具备自动化代码生成、漏洞检测和修复的能力, 从而帮助安全团队快速发现和响应潜在的安全风险。通过调研近十年来软件供应链安全领域的相关论文, 国内外研究人员对软件供应链安全防护研究进行了积极探索。本文基于软件供应链构件等级(Supply Chain Levels for Software Artifacts, SLSA)框架[8]中提出的顶级源、依赖项、软件包构件及其软件构建过程面临的安全问题, 对涉及的安全防护技术进行了文献调研。

为了便于研究人员对大模型在软件供应链安全防护方面进一步开展研究, 本文将综述软件开发环节中, 大模型赋能供应链安全防护的研究现状。现阶段, 大模型在解决软件供应链安全问题方面总体上处于起步阶段, 但已逐渐开始发挥作用, 例如, 在顶级源漏洞检测、模糊测试, 依赖项测试用例生成, 软件包二进制语义理解、自动程序修复等具体任务中的表现效果已经优于传统机制中的最先进方法(State of the Art, SOTA)。与此同时, 在软件包恶意代码分析、构建过程代码安全防护等方面与传统机制相结合的方法, 也取得了较理想效果。此外, 大模型在代码分析效率、长文本上下文语义信息获取等方面还处于探索阶段, 并且模型训练代价高, 存在幻觉等不稳定因素, 目前仍是传统机制所无法取代的。因此, 本文所述的大模型赋能技术方法并非完全优于传统机制, 而是与传统机制的互为补充。本文通过总结归纳目前已有研究工作的侧重点及存在的问题, 据此阐述该领域所面临的挑战, 旨在为大模型在软件供

应链安全防护中的赋能技术研究提供有价值的参考, 并提出未来可能的研究方向。

本文主要有三个方面的贡献:

1)本文调研了近年来大模型在软件供应链安全防护方面的工作, 深入分析了这些研究工作提出的基于大模型的安全检测技术及其方法, 对本领域的发展进程进行了全面的梳理;

2)本文从软件供应链自身的特性出发, 围绕软件供应链安全的主要威胁要素, 从面向顶级源的代码检测技术, 面向依赖项的威胁分析技术, 面向构建过程的防护机制, 面向软件包的安全检测技术 4 个角度对当前的研究工作进行了分类研究;

3)通过对软件供应链安全防护工作的调研总结, 本文对当前的工作进展进行了深入分析, 总结了大模型技术目前仍存在的不足和面临的挑战, 并展望了未来大模型赋能软件供应链安全防护的研究方向。

2 大模型及软件供应链安全的相关概念

2.1 大模型简介

2.1.1 通用领域大模型

GPT 模型[9]由 OpenAI 于 2018 年首次提出, 旨在解决通用自然语言处理领域的一个关键问题: 如何生成自然逼真的文本。GPT 模型的设计基于 Transformer 模型, 与传统的循环神经网络(Recurrent Neural Network, RNN)模型[10]不同, Transformer 模型使用了自注意力机制, 可以更好地处理长序列和并行计算, 因此具有更好的效率和性能。AlpacaEval[11]是斯坦福发布的 LLMs 排行榜, 从多样性、一致性、相关性等方面来测试模型遵循用户指令的能力, 从而更全面地评估大模型在实际应用场景中的表现。在通用领域大模型中, GPT-4、Contextual、Yi、Claude 等大模型位居 AlpacaEval 排行榜前列。本文根据 2024 年 3 月 AlpacaEval 2.0 排行榜, 从已验证的不同系列大模型中选取了该系列中排名靠前的 8 款模型, 从厂商、参数量、上下文长度、多模态的支持情况、开放性等维度进行简介, 如表 1 所示。

2.1.2 安全领域大模型

安全领域大模型是针对安全垂直领域开发的大语言模型, 借助海量的专业安全知识进行训练, 使其具备处理海量安全数据和执行安全领域特定任务的能力, 对保护企业和个人的信息安全、提升网络安全防护效率具有重要意义。由于网络安全领域场景复杂, 在该领域发布的大模型相对较少。已知的网络安全领域大模型, 如 SecGPT[12]、360 安全大模型[13]、Sangfor 安全 GPT[14]等, 以通用领域大模型为基座, 通过学习安全领域数据集, 进行二次预训练和微调, 在掌握通识知识的基础上领会专业领域的知识。然而, 这些大模型在网络安全领域的应用能力参差不齐, 难以进行有效评估。为此, 业内提出了首个用于网络安全大模型能力评测的平台 SecBench[15], 重点从能力、语言、领域、安全证书考试四个维度进行评估, 为安全大模型研发和学术研究提供模型选型参考。现阶段, 大模型应用于安全领域, 更多的是在网络安全的特定场景中, 对一些关键技术的尝试和探索。利用大模型强大的知识融合和内在特征的关联记忆能力, 安全研究人员对大模型在安全垂直领域中的应用进行了研究, 辅助提升安全防护的效率和准确性。

2.2 软件供应链安全的相关定义

在研究和定义软件供应链安全的相关定义前, 首先需要明确软件供应链的定义。目前, 学术界和工业界对软件供应链已经提出了一些定义, 何熙巽等人将软件供应链定义为通过一级或多级软件设计、开发阶段编写软件, 并通过软件交付渠道, 将软件从软件供应商送往软件用户的系统[16], 并从解决软件供应链安全问题的技术和管理手段出发综述了软件供应链安全的研究现状。纪守领等人从开源软件的角度对软件供应链的安全问题进行了综述, 将开源软件供应链定义为开源软件在开发和运行过程中, 涉及的所有开源软件上游社区、源码包、二进制包、第三方组件分发市场、应用软件分发市场, 以及开发者和维护者、社区、基金会等, 按照依赖、组合等形成的供应关系网络[17], 并从理论模型、风险分析、风险识别及加固防御层面描述了当前开源协作模式下软件供应链的安全风险。高恺等人将开源软件供应链定义为开源软件制品之间在各自开发、构建、发布和运维过程中通过各种形式的代码复用形成的供应关系网络[18], 并从代码复用的角度出发, 总结了典型的软件构件类型和代码复用方式。随着大模型的出现和 SLSA 框架的提出, 现有软件的开发模式和软件供应链的组成结构又有了新的变化, 因此, 对软件供应链的相关定义又提出了新的需求。

大模型技术使得人机边界越来越模糊化, 首位 AI 软件工程师 Devin[19]的出现加速了这一进程, 现阶段 Devin 已经具备代码编写、Bug 修复、模型训练和微调、云端部署等能力。SLSA 是谷歌于 2021 年提出的, 为软件开发提供的端到端软件供应链框架, 致力于解决软件供应链安全方面的局限性, 为提升软件产品安全, 帮助用户对可能遭受的软件供应链

表 1　通用领域大模型综合性排名 Top 8

Table 1　Comprehensive ranking of the Top 8 general domain LLMs

大模型	厂商	参数量	上下文长度 (Tokens)	基座模型	多模态	开放性	胜率 (%)	模型特点
GPT-4 Turbo	OpenAI	/	128K	GPT-4	√	API 接口	50.00	以速度、准确性和处理复杂、长任务的能力著称, 可生成高质量文本, 理解复杂上下文信息, 进行逻辑推理, 回答各种问题。
Contextual_KTO_Mistral_PairRM	Contextual AI	7B	/	Mistral	×	开源	33.23	可为多数据源定制模型, 提供更精准和个性化的语言处理服务, 在理解复杂语言结构和生成高质量文本方面具有出色能力。
Yi 34B Chat	零一万物	34B	200K	Yi-34B	×	开源	29.66	具备强大的交互对话、语义理解和高效的知识问答能力。在中文理解方面, 支持处理超长上下文, 满足复杂对话和任务需求。
Claude 3 Opus	Anthropic	/	200K	Claude 3	√	API 接口	29.04	在自然语言处理方面表现出色, 能够理解和生成接近人类水平的自然语言, 具备强大的计算机视觉处理能力。
Mistral Medium	Mistral AI	/	32K	Llama 2	×	API 接口	21.86	在基准测试中表现优异, 支持多语言处理, 具有深度理解和生成自然语言的能力。
Mixtral 8x7B v0.1	/	8x7B	32K	Mixtral	×	开源	18.26	具备跨语言能力, 支持英语、法语等多种语言。具有优秀的代码生成能力, 可微调为指令跟随模型, 满足用户不同任务需求。
Tulu 2+DPO 70B	AllenAI	70B	/	Llama 2	×	开源	15.98	Llama 2 的微调版本, 使用直接偏好优化在公开可用的合成和人类数据集上进行训练, 是 Llama 2 70b Chat 的有力替代品。
LLaMA2 Chat 70B	Meta	70B	4K	Llama 2	×	开源	13.87	能够理解并生成复杂的文本内容, 具备强大的上下文理解能力, 能够处理长文本和复杂的对话场景。经过人类反馈强化学习优化, 使得其在对话生成方面表现出色。

攻击提供指导策略。SLSA 框架将软件供应链抽象为一系列利用现有平台和构件创建新构件的过程, 并将其表示为围绕顶级源、依赖项和软件包的构件及其构建过程的有向无环图, 如图 1 所示。

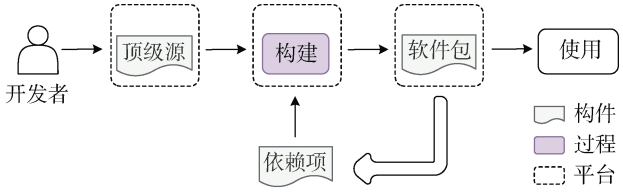

图 1　SLSA 框架元素构成的有向无环图

Figure 1　Directed acyclic graph constituted by elements of the SLSA framework

基于上述大模型技术的现状和 SLSA 框架中定义的元素, 软件供应链被定义为人机协同的软件设计与开发的各个阶段中来自顶级源编码、上游依赖项、生成软件包, 及其构建平台和过程的总和。其中, 顶级源是以当前的软件为起点的未经编译的程序代码及配置项, 通常由软件开发人员直接编写生成。这些代码及配置项是软件构件的基础, 直接影响整个软件的安全性、可靠性和完整性。依赖项是在软件开发过程中, 开发者引用或集成的来自第三方的代码和资源。这些依赖项可以是公开的开源库, 也可以是私有的或商业的二进制代码。软件包是软件开发过程的产物, 包含可执行的应用程序、库、配置文件等。构建过程不仅是将软件从输入构件转换为输出构件, 还从代码编辑、编译的角度为软件开发提供更加底层的安全保障。大模型赋能的软件供应链开发环节安全被定义为大模型介入的软件供应链开发环节中所有构件、过程和平台的安全总和。大模型赋能的软件供应链开发环节的安全防护, 旨在应用大模型技术, 赋能软件开发环节中的构件、平台及其全过程免受漏洞和恶意代码攻击, 以保障软件交付的安全性、可靠性和完整性。

2.3　软件供应链安全防护技术的发展历程

2.3.1　典型软件供应链安全事件分析

软件供应链的攻击相对于传统的针对软件漏洞的攻击, 攻击面的边界由软件本身扩大为软件内部的所有代码、模块和服务, 以及与这些模块相关的供应链上游的编码过程、开发工具和设备, 由此形成的嵌套风险显著降低了攻击的难度。近年来, 出现了多起攻击者利用软件供应链安全防范的缺失, 发起攻击的典型事件。例如, K.Thompson[20]提出了一种通过攻击软件编译工具的方法, 能够污染所有通过此编译工具编译生成的软件。此外, 攻击者利用开发人员从官方渠道获取 Xcode 开发工具难的情形, 通过注

入恶意代码污染从非官方渠道发布的 Xcode, 导致编译后的 APP 均存在后门。Sonatype 发现攻击者通过克隆 GitHub 存储库并修改 GitHub 执行脚本, 滥用 GitHub 自动化持续集成/部署(Continuous Integration/Continuous Delivery, CI/CD)通道挖掘加密数字货币。此外, 攻击者对 GitHub 库发起攻击, 将恶意代码注入到 event-stream 依赖中, 导致大量依赖此项目的 Node.js 应用受到影响。安全研究人员 Alex Birsan[21]利用依赖和命名空间误用的缺陷, 在无需社会工程的条件下向数十个高价值目标发送了恶意依赖包, 攻破了微软、苹果、优步和特斯拉等公司的系统。近期发生的 SolarWinds 事件[22]中, 攻击者通过篡改 Orion 软件包从官方渠道分发后门软件, 如图 2 所示, 用于实施隐蔽攻击, 扩大攻击范围和进行间谍活动。

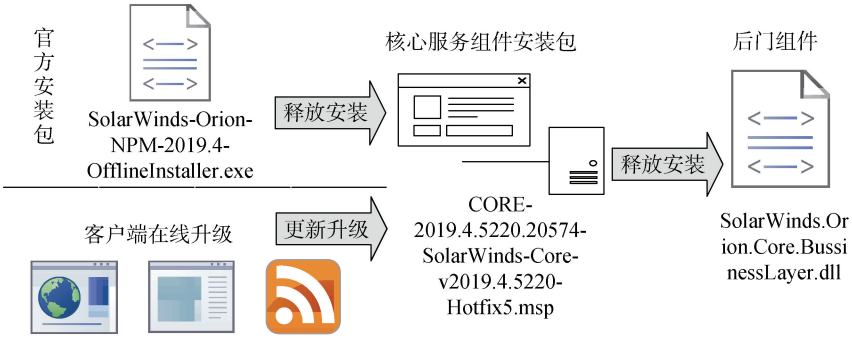

图 2 SolarWinds Orion 软件包污染

Figure 2 SolarWinds Orion software package pollution

软件设计与开发的任何阶段的安全问题都会直接影响软件供应链中所有下游的安全, 显著扩大了攻击的影响和范围。通过分析现有安全事件可以发现, 攻击者主要在软件开发环节以代码托管平台、编译环境、依赖项和软件包为污染目标。与此同时, 这些安全事件还体现出多样化的特点, 攻击的切入点和最终的受害者之间跨越了多个时空维度, 在表现形式以及最终影响上具有典型性, 值得引起安全研究人员的高度关注。

2.3.2 开发环节软件供应链安全防护的发展

软件供应链攻击防护的发展历程, 大致可以分为四个阶段。在初期阶段, 软件系统由少数人使用, 安全问题并未引起重视。到了中期阶段, 随着互联网普及, 计算机系统的安全问题变得更加复杂。此时, 软件供应链安全问题的重要性逐渐被认识到, 并出现了一些安全程序设计和分析技术, 包括安全需求分析、威胁建模、安全代码审查、安全测试等安全检测和防护技术。Web 应用的普及推动了软件的发展, 软件供应链安全由此进入了发展阶段。与传统的桌面应用程序相比, Web 应用程序面临着更多的安全威胁, 因此软件供应链的安全防护变得越来越重要。在这个时期, 安全被整合到软件构建过程的各个阶段中, 例如需求分析、设计、编码、测试、部署等。现阶段, 随着软件规模的增大和复杂度的提高, 手动处理所有的安全问题已经不再可行。因此, 现代软件供应链攻击防护更多需要自动化安全工具的支持, 包括代码检测、漏洞扫描、软件成分分析等, 这些自动化工具可以辅助开发人员及时发现并解决安全问题, 提高软件构建过程的安全性。

随着软件与系统安全的发展, 出现了一些与软件供应链攻击防护相关的框架模型, 如: 安全开发生命周期(Security Development Lifecycle, SDL)、安全软件开发生命周期(Secure Software Development Life Cycle, S-SDLC)、开发安全操作(Development Security Operation, DevSecOps)[23]、SLSA 框架等, 用于应对和缓解软件开发和构建过程中可能产生的安全问题。SDL 是微软提出的一种经典框架, 用于从安全角度出发指导可覆盖全生命周期的软件开发过程, 从需求分析、设计、代码开发到发布的所有阶段, 引入安全和隐私原则。S-SDLC 是由 OWASP 独立发布并主导的研究项目, 现已成为一种面向 Web 和 APP 厂商的安全工程方法。随着人工智能(Artificial Intelligence, AI)在全球范围内掀起的浪潮, 特别是 2022 年 11 月 ChatGPT 的发布以来, 大模型的能力不断迭代和完善, 在网络安全领域的应用落地也随之加速。总之, 软件供应链安全防护的发展是一个逐步深入的过程, 从最初安全意识的形成到传统安全防护技术的兴起, 再到大模型技术向软件供应链安全的整合, 都是推动软件供应链安全发展的重要历程。

2.3.3 大模型增强软件供应链安全防护的趋势

在软件供应链安全中, 传统的方法针对已知漏洞通常使用软件基因、软件成分分析(Software Com-

position Analysis, SCA)等技术进行识别, 针对未知漏洞通过静态分析、动态分析等技术判断代码质量, 并对引入的第三方代码进行安全检测。传统方法主要依赖于动静态代码分析、模糊测试等技术, 是在已知问题的认知和建模基础上提出的解决方案, 例如, 基于已知的漏洞模式编写特定的检测规则检测漏洞。然而, 由于软件供应链的安全威胁持续进化, 传统的防护方法逐渐显示出了其局限性。传统方法依赖于已有规则和模式, 人工编写成本高, 由于特征规则受限导致高误报和漏报率问题, 并且不具备检测未知漏洞能力。深度学习不需要显式地建模问题到目标的映射, 可以从大量的数据中学习特征, 这些特征可能包括代码的语法结构、函数调用关系、数据流等, 在处理程序分析、代码安全检测等任务时具有更强的学习能力。然而, 深度学习方法也存在一些问题, 如模型的可解释性较差, 以及需要大量标注数据进行训练, 过度依赖专家的知识和经验等。随着软件工程的复杂化, 软件涉及的代码库、依赖关系等持续增长, 而传统模型难以处理如此庞大且复杂的数据。此外, 传统模型通常针对单一任务在特定数据集上训练, 无法同时满足代码审计、依赖关系分析等多任务场景的需求, 适应新威胁变化的能力弱。而大模型具有更强的上下文理解和推理能力, 能够在不同的场景和任务中进行迁移学习, 尤其是在应对快速变化的安全威胁方面, 具有更好的适应性, 因此在复杂的软件供应链安全防护中体现出了更明显的优势。

首先, 大模型学习和融合了多个领域的知识, 包括覆盖全网的通用领域知识、安全领域知识、代码数据集以及安全漏洞和恶意行为检测模式等内容。这些大规模数据的训练无需人工标注, 极大减少了人工编写规则的成本。对于庞大的代码库及其元数据, 大模型有着传统模型无法比拟的数据融合能力。其次, 大模型具有更强的表示能力和理解能力, 可以学习到代码的深层特征和规律, 包括代码语法、语义、安全漏洞模式等, 分析软件依赖项中的复杂关系等, 从而更准确地识别代码中的安全问题。并且, 大模型由于其出色的多任务学习能力, 不需要为每个任务独立训练模型, 具备上下文语义理解、知识问答、多轮对话的能力, 在软件供应链安全赋能方面具有更强的泛化能力和迭代效率。通过学习大量代码库和已有编程知识, 大模型可以自动标记、解释复杂的代码结构和行为, 生成可读性强的代码注释和文档, 加快代码理解和分析过程。同时, 将代码生成和漏洞修复技术相融合, 大模型能够充分考虑代码的上下文和可能产生的附加效应, 生成针对具体漏洞的定制化补丁, 不仅提高了补丁修复的成功率, 还增强了代码的整体安全性。此外, 大模型拥有自主学习和增量学习能力, 可以主动理解和掌握最新的安全情报、漏洞知识和公告等内容, 实时更新和自我进化, 具备一定的推理和预测能力。同时, 大模型拥有未知攻击检测发现能力, 能够形成一定的知识“涌现”和创新, 将安全风险转化为修复建议, 以应对不断变化的未知安全威胁。

总之, 相对于传统方法和深度学习方法的专用能力而言, 大模型在应对复杂场景的综合分析和处理能力优势明显, 适用于对软件供应链多环节、多任务的赋能, 在安全防护中能够降低专家依赖, 左移人机边界, 为软件开发过程提供高效智能的全生命周期支撑。Singla T 等人[24]使用大模型分析历史上发生的软件供应链安全事件, 使用 LLM 对 69 起软件供应链安全事件进行分类, 从威胁类型、意图、性质和影响四个维度出发为 LLM 设计提示, 证明了大模型可以有效地实现对软件供应链安全事件的分类描述, 如图 3 所示。Huang L 等人[25]从 LLMs、模糊测试和基于 LLMs 生成的模糊测试三个方面, 对基于大模型的模糊测试技术进行了综述, 并且研究了基于 LLM 生成的模糊测试技术在未来广泛部署和应用的潜力。Fangzhou Wu 等人[26]研究了 ChatGPT 在七种软件安全应用中的局限性, 包括漏洞检测、调试、反编译、修复、成因分析、符号执行和模糊测试等。该研究表明, ChatGPT 不仅擅长代码生成, 而且在理解用户自然语言提示、推理程序控制流和数据流、生成复杂数据结构等方面表现出强大的能力, 甚至能够反编译汇编代码, 但在某些安全相关任务中也存在局限性, 例如, ChatGPT 在处理长代码上下文的能力受到限制。

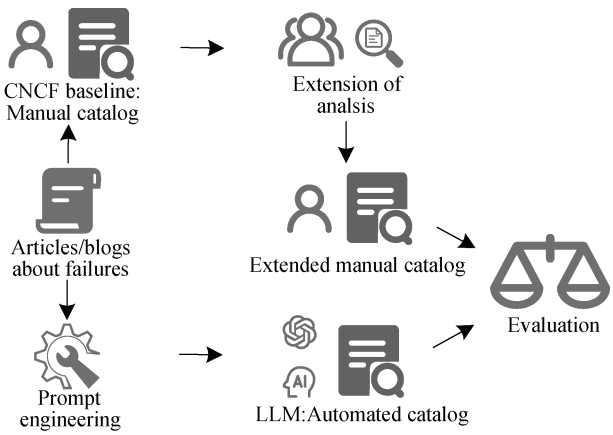

图 3 使用大模型分类软件供应链安全事件

Figure 3 Classifying software supply chain security events using LLMs

在漏洞检测方面, 安全研究人员将大模型与传统静态分析方法以及深度学习检测方法进行了比较, 大部分测试结果都显示了 LLM 的良好检测能力。Zeyu Gao 等人[27]通过对 16 个 LLM 与 6 个先进的基于深度学习的模型和静态分析工具进行了对比实验, 发现一些 LLM 在漏洞检测方面已经优于传统的深度学习方法, 揭示了 LLM 尚未开发的潜力。有研究团队将大模型应用于模糊测试工具的优化, 通过使用历史 Bug 触发程序漏洞作为参考, 引导变异种子生成。在代码生成方面, 虽然大模型生成了一些不安全的代码, 但是在明确要求下, 设计提示词引导大模型对漏洞代码进行修改, 大模型能够认识到自己生成代码的危害性并加以纠正, 进而生成更加安全的代码。在安全测试方面, 研究人员使用 ChatGPT AI 来自动化各种安全测试任务[28], 例如, 漏洞扫描、渗透测试等, 以发掘大模型在安全测试的准确性和效率方面的潜力。

目前, 将大模型应用于软件供应链安全领域仍处于起步阶段, 但总体上已经取得一些不错的效果。在未来, 大模型将在软件供应链安全方面发挥出更大的优势, 例如: 在软件集成过程中检测应用程序编程接口(Application Programming Interface, API)误用造成的逻辑漏洞; 将大模型进行代码调用图生成、代码语义理解等定向训练, 降低静态分析的误报率; 扮演安全培训工程师, 定向生成特定场景下的漏洞代码及相关文档, 辅助开发人员的安全培训等。总之, 大模型在软件供应链安全中的应用具有巨大的潜力和价值。随着技术的不断进步和应用的深入, 大模型将在未来为软件供应链安全提供更加全面、高效和智能的支持。

3 大模型赋能软件供应链安全防护的研究现状

本文重点从大模型赋能软件供应链安全的角度出发, 围绕在开发环节存在的软件供应链安全问题, 对相关防护技术研究进行综述, 依据 SLSA 框架将软件供应链安全防护的研究内容分为针对顶级源、依赖项、软件包构件及其构建过程的安全防护四个方面。

3.1 软件供应链开发环节的威胁要素

在开发过程中, 任何软件供应链的环节都可能引入安全风险。图 4 展示了以 SLSA 框架为指导, 从软件开发的角度出发, 结合文献调研梳理总结出的软件供应链顶级源、依赖项、软件包构件及其构建过程中存在的 17 种安全威胁要素, 包括: 顶级源中存在的编码缺陷漏洞、恶意代码提交、代码托管平台攻击和编程语言的不安全特性等问题; 依赖项中存在的缺陷漏洞、恶意代码、包名误用和依赖关系复杂等问题; 软件包中存在的缺陷漏洞、恶意代码、补丁缺失、逆向分析和重打包等问题; 构建过程中存在的不安全编译和优化、受威胁的编译组件、侧信道攻击和不安全的代码生成等问题。

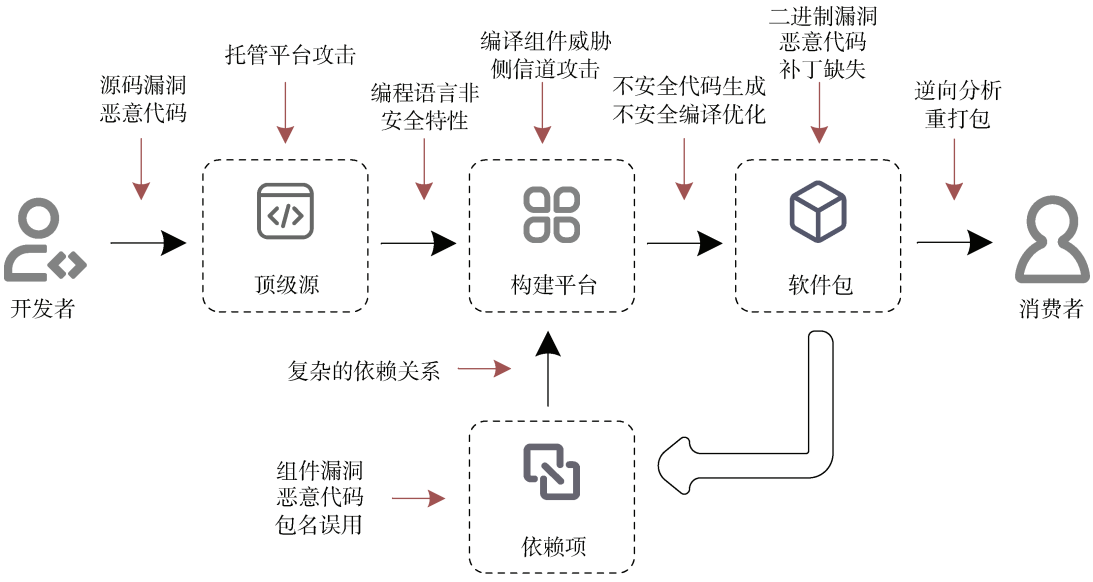

图 4 软件供应链构件安全威胁要素

Figure 4 Security threat elements of artifacts in software supply chain

本文后续章节将重点围绕大模型在顶级源、依赖项、软件包构件及其构建过程的安全赋能, 对 17 种威胁要素进行分类研究, 有针对性地对软件供应链安全问题进行梳理, 对大模型在软件供应链安全

防护技术和方法中的研究进行归纳和总结。

3.2 面向顶级源的代码检测技术研究

顶级源中主要存在源码漏洞、恶意代码提交、代码托管平台攻击、编程语言的不安全特性等威胁要素。随着软件复杂程度的提高带来的软件可追踪性的难度骤增, 顶级源的安全风险不断上升。沈力等人[29]的研究表明, 由于软件开发迭代速度快, 安全需求是否被满足, 代码是否完成检测等问题对于系统开发和维护人员来说已经越来越难以把控。开源软件的去中心化和开放式协作, 为源代码带来了难以预测的威胁[30]。攻击者会破坏上游服务器或代码存储库, 向其中注入恶意代码, 甚至存在开发人员故意将后门代码上传到代码托管平台的情况, 此类攻击具有极强的隐蔽性, 一旦成功会对下游用户产生严重影响。P. Ladisa 等人[31]的研究表明开源软件供应链的复杂性暴露出了巨大的攻击面, 对开源组件的广泛依赖使其成为被反复攻击的目标。Stack Overflow 等论坛的代码未经安全审查, 开发人员出于便利直接复制和使用这些代码, 导致不安全的代码流入到生产环境中。F. Fischer 等人[32]对 130 万个 Android 应用程序进行分析, 发现其中有 15.4%的代码来自 Stack Overflow, 而这其中有 97.9%的代码至少包含一个不安全的代码片段。此外, 开发者在使用 C/C++等编程语言时, 由于不遵守安全的编码规范很可能编写出具有破坏内存的漏洞代码。

针对顶级源中存在的软件漏洞问题, 传统的漏洞检测技术可以帮助开发人员发现和消除一部分安全威胁, 包括静态分析、动态分析, 以及动静态结合的分析方法。例如, Coverity 采用静态分析技术, 帮助开发和安全人员在 SDLC 的早期解决安全和质量缺陷; WhiteHat 采用动态分析技术, 测试应用程序的访问点, 监控是否存在安全威胁。近年来的一些工作, 研究人员提出了使用机器学习和深度学习方法来识别源代码中的安全漏洞。G. Tang 等人[33]采用神经网络自动提取漏洞特征的方法, 提高了漏洞检测的智能性。X. Li 等人[34]提出了一种基于深度学习的漏洞检测方法, 通过融合源代码和汇编代码切片, 提高了漏洞特征提取的准确性。H. Wang 等人[35]在神经网络的基础上开发了一种基于图的学习方法, 使用程序控制流图和依赖图捕获和推理程序的数据和调用关系, 重新构建漏洞特征, 取得了较好的检测效果。

传统方法主要是针对目标问题的认知进行建模, 进而提出解决方案。然而, 此类方法存在人工编写成本高、泛化能力弱且迭代滞后的问题, 并且不具备未知漏洞检测能力。深度学习不需要显式地建模问题到目标的映射, 而是在学习大量数据的基础上利用神经网络来拟合这种关系, 在处理复杂问题时具有更强的学习能力。然而, 深度学习方法也存在一些问题, 比如需要大量标注训练数据, 过度依赖专家的知识和经验, 模型可解释性较差等问题。通过调研大模型在顶级源代码检测技术与方法赋能方面的相关文献, 本文梳理总结了大模型在漏洞检测、缺陷修复、模糊测试、安全评估等具体技术环节中的赋能现状, 如表 2 所示。

表 2 大模型赋能的顶级源代码检测技术与方法

Table 2 Top-level source code detection techniques and methods empowered by LLMs

赋能对象	威胁要素	来源	大模型方法	赋能环节	数据集	年份
顶级源	①源码漏洞 ②恶意代码 ③代码托管平台攻击 ④编程语言的不安全特性	文献[36]	GPT-3、Codex	安全评估	LLMSecEval	2023
		文献[37]	ChatGPT	漏洞检测	SARD, NVD	2023
		文献[38]	GPT-4	静态分析	UBITect	2023
		文献[39]	CodeX	缺陷重现	Defects4J	2023
		文献[40]	GPT-2, CodeGPT, PolyCoder	漏洞检测	DiverseVul	2023
		文献[41]	SecureFalcon	漏洞推理	FormAI	2023
		文献[42]	LLM4Vuln	漏洞推理	Code4Rena	2024
		文献[43]	CodeRepair	缺陷修复	VulDeeLocator, NVD, SARD	2024
		文献[44]	LLM4FUZZ	模糊测试	DeFi	2024
		文献[45]	CovRL-Fuzz	模糊测试	JS Engine	2024
		文献[46]	ChatFuzz	模糊测试	Unibench, Fuzzbench, LAVA-M	2024

在一些方面, 大模型已经展现出了相对传统方案更好的适应性和更具明显的优势。首先, 大模型使用的训练数据无需人工标注, 在程序分析和代码语义理解方面具有更强的泛化能力, 极大减少了人工编写规则的成本。其次, 大模型具有更强的表示能力和学习能力, 能够处理更复杂的代码结构和模式,

在漏洞检测方面表现出了相较于图神经网络更优越的能力。通过在大规模代码数据集上进行训练, 大模型可以学习到代码的深层特征和规律, 从而更准确地识别代码中的缺陷。在模糊测试方面, 基于大模型的模糊测试引擎在代码覆盖率和错误发现的能力超过了当前最先进的模糊测试器。此外, 大模型可以自主学习和理解最新的安全情报、漏洞知识和公告等内容, 同时将知识转化为有针对性的代码缺陷修复建议, 缩短修复周期。

大模型利用强大的上下文推理能力, 能够理解代码的深层语义和逻辑结构, 识别出复杂的漏洞模式, 持续学习新的漏洞知识, 解决传统代码检测技术自动化效率低、存在较高误报率和漏报率的问题。在提示词优化方面, Tony C 等人[36]提出了名为 LLMSecEval 的自然语言提示数据集, 能够涵盖 MITRE Top 25 通用缺陷枚举(Common Weakness Enumeration, CWE)中列出的 18 个场景及其相应的安全代码示例, 可用于对 LLM 生成的代码安全评估。实验通过 GPT-3 和 Codex 使用该数据集生成代码, 并采用代码分析引擎 CodeQL 对其生成代码的安全性进行了验证。Zhang C 等人[37]针对当前使用 ChatGPT 进行漏洞检测没有充分考虑与漏洞检测相适应的提示词问题, 结合代码结构优化提示词设计, 利用 ChatGPT 多轮对话的记忆能力设计出了恰当的漏洞检测提示词, 证实了 ChatGPT 在漏洞检测方面的可行性, 其准确率比基于条件的图神经网络缺陷检测 SOTA 方法 CFGNN 提高了 58%。Li H 等人[38]利用 LLM 在复杂错误理解和逻辑推理中的能力, 对 LLM 辅助静态分析进行了深入研究, 通过精心设计提示词构建 LLift 框架, 在 Linux 内核中新发现了 13 个 UBI 未知错误, 展示了 LLM 在静态分析中的强大能力。为了从一般的错误报告中自动生成测试, Kang S 等人[39]提出了一种 LIBRO 技术, 是首个面向通用缺陷的自动化重现工作, 该技术有效利用大模型提升了缺陷重现的有效性。实验结果表明, 在主流数据集 Defects4J 中, LIBRO 能够根据缺陷报告自动生成测试用例并重现出 251 个缺陷, 比使用引导遗传算法搜索崩溃的 SOTA 重现方法 EvoCrash 多复现了 91 个缺陷。

在漏洞检测增强方面, Chen Y 等人[40]评估了 GNN、RoBERTa、T5、GPT 四类不同架构的模型在漏洞检测方面的能力, 结果表明在已知小规模数据集上进行训练时, 不同架构的模型性能之间没有明显差异。然而, 在更大规模的数据集上进行训练时, LLM 将明显优于代表性的图神经网络模型。其中, NatGen 模型的 F1 得分最高为 47.15%, 而图神经网络模型 ReVeal 模型仅为 29.76%。在未知项目上测试时, 尽管表现最好的 CodeBERT 模型 F1 得分为 11.94%, 但仍优于 ReVeal 模型的 F1 得分 8.67%。针对静态分析工具的高误报率问题, Ferrag M A 等人[41]通过对 FalconLLM 模型进行微调, 构建了名为 SecureFalcon 模型, 通过区分易受攻击和安全代码在 FormAI 数据集上进行测试, 模型准确率达到了 94%, 不仅减少了与传统静态分析相关的假阳性, 而且为软件漏洞检测提供了一种新的解决方案。Sun Y 等人[42]提出了 LLM4Vuln 评估框架, 将 LLM 的漏洞推理能力与其他能力解耦合, 包括主动寻求附加信息的能力。与没有额外知识的场景相比, 具有总结知识的 GPT-4 显著提高了脆弱性推理的精度和 F1 分数。例如, 在使用没有上下文提示的场景中, 精度从 12.12%提高到了 28.17%, F1 分数从 19.28%增加到了 27.58%, 并在漏洞悬赏平台成功提交了 9 个零日漏洞。在缺陷修复方面, 为了强化 LLM 生成代码的安全性, Islam N T 等人[43]提出了一种结合语义和句法奖励机制的强化学习方法, 用于修复程序缺陷。实验结果显示, 该方法模型与具有可比性或更少参数的模型相比性能优越, 超过了 7B 参数量 CodeGen2 模型的 0.06 BLEU 和 0.07 Rouge-L 得分。

在模糊测试方面, Shou C 等人[44]提出了一种 LLM4Fuzz 框架, 旨在通过利用 LLMs 将模糊测试引导至高价值代码区域和更有可能触发智能合约漏洞的输入序列。此外, LLM4Fuzz 还可以利用 LLMs 根据用户定义的参数指导模糊测试, 从而减少盲目探索的开销。在对收集的 117 个智能合约项目评估中, LLM4Fuzz 总共可以覆盖 160 万条指令, 而智能合约模糊测试 SOTA 方法 ITYFuzz 仅覆盖了 110 万条指令。对于特定的项目, LLM4Fuzz 实现的覆盖率甚至超过了该基线的两倍以上, 在漏洞检测发现效率方面取得了显著增益。针对现有技术通过代码覆盖率优化模糊测试中存在的问题, Eom J 等人[45]将大语言模型与覆盖反馈的强化学习相结合, 通过构建加权覆盖图和计算模糊测试奖励, 生成更可能发现新覆盖区域的测试用例, 从而提高漏洞检测的能力, 同时减少语法和语义错误。通过运行基于启发式或语言模型的 AFL、Superion、Token-Level AFL、Montage 等面向 JavaScript 的 SOTA 模糊测试器, 在相同时间下对 4 种 JavaScript 引擎重复了 5 次实验。结果表明, CovRL-Fuzz 的性能优于最先进的 JavaScript 模糊器, 边覆盖率分别平均增加了 102.62%/98.40%/19.49%/57.11%, 并且发现了最新引擎中的 39 个未知

安全漏洞。Hu J 等人[46]提出了一种结合生成式大模型的灰盒模糊测试方法, 使用 ChatGPT 理解格式化数据文件、不同编程语言的源代码、没有显式语法规则的文本等信息, 进而提高输入格式的合规性和质量。实验结果表明, ChatFuzz 在 12 个目标程序上比灰盒测试 SOTA 方法 AFL++提高了 12.77%的路径覆盖率, 并且在检测具有明确语法规则程序中的漏洞方面表现良好。

3.3 面向依赖项的威胁分析技术研究

依赖项中主要存在组件漏洞、恶意代码、包名误用、复杂的依赖关系等威胁要素。从软件开发的角度来看, 使用依赖项能有效地减轻开发人员的工作量。但是, 从软件安全的角度来看, 广泛使用依赖项不仅加重代码审计的负担, 还会引入不可控的安全风险。由于软件之间依赖关系复杂, 一旦依赖组件存在漏洞或恶意代码, 组件重用会产生蝴蝶效应, 加速漏洞传播[47]。R. Duan 等人[48]指出, 很多下游的软件开发者往往缺乏对第三方库的安全关注而没有及时进行更新。E. Derr 等人[49]对程序的版本控制研究发现, 开发人员为了避免重新集成带来的额外工作和版本不兼容问题, 会放弃依赖项更新。Y. He 等人[50]利用其研发的隐私泄露分析工具, 评估了 Android 第三方库的隐私泄露风险, 结果显示很多第三方库都拥有访问个人信息的权限, 对用户隐私安全造成了极大威胁。此外, 攻击者可以在公共存储库上以相同的名称注册具有高版本号的恶意依赖项, 从而覆盖现有版本的依赖项。第三方组件存在大量的包名伪造[51]问题, 利用依赖项生态系统的缺陷, Alex Birsan 对 PyPI、Npm 等发起包名误用攻击, 成功入侵了 35 家大型科技公司。同时, 包管理器也会被攻击者滥用, 攻击者毒化 Python 包管理工具的包索引[52], 进而发布污染的 PIP 包。

针对依赖项存在的安全威胁, 研究人员采取了多种方法减轻依赖项风险。Q. Li 等人[53]基于开源代码库中声明依赖关系的构建文件提出 PDGraph 方法来发现不同项目之间的依赖关系, 从不安全边界和依赖项两个方向分析并评估了项目依赖漏洞的影响。E. Derr 等人[49]对依赖项更新机制的研究指出, 在广泛使用且存在安全漏洞的依赖项版本中, 97.8%的漏洞可以通过替换的方式进行修复。软件物料清单(Software Bill of Materials, SBOM)通过提供软件开发中的组件和依赖项清单描述软件中的安全问题。Black Duck、OpenSCA、Dependency-Check 等 SCA 工具基于依赖关系提供 SBOM, 检测应用程序和容器中使用的开源和第三方组件的安全问题。针对包名误用攻击, 开源存储库所有者可以采用更严格的验证过程, 强制执行命名空间, 以防止冲突依赖项名称进入软件开发生命周期。例如, Java 依赖仓库 Maven Central 采用基于域的方法验证命名空间所有权, Go 依赖仓库以 GitHub 存储库的统一资源定位系统(Uniform Resource Locator; URL)命名, 以此提高攻击者实施攻击的难度。K. Yang 等人[54]提出了一种基于模糊测试发现组件库漏洞的方法, 结合组件的应用上下文发现了 Chrome、Safari 和 Firefox 等流行软件中的多个零日漏洞。

传统方法主要根据依赖项源码、更新日志、漏洞公告等数据, 通过手工定义检测规则和模式, 识别存在的安全威胁。然而, 传统方法受限于专家的认知经验, 并且会因为固定的特征规则导致高误报和漏报率出现。通过训练深度神经网络, 模型可以识别出与已知威胁模式相似的代码片段或行为, 从而发现潜在的威胁。然而, 深度学习仍需要对大量的数据进行标注和训练, 模型的可解释性较差。并且, 随着依赖项的不断增长和复杂化, 审查的效率和准确性也面临极大挑战。上述方法对依赖项分析的能力受限, 难以适应错综复杂的软件依赖关系。通过调研大模型在依赖项威胁分析技术与方法赋能方面的相关文献, 本文梳理总结了大模型在测试用例生成、依赖包完整性识别、测试驱动生成等具体技术环节中的赋能现状, 如表 3 所示。

表 3 大模型赋能的依赖项威胁分析技术与方法

Table 3 Dependency threat analysis techniques and methods empowered by LLMs

赋能对象	威胁要素	来源	大模型方法	赋能环节	数据集	年份
依赖项	①组件漏洞 ②恶意代码 ③包名误用 ④复杂的依赖关系	文献[55]	CodeGen	测试用例生成	DL library	2023
		文献[56]	CodeX, CodeGen	测试用例生成	DL library	2023
		文献[57]	LLaMa	依赖包完整性识别	VulLib	2023
		文献[58]	OSS-Fuzz	测试驱动编写	tinyxml2, OpenSSL	2023
		文献[59]	GPT-3.5, GPT-4	测试驱动生成	NAIVE-K, BACTX-K	2024
		文献[60]	ChatGPT-4, Copilot	测试驱动编写	OSS-Fuzz C	2023
		文献[61]	ChatGPT, GPT-4	测试驱动编写	自定义	2023

相对传统方案而言, 大模型在面对大型软件中复杂依赖关系网络分析的任务场景时, 能够自动化理解和处理大规模数据, 在较短时间内更高效的完成分析任务。其次, 大模型具有更强的上下文理解和推理能力, 通过深入分析软件依赖项中的复杂关系和模式, 融合多领域知识, 包括代码代码语法语义、文档注释、安全漏洞等, 提供更全面的安全威胁分析能力。此外, 大模型借助强大的自然语言和代码生成能力, 在测试用例生成、测试驱动编写、模糊测试种子生成方面表现出了巨大潜力。这些能力是现有传统方法无法比拟的, 并且已经在实际应用中发挥了的重要价值。

基于大模型进行依赖项威胁分析, 能够梳理复杂的依赖关系, 发现深层次的威胁, 解决了传统依赖项分析技术中难以调用、复杂依赖识别困难等问题。在依赖库模糊测试用例生成方面, Deng Y 等人[55]提出了一种名为 TitanFuzz 的大模型, 首次使用生成型(Codex)LLM 和填充型(InCoder)LLM 来生成多样化的输入, 对 TensorFlow、PyTorch 等深度学习库进行自动化模糊测试。该方法首先使用 Codex 合成高质量的种子输入, 然后将 InCoder 与进化算法相结合, 解决了传统测试方法在 API 库调用时的输入限制, 改善了传统模糊测试无法覆盖多样化 API 序列的缺陷。在对 PyTorch 和 TensorFlow 的评估中, 该方法比 SOTA 基线方法 DeepREL 的 API 覆盖率分别提高了 24.09%和 91.11%, 同时检测出了 44 个未知错误。FuzzGPT 框架[56]通过对 LLM 进行初始化以生成用于模糊测试的用例程序, 通过 LLM 的微调和上下文学习, 实现全自动化的测试。此外, FuzzGPT 还显示出利用 ChatGPT 的指令跟随能力进行有效模糊处理的潜力, 并对 PyTorch 和 TensorFlow 库进行测试, 其代码覆盖率比上述 TitanFuzz 方法分别高出了 60.70%和 36.03%, 共检测到 76 个漏洞, 其中 49 个已被确认为未知漏洞。在依赖包完整性识别方面, Chen T 等人[57]提出了一种名为 VulLibGen 的生成方法, 旨在解决现有 SCA 漏洞报告中缺乏记录受影响库列表的问题。该方法利用 LLM 对自然语言理解的巨大潜力, 通过漏洞描述生成与之相关的可能受影响的依赖库名称列表。此外, VulLibGen 还采用输入扩充技术, 用于辅助识别训练过程中未出现的零样本受影响库, 用于解决 VulLibGen 的错误识别问题。结果显示, VulLibGen 的平均 F1 得分为 0.626, 优于 SOTA 实践方法 VulLibMiner 的平均 F1 得分 0.561。

在依赖库模糊测试驱动编写方面, 针对编写高质量的模糊驱动程序耗费时间长, 现有的模糊驱动程序生成技术能力不足的问题, Google 提出了一种采用 LLM 赋能 OSS fuzz 进行辅助模糊测试方法[58], 该方法通过生成可用于模糊测试目标, 减少人工编写模糊测试驱动的工作, 实现了现有 OSS fuzz 项目的全自动化模糊测试目标生成, 使 14/31 测试的 OSS Fuzz 项目成功地编译了新的目标, 在项目 tinyxml2 中将代码覆盖率从 1.5%提高到 31%, 语句覆盖率从 38%提高到 69%, 并且在 LLM4FDG[59]中为基于 LLM 的模糊驱动程序生成研究提供了理论和技术支撑。对于自动化模糊测试中生成正确和健壮的高质量 API 驱动代码的挑战, C. Zhang 等人[60]利用 LLM 在代码 API 的理解和自动代码生成的能力, 对基于 LLM 自动化生成模糊驱动程序的有效性进行了探索, 通过半自动的框架依次设计了五种查询策略进行评估, 结果表明基于 LLM 生成的测试驱动具有 64%的正确率, 具备一定的实用性和可行性。Lyu 等人[61]提出了一种 PromptFuzz 方法, 通过提出指导性程序生成、错误程序净化、覆盖引导的提示变异和有约束的模糊器融合等关键技术, 迭代生成模糊驱动程序来探索未发现的依赖库代码。该方法在 14 个真实库中进行了有效性评估, 实验结果表明, PromptFuzz 生成的模糊驱动程序实现了更高的分支覆盖率, 是 OSS-Fuzz 的 1.61 倍, 是 SOTA 模糊测试驱动程序生成解决方案 Hopper 的 1.67 倍。此外, PromptFuzz 生成的模糊驱动程序成功发现了 33 个未知程序错误。

3.4 面向软件包的安全检测技术研究

软件包中主要存在二进制漏洞、恶意代码、补丁缺失、逆向分析、重打包等威胁要素。由于大多数软件包是以二进制代码发布的, 源代码不可用, 因此通过分析二进制程序检测代码漏洞存在更大的挑战性。在恶意代码的威胁方面, C. Li 等人[62]提出了一种针对模型的“后门”攻击, 根据模型的梯度使用标签反转算法对训练数据进行伪装, 污染检测模型的训练数据集, 误导模型把一些恶意软件分类为良性。Y. Shi 等人[63]的工作指出, 安全补丁通常针对最新的软件版本开发, 存在安全问题的旧版本软件让用户面临难以修复的安全风险。A. Bolat 等人[64]的研究指出, 攻击者采用静态方式分析可执行文件的反汇编, 或者利用侧信道进行逆向工程, 从而获取可执行文件的关键数据。在重打包的威胁方面, H. Ma 等人[65]的研究表明, 桌面应用中所使用的 Hook、在资源文件中隐藏恶意负载等传统攻击技术逐渐出现在重打包攻击中。A. Merlo 等人[66]指出, 攻击者在重打包后使用自己生成的私钥对恶意软件包签名, 诱导用户安装使用。随着攻防对抗技术的演进, 针对恶

意软件检测的对抗手段层出不穷。L. Demetrio 等人[67]提出了一种绕过 PE 文件的检测框架, 通过修改磁盘操作系统(Disk Operating System, DOS)头的方法, 实现了对基于机器学习的静态代码检测分析技术的绕过。X. Chen 等人[68]开发了一种自动生成对抗恶意软件检测用例的工具, 通过扰动 Android 应用程序包(Android Application Package, APK)的 Dalvik 字节码绕过安全检测。L. Shi 等人[69]介绍了一种将恶意软件重打包为应用程序的虚拟化插件, 将攻击载荷隐藏在虚拟化插件中远程加载, 通过在用户设备上运行程序的副本, 躲避防病毒软件的查杀。

针对软件的漏洞问题, Y. Wang 等人[70]提出了一种二进制代码漏洞检测系统 BinVulDet, 通过反编译二进制程序获取高级伪代码来提取漏洞特征, 通过此方案生成的漏洞特征更加接近底层实现。针对安全补丁缺失的问题, S.Wang 等人[71]提出了一种基于图神经网络的安全补丁检测系统 GraphSPD, 将补丁表示为具有更丰富语义的图, 取得了较好的检测效果。针对软件包逆向分析存在的安全问题, Y. Chen 等人[72]设计实现了 NORAX 系统, 在二进制文件运行时将程序代码动态加载到内存页中, 重定位嵌入数据和更新数据引用。D. Demicco 等人[73]提出了一种基于底层虚拟机(Low Level Virtual Machine, LLVM)消除二进制文件偏差的代码混淆方法, 削弱了攻击者对二进制文件的逆向分析能力。H. Ma 等人[65]提出了名为 AppWarder 的反重打包方案, 将分布式的思想应用到重打包防御框架中, 基于 Collatz 进行控制流混淆, 有效地抵御了多数运行时欺骗攻击。分析软件行为是目前检测恶意软件的常用手段, 但该方法存在固有的缺陷, 即此类方法严重依赖已知样本数据, 无法保证对未知恶意软件的有效性检测。为了解决此类问题, Manel Jerbi 等人[74]提出了一种恶意软件检测方案(Artificial Malware-based Detection, AMD), 通过遗传算法自动生成未知特征, 有效地提高了样本知识库的多样性。

传统方法通常涉及二进制代码反编译等逆向工程, 以分析其内部结构和行为。尽管反编译工具可以将二进制代码转换成伪代码, 但只提供了基础代码结构, 无法生成可读性强的函数名称和代码注释。深度学习方法可以自动化学习和识别二进制代码中的安全漏洞和恶意行为模式, 从大量的二进制数据中提取特征, 在不依赖特定规则或人工干预的情况下, 发现安全漏洞和恶意行为。然而, 随着攻击手段的不断演进, 一些新的安全漏洞和恶意行为往往难以被发现。在漏洞管理与修复方面, 存在无法对未公开漏洞进行有效识别和管理的问题。并且由于缺乏代码上下文理解, 导致修复不完全, 引入新的安全问题。通过调研大模型在软件包安全检测技术与方法赋能方面的相关文献, 本文梳理总结了大模型在二进制语义理解、污点分析、模糊测试、漏洞补丁等具体技术环节中的赋能现状, 如表 4 所示。

相较于传统方案, 首先大模型能够从更广泛的上下文中提取有用的信息, 学习到更全面的安全漏洞行为模式, 在不同的场景和任务中迁移学习, 在漏洞补丁的应用方面已经优于传统的自动修复技术。其次, 通过学习大量代码库和已有编程知识, 大模型可以自动标记、解释反编译的代码结构和行为, 生成可读性强的代码注释和文档。尽管基于深度学

表 4 大模型赋能的软件包安全检测技术与方法

Table 4 Software package security detection technology and methods empowered by LLMs

赋能对象	威胁要素	来源	大模型方法	赋能环节	数据集	年份
软件包	①二进制漏洞 ②恶意代码 ③补丁缺失 ④逆向分析 ⑤重打包	文献[75]	BAI-3.0	相似度比对	GitHub 全量 C/C++开源组件	2023
		文献[77]	LLM4Decompile	语义理解	Decompile-Eval	2024
		文献[78]	Code LLMs、General LLMs	语义理解	Github Real-world Projects	2024
		文献[79]	DeGPT	语义理解	AudioFlux, Mirai, Coreutils	2024
		文献[80]	ChatGPT	漏洞管理	iTAPE, Farsec, DKG, et al.	2023
		文献[81]	GPT4	污点分析	Juliet Test Suite, Karonte	2023
		文献[82]	Fuzz4All	模糊测试	SUTs	2023
		文献[83]	GPT-3.5, claude-2	模糊测试	Siesta	2024
		文献[84]	GPT-4-0613	模糊测试	BusyBox	2024
		文献[85]	CompVPD	补丁修复	Vulfix	2023
		文献[86]	GPT-Neo, GPT-J, et al.	补丁修复	Defects4j, Defects4j, et al.	2023
		文献[87]	RAP-Gen	补丁修复	tfix_rmd_rapgen, refine_rapgen	2023
		文献[88]	GPT4	恶意代码检测	Aliyun, Catak, Aliyun+Catak, et al.	2023

习的专家模型在模型尺寸和反应速度方面表现出色, 但这些专家模型的泛化能力较弱, 而大模型的零样本学习能力较强, 这正是大模型的一大优势。此外, 大模型通过持续学习能够有效适应新的漏洞模式, 同时能够充分考虑代码的上下文和可能产生的附加效应, 将代码生成和漏洞修复能力相融合, 生成针对具体漏洞的定制化补丁, 不仅提高了补丁修复的成功率, 还增强了整体的代码安全性。

在大模型赋能的软件包安全检测技术研究中, 借助于大模型强大的内在特征提取能力, 可以从函数和语义中学习到深层次的逻辑信息, 识别软件包中的不安全代码。在相似度比对方面, BinaryAI[75]提出二进制函数相似度匹配模型, 该模型在大模型的基础上应用启发式算法, 为每个二进制函数生成一个向量作为函数特征, 使用向量之间的距离反映函数的语义相似性, 并将两个文件的函数相似性问题转换为完全二分图的最大匹配问题, 提高了基于语义逻辑特征的二进制比对算法的准确率, 优于目前积极维护的流行开源二进制比对工具 Diaphora[76]。在语义理解方面, Tan H 等人[77]为反编译量身定制了开源的大模型, 提出了以开源反编译为重点的大模型和标准化的可重新编译基准, 探索了大模型在反编译增强方面的研究, 实验结果表明 LLM4Decompile 能够准确地反编译 21%的汇编代码, 比 GPT-4 的能力提高了 50%。Shang X 等人[78]通过函数名称恢复和二进制代码摘要两项下游任务, 评估了 8 个通用领域的大模型和 4 个基于深度学习的模型。研究结果表明, 尽管基于深度学习的模型在模型尺寸和推理速度方面表现出色, 然而现有小模型, 如 BinT5、HexT5 等, 对训练分布外的数据泛化能力较差, 其在函数名恢复、二进制代码摘要的性能远低于大模型。相反, 大模型虽然在推理速度上较慢, 但其零样本学习能力较强, 成为大模型赋能二进制语义理解的优势所在。Hu P 等人[79]提出了一种名为 DeGPT 的二进代码语义理解框架, 通过三角色机制将大模型应用于反编译器输出优化任务, 解决了现有反编译器存在的变量名无意义、变量冗余以及代码注释缺失等问题。实验结果表明, 与基于 Transformer 架构的 SOTA 反编译器优化框架 DIRTY 相比, DeGPT 采用了庞大数据集和模型参数, 能更好实现复杂反编译输出, 可读性提升表现显著。

在软件包漏洞管理方面, Peiyu Liu 等人[80]使用大规模数据集探索了 ChatGPT 在漏洞管理任务方面的能力, 并与 SOTA 方法进行比较。结果显示, 领域知识的总结可以提高 ChatGPT 在软件缺陷报告生成方面的性能, 但仍存在一些困难和挑战, 如有效引导 ChatGPT 关注有用信息等。在二进制污点分析方面, Liu P 等人[81]首次将大模型应用到二进制污点分析技术, 提出了名为 LATTE 的污点分析框架, 基于标准数据集 Juliet 中的漏洞测试用例来评估, 漏洞检测准确性和 F1 超过了 SOTA 方法 Emtaint 和 Arbiter。此外, 在测试用例的目标场景中, LATTE 识别安全敏感功能和外部输入源方面实现了 100%的正确覆盖率。此外, LATTE 在真实的二进制文件中检测到 119 个漏洞, 包括 37 个未知漏洞。

在模糊测试方面, Xia C S 等人[82]提出了一种 Fuzz4All 框架, 利用大模型来实现通用模糊测试的方法, 与传统 fuzzing 工具相比, 该框架能够针对多种输入语言和特性进行模糊测试, 通过生成多样化且真实的输入, 显著提高软件测试的覆盖率和发现漏洞的能力。通过 6 种流行编程语言和 9 种真实程序的实验表明, Fuzz4All 在模糊测试中达到了最高的覆盖率, 与不同维度的 SOTA 基线 GrayC、YARPGen、TypeFuzz、go-fuzz 相比, 平均提高了 36.8%, 并且检测到了 98 个错误, 其中 64 个被确认为未知错误。Qiu F 等人[83]提出了一种利用 LLMs 对二进制软件进行模糊测试的方法, 通过提供有价值的领域特定知识, 利用 LLMs 从种子输入中生成语法和语义有效的输入文件, 设计实现了全自动的模糊测试框架 CHEMFuzz, 检出了软件中 40 个缺陷。Oliinyk Y 等人[84]使用 LLM 生成针对性的初始化种子, 采用模糊测试技术检测 BusyBox 软件的潜在漏洞。实验结果表明, 与业内广泛使用的 AFL++工具生成的随机种子相比, LLM 产生的初始化种子在模糊过程中识别出了更多的路径, 发现了更多的崩溃, 有效提高了嵌入式系统的漏洞检测能力。

在补丁识别修复方面, Chen T 等人[85]提出了一种微调 StarCoder 大模型来识别漏洞补丁的方法 CompVPD, 通过删除不相关的文件、方法和语句, 并自适应地扩展每个上下文, 在给定的窗口大小内综合理解提交代码的上下文信息。实验结果显示, 与 SOTA 基线方法 StarCoder 相比, 该方法在识别漏洞补丁方面展现出了更高的有效性, AUC 指标提高了 11%, F1 指标提高了 30%, 并从 5 个高度流行项目最近提交的 2500 次代码中识别出了 20 个漏洞补丁和 18 个高风险漏洞。针对漏洞补丁种类有限, 无法修复复杂错误的问题, Xia C S 等人[86]的研究表明大模型在漏洞补丁技术上的应用已经优于传统的自动修复技术, 并且可以通过增加样本大小、合并修复模板信息等方式提高修复能力。与传统的和包含深度学

习在内的基于机器学习的自动程序修复方法相比, 直接应用 LLMs 在 Defects4J 数据集上进行自动程序修复时, Codex 模型无需进行任何特定更改或微调, 已经能够比现有基准获得最高的程序正确修复数。针对传统补丁修复技术的性能限制, Wang W 等人[87]利用 CodeT5 大模型构建了名为 RAP-Gen 的增强补丁生成框架, 采用语言无关的方式进行词汇解释和语义匹配, 可以灵活的集成不同补丁搜索和生成器来修复各种类型的错误, 将 SOTA 方法 T5-large 在 TFix 上的实验准确率从 49.70%提高到了 54.15%。此外, RAP-Gen 作为一个语言无关的模型, 具有更强的通用性, 能够处理其他基于 AST 或测试诊断的模型无法处理的代码片段和场景, 在自动程序修复方面更具优势。在恶意代码检测方面, 传统基于 API 的深度学习检测方法存在 API 表征质量有限、无法生成未知 API 调用表征、难以应对概念漂移的问题, Yan P 等人[88]提出了一种将 LLM 和卷积神经网络(Convolutional Neural Networks, CNN)相结合的方法, 首先引入 GPT-4 提示工程为 API 序列中的每个 API 调用生成可解释的文本, 再使用预训练 BERT 模型获取可解释的 API 序列表征, 最终构建一个基于 CNN 的模块来学习表征, 从而辅助恶意代码动态分析。实验结果表明, 与基于 RNN、CNN、CNN+RNN 和 Transformer 的 SOTA 方法相比, 该方法在数据集上的检测性能均有所提高, 利用在训练过程中整合的外部知识, 在一定程度上提高了模型的性能, 并且在小样本中展现出了较好的检测效果和模型泛化能力。

3.5 面向构建过程的防护机制研究

构建过程中主要存在不安全的代码生成、不安全的编译和优化、受威胁的编译组件、侧信道攻击等威胁要素。不安全的编译工具允许软件不受限制地访问内存空间[89], 包含缺陷的代码逻辑将导致内存破坏。S. Mahmud 等人[90]对支付服务提供商的 SDK 进行了安全评估, 发现众多 SDK 存在安全问题, 包括将加密的信用卡数据保存到文件, 使用不安全的口令等。Pearce H 等人[91]通过提示 Copilot 生成代码, 探索 Copilot 在弱点多样性、提示多样性和领域多样性方面的代码脆弱性表现, 发现约 40%的程序代码存在漏洞。Perry N 等人[92]进行了一项大规模研究, 分别观察用户与 AI 代码助手在不同编程语言下的代码安全性。该研究发现, 有 AI 助手参与编写的代码, 存在明显的安全问题。C. Abate 等人[93]指出编译工具存在受威胁的编译组件, 攻击者可以通过控制这些组件来发起攻击。由于执行 CI 任务的代码可能来自不受信任的用户, 配置不当或隔离机制较弱的 CI 可能使攻击者将恶意代码注入到软件中。Gu Y 等人[94]系统地研究了 CI 工作流程中的潜在安全威胁。F.Ullah 等人[95]的研究表明, 来自开发、测试、运维等不同团队的用户对 CD 管道上的资源存在相同的访问权限。Y. Lin 等人[96]的研究指出, 编译优化可能在识别函数参数时产生错误, 引发程序安全问题。Xu J 等人[97]发现编译器优化会对地址随机化、控制流完整性保护等代码重用攻击的防御产生负面影响。F. Besson 等人[98]的工作表明, 使用编译器进行程序转换时可能存在信息流泄漏, 当攻击者进行内存访问时, 通过观察缓存数据能够使目标程序比源程序更容易受到侧信道攻击。

针对构建过程存在的安全问题, A. El-Korashy 等人[99]提出了一种编译时的内存安全机制, 即使编译单元关联到攻击代码, 编译器仍可为编译单元提供额外的安全保障。针对构建平台编译时无法检查不安全语言未定义错误产生的未知运行风险, C. Abate 等人[93]提出了一种评估不安全语言的安全编译方案, 防止动态受损的组件破坏系统的安全属性。为了检测构建过程篡改源代码的问题, C. Lamb 等人[100]创新性地提出了可重现编译过程的方法, 基于源代码树构建时生成逐位相同的结果, 有效地避免了编译器层面的恶意代码植入问题。针对编译优化时由于保留某些非功能属性引入的信息流泄露问题, F. Besson 等人[98]提出了针对信息流保护的方法, 以确保源代码编译后不会提升侧信道攻击的风险。

传统方法在软件构建过程方面主要依赖于静态代码分析、规则匹配和模板生成等技术, 提供代码格式化、语法和语义检查功能, 基于关键词匹配和预定义代码模板, 为开发者提供有限的代码补全选项。构建过程的代码安全审计, 主要通过静态分析工具来检查代码中的潜在安全问题, 缺乏有效的检测机制来确保生成代码的质量和安全性, 还可能引入新的安全问题。深度学习方法可以从大量代码中学习语法、语义和上下文信息, 从而提供更智能的代码补全和安全提示。根据开发者输入的上下文信息, 预测并推荐可能的代码片段或函数, 通过学习安全漏洞的特征, 自动检测代码中的潜在安全问题。通过调研大模型在软件构建过程防护机制赋能方面的相关文献, 本文梳理总结了大模型在代码补全与生成、安全检测与评估、代码调试与修复、补丁验证等具体技术环节中的赋能现状, 如表 5 所示。

首先, 大模型学习了覆盖全网的高质量编程数据知识, 可以根据开发者的输入和上下文信息, 自

表 5 大模型赋能的构建过程防护机制

Table 5 Building process protection mechanisms empowered by LLMs

赋能对象	威胁要素	来源	大模型方法	赋能环节	数据集	年份
构建过程	①不安全编译和优化 ②受威胁的编译组件 ③不安全的代码生成 ④侧信道攻击	文献[101]	GPT、Codex	代码补全	HumanEval, AAPS	2021
		文献[102]	Codex、CodeBERT	代码生成检测	GitHub LGTM	2023
		文献[103]	Copilot	代码生成评估	自定义	2023
		文献[104]	Copilot、StarCoder、CodeLlama	代码生成评估	GitHub	2024
		文献[105]	GPT-J 6B	代码生成检测	自定义智能合约	2023
		文献[106]	Mistral、Llama2、GPT-3.5	编译器优化	Alive2	2024
		文献[107]	GPT-4	代码调试、检测与修复	未公开	2023
		文献[108]	GPT-4	漏洞验证	CVEs	2024
		文献[109]	InferFix	错误修复、补丁验证	InferredBugs	2023

动化生成高质量的代码, 提供更丰富、更精准的代码补全和推荐, 在VSCode等集成开发环境中被开发人员广泛应用, 用实际效果验证了大模型在该方面相较于传统方案的领先能力。其次, 大模型还可以结合自然语言处理技术, 实现更符合人类习惯的代码编辑和注释功能。此外, 借助已学习的安全知识和漏洞模式, 大模型在代码生成过程中能够对代码语义和安全性进行评估, 从而更准确地识别代码中的潜在安全问题, 为软件构建过程提供高效安全和智能化的全生命周期支持。

大模型在软件构建防护机制方面具有更深的数据提取能力, 能够理解代码、配置文件以及构建过程元素的特殊含义, 从而识别构建过程中不同编程语言的特定风险, 实现相对安全通用的软件构建机制。在代码补全方面, Chen M 等人[101]提出了名为 Codex 的代码生成模式, OpenAI 和 Microsoft 基于 Codex 合作开发了 Copilot 代码生成大模型, 用户在使用 Visual Studio Code、Microsoft Visual Studio 等集成开发环境时, 可通过注释、方法名称、上下文代码等信息自动补全代码。然而, 大模型生成的代码可能伴随着一些列安全问题。因此, 有学者研究大模型生成的代码质量, 以及如何微调大模型进而生成更加安全可靠的代码。Chan A 等人[102]使用易受攻击代码数据集对 LLM 进行预训练, 学习 250 多种漏洞类型的复杂表现, 当开发人员使用 EditTime 编写代码时检测易受攻击的代码, 相比擅长于各种编码任务的大模型Codex, 该方法以10%的参数量达到相同的精度和更好的召回率, 并将代码大模型的漏洞率降低 90%。Dakhel A M 等人[103]发现 Copilot 几乎可以为所有的代码算法提供解决方案, 并且这些方案与开发人员所能提出的最优方案性能相当, 但仍需要通过开发人员来修复其中的错误。Finkman A 等人[104]提出了一种将深度强化学习和大模型相结合的代码评估方法, 针对 LLM 代码助手存在的代码泄露风险, 在发送提示给助手服务前, 通过深度强化学习代理 CodeCloak 最小化代码泄露, 同时保留对开发者有用的建议, 具备跨模型迁移的能力。Storhaug A 等人[105]发现超过 70%的自动生成的以太坊智能合约代码存在安全问题, 通过使用漏洞代码标记数据集对大模型进行微调, 识别出62%的生成代码中存在漏洞, 并且成功避免了其中 67%的代码漏洞。

在编译器优化方面, Wang Y 等人[106]提出了一种将大模型应用于 LLVM 编译器转换的验证框架, 遇到无法确认转换合理性的情况时, 利用微调后的 LLMs 进行预测, 弥补了传统形式验证工具的不足, 并且在深度学习加速器设计等复杂应用场景中表现出了良好的效果, 未进行微调的 GPT-3.5 准确率仅为 50%, 而微调后的 GPT-3.5 准确率高达 88%, 解决了现有工具难以解决的问题。在单元测试方面, Auto-Dev 是一款开源、自动化的 AI 辅助编程工具, 支持自定义大模型, 采用代码静态分析构建上下文的技术, 在单元测试等场景构造更准确的测试结果。在漏洞修复方面, 安全研究人员 Mark Harbottle[107]指出, GPT-4 是一个强大的人工智能语言模型, 可以用来自动执行重复的编程任务, 修复代码中的错误, 完成 C++、C#、Python 等编程语言的代码调试。在代码文档注释生成方面, 大模型可以扮演代码编辑器的角色, 根据开发人员提供的高级指令或意图, 生成相应的代码, 帮助编写文档和生成注释, 改善代码的可读性。

在漏洞验证方面, 大模型作为一种语言模型, 应用于代码这种另一形式的语言时, 有着良好的表现。Fang R 等人[108]借助大模型强大的自主阅读和学习能力, 仅提供描述漏洞 CVE 公告, 便可以成功地利用并验证现实世界真实存在的安全漏洞, 且成功率高达 87%。在端到端自动化软件开发方面, Jin M

等人[109]针对软件开发生命周期中缺陷的引入、识别和解决等方面, 提出了一种基于 Transformer 的程序修复框架 InferFix, 通过增加语义类型注释和从外部检索语义信息细化程序修复过程。实验证明, InferFix 框架比 Codex、Davinci 等 LLM 基线方法表现更好, C#语言生成修复的 Top-1 准确率为 65.6%, Java 语言为 76.8%。同时, InferFix 与 Infer 整合, 提供了一种端到端的自动化软件开发框架。

4 大模型赋能软件供应链安全防护技术的挑战与展望

4.1 当前工作面临的挑战

4.1.1 大模型训练成本和资源消耗问题

大模型的训练成本主要体现在算力需求、人工及时间成本、存储空间等方面。首先, 训练大模型需要消耗大量的计算资源, 包括提供高性能计算的 GPU、TPU 设备。这些设备价格昂贵, 而且运行成本很高。其次, 训练大模型通常需要数据科学家、AI 工程师、开发人员等多方面的专业人员, 耗费大量的时间和精力, 这些人力和时间成本也是训练成本的重要组成部分。根据斯坦福大学发布的《2024 年人工智能指数报告》[110], OpenAI 的 GPT-4 等大模型的训练成本预估在 7800 万美元, 而谷歌的 Gemini Ultra 的成本花费预估为 1.91 亿美元。此外, 训练大模型需要大量的数据支撑, 这些数据会占用大量的存储资源。随着模型规模的增加, 所需的存储空间也会成倍增长, 这也会增加训练成本。

大模型的训练成本高昂, 对于很多机构和个人来说, 这样的资金压力是巨大的, 甚至可能超出他们的承受能力。此外, 长时间的训练也可能因为设备故障、数据问题等原因而中断, 导致训练失败。因此, 在实际应用中, 如何处理大规模数据, 优化模型的训练过程以提高训练效率, 以及如何保证模型的泛化能力以进一步提高模型的稳定性和可靠性, 这些技术挑战都需要进行深入的研究和探索。

4.1.2 大模型稳定性和可靠性问题

大模型在处理复杂的软件供应链安全数据时有着卓越的能力, 但其理解和分析方面的稳定性和可靠性仍面临一些挑战, 例如, 模型上下文长度的限制、模型幻觉等问题。大多数基于 Transformer 架构的模型, 受限于固定的上下文窗口长度, 直接影响了它们处理长文本时的性能。在处理长代码文件或文档时, 模型可能无法捕获全文的相关性和上下文依赖, 导致理解出现断层, 影响决策的有效性。虽然有技术试图通过分层注意力或递归技术来突破这一限制, 但这些方法往往增加了计算复杂度和资源需求。模型幻觉是指大模型在没有足够证据支持的情况下生成不准确或虚构的信息。例如, 在自动生成代码注释或文档时, 模型可能产生与实际代码逻辑不符的描述。这种不稳定的输出结果, 不仅会误导开发人员和安全分析师, 还可能引发错误的安全决策, 为软件供应链的安全风险带来负面影响。

为此, 研究者们正在积极探索各种方法和技术。例如, 通过改进模型的训练方法、优化模型的结构等来提高模型的稳定性和可靠性。然而, 解决大模型的稳定性和可靠性问题并非一蹴而就的过程, 需要不断深入探索、持续改进, 并结合实际应用场景进行针对性的优化, 以最大限度激发大模型的潜力。

4.1.3 大规模高质量数据集构建难

在安全领域大模型的构建过程中, 特定的安全威胁和场景呈现多样化的特征, 这使得构建一个具有完备性的高质量数据集成为迫切需求。软件供应链安全所涉及的数据规模庞大且种类繁多, 包括代码库、文档、补丁、第三方库等, 从文本到程序代码, 再到图像和文件等多种形式, 每一种数据类型都蕴含了软件供应链的关键信息。然而, 对这些数据的有效利用并非易事。它不仅仅依赖于丰富的专家经验来指导数据的筛选、预处理和标注, 更需要对各种安全威胁的本质和表现形式有深入的理解。这一过程并非简单的数据堆砌所能完成, 而是需要专家的精心的设计, 以确保所构建的数据集在质量和实用性上达到较高的水平。此外, 软件供应链中也存在跨平台、跨语言、跨架构的多元异构问题, 这无疑增加了数据集构建的复杂度。

因此, 构建适用于安全垂直领域大模型的高质量数据集是一项极具挑战性和复杂性的任务, 需要综合考虑数据规模、数据类型、安全威胁、专家经验以及多元异构等多个方面的问题。同时, 由于模型的训练数据可能存在偏差或不足, 也可能导致模型在特定场景下的性能下降。只有通过精准的设计和策略优化才能确保数据集的质量, 为安全垂直领域大模型的构建提供坚实的数据基础。

4.1.4 微调垂直领域大模型是系统工程

微调作为大模型构建中的关键环节, 在通用大模型的基础上, 通过针对特定任务数据集的精细调整与优化, 实现模型参数的有效适配, 从而使其更精准地满足软件供应链安全垂直领域的特定需求。RLHF 的微调过程本质上是一项高度复杂且任务繁重的系统工程, 需要投入大量的时间与人力资源。这一过程中, 不仅涉及大量数据的准备与预处理, 还

需对模型结构进行优化, 并针对特定的威胁模式进行训练。由于软件供应链安全涉及多种复杂的安全威胁, 每种威胁可能需要不同的检测技术, 这些技术的开发与应用依赖于深厚的专家经验和行业知识。此外, 大模型微调过程中还可能出现一系列未知的问题, 如灾难性遗忘、模型幻觉等, 这些问题都可能导致模型性能下降或产生误判。

总之, 微调垂直领域大模型使其适用于软件供应链安全场景是一项复杂的任务, 需要综合考虑多个方面的因素, 以确保模型的准确性和有效性。即便是最先进的大模型, 也需要与领域专家紧密协作, 共同探索更具先进性的解决思路, 以构建适用于软件供应链安全的垂直领域大模型, 实现对供应链安全威胁的精准识别与有效防御。

4.1.5 大模型引入新的供应链安全问题

尽管大模型在理解和生成复杂信息方面表现出强大的能力, 但其固有的复杂性和不透明性使得自身面临一系列安全挑战[111-115], 并且存在从开发、训练到部署应用全过程的供应链安全问题[116]。这些问题不仅关乎模型自身的安全性, 还引入了新的未知安全风险。首先是隐私问题, 模型通常需要训练大量数据, 其中包含了敏感信息。模型训练过程可能会无意中暴露或学习到数据中的敏感特征。此外, 模型在生成结果时可能会“回忆”训练数据中的细节, 进一步增加隐私泄露的风险。其次, 大模型可能因训练数据中的偏差、模型的使用不当或对抗性攻击, 产生误导性的输出或安全漏洞。攻击者可能会利用精心设计的提示词, 对大模型进行诱导攻击, 以产生恶意的输出结果。若大模型在开发过程中不慎使用了被恶意篡改的第三方库或依赖项, 这些潜藏的恶意代码会融入到最终模型中, 对使用该模型的软件或系统构成安全威胁。

由于复杂性和不透明性导致的安全问题, 给大模型在软件供应链安全领域的研究和应用带来了现实挑战。针对大模型存在的上述安全问题, 亟需采取一系列有效措施, 以确保模型的开发、训练和使用过程均符合安全标准, 保障用户数据的隐私和安全, 同时需要探索和研究提升模型自身可靠性和安全性的方法。

4.2 未来研究方向

结合上述研究现状, 可以总结出现阶段大模型在软件供应链安全中围绕顶级源、依赖项、软件包构件及其构建过程所面临的 5 种现实挑战, 以及未来可能的 5 个研究方向。

4.2.1 大模型赋能的软件工程开发范式研究

大模型技术的突破性进展和创新方法, 引领着软件工程开发范式迈入崭新的时代。以 GPT-4 为代表的大语言模型, 借助 RLHF 和多模态技术的融合, 展现出强大的图像识别能力, 显著提升了语义理解和回答的准确性, 并具备执行一系列复杂任务的能力, 包括代码生成、错误检测以及软件设计等。深入研究 GPT-4 及 GPT-5 在软件研发生命周期中的新技术新应用, 将彻底改变开发人员构建、维护和改进软件开发的传统范式。大模型能够自动化地理解研发人员所下发的任务, 并自主完成软件的各项开发工作, 如需求理解、用户界面生成、产品代码生成以及测试脚本生成等任务。

展望未来, 研发团队的主要工作将不再局限于编写代码和执行测试用例, 而是聚焦于大模型训练、参数调优以及围绕业务开展提示工程。大模型技术赋能于软件的需求分析、开发、测试、运维、项目管理等多个关键环节, 进而开启“软件工程 3.0”的新纪元。这一变革将极大地提升软件开发的效率和质量, 推动软件工程领域向更高的层次发展。

4.2.2 大模型赋能的攻击代码意图理解技术研究

针对第三方组件的生态脆弱性等安全挑战, 大模型不仅能够深入解析恶意代码的复杂行为, 还能自动化地生成针对特定攻击代码的分析报告, 对于快速准确地识别并响应软件供应链中的攻击威胁具有至关重要的作用。通过挖掘攻击代码的语义信息, 大模型能够理解其执行逻辑以及潜在的攻击意图, 这一强大能力来源于对海量恶意代码数据集的学习与训练, 使得模型能够敏锐地捕捉到微小的攻击模式和行为差异, 从而实现对恶意行为的精准识别。通过对代码行为特征的细致分析, 大模型还能进一步揭示攻击者的策略和战术, 如数据窃取、权限提升、检测逃避等技战术手法。

基于这些深入的分析, 大模型能够自动化地生成攻击意图分析报告, 为安全团队提供包括攻击载荷、攻击手法、潜在影响以及修复建议在内的全方位信息。这一过程不仅提高了攻击代码分析的效率和准确性, 还为安全团队提供了更加科学、系统的决策支持, 有助于加强软件供应链安全防护的能力。

4.2.3 基于小样本数据集的大模型调优技术研究

基于小样本数据集的大模型调优技术旨在攻克传统大模型训练中对海量标注数据的依赖瓶颈, 特别是在面临标注稀缺数据的问题, 该技术能有效提升模型的学习效率。针对软件供应链安全的特定需求, 我们不仅要在大模型的训练过程中寻求创新, 例如, 通过引入零样本学习、少样本学习、迁移学习、模型蒸馏等技术来优化模型在有限数据条件下的性

能, 还需深入探索大模型架构本身的优化策略, 使其具备更强的学习能力和对新威胁的适应能力。研究生成模型也是一种解决小样本问题的手段, 通过学习数据的分布, 生成更多的样本数据, 从而扩充原始样本数据集规模, 提高模型的泛化能力。

此外, 融合其他小模型技术也是大模型调优的有效途径。通过集成针对特定安全威胁而设计的小模型, 利用其在特定领域的专业优势, 结合不同模型的分析结果, 形成多策略融合的解决方案。这种策略在保持系统总体效能的同时, 能够显著增强对未知攻击的识别和检测能力, 为软件供应链安全提供更加全面和高效的保障。

4.2.4 基于思维链的软件供应链安全防护技术

思维链提示, 作为提示工程中的一种前沿方法, 旨在引导大模型进行有效的推理与问题解决。利用大模型进行思维链推理的关键优势在于其能够处理和分析海量的数据, 从而提供更为深入且全面的安全见解。在软件供应链安全防护的场景下, 基于思维链推理的技术通过精心设计的提示词, 牵引大模型深入剖析潜在的威胁、行为模式及安全漏洞。这一过程中, 大模型所具备的强大数据处理能力和自然语言深度理解能力能够得到充分利用, 从而适应软件供应链多样化的安全需求, 包括对代码、配置及依赖项等的综合安全评估。

具体而言, 通过对历史安全事件数据的深入分析, 基于提示词的方法能够引导大模型专注于软件供应链安全的特定问题, 进而提升其分析效率。这不仅有助于识别出新的攻击模式, 更能在软件供应链安全的初期阶段预测潜在的安全威胁, 为及时采取防范措施提供有力支持。基于思维链推理的安全防护技术为软件供应链安全防护提供了新的思路, 有望为该行业的发展提供坚实的技术保障。

4.2.5 基于提示词注入防范的大模型安全研究

随着大模型在软件供应链安全场景的应用研究日益深入, 确保模型的安全性和可靠性成为亟待解决的关键问题。大模型在各类下游任务中展现出卓越的推理能力, 使其成为潜在的攻击目标。在基于提示词的大模型注入攻击中, 当训练数据来源不可靠或含有恶意植入的内容时, 模型在训练过程中可能学习到不良行为或安全漏洞, 从而对其性能和安全性产生负面影响。

在 LLMs 应用程序 OWASP Top 10 中, 提示词注入攻击位居 LLMs 安全风险的榜首, 为了应对这一挑战, 我们不仅需要深入理解大模型如何处理和响应提示词, 还需深入探究如何有效防御基于提示词注入的攻击。基于提示词注入的安全防护方法核心在于研究能够精准识别和拦截异常提示词的安全机制。同时, 还需要对大模型的响应内容进行实时监控和分析, 以便在攻击发生的初始阶段迅速识别并采取响应措施。通过这种方法, 我们可以提高大模型在供应链安全应用中的可靠性和安全性, 为软件供应链的安全防护提供更为坚实的能力支撑。

5 总结

随着信息技术的日新月异, 软件功能的日益复杂化、模块化和产业化, 软件供应链安全问题随之产生, 成为软件产业从新兴迈向成熟的必由之路。近年来, 软件供应链安全事件层出不穷, 其严重性已引起学术界和产业界的广泛关注。本文聚焦于大模型在开发环节软件供应链安全防护中的赋能技术, 对其进行研究和综述。首先, 本文阐述了大模型及软件供应链安全的相关概念, 梳理了近期发生的典型软件供应链安全事件, 回顾了开发过程中软件供应链安全防护技术的发展脉络。随后, 介绍了通用领域大模型及其在软件供应链安全中的应用趋势, 为后续研究提供理论支撑和实践参考。通过深入调研该领域的相关研究成果, 本文从软件开发环节的 17 种威胁要素出发, 围绕顶级源的代码检测技术、依赖项的威胁分析技术、软件包的安全检测技术、构建过程的防护机制四个方面, 系统性地总结了大模型在软件供应链安全防护方面的研究现状。在此基础上, 本文进一步提出了当前研究所面临的五种现实挑战, 旨在为后续研究提供方向指引。最后, 结合当前研究现状和现实挑战, 本文对未来大模型赋能软件供应链安全的研究进行了展望。

大模型的综合能力表现是单个垂直领域的深度学习模型难以超越的, 并且能够为软件开发提供全生命周期的赋能支撑。传统方法基于已知问题的认知建模, 面对持续进化的安全威胁时, 其局限性已成为业内共识。深度学习能从大量数据中学习特征, 具有更强学习能力, 但可解释性差且需大量标注数据。随着软件开发的复杂化, 传统模型难以处理庞大的数据和适应多种任务场景。相比于垂直领域的深度学习模型而言, 大模型在处理软件供应链安全防护中的复杂性和多任务需求上更具优势。大模型融合了全网多领域知识, 能够高效处理庞大代码数据, 并具备深层特征和规律的学习能力, 其强大的上下文理解和推理能力使其适应性强, 能应对快速变化的安全威胁。大模型更强的泛化能力和迭代效率提升了其多任务学习和处理的能力, 能够主动学习安

全领域知识而无需人工标注, 在开发过程中辅助开发人员生成代码, 编写代码注释和文档, 自动检测和发现软件安全问题, 定制漏洞补丁进行漏洞修复等, 在软件开发全流程中显著降低了对专家的依赖。

与此同时, 大模型也存在一些现有的局限性。首先, 由于参数量巨大, 大模型通常需要大量的计算资源和时间进行训练和推理, 这增加了训练成本和部署难度。其次, 大模型的稳定性和可靠性问题也面临着现实挑战, 如训练过程中的不稳定性、推理阶段的可靠性等问题。此外, 大模型往往缺乏透明度, 这使得用户对其预测结果产生质疑, 也限制了在某些领域的应用。相比之下, 深度学习模型和小模型在某些方面具有优势。小模型具有参数量少、计算量小、易于部署等优点, 适用于资源受限的环境和简单的任务。此外, 小模型在处理特定任务时可能更加高效和准确, 因为它们可以针对特定任务进行优化和定制。因此, 现阶段大模型并不能完全取代深度学习模型或其他小模型。

综上所述, 在实际应用中, 应根据具体任务和数据特点选择合适的模型。对于大规模数据集和复杂任务场景, 大模型更具优势; 而对于资源受限的环境或简单任务场景, 深度学习模型或小模型可能更加适用。此外, 不同模型之间也可以进行融合和协作, 以充分利用各自的优势, 提高整体性能。大模型在解决复杂软件供应链安全防护场景的任务中具有相对传统方法而言更大的潜力和应用价值, 未来的研究可以围绕软件工程开发范式、攻击代码意图理解、基于小样本数据集的大模型调优、基于思维链和提示词注入防范等关键技术开展工作。首先, 大模型赋能的软件工程开发范式和攻击代码意图理解将成为研究热点, 能够为软件供应链安全防护的技术和方法提供新的思路。其次, 基于小样本数据集的大模型调优技术有助于提高大模型在有限数据条件下的性能, 降低对大量标注数据的依赖。此外, 基于思维链以及基于提示词注入防范的大模型安全防护技术也将成为未来可能的研究方向, 有望为软件供应链安全防护提供更加全面有效的保障措施。

致　谢　本课题得到中国科学院青年创新促进会; 中国科学院网络测评技术重点实验室; 网络安全防护技术北京市重点实验室项目资助; 国家电网有限公司科技项目资助(5700-202352606A-3-2-ZN)。

参考文献

[1] Addressing Cybersecurity Challenges in Open Source Software. Synk. https://resources.snyk.io/state-of-open-source-security-report-2022. Jul. 2022.

[2] 8th Annual State of the Software Supply Chain. Sonatype. https://www.sonatype.com/state-of-the-software-supply-chain/open-source-dependency-management-trends-and-recommendations. Oct. 2022.

[3] Open Source Security and Risk Analysis 2023. Synopsys. https://www.gcomtw.com/mailshot/Synopsys/2303BlackDuck/reposra2023ch.pdf. Mar. 2023.

[4] Gartner Identifies Top Security and Risk Management Trends for 2022. Gartner.https://www.gartner.com/en/newsroom/press-releases/2022-03-07-gartner-identifies-top-security-and-risk-management-trends-for-2022. Mar. 2022.

[5] Biden-Harris Administration Launches Artificial Intelligence Cyber Challenge to Protect America's Critical Software. Whitehouse. https://www.whitehouse.gov/briefing-room/statements-releases/2023/08/09/biden-harris-administration-launches-artificial-intelligence-cyber-challenge-to-protect-americas-critical-software. Aug. 2023.

[6] Divakaran D M, Peddinti S T. LLMS for Cyber Security: New Opportunities[EB/OL]. 2024: 2404.11338. https://arxiv.org/abs/2404.11338v1.

[7] SWE-agent. Github. https://github.com/princeton-nlp/swe-agent. Apr. 2024.

[8] Safeguarding artifact integrity across any software supply chain. SLSA. https://slsa.dev. 2023.

[9] Radford A, Narasimhan K, Salimans T, et al. Improving language understanding by generative pre-training[J]. *OpenAI*, 2018.

[10] Elman J. Finding Structure in Time[J]. *Cognitive Science*, 1990, 14(2): 179-211.

[11] An Automatic Evaluator for Instruction-following Language Models. Github. https://tatsu-lab.github.io/alpaca_eval. 2024.

[12] Security: Large Language Model. Github. https://github.com/Clouditera/secgpt. 2024.

[13] 360 officially releases the industry's first deliverable security big model application. 360. https://qycloud.360.cn/news/4138.html. Aug. 2023.

[14] Security GPT Detection. Sangfor. https://support.sangfor.com.cn/productDocument/read?product_id=141&version_id=736&category_id=273911. Dec. 2023.

[15] SecBench: Large Language Model Evaluation. Secbench. https://secbench.org. 2024.

[16] He X X, Zhang Y Q, Liu Q X. Survey of Software Supply Chain Security[J]. *Journal of Cyber Security*, 2020, 5(1): 57-73.
(何熙巽, 张玉清, 刘奇旭. 软件供应链安全综述[J]. *信息安全学报*, 2020, 5(1): 57-73.)

[17] Ji S L, Wang Q Y, Chen A Y, et al. Survey on Open-Source Software Supply Chain Security[J]. *Journal of Software*, 2023, 34(3): 1330-1364.
(纪守领, 王琴应, 陈安莹, 等. 开源软件供应链安全研究综述

[J]. *软件学报*, 2023, 34(3): 1330-1364.)

[18] Gao K, He H, Xie B, et al. Survey on Open Source Software Supply Chains[J]. *Journal of Software*, 2024, 35(2): 581-603.
(高恺, 何昊, 谢冰, 等. 开源软件供应链研究综述[J]. *软件学报*, 2024, 35(2): 581-603.)

[19] SWE-bench technical report. https://www.cognition-labs.com/post/swe-bench-technical-report. Mar. 2024.

[20] Thompson K. Reflections on Trusting Trust[J]. *Communications of the ACM*, 1984, 27(8): 761-763.

[21] Dependency Confusion: How I Hacked into Apple, Microsoft and Dozens of Other Companies. Medium. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610. 2024.

[22] Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims with SUNBURST Backdoor. Mandiant. https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor. 2024.

[23] Dai Q M, Mao R F, Huang H, et al. DevSecOps: Exploring Practices of Realizing Continuous Security in DevOps[J]. *Journal of Software*, 2021, 32(10): 3014-3035.
(戴启铭, 毛润丰, 黄璜, 等. DevSecOps: DevOps 下实现持续安全的实践探索[J]. *软件学报*, 2021, 32(10): 3014-3035.)

[24] Singla T, Anandayuvaraj D, Kalu K G, et al. An Empirical Study on Using Large Language Models to Analyze Software Supply Chain Security Failures[EB/OL]. 2023: 2308.04898. https://arxiv.org/abs/2308.04898v1.

[25] Huang L H, Zhao P Z, Chen H M, et al. Large Language Models Based Fuzzing Techniques: A Survey[EB/OL]. 2024: 2402.00350. https://arxiv.org/abs/2402.00350v2.

[26] Wu F Z, Zhang Q Z, Bajaj A P, et al. Exploring the Limits of ChatGPT in Software Security Applications[EB/OL]. 2023: 2312.05275. https://arxiv.org/abs/2312.05275v1.

[27] Gao Z Y, Wang H, Zhou Y C, et al. How far Have we Gone in Vulnerability Detection Using Large Language Models[EB/OL]. 2023: 2311.12420. https://arxiv.org/abs/2311.12420v3.

[28] ChatGPT AI in Security Testing: Opportunities and Challenges. Cyfirma. https://www.cyfirma.com/outofband/chatgpt-ai-in-security-testing-opportunities-and-challenges. 2024.

[29] Shen L. Research on software tracking method based on code version control[D]. Wuhan: Wuhan University of Technology, 2018.
(沈力. 基于代码版本控制的软件跟踪方法研究[D]. 武汉: 武汉理工大学, 2018.)

[30] Wermke D, Wöhler N, Klemmer J H, et al. Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects[C]. *2022 IEEE Symposium on Security and Privacy* (*SP*), 2022: 1880-1896.

[31] Ladisa P, Plate H, Martinez M, et al. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains[C]. *2023 IEEE Symposium on Security and Privacy*, 2023: 1509-1526.

[32] Fischer F, Böttinger K, Xiao H, et al. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security[C]. *2017 IEEE Symposium on Security and Privacy*, 2017: 121-136.

[33] Tang G G, Yang L, Ren S Y, et al. An Automatic Source Code Vulnerability Detection Approach Based on KELM[J]. *Security and Communication Networks*, 2021, 2021: 5566423.

[34] Li X Z, Feng B W, Li G F, et al. A Vulnerability Detection System Based on Fusion of Assembly Code and Source Code[J]. *Security and Communication Networks*, 2021, 2021: 9997641.

[35] Wang H T, Ye G X, Tang Z Y, et al. Combining Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection[J]. *IEEE Transactions on Information Forensics and Security*, 2021, 16: 1943-1958.

[36] Tony C, Mutas M, Ferreyra N E D, et al. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations[C]. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories*, 2023: 588-592.

[37] Zhang C Y, Liu H, Zeng J T, et al. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT[EB/OL]. 2023: 2308.12697. https://arxiv.org/abs/2308.12697v2.

[38] Li H N, Hao Y, Zhai Y Z, et al. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models[EB/OL]. 2023: 2308.00245. https://arxiv.org/abs/2308.00245v3.

[39] Kang S, Yoon J, Yoo S. Large Language Models Are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction[C]. *2023 IEEE/ACM 45th International Conference on Software Engineering*, 2023: 2312-2323.

[40] Chen Y Z, Ding Z J, Alowain L, et al. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection[C]. *The 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023: 654-668.

[41] Ferrag M A, Battah A, Tihanyi N, et al. SecureFalcon: Are we there yet in Automated Software Vulnerability Detection with LLMS? [EB/OL]. 2023: 2307.06616. https://arxiv.org/abs/2307.06616v2.

[42] Sun Y Q, Wu D Y, Xue Y, et al. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMS' Vulnerability Reasoning[EB/OL]. 2024: 2401.16185. https://arxiv.org/abs/2401.16185v1.

[43] Islam N T, Karkevandi M B, Najafirad P. Code Security Vulnerability Repair Using Reinforcement Learning with Large Language Models[EB/OL]. 2024: 2401.07031. https://arxiv.org/abs/2401.07031v2.

[44] Shou C F, Liu J, Lu D D, et al. LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models[EB/OL]. 2024: 2401.11108. https://arxiv.org/abs/2401.11108v1.

[45] Eom J, Jeong S, Kwon T. CovRL: Fuzzing JavaScript Engines with Coverage-Guided Reinforcement Learning for LLM-Based Mutation[EB/OL]. 2024: 2402.12222. https://arxiv.org/abs/2402.12222v1.

[46] Hu J, Zhang Q, Yin H. Augmenting Greybox Fuzzing with Generative AI[EB/OL]. 2023: 2306.06782. https://arxiv.org/abs/2306.06782v1.

[47] Xiao Y, Chen B, Yu C, et al. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures[C]. *USENIX Security Symposium*. 2020: 1165-1182.

[48] Duan R A, Bijlani A, Ji Y, et al. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries[C]. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[49] Derr E, Bugiel S, Fahl S, et al. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android[C]. *The 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 2187-2200.

[50] He Y Z, Yang X J, Hu B H, et al. Dynamic Privacy Leakage Analysis of Android Third-Party Libraries[J]. *Journal of Information Security and Applications*, 2019, 46: 259-270.

[51] Analysis of large-scale attacks and abuse events on PyPI software sources.Qianxin. https://tianwen.qianxin.com/blog/2024/03/29/batch_typosquatting. Mar. 2024.

[52] Liang G P, Zhou X Y, Wang Q Y, et al. Malicious Packages Lurking in User-Friendly Python Package Index[C]. *2021 IEEE 20th International Conference on Trust*, *Security and Privacy in Computing and Communications*, 2021: 606-613.

[53] Li Q, Song J K, Tan D W, et al. PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities[C]. *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2021: 161-173.

[54] Yang K, Deng Y, Zhang C, et al. ICUFuzzer: Fuzzing ICU Library for Exploitable Bugs in Multiple Software[M]. Developments in Language Theory. Cham: Springer International Publishing, 2018: 67-84.

[55] Deng Y L, Xia C S, Peng H R, et al. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries *via* Large Language Models[C]. *The 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023: 423-435.

[56] Deng Y L, Xia C S, Yang C Y, et al. Large Language Models Are Edge-Case Fuzzers: Testing Deep Learning Libraries *via* FuzzGPT [EB/OL]. 2023: 2304.02014. https://arxiv.org/abs/2304.02014v1.

[57] Chen T, Li L, Zhu L, et al. Vullibgen: Identifying vulnerable third-party libraries via generative pre-trained model[EB/OL]. 2023: ArXiv Preprint ArXiv:2308.04662.

[58] AI-Powered Fuzzing: Breaking the Bug Hunting Barrier. Google Security Blog. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html, Aug. 2023.

[59] Study of Zero-Shot Fuzz Driver Generation. Google. https://sites.google.com/view/llm4fdg/home. 2024.

[60] Zhang C, Bai M Q, Zheng Y W, et al. Understanding Large Language Model Based Fuzz Driver Generation[EB/OL]. 2023: 2307.12469. https://arxiv.org/abs/2307.12469v4

[61] Lyu Y L, Xie Y X, Chen P, et al. Prompt Fuzzing for Fuzz Driver Generation[EB/OL]. 2023: 2312.17677. https://arxiv.org/abs/2312. 17677v2

[62] Li C R, Chen X, Wang D R, et al. Backdoor Attack on Machine Learning Based Android Malware Detectors[J]. *IEEE Transactions on Dependable and Secure Computing*, 2022, 19(5): 3357-3370.

[63] Shi Y, Zhang Y, Luo T, et al. Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches[C]. *31st USENIX Security Symposium,* 2022: 1993-2010.

[64] Bolat A, Çelik S H, Olgun A, et al. ERIC: An Efficient and Practical Software Obfuscation Framework[C]. *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2022: 466-474.

[65] Ma H Y, Li S J, Gao D B, et al. Secure Repackage-Proofing Framework for Android Apps Using Collatz Conjecture[J]. *IEEE Transactions on Dependable and Secure Computing*, 2022, 19(5): 3271-3285.

[66] Merlo A, Ruggia A, Sciolla L, et al. You shall Not Repackage! Demystifying Anti-Repackaging on Android[J]. *Computers & Security*, 2021, 103: 102181.

[67] Demetrio L, Coull S E, Biggio B, et al. Adversarial EXEmples[J]. *ACM Transactions on Privacy and Security*, 2021, 24(4): 1-31.

[68] Chen X, Li C R, Wang D R, et al. Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection[J]. *IEEE Transactions on Information Forensics and Security*, 1001, 15: 987-1001.

[69] Shi L M, Ming J, Fu J M, et al. VAHunt: Warding off New Repackaged Android Malware in App-Virtualization's Clothing[C]. *The 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020: 535-549.

[70] Wang Y, Jia P, Peng X, et al. BinVulDet: Detecting Vulnerability in Binary Program *via* Decompiled Pseudo Code and BiLSTM-Attention[J]. *Computers & Security*, 2023, 125: 103023.

[71] Wang S, Wang X D, Sun K, et al. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics[C]. *2023 IEEE Symposium on Security and Privacy*, 2023: 2409-2426.

[72] Chen Y H, Zhang D L, Wang R W, et al. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64[C]. *2017 IEEE Symposium on Security and Privacy*, 2017: 304-319.

[73] Demicco D, Erinfolami R, Prakash A. Program Obfuscation *via* ABI Debiasing[C]. *Annual Computer Security Applications Con-*

ference, 2021: 146-157.

[74] Jerbi M, Dagdia Z C, Bechikh S, et al. On the Use of Artificial Malicious Patterns for Android Malware Detection[J]. *Computers & Security*, 2020, 92: 101743.

[75] BinaryAI. BinaryAI. https://www.binaryai.cn. 2023.

[76] Diaphora. Github. https://github.com/joxeankoret/diaphora. 2023.

[77] Tan H Z, Luo Q, Li J, et al. LLM4Decompile: Decompiling Binary Code with Large Language Models[EB/OL]. 2024: 2403.05286. https://arxiv.org/abs/2403.05286v2.

[78] Shang X W, Cheng S Y, Chen G Q, et al. How far Have we Gone in Stripped Binary Code Understanding Using Large Language Models[EB/OL]. 2024: 2404.09836. https://arxiv.org/abs/2404.09836v2.

[79] Hu P, Liang R, Chen K. DeGPT: Optimizing Decompiler Output with LLM[C]. *NDSS*. 2024.

[80] Liu P, Liu J, Fu L, et al. How ChatGPT is Solving Vulnerability Management Problem[EB/OL]. 2023: ArXiv Preprint ArXiv:2311.06530.

[81] Liu P Z, Sun C N, Zheng Y W, et al. Harnessing the Power of LLM to Support Binary Taint Analysis[EB/OL]. 2023: 2310.08275. https://arxiv.org/abs/2310.08275v1.

[82] Xia C S, Paltenghi M, Tian J L, et al. Fuzz4All: Universal Fuzzing with Large Language Models[EB/OL]. 2023: 2308.04748. https://arxiv.org/abs/2308.04748v2.

[83] Qiu F, Ji P, Hua B J, et al. CHEMFUZZ: Large Language Models-Assisted Fuzzing for Quantum Chemistry Software Bug Detection[C]. *2023 IEEE 23rd International Conference on Software Quality*, *Reliability*, *and Security Companion*, 2023: 103-112.

[84] Asmita, Oliinyk Y, Scott M, et al. Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing[EB/OL]. 2024: 2403.03897. https://arxiv.org/abs/2403.03897v1.

[85] Chen T, Li L, Qian T, et al. Identifying Vulnerability Patches by Comprehending Code Commits with Comprehensive Change Contexts[EB/OL]. 2023: ArXiv Preprint ArXiv:2310.02530.

[86] Xia C S, Wei Y X, Zhang L M. Automated Program Repair in the Era of Large Pre-Trained Language Models[C]. *2023 IEEE/ACM 45th International Conference on Software Engineering*, 2023: 1482-1494.

[87] Wang W S, Wang Y, Joty S, et al. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair[C]. *The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023: 146-158.

[88] Yan P, Tan S Q, Wang M H, et al. Prompt Engineering-Assisted Malware Dynamic Analysis Using GPT-4[EB/OL]. 2023: 2312.08317. https://arxiv.org/abs/2312.08317v1.

[89] Papaevripides M, Athanasopoulos E. Exploiting Mixed Binaries[J]. *ACM Transactions on Privacy and Security*, 2021, 24(2): 1-29.

[90] Mahmud S Y, English K V, Thorn S, et al. Analysis of Payment Service Provider SDKS in Android[C]. *The 38th Annual Computer Security Applications Conference*, 2022: 576-590.

[91] Pearce H, Tan B, Ahmad B, et al. Examining Zero-Shot Vulnerability Repair with Large Language Models[C]. *2023 IEEE Symposium on Security and Privacy*, 2023: 2339-2356.

[92] Perry N, Srivastava M, Kumar D, et al. Do Users Write More Insecure Code with AI Assistants? [C]. *The 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023: 2785-2799.

[93] Abate C, Azevedo de Amorim A, Blanco R, et al. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 1351-1368.

[94] Gu Y C, Ying L Y, Chai H J, et al. Continuous Intrusion: Characterizing the Security of Continuous Integration Services[C]. *2023 IEEE Symposium on Security and Privacy*, 2023: 1561-1577.

[95] Ullah F, Raft A J, Shahin M, et al. Security Support in Continuous Deployment Pipeline[EB/OL]. 2017: 1703.04277. https://arxiv.org/abs/1703.04277v1.

[96] Lin Y, Gao D B. When Function Signature Recovery Meets Compiler Optimization[C]. *2021 IEEE Symposium on Security and Privacy*, 2021: 36-52.

[97] Xu J H, Di Bartolomeo L, Toffalini F, et al. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches[C]. *2023 IEEE Symposium on Security and Privacy*, 2023: 1271-1288.

[98] Besson F, Dang A, Jensen T. Information-Flow Preservation in Compiler Optimisations[C]. *2019 IEEE 32nd Computer Security Foundations Symposium*, 2019: 230-23012.

[99] El-Korashy A, Tsampas S, Patrignani M, et al. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-As-Capabilities Principle[C]. *2021 IEEE 34th Computer Security Foundations Symposium*, 2021: 1-16.

[100] Lamb C, Zacchiroli S. Reproducible Builds: Increasing the Integrity of Software Supply Chains[J]. *IEEE Software*, 2022, 39(2): 62-70.

[101] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code[EB/OL]. 2021: ArXiv Preprint ArXiv:2107.03374.

[102] Chan A, Kharkar A, Moghaddam R Z, et al. Transformer-Based Vulnerability Detection in Code at EditTime: Zero-Shot, Few-Shot, or Fine-Tuning? [EB/OL]. 2023: 2306.01754. https://arxiv.org/abs/2306.01754v1.

[103] Moradi Dakhel A, Majdinasab V, Nikanjam A, et al. GitHub Copilot AI Pair Programmer: Asset or Liability?[J]. *Journal of Systems and Software*, 2023, 203: 111734.

[104] Finkman A, Bar-Kochva E, Shapira A, et al. CodeCloak: A Method

for Evaluating and Mitigating Code Leakage by LLM Code Assistants[EB/OL]. 2024: 2404.09066. https://arxiv.org/abs/2404.09066v1.

[105] Storhaug A, Li J Y, Hu T Y. Efficient Avoidance of Vulnerabilities in Auto-Completed Smart Contract Code Using Vulnerability-Constrained Decoding[C]. *2023 IEEE 34th International Symposium on Software Reliability Engineering*, 2023: 683-693.

[106] Wang Y Z, Xie F. Enhancing Translation Validation of Compiler Transformations with Large Language Models[EB/OL]. 2024: 2401.16797. https://arxiv.org/abs/2401.16797v2.

[107] Leverage GPT-4 for Debugging and Bug Fixes. Sitepoint. https://www.sitepoint.com/gpt-4-for-debugging. March 25, 2023.

[108] Fang R, Bindu R, Gupta A, et al. LLM Agents Can Autonomously Exploit One-Day Vulnerabilities[EB/OL]. 2024: 2404.08144. https://arxiv.org/abs/2404.08144v2.

[109] Jin M, Shahriar S, Tufano M, et al. InferFix: End-to-End Program Repair with LLMS[C]. *The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023: 1646-1656.

[110] Artificial Intelligence Index Report 2024. Stanford University. https://aiindex.stanford.edu/wp-content/uploads/2024/04/HAI_AI-Index-Report-2024.pdf.

[111] Deng B Y, Wang W J, Feng F L, et al. Attack Prompt Generation for Red Teaming and Defending Large Language Models[EB/OL]. 2023: 2310.12505. https://arxiv.org/abs/2310.12505v1.

[112] Yu J H, Lin X W, Yu Z, et al. GPTFUZZER: Red Teaming Large Language Models with Auto-Generated Jailbreak Prompts[EB/OL]. 2023: 2309.10253. https://arxiv.org/abs/2309.10253v4.

[113] Xu Z E, Zhang J P, Cui S W, et al. TroubleLLM: Align to Red Team Expert[EB/OL]. 2024: 2403.00829. https://arxiv.org/abs/2403.00829v1.

[114] Wei A, Haghtalab N, Steinhardt J. Jailbroken: How does llm safety training fail?[J]. *Advances in Neural Information Processing Systems*, 2024: 36.

[115] Greshake K, Abdelnabi S, Mishra S, et al. Not What You've Signed up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection[C]. *The 16th ACM Workshop on Artificial Intelligence and Security*, 2023: 79-90.

[116] Wang S N, Zhao Y J, Hou X Y, et al. Large Language Model Supply Chain: A Research Agenda[EB/OL]. 2024: 2404.12736. https://arxiv.org/abs/2404.12736v1.

刘井强 于 2017 年在哈尔滨工业大学计算机技术专业获得硕士学位。现在中国科学院大学网络空间安全专业攻读博士学位。主要研究方向为软件供应链安全、网络空间安全大模型、网络攻防技术。Email: liujingqiang@iie.ac.cn

田星 于 2022 年在青岛大学信息安全专业获得学士学位。现在中国科学院大学网络与信息安全专业攻读硕士学位。主要研究方向为软件供应链安全、网络威胁追踪溯源。Email: tianxing@iie.ac.cn

舒钰淇 于 2022 年在东南大学网络空间安全专业获得学士学位。现在中国科学院大学网络空间安全专业攻读硕士学位。研究领域为软件供应链安全、Web 安全。Email: shuyuqi@iie.ac.cn

朱小溪 于 2020 年在悉尼大学网络与分布式系统专业取得硕士学位。现任中国科学院信息工程研究所工程师。主要研究方向为网络空间安全大模型、网络安全评测。Email: zhuxiaoxi@iie.ac.cn

刘玉岭 于 2013 年在中国科学院软件研究所获得博士学位。现任中国科学院信息工程研究所正高级工程师、CCF 会员。研究领域为网络空间安全大模型、网络安全态势感知、网安大数据分析、安全测评认证等。Email: liuyuling@iie.ac.cn

刘奇旭 于 2011 年在中国科学院研究生院信息安全专业获得博士学位。现任中国科学院信息工程研究所研究员、中国科学院大学网络空间安全学院教授、CCF 会员。主要研究方向为网络攻防技术、网络安全评测。Email: liuqixu@iie.ac.cn