软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(6):2370−2389 [doi: 10.13328/j.cnki.jos.007618] [CSTR: 32375.14.jos.007618]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

MARC: 基于多智能体协同的硬件安全缺陷早期检测方法*

芮志清[1,2], 凌 祥[1], 曹方泽[1,2], 罗天悦[1], 吴敬征[1]

[1](中国科学院 软件研究所 智能软件研究中心, 北京 100190)

[2](中国科学院大学, 北京 100049)

通信作者: 吴敬征, E-mail: jingzheng08@iscas.ac.cn

摘 要: 随着开源 RISC-V 架构的迅猛发展, 其开放与模块化的特性在催生繁荣硬件生态的同时, 也给硬件电路的安全性保障带来巨大挑战. 在硬件设计流程的前期进行安全缺陷早期检测, 能够以最低成本在缺陷固化于物理芯片前将其消除. 尽管静态分析已用于硬件安全缺陷早期检测, 但由于规约知识未充分利用以及代码上下文语义理解不足, 现有检测方法存在高漏报率和高误报率的问题. 针对这些问题, 提出 MARC, 一种基于大语言模型多智能体协同的硬件安全缺陷早期检测方法. 该方法通过构建设计依赖分析、文档分析、安全缺陷检测、安全缺陷确认这 4 类智能体及协同工作框架, 从补充跨模块上下文、结构化模块文档知识、快速初筛安全缺陷、深度推理安全缺陷风险分析等多维度协同作用, 有效降低硬件电路设计阶段早期安全缺陷检测的误报率, 提升检测准确性. 实验结果显示, MARC 方法在工业级数据集上将早期缺陷检测的漏报率和误报率分别降低至 0.3829 和 0.3695, 相较于基准方法分别降低了约 18.2% 和 19.1%. 上述实验结果充分表明, MARC 有效缓解了硬件安全缺陷早期检测中的误报问题, 提升了安全缺陷检测的准确性与效率, 为硬件安全提供了更可靠的技术支撑. 另外, 还支撑作者团队夺得 HACK@DATE 2025 硬件漏洞挖掘竞赛全球冠军, 成功挖掘了 1 个获 CVE 编号的硬件漏洞, 在真实世界中验证了其有效性.

关键词: 硬件安全; 寄存器传输级; 安全缺陷检测; 大语言模型; 多智能体系统

中图法分类号: TP311

中文引用格式: 芮志清, 凌祥, 曹方泽, 罗天悦, 吴敬征. MARC: 基于多智能体协同的硬件安全缺陷早期检测方法. 软件学报, 2026, 37(6): 2370–2389. http://www.jos.org.cn/1000-9825/7618.htm

英文引用格式: Rui ZQ, Ling X, Cao FZ, Luo TY, Wu JZ. MARC: Multi-agent Collaborative Approach for Early Detection of Hardware Security Vulnerabilities. Ruan Jian Xue Bao/Journal of Software, 2026, 37(6): 2370–2389 (in Chinese). http://www.jos.org.cn/1000-9825/7618.htm

MARC: Multi-agent Collaborative Approach for Early Detection of Hardware Security Vulnerabilities

RUI Zhi-Qing[1,2], LING Xiang[1], CAO Fang-Ze[1,2], LUO Tian-Yue[1], WU Jing-Zheng[1]

[1](Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: With the rapid proliferation of the open-source RISC-V architecture, its openness and modular design foster a thriving hardware ecosystem while simultaneously posing significant challenges to hardware security assurance. Early detection of security vulnerabilities during the initial stages of the hardware design flow enables vulnerabilities to be eliminated at minimal cost before being permanently embedded into physical silicon. Although static analysis techniques are applied to early-stage hardware security vulnerability detection,

* 基金项目: 国家自然科学基金 (62202457); “源图”重大基础设施
本文由“RISC-V 与人工智能系统软件前沿进展”专题特约编辑武延军研究员、谢涛教授、侯锐研究员、宋威研究员、邢明杰高级工程师推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-20; 采用时间: 2025-12-17; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-03-19

existing methods suffer from high false negative rate (*FNR*) and false discovery rate (*FDR*) due to insufficient utilization of specification knowledge and an inadequate semantic understanding of code context. To address these challenges, this study proposes MARC, an early detection method for hardware security vulnerabilities based on collaborative multi-agent systems powered by large language models (LLMs). The proposed method constructs a collaborative framework consisting of four specialized agents: design dependency analysis, documentation analysis, security vulnerability detection, and security vulnerability confirmation. Through multi-dimensional collaboration, including cross-module context augmentation, structured utilization of module documentation knowledge, rapid preliminary screening of potential vulnerabilities, and deep reasoning-based risk analysis. Experiments conducted on an industrial-grade dataset demonstrate that the MARC framework achieves an *FNR* of 0.3829 and an *FDR* of 0.3695, representing reductions of approximately 18.2% and 19.1%, respectively, compared to the baseline method. The proposed framework effectively reduces false positives and improves detection accuracy in early-stage hardware design. Furthermore, its real-world effectiveness is validated by the discovery of a hardware vulnerability that has been assigned a CVE identifier. By its successful application, the authors' team won the global championship in the HACK@DATE 2025 hardware security competition.

Key words: hardware security; register transfer level (RTL); security vulnerability detection; large language model (LLM); multi-agent system

随着开源指令集 RISC-V 架构的迅猛发展, 其开放与模块化的特性催生了空前繁荣的硬件生态. 据 SHD Group 预测, RISC-V SoC 的出货量将以近 47% 的年复合增长率增长, 至 2030 年占据全球近 35% 的市场份额[1]. 由于这种爆炸性的增长与广泛应用, 构建在 RISC-V 之上的硬件电路的安全性与可靠性, 已然成为整个信息安全体系的信任根基. 然而, 与易于迭代更新的软件漏洞不同, 源于设计阶段的硬件安全缺陷, 一旦经过流片制造便被永久性地固化在物理实体之中, 后期修复成本极其高昂甚至无法被修复. 更为严重的是, 这些硬件层面的缺陷能够轻易绕过上层的软件安全防护机制, 使得构建于其上的操作系统、加密算法等所有安全措施失效[2]. 然而, 现有安全体系对硬件缺陷的防御能力严重不足. 一方面, 主流安全机制多以缓解软件漏洞攻击为核心目标, 难以抵御跨层攻击对硬件层面的渗透; 另一方面, 即便是针对硬件设计的安全扩展 (如 SGX (software guard extension)[3]、Penglai-Enclave[4]), 也并非为应对硬件自身安全缺陷而设计. 因此, 其实现过程中仍易受设计阶段遗留的未检测缺陷影响, 已多次成为跨层攻击的成功目标[5,6]. 即便 Sanctum[7]、SANCTUARY[8]、Keystone [9]等安全研究项目, 也未将硬件实现层面的安全性保障纳入核心设计目标. 进一步分析表明, 硬件安全缺陷的成因具有多元性与隐蔽性. 其安全缺陷既可能源于安全规范的不明确或错误、设计阶段的逻辑缺陷, 也可能是设计方案在门级综合过程中因人为失误或转换故障导致的实现偏差. 由于这些缺陷的普遍存在与防御体系的滞后性相互叠加, 硬件安全危机愈发突出, 已成为威胁整个计算系统安全的核心风险点.

由于在硬件设计中, 缺陷的发现与修复成本会随着设计流程的推进呈指数级增长, 因此在设计的早期阶段检测并修复潜在缺陷至关重要. 寄存器传输级 (register transfer level, RTL) 作为设计意图的首次代码化实现, 是在缺陷固化于下游的逻辑综合与物理版图前、以最小代价进行拦截和修正的关键环节. 现有的主流硬件缺陷检测技术, 如仿真、形式化断言、硬件模糊测试以及信息流追踪等, 其共同点在于均需对一个功能完备的 RTL 模块进行事后验证, 故无法满足在编码过程中进行即时反馈的伴随式检查需求. 伴随式检查中的常见方法是 Linter 静态代码检查工具, 其工作原理是通过数据流分析、控制流分析, 实现对源代码中风格、语法、结构、设计问题的自动化检查, 以及对特定缺陷模式的启发式识别. 然而, 由于 Linter 的本质是一种语法和浅层语义层面的分析器, 其分析范式高度依赖于一个预设的、基于语法与结构模式匹配的静态规则库, 此范式无法对代码设计意图及上下文逻辑进行有效建模与理解.

现有硬件安全缺陷早期检测技术存在以下挑战.

挑战 1: 缺乏设计安全知识导致的高漏报率. 当前早期 RTL 检测技术在安全缺陷方面的高漏报率, 根本上源于设计规约中的安全知识与自动化工具所能处理的规则库之间的巨大鸿沟. 关键的安全属性、资产定义与信任边界等规约, 通常散布于非结构化的自然语言文档中, 或是在快速迭代的开发流程中缺失不全. 将这些非结构化的知识手动转化为精确的 Linter 规则过程, 不仅效率低下、极易出错, 且难以随设计的演进进行同步维护, 在工程上不具备可扩展性. 由于将设计安全知识有效结构化存在困境, 现有方法只能局限于通用的、与设计意图无关的缺陷模式匹配, 因而无法检测出与特定设计语义和安全需求紧密耦合的深层次漏洞.

挑战 2: 缺乏对代码语义的深层理解导致的高误报率. 现有早期检测技术普遍依赖基于语法和结构的浅层模式匹配技术. 然而, 硬件设计的正确性本质上是由深层的功能与上下文语义所决定的. 这种分析能力与验证需求之间的不匹配, 是导致高误报率的核心原因. 现有方法虽能识别出局部代码中与安全缺陷模式相似的结构, 但无法推断出代码的安全意图, 因而无法判断数据流的信任等级、对关键资产的访问权限, 以及是否存在违反安全协议的状态转换等深层安全问题. 因此, 许多在特定功能或协议上下文中完全正确的设计实现, 会因其结构触发了规则集中通用的、与上下文无关的启发式规则, 而被错误地标记为安全缺陷. 这种由语义缺失所引致的海量误报, 极大地增加了硬件工程师的人工甄别负担, 削弱了自动化工具在工程实践中的应用价值.

大语言模型 (large language model, LLM)[10]的发展与应用, 为在无需依赖完全成熟的测试框架下检测代码缺陷提供了一种可能的途径. LLM 在 RTL 代码生成[11]和修复[12]等任务上已取得相当程度的成功, 但其在安全缺陷检测方面的能力仍有待验证. 在软件领域, 已有前期工作探索了将 LLM 与静态分析相结合的方法. 例如, IRIS[13]利用 CodeQL 作为静态分析工具并与 LLM 协同, 以检测 Java 代码中的代码注入漏洞. Chapman 等人[14]提出了一种将 LLM 与 EESI 静态分析工具相结合的方法, 用于检测特定错误推断问题. LLIFT [15]利用 LLM 与静态分析的组合, 来检测 Linux 内核中使用前未初始化的缺陷. 此外, Li 等人[16]使用微调后的 LLM 生成领域代码, 用以增强对深度学习库的动态模糊测试. LLM 的决策过程可用于判断一段代码是否不安全, 但它无法控制工具使用的流程或信息收集的过程. 这种在自主思考上的差距可以通过在多智能体中使用 LLM 来填补. 通过赋予 LLM 制定计划、在适当时使用工具以及在框架内控制流程的能力, LLM 就如同一个能够做出类似人类决策的智能体[17]. 这能更好地模拟 RTL 设计者或验证工程师审计代码时的思维过程. 此外, 寻找漏洞的过程可能需要迭代地使用多种方法来定位设计行为异常的原因. 这个过程可以在一个多智能体框架中进行模拟.

受到上述基于大语言模型的代码分析工作启发, 本文提出了一种基于多智能体协同的硬件安全缺陷早期检测方法——MARC (multi-agent RTL checker). 该方法的核心机制在于通过由多个专业智能体构成的协同系统, 系统性地弥合设计知识与代码分析间的鸿沟. 具体而言, 首先, 由依赖分析智能体基于 AST 构建包含跨模块依赖的深度代码语义图; 随后, 由文档分析智能体从非结构化文档中提取结构化的设计规约以解决漏报问题 (以应对挑战 1); 然后, 缺陷检测智能体对代码进行深度分析, 以识别潜在风险点; 最终, 由缺陷确认智能体扮演验证专家角色, 利用前序智能体构建的完整知识与上下文进行深度逻辑推理, 从而精准甄别真实缺陷, 并滤除因缺乏语义理解而产生的误报 (以应对挑战 2). 通过这种分层协同框架, MARC 能够将设计规约与代码语义深度融合, 实现远超单一方法的检测精度与可靠性. 本文主要贡献如下.

1) 提出了一种基于多智能体协同的硬件安全缺陷早期检测方法 MARC, 通过模拟专家团队的协同审查流程, 系统性地解决了传统方法在 RTL 级硬件安全缺陷早期检测问题上的高漏报率与高误报率问题.

2) 通过基于检索增强生成 (retrieval-augmented generation, RAG)[18]技术将非结构化的硬件文档知识和安全规范转换为模块相关的结构化安全规范图谱, 并将其应用于 RTL 级硬件安全缺陷早期检测.

3) 通过实验证明, MARC 能够将漏报率与误报率相较于基准方法分别降低约 18.2% 与 19.1%; 并支撑团队获得 HACK@DATE 2025 硬件漏洞挖掘竞赛全球冠军[19], 成功挖掘 1 个获 CVE 编号的真实漏洞.

本文第 1 节介绍研究背景、现有主流技术及其局限性, 并对相关工作进行综述. 第 2 节对本文所研究的问题进行形式化定义, 并系统阐述现有技术在漏报率与误报率方面面临的核心挑战. 第 3 节详细阐述本文提出的基于多智能体协同的硬件安全缺陷早期检测框架, 包括其整体架构、各智能体的设计与各智能体的协同工作机制. 第 4 节通过全面的量化实验, 将 MARC 与现有基准方法进行多维度性能比较与分析. 第 5 节讨论本文方法的潜在局限性并展望未来研究方向. 第 6 节对全文工作进行总结.

1 背景及相关工作

1.1 RISC-V 生态下的硬件安全新挑战

开放指令集架构 RISC-V 正以前所未有的态势重塑全球半导体格局, 其开放、模块化与可扩展的特性在极大促进硬件生态创新与自主可控生态构建的同时[20], 也对硬件设计的内在安全性与可靠性提出了更为严苛的要求.

一方面, RISC-V 灵活的自定义指令集扩展机制, 在催生大量定制化处理器的同时也带来了实现多样性与生态碎片化的挑战, 不同厂商实现的行为差异可能被利用, 从而引入新的攻击面[21]. 另一方面, 在日益复杂的片上系统设计中, 硬件电路作为承载上层软件与应用的核心实体, 构成了整个计算系统的信任根基[22]. 任何源于硬件设计阶段的缺陷, 都可能被利用以发起绕过软件安全机制的跨层攻击[2], 从而使整个系统的安全体系土崩瓦解. 因此, 在 RISC-V 生态蓬勃发展的背景下, 如何确保硬件设计自身的安全性, 已成为保障整个信息系统安全的重要挑战.

硬件设计一旦经过流片环节, 其逻辑功能便被永久性地固化在物理芯片实体之中. 这意味着在硬件设计阶段遗留的任何微架构层面的安全缺陷, 例如导致 Meltdown[23]或 Spectre[24]攻击的推测执行漏洞, 都无法进行根本性修复. 这些缺陷并非软件层面的编程错误, 而是源于处理器为追求高性能而做出的基础设计决策, 这类缺陷的利用与具体操作系统或应用软件无关[25]. 这种无法被更改的物理特性, 决定了其修复代价会随设计流程的推进而急剧攀升. 在 RTL 编码阶段发现的缺陷, 其修复成本可能仅为数小时的工程师时间; 然而, 一旦设计进入后端乃至流片之后, 同样缺陷的修复则可能需要耗费数百万美元进行掩模重制和芯片重新流片, 甚至导致产品的全面召回. 更严峻的是, 如 Spectre 等漏洞的修复方案往往只能依赖于性能损失巨大的固件级缓解措施, 而彻底的解决方案则必须等待下一代处理器的重新设计[24]. 这种跨层漏洞的严重性[26]与高昂的修复代价, 凸显了在设计早期进行高精度缺陷检测的重要性.

在整个硬件设计流程中, RTL 设计阶段占据了承上启下的核心位置. 硬件设计本质上是一个从高级抽象向物理实现逐层精化的过程, 它始于系统级的功能规约或高级语言模型, 终于物理版图[27]. RTL 作为该流程的中间环节, 是设计意图首次被代码化、结构化和时序化表达的阶段, 它将高层的功能算法转化为可被下游工具综合、布局布线的具体电路描述[28]. 正是这种独特的中间位置, 使 RTL 阶段成为识别并修正逻辑与安全类缺陷成本最低、效率最高的时期. 一方面, 相较于上游抽象的规约, RTL 代码提供了足够的设计细节以进行深度的自动化分析和逻辑验证. 另一方面, 相较于下游经过综合优化的门级网表和固化的物理版图, RTL 代码仍保持着高度的可读性与可修改性. 在此阶段, 修正一个缺陷通常仅需修改几行源代码; 而一旦缺陷遗漏至后端乃至流片环节, 其修正成本将呈指数级增长. 无论是常规的功能性错误, 还是如硬件木马等难以在后期通过物理测试发现的恶意设计[29], 在 RTL 源码层面进行审查和验证都是最直接、最有效的拦截手段. 因此, 将研究聚焦于 RTL 阶段的早期检测, 是保障硬件安全、降低开发成本的关键.

1.2 硬件安全缺陷检测技术

在 RTL 设计完成后, 业界主要依赖动态仿真 (dynamic simulation)[30]、形式化验证 (formal verification, FV)[31–33]以及硬件模糊测试 (hardware fuzzing)[21]等验证技术来保障设计的正确性与安全性. 这些技术是现代芯片设计流程中不可或缺的组成部分, 构成了确保芯片质量的黄金标准[34]. 例如, 形式化验证能够通过数学方法对设计的所有状态空间进行穷尽性探索, 为关键模块提供完备的正确性证明[32,33]. 然而, 这些传统验证方法均具有较大的局限性, 主要体现在以下 3 个方面: 首先, 高度依赖完备的验证环境. 无论是动态仿真所需的大量测试用例和通用验证方法学 (UVM) 平台[35], 还是形式化验证所需的精确规约和断言, 都需要验证工程师投入大量精力进行专门构建. 其次, 资源消耗巨大且反馈周期长. 完整的验证流程在芯片开发中占据近一半的时间与成本[31], 且形式化验证等技术面临着严峻的可扩展性挑战, 对于复杂设计可能需要数天甚至数周的运行时间, 这种延迟无法满足设计者在编码时对即时反馈的需求. 最后, 专业门槛高. 尤其是形式化验证, 其陡峭的学习曲线和对工程师专业能力的高要求, 使其通常由专门的验证团队负责, 无法被 RTL 设计者在日常开发中便捷使用[32]. 这些局限性使得上述方法只能作为设计完成后独立的、重量级的验证阶段, 难以胜任设计编码过程中的伴随式检查任务.

为了弥补传统验证技术在开发早期的缺失, 静态代码检查技术 (Linter) 成为当前主流的伴随式早期检测方法. 与重量级的验证方法不同, Linter 是一种轻量化的自动化代码审查工具[36], 能够为硬件工程师在 RTL 编码阶段提供即时的、自动化的反馈. 在硬件设计领域, 以 Synopsys 的 SpyGlass Lint 等商业 EDA 工具和 Verible Lint 等开源项目为代表的 Linter 得到了广泛应用[37,38]. Linter 的核心工作原理是基于一个预设的、静态的规则库. 其分析过程无需编译或执行代码, 而是直接对 RTL 源代码进行处理. 首先, 工具会利用内置的解析器将 SystemVerilog 等硬件

描述语言代码转化为抽象语法树等结构化表示. 随后, Linter 遍历该结构化表示, 通过语法分析、结构匹配和模式识别等手段, 逐一检查代码是否违反了规则库中定义的规则, 这些规则覆盖了从编码风格、设计结构到可综合性等多个维度. 这种基于模式匹配的自动化检查机制, 使得 Linter 能够高效地识别出常见的编码缺陷和潜在的设计风险, 尽早提升代码质量.

然而, Linter 的工作模式导致了高漏报率和高误报率的瓶颈. 首先, 现有静态分析技术存在显著的“设计知识鸿沟”, 这是导致高漏报率的根本原因. 关键的安全规约, 例如信任边界的划分、关键资产的定义以及特定的安全协议状态机等, 通常仅存在于非结构化的自然语言设计文档之中. 传统的 Linter 无法自动解析和理解这些非结构化的设计知识, 因而其规则库只能包含通用的、与具体设计意图无关的语法或结构模式. 这使得工具无法检测出那些与特定设计语义和安全需求紧密耦合的深层次漏洞, 从而造成了大量危险的安全缺陷被漏报. 其次, 这些工具普遍存在语义理解缺失的问题. 由于 Linter 依赖于浅层的模式匹配, 无法推断代码背后真实的设计意图和功能逻辑. 因此, 许多在特定协议或算法上下文中完全正确、安全的设计实现, 会因为其代码结构偶然触发了通用规则而被错误地标记为潜在缺陷. 正如文献 [39] 指出的, 高误报率是当前 Linter 类工具面临的普遍问题. 这种由语义缺失所引致的海量误报, 增加了硬件工程师的人工甄别负担, 并严重削弱了自动化工具在工程实践中的应用价值.

1.3 基于大语言模型的代码分析技术

近年来, 以大规模预训练模型为代表的大语言模型为解决代码的深层语义理解难题提供了全新的范式, 有望突破传统静态分析技术的瓶颈. 与依赖预设规则进行语法和结构匹配的 Linter 不同, 大语言模型通过在海量代码和自然语言文本上进行训练, 获得了强大的上下文学习与逻辑推理能力, 使其能够理解代码背后更为复杂的语义和设计意图[40]. 这一潜力已在众多代码相关任务中得到验证, 在代码生成与补全领域, 研究表明经过微调的 LLM 能够高效生成功能正确且语法规范的 Verilog 代码, 其表现在某些场景下甚至超越了先进的商用模型[11], 并且已发展出如 RTLCoder[41]等在性能和模型轻量化上取得平衡的开源模型, 以及如 ChatCPU[42]的覆盖敏捷设计与验证的自动化平台. 在更为复杂的自动程序修复任务中, LLM 不仅能够修复简单的函数级错误, 在具备充分的仓库级上下文后, 其修复复杂跨文件漏洞的能力也得到显著增强[43].

而在更为关键的硬件安全领域, LLM 的应用已经初见成效[44], 现有工作证明了 LLM 能够有效修复 Verilog 代码中的安全相关缺陷, 其性能已超越部分最先进的专用硬件漏洞修复工具[12]. 进一步地, 研究界正积极探索利用 LLM 进行主动安全缺陷检测与定位. 例如, FLAG[45] 结合静态分析与 LLM, 通过对比分析原始代码与 LLM 生成代码之间的词法单元及行级差异, 实现了一种无需测试用例的 RTL 功能及安全缺陷定位方法. 此外, 亦有工作从安全感知的代码生成角度切入, 如 SecV[46]框架利用从硬件 CWE (common weakness enumeration) 语料库中构建的知识图谱, 引导 LLM 在代码生成阶段即主动规避已知的安全漏洞. 这些研究成果充分表明, LLM 具备处理和理解 RTL 代码复杂语义的能力, 为其应用于更具挑战性的硬件安全缺陷早期检测任务奠定了坚实的技术可行性基础. 相较于硬件领域, LLM 在软件安全领域的应用则更为深入. 例如, IRIS 框架利用 LLM 自动为静态分析工具 CodeQL 推断和生成污点分析规范, 显著提升了后者在真实 Java 项目中的漏洞检出率, 并发现了多个未知漏洞[47]. 同样, LLIFT 通过将 LLM 与静态分析相结合, 增强对 Linux 内核代码的路径分析能力, 成功定位了多个此前未被发现的使用前未初始化缺陷[15].

但是真实的代码分析任务并非一个简单过程, 而是需要结合规划、工具使用和流程控制的系统性工程[47]. 单一的 LLM 调用本质上是无状态的, 缺乏主动分解任务、与外部工具交互以及根据中间结果动态调整策略的能力. 因此, 针对单一大语言模型对于多阶段、多工具协同的复杂任务处理能力不足的问题. 研究界开始将 LLM 从一个被动的语言模型演进为一个主动的智能体[47], 并构建多智能体协同框架[16]. 基于多智能体的方法论已在硬件领域开始应用. 在 RTL 代码生成任务中, MAGE[48]采用多智能体架构, 通过调试机制探索候选代码空间, 提升生成代码的功能正确性. 在多智能体框架下, 不同的 LLM 实例被赋予特定的专家角色, 每个智能体负责解决一个子问题. 这些智能体之间可以相互协作、传递信息, 并调用外部工具来获取和处理数据, 从而模拟一个人类专家团队的复杂决策与工作流程. 通过这种角色分工与协同推理的模式, 能够将一个宏大的审计任务分解为一系列可管理的步骤,

从而更有效地解决深度依赖上下文的漏洞挖掘等复杂问题. 这些工作证明了通过让 LLM 扮演领域专家角色来指导静态分析工具, 可以有效克服传统方法在规约缺失和上下文理解方面的短板, 显著提升代码分析的精度与广度.

然而, 值得注意的是, 当前基于大语言模型的代码分析研究主要集中于 Java、C/C++等软件编程语言, 其方法和经验难以直接迁移至硬件领域. 这种迁移的根本障碍源于软件编程语言与硬件描述语言的本质差异[48]. 首先, 软件代码旨在为处理器编写一系列顺序执行任务的计算指令; 而硬件描述语言并非执行任务, 而是在描述一个物理硬件架构, 其核心是定义数据如何在寄存器间并发流动、精确时序状态以及电路构建结构. 其次, 硬件描述语言的安全意图高度依赖非结构化的自然语言设计文档, 这与软件逻辑在代码或 API 中通常具有更清晰结构化体现的情况截然不同. 这些本质差异导致了软件领域方法无法应对的独特挑战, 即为通用软件训练的大语言模型难以理解 RTL 独特的时序与并发约束, 而传统静态分析工具亦无法处理 RTL 的并发语义和非结构化规约依赖, 进而导致了前文所述的高漏报率 (挑战 1) 与高误报率 (挑战 2). 因此, 尽管软件领域基于 LLM 与多智能体协同范式提供了宝贵启示, 但如何针对硬件描述语言的独有特性进行适应性设计, 构建能同时理解硬件描述语言并发时序语义和硬件设计规约的领域特定方法, 仍是一个亟待专门研究的开放性问题, 也正是本文工作的出发点.

1.4 小　结

针对当前 RTL 缺陷检测问题, 当前研究仍面临挑战. 一方面, 形式化验证等传统技术虽能提供高可信度的验证, 但因其资源消耗巨大、反馈周期长而时机太晚; 另一方面, 作为主流早期检测手段的静态分析工具, 则因其分析原理无法跨越非结构化设计文档与源代码之间的知识鸿沟, 导致了高漏报率与高误报率并存的问题.

本文旨在研究能够深度融合设计知识与代码上下文的早期检测, 提出一种基于大语言模型多智能体协同的硬件安全缺陷检测方法, 该方法的核心在于构建一个模拟领域专家团队协同审计的多智能体框架, 通过角色化的智能体分工与协作, 系统性地将蕴含在自然语言文档中的设计规约与 RTL 代码的深层功能语义相结合, 从而在保证检测时效性的同时, 显著提升缺陷识别的准确性与召回率, 以期突破现有技术的瓶颈.

2 问题定义

本文的研究聚焦于硬件安全缺陷的早期检测. 本文所要解决的问题场景, 是为正在进行 RTL 级设计的硬件工程师提供一种伴随式的安全缺陷早期检测方法. 在该场景下, 工程师对一个或多个 RTL 模块进行编写或修改, 该方法应能实时或近实时地对当前的代码片段进行自动化分析, 并即时反馈潜在的安全隐患. 此机制旨在将缺陷发现的节点从传统的验证阶段大幅提前至编码阶段, 从而以最小的成本消除安全风险. 该场景的核心要求是, 在功能设计尚未完善、缺乏完整测试环境或形式化规约的环境下, 实现高精度、低误报的自动化安全缺陷早期检测.

2.1 概念定义

为精确描述本问题, 我们引入以下定义.

定义 1. RTL 设计. 一个 RTL 设计是描述硬件逻辑与结构的源代码集合, 是设计意图在物理实现前的最高层次代码化表达. 一个 RTL 设计 $\mathcal{D}$ 可被定义为一个模块的集合 $\mathcal{D}=\{m_1,m_2,\ldots,m_n\}$, 其中每个模块 m_i 由其源代码 $\mathcal{C}_i$、端口集 $\mathcal{P}_i$ 与实例化关系 $\mathcal{I}_i$ 共同描述, 即 $m_i=(\mathcal{C}_i,\mathcal{P}_i,\mathcal{I}_i)$.

定义 2. 设计知识. 设计知识是理解代码安全意图所必需的上下文, 它通常蕴含在非结构化的设计文档 $\mathcal{T}_{\text{doc}}$ 之中. 在形式上, 设计知识 $\mathcal{K}$ 可被定义为一个包含关键资产 $\mathcal{A}$、信任边界 $\mathcal{B}$ 以及安全属性 $\mathcal{S}$ 的集合, 即 $\mathcal{K}=\{\mathcal{A},\mathcal{B},\mathcal{S}\}$. 这些元素共同定义了需要保护什么、在哪里保护以及应遵循何种规则.

定义 3. 安全缺陷. 一个安全缺陷本质上是代码实现与设计安全意图之间的偏差, 是构成安全风险的具体代码点. 在形式上, 它指 RTL 源代码 $\mathcal{C}$ 的具体实现违背了设计知识 $\mathcal{K}$ 中某项安全属性 $s\in\mathcal{S}$ 的情况, 可记为 $\mathcal{C}\nvDash s$. 每一个被识别的缺陷 d 可由一个三元组 $d=\langle l,t,desc\rangle$ 来精确描述, 分别代表其在代码对应位置的代码片段、所属缺陷类型和具体缺陷描述.

定义 4. 早期检测. 早期检测是一种在伴随式编码场景下, 对不完整设计进行实时分析以尽早发现潜在安全风险的过程. 其目标是构建一个映射函数 $f:(\mathcal{D},\mathcal{T}_{\text{doc}})\rightarrow\mathcal{D}_{\text{found}}$, 该函数通过分析 RTL 设计 $\mathcal{D}$ 及其相关文档 $\mathcal{T}_{\text{doc}}$, 输出

一个被检测到的缺陷集合 $\mathcal{D}_{\text{found}}$, 并使其尽可能地逼近真实存在的缺陷集 $\mathcal{D}_{\text{true}}$.

2.2 核心挑战

结合上述定义, 引言中所述的挑战 1 和挑战 2 可被更精确地做以下阐述.

挑战 1: 缺乏设计安全知识导致的高漏报率. 该挑战源于从非结构化设计文档 T_{doc} 到结构化设计知识 K 的转化缺失. 现有早期检测方法 f 通常只将 RTL 设计 $\mathcal{D}$ 作为输入, 其分析范式无法有效融入针对特定设计的关键资产 $\mathcal{A}$、信任边界 $\mathcal{B}$ 或安全属性 $\mathcal{S}$. 因此, 检测工具只能依赖一组通用的、与特定设计意图无关的规则进行分析. 这导致大量需要结合具体设计知识才能判定的真实安全缺陷 d 被遗漏, 即 $\mathcal{D}_{\text{found}}$ 集合远小于 $\mathcal{D}_{\text{true}}$ 集合, 从而产生高漏报率.

挑战 2: 缺乏对代码语义的深层理解导致的高误报率. 该挑战源于检测方法对 RTL 设计 $\mathcal{D}$ 的分析停留在语法或浅层语义层面. 现有方法在判断一段代码 $\mathcal{C}$ 是否构成安全缺陷 d 时, 通常未能充分考虑其在整个设计中的跨模块调用关系 $\mathcal{I}$ 和数据流上下文. 因此, 大量在特定功能或协议上下文中完全合规的实现, 会因其代码结构触发了通用的、与上下文无关的启发式规则, 而被错误地标记为缺陷. 这导致 $\mathcal{D}_{\text{found}}$ 集合中包含大量并非存在于 $\mathcal{D}_{\text{true}}$ 中的伪缺陷, 从而产生高误报率.

3 基于多智能体协同的硬件安全缺陷早期检测方法

针对上述挑战, 本文设计了一种基于大语言模型多智能体协同的硬件安全缺陷早期检测方法 MARC. MARC 核心思想在于模拟硬件安全专家团队的工作模式, 通过关注点分离原则将复杂的检测任务分解, 并设计了 4 类职能明确、分工协作的智能体, 以流水线式的架构系统性地完成从设计理解、缺陷检测到根因验证的全过程, 方法整体框架如图 1 所示.

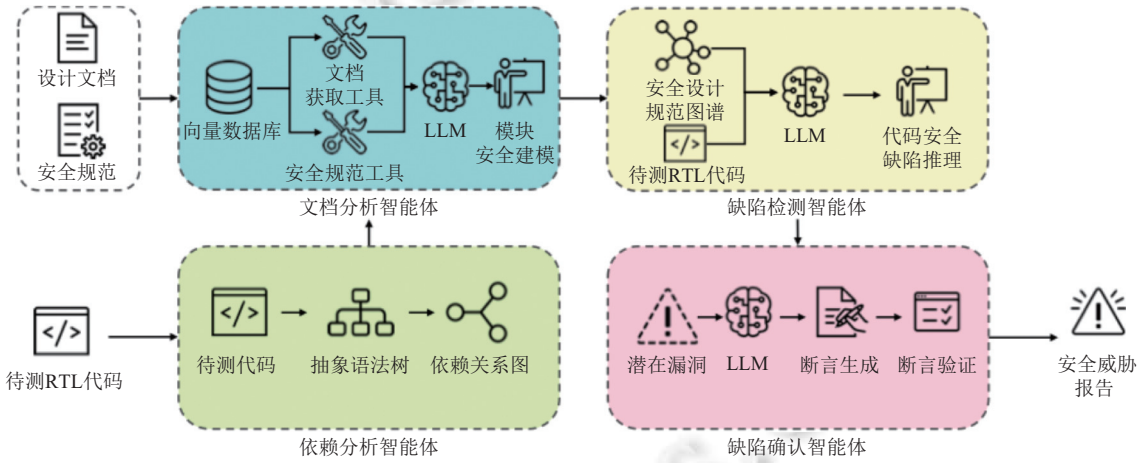

图 1 基于多智能体协同的硬件安全缺陷早期检测方法

3.1 依赖分析智能体

由于单一硬件模块的安全性会受到其在整个系统中的模块间交互关系的影响, 因此需要从更全局的视角分析其交互边界并评估由此引入的潜在安全风险. 因此, 本方法设计了依赖分析智能体, 旨在自动化地构建并解析模块间的依赖关系图, 在对模块内部代码分析时补充系统级的信息.

该智能体基于对项目中 RTL 源文件进行抽象语法树分析实现依赖关系分析. 具体的, 本文通过遍历 AST 中实例列表的所有节点, 识别出每个父模块所实例化的所有子模块. 基于此信息构建出一个有向图, 其中节点代表硬件模块, 有向边代表实例化或依赖的关系. 同时, 该流程也生成一个逆向依赖图, 用于快速追溯一个底层模块被哪些上游模块所调用, 为理解影响传播路径提供了基础.

为了使构建的依赖图能够被大模型语义化的理解, 本文中的文档分析智能体在接收此依赖图后, 其内部提示

词会引导大语言模型将此结构化信息解读为安全层面的交互边界. 模型被要求在其依赖关系的输出中, 重点分析目标模块与外部世界的交互关系, 并根据依赖源的性质, 初步推断哪些交互是可信的, 而哪些构成了潜在的攻击向量, 实现了从结构连接信息到潜在威胁攻击面的转化.

通过自动化地识别交互边界并初步评估其信任级别, 本方法为后续的缺陷检测智能体提供了关键的潜在威胁攻击面信息, 使其能够将分析注意力优先投入处理更具安全威胁代码的逻辑分析上, 从而显著提高了安全威胁检测的针对性.

3.2 文档分析智能体

文档分析智能体的核心任务是系统地弥合硬件设计意图和 RTL 代码物理实现的鸿沟, 将多源、异构的模块信息系统性地整合为一个结构化的设计安全知识库, 以应对缺乏设计安全知识的挑战, 实现对设计意图的理解.

由于模块安全缺陷检测所需的关键知识往往分布于功能设计文档、安全规范、RTL 源代码和验证脚本等多种信息载体之中, 这些载体不仅信息体量庞大、关键内容密度低, 且不同来源信息之间的内在关联往往是隐晦的. 传统自动化方法, 难以处理非结构化文本并建立跨域关联. 如果直接将海量原始信息输入大语言模型, 不仅会超出大模型的上下文窗口限制而导致关联失败, 还可能因为关键信息被稀释而导致严重的内容幻觉. 为解决此问题, 文档分析智能体采用了检索增强生成技术, 实现对海量、稀疏知识源中高价值信息片段的筛选融合与总结. 为进行高效的检索, 该智能体首先对目标模块的全部文档进行嵌入, 处理为向量. 考虑到大语言模型对输入长度的限制以及长文本向量表征的稀疏性问题, 本文采用了一种滑动窗口分块策略. 具体而言, 原始文档被切割为最大长度不超过 1 000 字符的文本块, 并设置了 200 个字符的重叠区, 以确保上下文语义在块与块之间的连续性与完整性, 避免因硬性切分而导致关键信息的割裂. 随后, 每一个文本块均通过文本嵌入模型转换为能够表征其深层语义的高维向量, 并将这些包含了原始文本、元数据及其对应向量的完整信息批量存入 ChromaDB 向量数据库中, 建立起一个支持高效相似性搜索的知识索引. 设计了一个可被大语言模型调用的文档检索工具, 可根据模块名称或功能描述, 从向量化知识库中查询相关的 RTL 设计、实现细节或安全信息.

根据对设计文档的观察, 硬件开发团队通常在文档中较少提及硬件自身的安全问题, 其关注点多集中于功能实现. 因此, 为了能够系统性地将已知的硬件攻击模式引入分析流程, 弥补设计阶段可能存在的安全知识盲点, 本文构建了一个通用缺陷枚举 (CWE) 向量数据库. 本文选用官方 CWE 列表中与硬件设计强相关的 110 个类型作为数据源, 包含编号、名称、描述、威胁、示例代码等信息. 在向量化和存储策略上采取和文档相同的处理方式, 同样提供了一个可被大模型调用的检索工具, 以实现语义相关的潜在缺陷模式检索.

由于在伴随式检测场景下代码处于快速迭代周期, 其实现细节常与设计文档存在不同步的情况. 因此, 为了确保分析的准确性, 本文方法融入了代码自身的结构化信息. 本文基于抽象语法树对 RTL 代码进行建模, 从而能够精确地提取模块的层次结构、端口定义、寄存器列表、数据流路径以及控制逻辑等关键结构化信息.

在对模块建立全面认知后, 本智能体将进行潜在威胁分析并针对该模块生成结构化的安全设计规范图谱, 图谱字段结构如后文表 1 所示. 该过程旨在将前序步骤中融合的多模态信息, 系统性地映射到已知的硬件安全缺陷模式上. 具体而言, 本智能体利用其对模块功能、状态机、接口协议和寄存器配置的深度理解, 自主生成多个语义查询, 并调用上文介绍的 CWE 检索工具, 在 CWE 向量数据库中检索与之最相关的潜在缺陷模式. 最终, 本智能体将文档分析、代码结构概要、依赖关系以及潜在 CWE 威胁等全部认知成果, 整合并输出为一个严格遵循预定义字段结构的 JSON 对象, 作为最终的模块安全设计规范图谱, 为后续缺陷检测智能体提供了高质量、高密度的上下文信息.

3.3 缺陷检测智能体

缺陷检测智能体是 MARC 方法中执行具体代码级安全缺陷检测的核心模块. 传统的静态分析工具因无法理解设计意图而产生大量误报, 而通用的大模型也因为本身缺乏对代码上下文的理解能力而出现严重的幻觉现象. 本智能体利用前序智能体产出的模块安全设计规范图谱, 将高密度知识与代码分析紧密结合, 把通用的大语言模型增强为一个具备上下文感知能力的代码审计者, 旨在解决缺乏对代码语义深层理解的研究挑战.

表 1 模块安全设计规范图谱字段结构表

字段ID及结构	名称	父节点	数据类型	字段描述
ROOT	根结点	—	Object	IP模块安全设计规范图谱根结点
├─ name	IP模块名称	ROOT	String	IP模块的具体名称, 例如“gpio”
├─ doc_analysis	文档分析结果	ROOT	Object	从IP模块设计文档中分析的全部信息
│ ├─ summary	功能总结	doc_analysis	String	对核心功能和安全机制的简要总结
│ ├─ mechanism	工作原理	doc_analysis	Object	包含对模块内部工作逻辑的详细分析
│ │ ├─ core_func	核心功能	mechanism	String	描述模块的主要功能和操作模式
│ │ ├─ fsms	关键状态机列表	mechanism	Array	列出模块中实现关键逻辑的状态机
│ │ │ ├─ fsm_name	状态机名称	fsms	String	状态机的唯一标识名称
│ │ │ ├─ fsm_desc	状态机功能描述	fsms	String	描述该状态机负责的具体功能
│ │ │ └─ states	状态列表	fsms	Array	列出该状态机包含的所有状态
│ │ └─ data_flow	数据流路径	mechanism	String	描述数据在模块内部及接口之间的流动路径
│ ├─ access_points	接口与攻击面	doc_analysis	Object	识别物理和逻辑接口及潜在攻击面
│ │ ├─ bus_if	总线接口	access_points	Array	模块连接到的总线接口
│ │ ├─ io_pins	物理IO引脚	access_points	Array	列出模块的物理输入/输出引脚
│ │ └─ clk_rst	时钟与复位	access_points	String	分析模块的时钟域和复位策略及其安全隐患
│ ├─ prog_model	编程模型	doc_analysis	Object	分析软件如何通过寄存器等方式控制模块
│ │ ├─ reg_map	寄存器映射表	prog_model	Array	每个寄存器的功能、偏移量和安全影响
│ │ └─ interrupts	中断列表	prog_model	Array	模块可生成的中断及其安全相关性
│ └─ sec_features	安全特性	doc_analysis	Array	模块文档中提到的安全功能及其潜在弱点
├─ ast_summary	AST分析总结	ROOT	String	AST的总结, 识别关键逻辑和结构
├─ dep_summary	模块依赖总结	ROOT	String	总结对其他模块的依赖, 特别是安全依赖
└─ cwes	潜在CWE识别	ROOT	Array	识别出的与模块相关的潜在硬件CWE
├─ cwe_id	CWE编号	cwes	String	对应的CWE编号, 例如“CWE-1262”
├─ cwe_name	CWE名称	cwes	String	对应CWE的官方名称
├─ cwe_desc	CWE官方描述	cwes	String	CWE的官方详细描述
└─ cwe_reason	关联理由	cwes	String	CWE与当前IP模块相关联的具体理由

为实现上下文感知的缺陷检测, 该智能体采用知识增强的提示工程策略. 首先, 利用知识增强的上下文注入策略, 即将文档分析智能体生成的模块安全规范图谱作为前序知识, 与待测 RTL 源码一并注入提示词. 这使得大语言模型在分析代码前, 已对模块的预期功能、状态机定义、接口协议、关键寄存器等拥有了全面认知. 其次, 为了使大语言模型的注意力集中到安全分析上, 提示词中还明确列出了一系列硬件设计中常见的安全威胁类型, 例如访问控制与权限提升、不安全调试接口等, 以审计清单的形式引导大语言模型将推理能力集中于最可能出现安全问题的方向. 与依赖预定义规则集的传统静态分析工具相比, 本智能体通过注入丰富的设计意图上下文, 能够有效地区分设计中的正常行为与潜在安全缺陷. 例如, 当大语言模型从图谱中了解到某个寄存器被设计为可由软件读写以控制功能时, 它就不会轻易地将其标记为不安全的直接访问漏洞, 从而显著降低误报率. 另外, 大语言模型能够基于其对模块预期行为的理解, 发现那些在语法上正确但与设计逻辑或安全需求相悖的逻辑漏洞, 例如, 识别出状态机在某种异常输入下可能进入的未定义或不安全状态.

3.4 缺陷确认智能体

缺陷确认智能体在 MARC 框架中扮演着“验证工程师”的角色, 其核心任务是对缺陷检测智能体发现的潜在安全缺陷进行自动化验证与可信度评估. 经由前序智能体检测出的潜在安全缺陷, 本质上仍是基于启发式推理的假设, 仍可能包含误报. 为解决此问题, 本智能体设计了一种断言生成及评估的快速验证方法, 旨在人工审计之前, 以低成本的方式对发现的缺陷进行验证.

由于前序智能体生成的结果为对安全缺陷的自然语言描述, 无法进行精准验证. 由于硬件描述语言的特性, 其安全缺陷的确认高度依赖于对复杂时序逻辑和并发状态的精确理解. 因此, 本智能体采取缺陷断言生成的方法, 将前序智能体输出的、以自然语言描述的潜在漏洞, 转化为一种机器可验证的断言. 本文在此处采用业界常用的 SystemVerilog 断言 (SVA), SVA 是一种用于形式化描述硬件设计时序逻辑的语言, 能够将设计规范转化为机器可

分析的属性. 该智能体向大语言模型提供 RTL 源代码和漏洞描述, 并指示其为该漏洞编写对应的 SVA 代码. 这使得 LLM 必须深度理解代码的时序语义和并发逻辑, 才能将一个自然语言形式的漏洞描述转化为一个精确的、机器可分析的 SVA 语句. 如果一个安全缺陷本身描述很模糊, 定义不清晰, 无法生成逻辑自洽且与代码相关的断言, 则说明该安全缺陷本身的可信性较低, 也就实现了一层初步过滤.

在断言生成之后, 本智能体将进行缺陷断言验证阶段. 在此阶段, 本智能体发起另一个大模型推理请求, 将原始 RTL 代码、原始漏洞描述以及前一阶段生成的 SVA 代码注入提示词, 并要求大模型判断给定的断言能否在给定的 RTL 代码中成立.

通过这种精准验证方案, 本智能体能够对潜在缺陷进行有效的筛选与确认, 使得硬件安全专家能够将精力聚焦于可信度更高的缺陷上, 从而优化审计资源分配. 该流程生成的 SVA 代码也可以集成到标准的仿真或形式化验证流程中, 帮助进行高效的安全验证.

3.5 多智能体协作流程

MARC 方法的整体运行流程是一个高度协同的流水线式架构, 通过模拟一个人类硬件安全专家团队的工作模式, 系统性地完成从设计理解到漏洞验证的全过程. 为了提高检测效果, 本协作流程应用关注点分离的理念, 即每个智能体都专注于一个特定的、定义明确的任务, 并通过标准化的结构化数据接口进行信息传递, 从而在各自领域内达到更高的专业水平. 首先, 依赖分析与文档分析智能体协同解析系统级接口与设计规范, 生成一份结构化的模块安全规范图谱作为上下文基础. 随后, 缺陷检测智能体利用该图谱, 对 RTL 源码进行上下文感知的深度逻辑审计, 产出潜在安全威胁列表. 最后, 缺陷确认智能体对清单中的每一项通过精准断言生成及验证的方式进行审查, 最终输出一份经过误报过滤的安全缺陷报告.

另外, 在具体任务中, 高效的提示词有助于激活每一个智能体的能力[49]. 提示词的元素包括 4 个部分: 指令 (instruction)、上下文 (context)、输入数据 (input data) 和输出指示器 (output indicator). 其中, 指令用于指导模型的行为, 帮助其完成特定任务; 上下文提供背景知识或其他外部信息, 增强模型生成响应的准确性; 输入数据是期望模型处理并据此生成响应的数据; 输出指示器则指定所需的输出格式或类型. 将这些要素整合到提示词中, 可以有效传达设计者的意图. 为提高模型对复杂任务的理解, 本文在不同智能体中采用了多种优化策略, 包括人格设定策略、任务分解策略、少量样本提示策略等. 具体而言, 人格设定策略[50]是指通过在提示词中为大语言模型分配一个明确的专家角色或身份, 以激活其在该领域的专业知识, 并引导其输出风格与内容贴近该角色的行为模式. 例如, 在缺陷检测智能体的提示词中, 我们将模型角色设定为“硬件安全专家”, 以确保其分析聚焦于硬件设计的独特安全视角. 任务分解策略[51]是指将一个宏观的复杂指令, 在提示词中细化为一系列清晰、可执行的微观步骤或检查清单, 以引导模型进行结构化、逐步的思考与执行. 例如, 文档分析智能体的提示词中包含从识别目标模块到生成 JSON 报告的工作流要求, 确保了其信息收集与分析过程的质量. 少量样本提示策略[52]是指在提示词中提供若干个任务的完整示例, 通过上下文学习的方式, 让模型从样本中归纳出解决问题的模式、遵循特定的输出格式或采纳某种推理逻辑. 例如, 在指导缺陷检测智能体输出结构化报告时, 提示词中会提供一个完整的漏洞报告作为范例, 清晰地展示缺陷描述应如何撰写、代码位置应如何格式化等具体要求, 从而确保模型后续生成的每一个漏洞报告都能严格遵循同样的标准.

此外, 为确保 LLM 在多智能体协作流程中的稳定性与正确性, 并有效抑制输出幻觉与输出格式错误[53], 本文设计并实现了一套多层保障机制. 首先, 在提示词工程层面, 如前文所述, 本文使用少量样本提示策略为 LLM 提供清晰、完整的 JSON 输出范例, 并强制模型在输出的开头和结尾使用特定分隔符, 以便后端进行精确的解析和提取. 其次, 在智能体接收 LLM 的响应后, 本文引入严格的 JSON Schema 校验机制. 系统会根据预先定义的模板, 对 LLM 返回内容的字段、类型和结构进行强制校验. 最后, 本文实现了一个包含错误处理的健壮重试机制. 当发生网络错误或内容校验失败时, 系统将自动重试请求, 以确保最终获得正确有效的输出. 通过这套机制, 本文方法能够规避大模型幻觉问题, 保障了交互流程的正常运行.

通过以上方式, MARC 的多智能体协作流程将复杂的硬件安全缺陷早期检测任务, 分解为一系列可管理、可

自动化、可验证的步骤, 不仅提升了每个单一任务的质量, 更通过流水线式的协同工作, 实现更为可靠的硬件安全缺陷早期检测能力.

4 实验分析

本节旨在通过一系列量化实验, 对本文提出的 MARC 方法进行系统性的性能评估与分析.

4.1 研究问题

为了验证 MARC 相较于现有技术的优越性, 并具体衡量其在应对上述核心挑战方面的能力, 本文设计了以下 3 个核心研究问题.

RQ1. MARC 在硬件安全缺陷早期检测方面的整体效果如何?

RQ2. 针对挑战 1, MARC 通过融入设计知识的机制在降低漏报率方面的效果如何?

RQ3. 针对挑战 2, MARC 通过深度代码语义分析在精准识别真实缺陷方面的效果如何?

4.2 实验准备

(1) 实验环境

本文所提出的 MARC 方法基于 Python 语言实现, 并选用 LangChain 作为其多智能体系统的核心构建与调度框架. 在底层工具链方面, 我们采用 Google 开源的 Verible 工具对 RTL 源代码进行解析, 以构建精确的抽象语法树. 同时, 框架内的检索增强生成功能亦基于 LangChain 实现. MARC 的整体工作流程被设计为高度自动化, 并且其核心逻辑不依赖具体的大语言模型, 从而保证框架的灵活性与可扩展性, 便于未来集成更多先进模型. 在实验中, 我们选用当前性能领先的 DeepSeek-Chat、Google Gemini-2.5-Pro 和 GPT-5 作为驱动各个智能体的基础模型.

本文所有实验均在同一台高性能工作站上执行, 其硬件配置为 Intel Core i9-12900H 处理器, 核心数为 20 核, 主频 5.00 GHz, 配备 32 GB 系统内存. 软件环境为 Arch Linux 操作系统, 内核版本为 6.16.4-4.

(2) 数据集

为了系统性地评估 MARC 方法在真实工业级 RTL 设计中的有效性, 并验证其应对核心挑战 1 和 2 的能力, 本文选取 HACK@DATE 2025 竞赛作为基准测试数据集. HACK@DATE 2025 是一项顶级硬件安全竞赛, 其赛题采用了 Google 开源的 RISC-V 架构 OpenTitan SoC 作为基础, 由 RISC-V 架构的 Ibex CPU 核心及相关 IP 模块共同组成, 具备端到端数据完整性校验、安全启动以及侧信道掩码等安全特性. OpenTitan 采用 Ibex RISC-V 处理器作为主核心, 并集成了多种 IP 模块作为外设, 其中包括加密算法加速计算模块、系统管理模块和各类输入输出模块等. 本文使用 MARC 进行分析的 IP 模块汇总于表 2, 表中统计了每个 IP 的设计文件数、代码行数、文档文件数量以及文档字符数. 其中, 设计文件仅包含用于实现 IP 的 RTL 源码文件, 而设计代码行数则为这些源码文件中剔除注释和空白行后的代码行总数.

本文选择该基准测试数据集的核心优势在于, 其各项特征能够全面且精准地回应本研究的 3 个核心问题. 首先, 比赛主办方委托硬件安全专家, 在测试数据集中手动植入了与真实产品漏洞相仿的代表性安全缺陷, 其真实性与代表性为评估 MARC 的整体有效性 (RQ1) 提供了高标准的测试基准. 其次, OpenTitan 项目丰富的开源且非结构化的设计文档, 能够有效检验 MARC 能否融入设计知识以降低漏报率 (RQ2). 最后, 该 SoC 的功能模块涵盖了密码学、系统管理和通信协议等多个领域, 其包含复杂的功能逻辑所导致的上下文相关缺陷, 则为验证 MARC 能否通过深度语义分析以精准识别缺陷 (RQ3) 提供了验证条件.

(3) 评价指标

为对 MARC 框架的检测性能进行客观且量化的评估, 本文采用了一系列缺陷检测领域的标准评价指标. 评估的核心在于比较检测方法 f 所输出的缺陷集 $\mathcal{D}_{\text{found}}$ 与数据集中预先定义的真实缺陷集 $\mathcal{D}_{\text{true}}$ 之间的一致性. 根据本文对安全缺陷的定义, 每一个缺陷 d 均由其代码位置 l、类型 t 和描述 $desc$ 构成. 在进行结果比对时, 我们采用严格的匹配标准: 当且仅当 $\mathcal{D}_{\text{found}}$ 中某个被报告的缺陷 d_{found} 与 $\mathcal{D}_{\text{true}}$ 中的某个真实缺陷 d_{true} 在代码位置 l 上存在重叠, 且缺陷类型 t 完全一致时, 本文才判定该缺陷被成功检出.

表 2 数据集中的 IP 核及其设计规模

IP模块	功能描述	设计文件数	设计代码行数	文档文件数	文档字符数
adc_ctrl	用于双通道模数转换器的低功耗控制器	7	5679	8	78491
aes	具备侧信道对抗和故障注入对抗能力的AES加速器	37	14742	8	121870
csrng	密码学安全随机数生成器	12	7820	9	127943
entropy_src	对来自随机噪声源的原始熵比特进行过滤与校验, 并将其转发至密码学安全随机数生成器	18	10511	8	183955
gpio	通用输入输出通信模块, 允许软件通过I/O引脚通信	3	1326	8	60220
hmac	SHA-2密钥散列消息认证码及哈希加速器	4	4595	8	98103
keymgr	密钥管理器, 可以管理身份标识与根密钥, 保护机密资产免受软件访问, 提供密钥派生接口	14	7606	8	131282
lc_ctrl	管理设备的生命周期状态与状态转换, 并控制对密钥管理器、闪存、一次性可编程存储器以及调试接口的访问	10	5403	8	151664
lowrisc_ibex	兼容RV32指令集的RISC-V CPU核心	30	18939	8	56700
otbn	非对称密码学的可编程协处理器	24	12033	21	269330
otp_ctrl	一次性可编程存储器的控制器	15	11997	9	203156
prim	一组可复用的底层硬件组件的集合	157	22843	12	55971
spi_device	负责通过 SPI 总线进行命令和数据交互	17	31983	5	164122

基于此判定方式, 我们将检测结果分为以下 4 种情况: 1) 真阳性 (true positive, *TP*): 检测工具报告了一个缺陷, 且该缺陷与真实缺陷集中的一个条目成功匹配. *TP* 的数量表示被正确检出的真实缺陷数量. 2) 假阳性 (false positive, *FP*): 检测工具报告了一个缺陷, 但在真实缺陷集中不存在与之匹配的条目. *FP* 的数量表示被错误报告的非缺陷代码, 即误报. 3) 假阴性 (false negative, *FN*): 真实缺陷集中的某个缺陷, 没有被检测工具报告. *FN* 的数量表示被遗漏的真实缺陷, 即漏报. 4) 真阴性 (true negative, *TN*): 在代码缺陷检测的场景中, *TN* 代表未被报告的、非缺陷的广大代码片段. 由于其数量极其庞大且难以界定, 该指标在本文的性能计算中不被直接使用.

本文选取漏报率 (false negative rate, *FNR*)、误报率 (false discovery rate, *FDR*)、*F*1 分数作为评估模型的检测准确程度的核心指标. *FNR*、*FDR* 和 *F*1 的计算方法如下:

$$FNR = \frac{FN}{TP + FN} \tag{1}$$

$$FDR = \frac{FP}{FP + TP} \tag{2}$$

$$F1 = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{3}$$

漏报率衡量所有真实存在的缺陷中被遗漏的比例, 是评估安全缺陷检测工具能力的关键负向指标, 漏报率越低则模型检测效果越理想. 误报率衡量的是所有被检测工具判定为存在缺陷的结果中, 实际上并不存在缺陷的比例, 直接反映了检测结果的可靠性与实用价值, 误报率越低则意味着工具产生的噪音越少, 使用者需要付出的额外审计成本也越低. *F*1 分数作为查准率与查全率的调和平均值, 是一个用于综合评估模型整体性能的平衡指标, 能够同时量化模型对误报与漏报的抑制效果, 提供一个更为鲁棒的性能评判基准.

4.3 实验方法与结果分析

为了系统性地评估 MARC 方法在硬件安全缺陷早期检测方面的整体有效性 (RQ1), 本文将其与业界主流的传统静态分析工具 Verilator、仅基于大模型的 LLMOnly 方法进行了全面的量化比较. LLMOnly 方法直接向大语言模型提供待测 RTL 源代码并要求其找出其中存在的安全缺陷. 参与对比的 3 种方法均在同一数据集上进行实验. 其中, 本文提出的 MARC 与 LLMOnly 方法分别使用了 DeepSeek-Chat、Gemini-2.5-Pro 和 GPT-5 这 3 款不同能力层级的大语言模型进行测试. Verilator 作为传统静态 Linter 工具的代表, 其检测结果为本研究提供了对比传

统方法的基准.

整体实验结果如表 3 所示. 可以看出, 本文提出的 MARC 在所有测试模型上的综合性能显著优于 LLMOnly 基准方法, 并且全面超越了传统的 Verilator 工具. 其中, 当使用 GPT-5 模型时, MARC 取得了最佳的检测效果, *F*1 分数达到了 0.6236, 相较于同等模型下的 LLMOnly (0.5376) 提升了约 16%. 同时, MARC 也实现了最低的漏报率 (*FNR*=0.3829) 与最低的误报率 (*FDR*=0.3695), 这表明其在缺陷检出率与结果精确率之间取得了最优的平衡.

表 3 MARC 整体实验结果

Analyzer	Model	*TP*	*FP*	*FN*	*TP*+*FP*	*F*1	*FNR*	*FDR*	Token消耗	Cost ($)	Time (s)
MARC	DeepSeek-Chat	17	108	30	125	0.1976	0.6383	0.8640	517347	0.144	176.16
	Gemini-2.5-Pro	28	43	19	71	0.4745	0.4042	0.6056	686678	1.781	247.54
	GPT-5	**29**	17	**18**	46	**0.6236**	**0.3829**	**0.3695**	640764	2.346	515.81
LLMOnly	DeepSeek-Chat	16	87	31	103	0.2133	0.6595	0.8446	**434475**	**0.125**	309.14
	Gemini-2.5-Pro	18	46	29	64	0.3243	0.6170	0.7187	610659	1.184	313.05
	GPT-5	25	21	22	46	0.5376	0.4680	0.4565	574696	2.401	419.17
Verilator	—	7	**8**	40	15	0.2258	0.8511	0.5333	—	0	**136.81**

注: 加粗数据表示最优结果

从结果对比中可以观察到两个核心趋势. 首先, 多智能体协同方法的优越性得到了充分验证. 在任意一款模型的驱动下, MARC 的 *F*1 分数均高于 LLMOnly. 以 GPT-5 为例, MARC 不仅多找出了 4 个真实缺陷 (*TP*=29 vs. *TP*=25), 还减少了 4 个误报 (*FP*=17 vs. *FP*=21), 证明了本文所设计的融合设计知识与验证流程的多智能体协同方法能够有效引导大语言模型进行更精准、更全面的分析, 从而系统性地降低了漏报与误报. 其次, 引入 Verilator 作为传统工具基准, 进一步凸显了 MARC 的先进性. 实验数据显示, Verilator 的漏报率高达 0.8511, 仅能识别出 7 个真实缺陷, 表明传统 Linter 工具在发现复杂安全缺陷方面存在显著不足. 再次, 框架整体性能与底层大语言模型的能力正相关. 无论是 MARC 还是 LLMOnly, 其检测效果都随着模型从 DeepSeek-Chat 向 GPT-5 的升级而稳步提升. 这表明, 一个更强大的基础模型能够更好地执行框架赋予的复杂指令, 而 MARC 则能有效地放大这种能力优势, 将其转化为实际的检测精度提升. 综上所述, 本实验有力地回答了研究问题 RQ1, 证实了 MARC 方法在硬件安全缺陷早期检测任务上的能力.

在分析检测性能的同时, 本研究亦对不同方法的计算开销进行了量化, 涵盖 Token 消耗、经济成本 (Cost) 与执行时间 (Time), 如表 3 最后 3 列所示. 数据显示, MARC 在 Token 消耗量上普遍高于 LLMOnly, 这是由于其提示词更为复杂. 在经济成本方面, MARC 的开销通常高于 LLMOnly. 然而, 当使用 GPT-5 模型时, MARC 的成本反而略低于 LLMOnly. 这是由于 MARC 框架针对 Token 成本的优化. 由于 GPT-5 输出 Token 价格是输入价格的 8 倍, 因此 MARC 通过优化的提示词策略, 使其输出 Token 数显著低于 LLMOnly, 从而实现了经济成本的有效压缩. 在执行时间方面, Verilator 由于其不依赖 LLM, 耗时最短. 对于基于 LLM 的方法, 同一分析器下基于 DeepSeek-Chat 模型的实验轮次耗时最短, 反映出执行时间与模型能力呈正相关. 值得注意的是, 尽管 MARC 方法的框架结构更为复杂, 其执行时间反而低于 LLMOnly. 这是由于 LLM 的响应时间与输出 Token 数量有着强相关, LLMOnly 的提示词策略未经妥善优化, 消耗了更多的输出 Token 数量, 也导致了更长的执行时间.

为进一步深入分析 MARC 方法的性能, 本文还统计了其在不同硬件 IP 模块上的细分检测效果, 以实现 MARC 方法在不同功能模块上的性能一致性与差异性分析, 并评估底层模型能力对具体检测任务的影响. 本文把数据集中的 13 个 IP 模块的缺陷检测所取得的 *F*1 分数, 以热力图的形式进行可视化呈现, 如图 2 所示. 其中纵轴代表不同的 IP 模块, 横轴代表所使用的模型, 单元格的颜色深浅表示 *F*1 分数的高低, 即颜色越深, 代表检测效果越好. 从统计结果来看, 检测性能与模型能力普遍存在强正相关性, 对于绝大多数 IP 模块, 检测效果随着模型能力的提升而显著增强. 例如, 在 aes 模块上, *F*1 分数从 0.17 提升至 0.71; 在 keymgr 模块上, 更是从 0.17 大幅提升至 0.80. 在 gpio 模块上的效果尤为突出, *F*1 分数从 0 提升至 1.00, 实现了最优. 这一趋势有力地印证了更强大的基础模型能够更精准地理解复杂硬件逻辑, 从而在 MARC 方法的引导下取得更优的检测成果. 另外, 不同 IP 模块的检测难度

存在显著差异, MARC 框架的性能在不同模块间表现出明显的差异性, 这反映了不同硬件设计的内在复杂性和缺陷的隐蔽程度. 在如 gpio、aes、keymgr 等模块上, 先进模型能够取得非常高的 *F*1 分数, 表明这些模块的缺陷模式或设计意图相对更容易被模型所捕获. 而在 csrng 模块上, 所有模型均未能成功检出任何缺陷, 这是由于 LLM 缺乏对 RTL 代码物理综合语义的因果推理能力, 使得在面对特定类型漏洞时分析效果不佳, 具体将在第 4.4.2 节进行详细分析. 综上, 该细分结果表明, MARC 方法的有效性虽已得到证实, 但其在具体模块上的表现并非完全均衡, 而是同时受到底层模型能力和目标模块自身复杂性的双重影响.

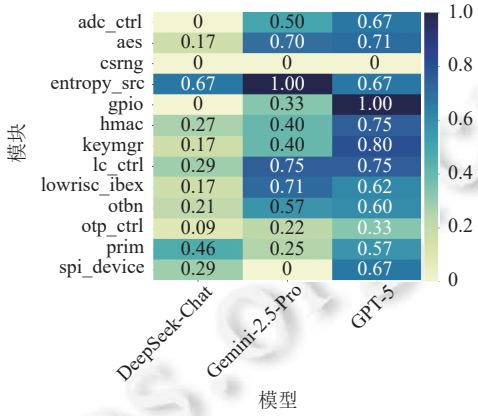

图 2 MARC 基于不同模型在不同 IP 模块下检测效果的 *F*1 分数热力图

为了验证 MARC 方法中各个核心组件的实际贡献, 并针对性地回答研究问题 RQ2 与 RQ3, 我们设计了消融实验. 该实验通过在完整的 MARC 方法基础上, 系统性地移除特定功能模块, 以评估其对最终检测性能的影响. 我们构建了两个 MARC 的降级版本, 分别对应: 1) 去除文档分析智能体, 移除完整的安全设计规范图谱和 CWE 安全规范, 旨在评估将非结构化设计知识融入检测流程对降低漏报率 (RQ2) 的整体效果; 2) 去除缺陷确认智能体, 旨在评估基于断言的精准验证对深度代码语义理解 (RQ3) 的贡献. 本文将这些降级版本的性能与完整版 MARC 进行对比, 实验结果如表 4 所示.

表 4 MARC 消融实验结果

Analyzer	Model	Precision	Recall	*F*1	*FNR*	*FDR*
MARC	DeepSeek-Chat	0.4048	0.3617	0.3820	0.6383	0.5952
	Gemini-2.5-Pro	0.5909	0.5532	0.5714	0.4468	0.4091
	GPT-5	0.6111	0.7021	0.6535	0.2979	0.3889
MARC (去除文档分析智能体)	DeepSeek-Chat	0.3659	0.3192	0.3409	0.6809	0.6342
	Gemini-2.5-Pro	0.5476	0.4894	0.5169	0.5106	0.4524
	GPT-5	0.6346	0.7021	0.6667	0.2979	0.3654
MARC (去除缺陷确认智能体)	DeepSeek-Chat	0.4667	0.4468	0.4565	0.5532	0.5333
	Gemini-2.5-Pro	0.5476	0.4894	0.5169	0.5106	0.4524
	GPT-5	0.5882	0.6383	0.6122	0.3617	0.4118

研究问题 RQ2 的核心是验证融入安全设计知识的有效性. 在 MARC 中, CWE 安全规范和最终生成的安全设计规范图谱是安全设计知识的主要载体. 从实验结果来看, 对于 DeepSeek-Chat 和 Gemini-2.5-Pro 等模型, 去除文档分析智能体导致了 *F*1 分数的明显下降, 例如 Gemini-2.5-Pro 从 0.5714 降至 0.5169, 这表明对于这些模型, 外部注入的结构化设计知识是其准确理解设计意图、降低漏报的关键. 值得注意的是, 当使用最先进的 GPT-5 模型时, 去除文档分析智能体后, *F*1 分数并未下降, 反而从 0.6535 微弱提升至 0.6667. 我们分析认为, 这是因为 GPT-5 这样的顶级模型已在其海量预训练数据中内化了极其丰富的通用安全知识. 因此, 由文档分析智能体从外部再次注入这部分通用知识时, 其边际效益递减, 甚至可能产生信息冗余或注意力噪声, 对模型的推理产生轻微干扰, 并导

致少量无法被彻底消除的误报. 另外, 结合未检出结果的案例分析, 由于本方法缺乏物理综合层面的知识和推理能力, 在部分案例上已达到能力瓶颈. 但整体而言, 该知识图谱对于引导中等能力模型、确保框架的普适性而言, 其价值是不可或缺的, 证明了融入安全设计知识的机制在整体上是成功的.

研究问题 RQ3 的核心是评估深度代码语义分析在精准识别缺陷方面的作用. MARC 中的缺陷验证模块通过精准的断言生成和验证, 是深度语义理解的关键模块. 从实验结果来看, 当去除缺陷验证模块后, 所有模型都表现出一致的性能下滑. 以 GPT-5 为例, $F1$ 分数从 0.6535 下降至 0.6122, 同时漏报率从 0.2979 显著上升至 0.3617. 这一结果清晰地表明, 将自然语言描述的潜在缺陷转化为形式化的断言是深度语义理解的最终体现, 缺乏这一关键的验证环节将严重削弱整个框架的逻辑推理与确认能力, 从而导致对复杂缺陷的误判与遗漏, 回答了 RQ3 并验证了深度语义分析的必要性.

4.4 案例分析

4.4.1 通用输入输出模块访问控制边界绕过缺陷

本缺陷位于 HACK@DATE 2025 数据集的通用输入输出 (general-purpose input/output, gpio) 模块的核心功能子模块中, 该子模块负责处理寄存器的写请求并更新 gpio 的输出状态. 缺陷具体存在于负责更新输出的使能寄存器 cio_gpio_en_q 的一个时序逻辑块中, 由于该逻辑块内的一个用于处理低 16 位掩码写操作 (MASKED_OE_LOWER) 的条件分支逻辑存在错误, 导致了控制边界绕过缺陷. 根据设计规约, 该操作本应只对输出使能信号的低 16 位进行掩码更新, 但代码实现却错误地对整个 32 位总线执行了赋值操作. 此漏洞在硬件安全分类中归属于 CWE-1262, 即寄存器接口访问控制不当. 该缺陷导致对 MASKED_OE_LOWER 寄存器的越权写操作影响了本应隔离的高 16 位输出使能状态. 攻击者仅需构造一次对 MASKED_OE_LOWER 寄存器的合法写操作, 即可在修改低 16 位 gpio 使能状态的同时, 非法控制高 16 位的 gpio 引脚. 这会破坏硬件强制执行的引脚隔离机制, 允许低权限实体非法获取高位引脚的控制权, 构成控制边界绕过安全缺陷.

该缺陷的检出过程充分体现了 MARC 的设计优势. 首先, 文档分析智能体利用 RAG 技术解析非结构化设计文档, 从寄存器映射表中提取出关键设计规约, 即 MASKED_OE_LOWER 寄存器的预期行为是其写操作应仅影响使能信号的低 16 位, 并将此知识整合至模块安全设计规范图谱, 从而克服了传统工具因知识鸿沟导致的漏报问题. 在此基础上, 缺陷检测智能体依据该图谱提供的上下文, 对 RTL 源码执行上下文语义感知的代码审计, 通过数据流分析追踪到 MASKED_OE_LOWER 的实际行为是错误地对整个 32 位总线执行了赋值. 通过对比预期行为与实际行为之间的不一致, 该智能体识别出此越权写操作, 解决了传统 Linter 因缺乏设计意图和上下文语义理解而无法检出的难题. 最后, 缺陷确认智能体进一步将此不一致转化为 SVA 断言, 即 MASKED_OE_LOWER 的写操作不应影响高 16 位, 从而高置信度地确认此边界绕过缺陷. 最终, MARC 框架通过智能体协同, 实现了从非结构化文档中自动提取安全规约, 并将其与代码的深层语义实现进行交叉验证的过程, 进而发现了传统自动化工具无法覆盖的、与设计意图紧密相关的硬件安全缺陷.

4.4.2 密码学安全随机数生成器模块信息泄漏缺陷

本缺陷位于 HACK@DATE 2025 数据集的密码学安全随机数生成器 (cryptographically secure random number generator, csrng) 模块的 csrng_reg_top 子模块中, 该子模块负责实现总线接口与寄存器文件的访问控制. 缺陷具体存在于负责处理寄存器读请求的组合逻辑块中, 由于该组合逻辑块内的一个条件选择逻辑未能完整覆盖所有可能的地址输入情况, 特别是当总线发起一次对未映射地址的读请求时, 由于默认分支不存在, 导致输出信号 reg_rdata_next 在该条件下无确定赋值. 而根据 SystemVerilog 的硬件综合语义, 这种不完整的赋值将使得综合工具为了保持信号的稳定性而推断出一个隐式的锁存器. 此漏洞在硬件安全分类中归属于 CWE-1277, 即组合逻辑中的锁存器使用问题. 该隐式锁存器引入了非预期的状态记忆功能, 当一个合法的敏感寄存器被读取后, 其数据将被该锁存器捕获. 攻击者仅需在下一个时钟周期对任一未映射的无效地址执行读操作, 便可触发该锁存器, 使其在总线上再次输出先前锁存的敏感数据, 从而导致 csrng 的内部状态被窃取, 破坏模块安全性, 即引发 CWE-200 敏感信息泄漏安全缺陷.

然而, 本缺陷未被基于任何基础模型的 MARC 分析框架所识别, 具体原因如下. 首先, 该缺陷具有高度的隐蔽性. LLM 虽擅长识别错误赋值、逻辑紊乱等显式错误模式, 但对于由代码缺失所引发的隐式缺陷, 其检测能力存在局限. 其次, 缺陷的领域特殊性超越了当前框架的设计范畴. 本文所提出的 MARC 框架其核心设计旨在解决设计知识鸿沟与代码语义缺失的挑战, 其优势在于理解高层设计意图及复杂时序逻辑. 然而, 这一缺陷并非源于设计逻辑或时序错误, 而是由于 RTL 代码在综合阶段的工程后果.

这一现象反映了当前 LLM 在硬件 RTL 代码审计方面所面临的深层挑战. 当前主流的 LLM 普遍缺乏从 RTL 描述到综合后物理电路的推理能力. 尽管如此, 这并非否定了基于 LLM Agent 的静态分析方法的价值, 反而为其指明了关键的改进路径. 未来的 Agent 架构可通过集成轻量级的形式化 Linter 与综合工具, 将推断锁存器等警告信息作为关键特征反馈给 LLM, 以有效弥补其在物理综合推理层面的短板. 与此同时, 还可以通过构建特定于硬件综合缺陷的微调数据集, 实现对 LLM 的领域微调, 以显著增强模型对 RTL 的深层理解能力. 本文认为, 这种 LLM 结合 EDA 工具的混合 Agent 架构, 有望大幅提升对早期设计阶段中隐蔽硬件缺陷的检测覆盖率, 在硬件安全领域依然具有广阔的应用前景.

4.4.3 SHA256 硬件寄存器敏感信息未清除安全缺陷

除了 OpenTitan, 本文还尝试在多个硬件 RTL 模块中进行安全缺陷检测. 其中, CVE-2025-50418 是一个基于本文方法发现的真实 RTL 模块安全缺陷, 现已获得 CVE 官方确认. 该安全缺陷存在于广泛使用的开源 SHA256 加密 IP 核 SystemVerilogSHA256 中. 该安全缺陷的类型为 CWE-1239, 即硬件寄存器敏感信息未清除. 其问题根源在于, 当哈希计算完成后或在不同用户上下文之间切换时, 未能清除存储在内部寄存器中的中间哈希值、最终状态值及其他临时计算数据. 具体而言, 在 sha_mainloop.sv 和 miner.sv 模块中, 包括 8 个中间哈希寄存器、8 个最终状态寄存器以及 1 个结果寄存器在内的多个关键状态寄存器, 在 done 信号置位后, 其内容并未被清零或擦除. 这些残留的敏感数据可被后续访问该硬件模块的非授权用户读取, 从而导致严重的信息泄露.

传统方法对该安全缺陷的检测存在局限性. 由于该安全缺陷并未影响该模块的功能正确性, 所以动态仿真方法无法发现该安全缺陷. 而常规语义级静态分析工具 Linter 由于其规则库中缺乏“加密状态在操作完成后必须擦除”这类领域特定的高级安全知识, 且无法进行深层语义推理, 不能理解 done 信号标志着关键状态转换, 因此也无法发现该安全缺陷.

MARC 能够识别此安全缺陷的关键在于多智能体协同机制系统性地克服了传统方法的局限性. 模块文档分析智能体通过检索增强生成从 CWE 通用设计规范中提取出“操作完成后必须清除敏感数据”这一核心安全规约, 并结合模块主要功能描述, 将存储哈希计算敏感数据相关的中间寄存器、最终状态寄存器和结果寄存器识别为关键安全信号, 从而弥合了传统工具在安全知识领域的空白. 在此基础上, 依赖分析与缺陷检测智能体协同进行深度代码语义分析, 它们构建代码语义图并追踪控制流, 最终确认在 done 信号置位后, 并未执行任何针对上述关键安全信号的清除操作. 最后, 缺陷确认智能体基于“安全资产在其生命周期结束后未被清除”这一事实, 进行断言生成及验证, 高置信度地判定其为 CWE-1239 类型漏洞. 通过本案例可见, 通过将非结构化的设计安全知识与对代码执行流程的深度语义理解进行结合, 才得以发现传统自动化工具无法覆盖的、与设计意图和安全规约紧密相关的硬件安全缺陷.

5 讨 论

本文清晰地展示了 MARC 方法在硬件安全缺陷早期检测方面的显著有效性. 本节将围绕实验数据, 从多智能体框架的必要性、影响性能的关键因素以及本方法在现有验证流程中的定位等维度展开深入讨论.

5.1 多智能体协同的必要性

从实验结果可以得出结论, 简单地将大语言模型应用于代码审计是不足的. 在所有测试中, MARC 方法的综合性能均显著优于 LLMOnly 基准方法, 当搭载 GPT-5 时, $F1$ 分数领先约 16%, 并同时降低了漏报与误报. 这一优势源于多智能体协同机制系统性地克服了单一模型调用的内在局限性.

LLMOnly 方法本质上是一种零上下文的概率预测, 其效果完全依赖于模型在预训练中内化的知识, 面对特定、复杂的设计意图时容易产生幻觉. 而 MARC 框架通过关注点分离原则构建多智能体范式协同架构, 系统性地将硬件领域特有的设计知识与验证手段相融合, 为大模型的强大推理能力构建了一个结构化的工作流程. 文档分析智能体将非结构化的设计规约和安全知识系统性地转化为高密度的结构化上下文, 从根本上解决了单一模型因缺乏设计意图理解而导致的漏报问题. 缺陷确认智能体扮演了验证过滤器的角色, 通过生成 SVA 断言进行二次验证, 有效抑制了大模型的幻觉, 实现误报的过滤. 这一点在成功发现 CVE-2025-50418 的案例中可以体现, 正是通过融合 CWE 安全规约与代码的深度时序逻辑分析, MARC 才得以识别出传统静态工具无法覆盖的、与设计意图紧密相关的逻辑漏洞.

5.2 性能影响因素分析

实验数据揭示了影响 MARC 框架性能的两个核心因素, 分别是底层大语言模型的推理能力和目标 IP 模块自身的复杂性. 首先, 检测性能与模型推理能力强正相关. 无论是整体结果还是细分的 IP 模块热力图, 都清晰地显示出检测效果随着模型从 DeepSeek 向 GPT-5 的升级而稳步提升. 这表明, 一个更强大的基础模型能够更精准地理解硬件描述语言的复杂语义、遵循框架赋予的复杂指令, 从而在 MARC 的引导下取得更优的检测成果. 其次, 不同 IP 模块的检测难度存在显著差异. 根据热力图统计结果, MARC 在 gpio、aes 等模块上表现优异, 但在 csrng 等模块上则表现不佳. 这反映了 MARC 的设计模式和不同缺陷的检出率有较强对应关系. MARC 的核心优势在于解决设计规约与代码实现不一致的核心挑战, 因此, 对于第 4.4.1 节中描述的 gpio 模块的缺陷, MARC 能通过融合文档规约与代码语义来精准识别此类设计逻辑缺陷. 然而, 对于第 4.4.2 节中描述的 csrng 模块缺陷, 其缺陷根源于物理综合语义问题, 并非高层设计意图的违背, 已经超出了 MARC 希望解决的硬件安全缺陷早期检测核心挑战. 但本文仍然为该问题的解决提供了思路, 即通过构建与 EDA 工具结合的 Agent 架构, 有希望提升设计阶段中隐蔽硬件缺陷的检测覆盖率, 在硬件安全领域依然具有广阔的应用前景.

6 总 结

硬件安全是信息技术体系的信任根基, 在设计早期阶段高效、精准地发现并修复 RTL 级安全缺陷, 对于降低研发成本、保障系统安全至关重要. 然而, 现有的 RTL 早期检测技术, 特别是静态分析工具, 普遍受困于设计知识鸿沟与代码语义理解缺失两大核心挑战, 导致高漏报率与高误报率问题并存, 难以满足工业界对高质量伴随式代码审计的需求. 为应对上述挑战, 本文提出了一种基于大语言模型多智能体协同的硬件安全缺陷早期检测方法 MARC. 该方法通过构建一个模拟硬件安全专家团队工作模式的协同框架, 系统性地解决了传统方法的瓶颈. 该框架包含 4 类职能明确的智能体: 依赖分析智能体负责构建系统级上下文; 文档分析智能体利用检索增强生成技术, 将非结构化的设计文档与安全规范转化为结构化的模块安全设计规范图谱, 以解决高漏报问题; 缺陷检测智能体利用该图谱进行上下文感知的深度代码审计; 缺陷确认智能体则通过自动生成并验证 SVA 断言, 对潜在缺陷进行精准过滤, 以解决高误报问题. 为了验证 MARC 的有效性, 本文在开源工业级 SoC 数据集上进行了一系列量化实验. 实验结果表明, MARC 方法的性能显著优于基线方法, 其漏报率降低至 0.382 9, 降幅约 18.2%, 误报率降低至 0.369 5, 降幅约 19.1%, 证明了多智能体协同框架在融合设计意图与代码语义方面的优越性. 这些量化优势进一步转化为实际成果, 作者团队凭借本方法不仅成功挖掘了一个已获官方 CVE 编号的真实硬件漏洞, 还获得了国际顶级硬件安全赛事 HACK@DATE 2025 的全球冠军, 充分展示了其在实际工程中的应用潜力.

未来的研究工作可从以下几个方面展开: 首先, 可以将 MARC 生成的 SVA 断言与形式化验证工具链进行深度集成, 实现从缺陷初筛到完备性证明的自动化流程. 其次, 可以进一步扩展框架以支持更多硬件描述语言, 并探索更复杂的智能体协作与自主学习机制, 以应对更大规模、更复杂的 SoC 设计. 我们相信, 随着大语言模型与多智能体技术的不断发展, MARC 方法将为构建更智能、更可靠的硬件安全自动化验证体系提供有力的技术支撑.

References

[1] The SHD Group. RISC-V market report: Application forecasts in a heterogeneous world. 2024. https://theshdgroup.com/wp-content/

uploads/2024/01/RISC-V-Market-Analysis-2024-Abridged-Report-2.pdf

[2] Dessouky G, Gens D, Haney P, Persyn G, Kanuparthi AK, Khattri H, Fung JM, Sadeghi AR, Rajendran J. HardFails: Insights into software-exploitable hardware bugs. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX Association, 2019. 213–230.

[3] Wikipedia. Software guard extensions. 2015. https://en.wikipedia.org/w/index.php?title=Software_Guard_Extensions&oldid=1306308365

[4] IPADS. Penglai-Enclave. 2025. https://github.com/Penglai-Enclave/Penglai-Enclave

[5] van Bulck J, Minkin M, Weisse O, Genkin D, Kasikci B, Piessens F, Silberstein M, Wenisch TF, Yarom Y, Strackx R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: Proc. of the 27th USENIX Security Symp. Baltimore: USENIX Association, 2018. 991–1008.

[6] Tang A, Sethumadhavan S, Stolfo SJ. CLKSCREW: Exposing the perils of security-oblivious energy management. In: Proc. of the 26th USENIX Security Symp. Vancouver: USENIX Association, 2017. 1057–1074.

[7] Costan V, Lebedev IA, Devadas S. Sanctum: Minimal hardware extensions for strong software isolation. In: Proc. of the 25th USENIX Security Symp. Austin: USENIX Association, 2016. 857–874.

[8] Brasser F, Gens D, Jauernig P, Sadeghi AR, Stapf E. SANCTUARY: ARMing TrustZone with user-space enclaves. In: Proc. of the 26th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2019. [doi: 10.14722/ndss.2019.23448]

[9] Lee D, Kohlbrenner D, Shinde S, Asanović K, Song D. Keystone: An open framework for architecting trusted execution environments. In: Proc. of the 15th European Conf. on Computer Systems. Heraklion: ACM, 2020. 38. [doi: 10.1145/3342195.3387532]

[10] Xu PY, Kuang BY, Su M, Fu AM. Survey of large-language-model-based automated program repair. Journal of Computer Research and Development, 2025, 62(8): 2040–2057 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.202440467]

[11] Thakur S, Ahmad B, Pearce H, Tan B, Dolan-Gavitt B, Karri R, Garg S. VeriGen: A large language model for Verilog code generation. ACM Trans. on Design Automation of Electronic Systems, 2024, 29(3): 46. [doi: 10.1145/3643681]

[12] Ahmad B, Thakur S, Tan B, Karri R, Pearce H. On hardware security bug code fixes by prompting large language models. IEEE Trans. on Information Forensics and Security, 2024, 19: 4043–4057. [doi: 10.1109/TIFS.2024.3374558]

[13] Li ZY, Dutta S, Naik M. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In: Proc. of the 13th Int'l Conf. on Learning Representations. Singapore: OpenReview.net, 2025. 1–24.

[14] Chapman PJ, Rubio-González C, Thakur AV. Interleaving static analysis and LLM prompting. In: Proc. of the 13th ACM SIGPLAN Int'l Workshop on the State of the Art in Program Analysis. Copenhagen: ACM, 2024. 9–17. [doi: 10.1145/3652588.3663317]

[15] Li HN, Hao Y, Zhai YZ, Qian ZY. Enhancing static analysis for practical bug detection: An LLM-integrated approach. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA1): 111. [doi: 10.1145/3649828]

[16] Li ZY, Wu JZ, Ling X, Luo TY, Rui ZQ, Wu YJ. The seeds of the future sprout from history: Fuzzing for unveiling vulnerabilities in prospective deep-learning libraries. In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Ottawa: IEEE, 2025. 1616–1627. [doi: 10.1109/ICSE55347.2025.00132]

[17] Wang LQ, Zhou YT, Li QS, Wang MK, Xu ZX, Cui D, Wang L, Luo YX. Multi-agent collaborative code reviewer recommendation based on large language model. Ruan Jian Xue Bao/Journal of Software, 2025, 36(6): 2558–2575 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7326.htm [doi: 10.13328/j.cnki.jos.007326]

[18] Lewis P, Perez E, Piktus A, Petroni F, Karpukhin V, Goyal N, Küttler H, Lewis M, Yih W T, Rocktäschel T, Riedel S, Kiela D. Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Proc. of the 34th Conf. on Neural Information Processing Systems. Vancouver: Neural Information Processing Systems Foundation, 2020. 9459–9474 [doi: 10.5555/3495724.3496517]

[19] DATE 2025 awards. 2025. https://date25.date-conference.com/awards

[20] Rui ZQ, Mei Y, Chen ZZ, Wu JZ, Ling X, Luo TY, Wu YJ. SeChain: Design and implementation of RISC-V secure boot mechanism based on domestic cryptographic algorithms. Journal of Computer Research and Development, 2024, 61(6): 1458–1475 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.202440088]

[21] Thomas F, Hetterich L, Zhang RY, Weber D, Gerlach L, Schwarz M. RISCVuzz: Discovering architectural CPU vulnerabilities via differential hardware fuzzing. 2024. https://ghostwriteattack.com/riscvuzz.pdf

[22] Liu C, Wu YJ, Wu JZ, Zhao C. Survey on RISC-V system architecture research. Ruan Jian Xue Bao/Journal of Software, 2021, 32(12): 3992–4024 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6490.htm [doi: 10.13328/j.cnki.jos.006490]

[23] Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M. Meltdown: Reading kernel memory from user space. In: Proc. of the 27th USENIX Security Symp. Baltimore: USENIX Association, 2018. 973–990.

[24] Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y. Spectre attacks: Exploiting speculative execution. In: Proc. of the 2019 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2019. 1–19. [doi: 10.1109/SP.2019.00002]

[25] Gerlach L, Weber D, Zhang RY, Schwarz M. A security RISC: Microarchitectural attacks on hardware RISC-V CPUs. In: Proc. of the 2023 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2023. 2321–2338. [doi: 10.1109/SP46215.2023.10179399]

[26] Papadimitriou G, Gizopoulos D. Demystifying the system vulnerability stack: Transient fault effects across the layers. In: Proc. of the 48th ACM/IEEE Annual Int'l Symp. on Computer Architecture (ISCA). Valencia: IEEE, 2021. 902–915. [doi: 10.1109/ISCA52012.2021.00075]

[27] Raia G, Rigano G, Vincenzoni D, Martina M. A case study on formal equivalence verification between a C/C++ model and its RTL design. In: Proc. of the 26th Int'l Conf. on Formal Methods. Milan: Springer, 2025. 373–389. [doi: 10.1007/978-3-031-71177-0_23]

[28] Godbole A, Huffman B, Liu FF, Rozas CV, Seshia SA. PYCALIPER: Python-embedded infrastructure for RTL verification and specification synthesis. In: Proc. of the 37th Int'l Conf. on Computer Aided Verification. Zagreb: Springer, 2025. 408–425. [doi: 10.1007/978-3-031-98682-6_21]

[29] Rajendran SR, Mukherjee R, Chakraborty RS. SoK: Physical and logic testing techniques for hardware Trojan detection. In: Proc. of the 4th ACM Workshop on Attacks and Solutions in Hardware Security. ACM, 2020. 103–116. [doi: 10.1145/3411504.3421211]

[30] Li XP, Li XH, Dall C, Gu RH, Nieh J, Sait Y, Stockwell G. Design and verification of the ARM confidential compute architecture. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2022. 465–484.

[31] Zhan BH, Wu ZL. Formal verification of circuit design. Science and Technology Foresight, 2023, 2(1): 23–32 (in Chinese with English abstract). [doi: 10.3981/j.issn.2097-0781.2023.01.002]

[32] Orenes-Vera M, Manocha A, Wentzlaff D, Martonosi M. AutoSVA: Democratizing formal verification of RTL module interactions. In: Proc. of the 58th ACM/IEEE Design Automation Conf. (DAC). San Francisco: IEEE, 2021. 535–540. [doi: 10.1109/DAC18074.2021.9586118]

[33] Tan QH, Yang YH, Bourgeat T, Malik S, Yan MJ. RTL verification for secure speculation using contract shadow logic. In: Proc. of the 30th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (Vol. 1). Rotterdam: ACM, 2025. 970–986. [doi: 10.1145/3669940.3707243]

[34] Trippel T, Shin KG, Chernyakhovsky A, Kelly G, Rizzo D, Hicks M. Fuzzing hardware like software. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 3237–3254.

[35] Ma RY, Huang JY, Zhang SJ, Xie Y, Luo GJ. NoCFuzzer: Automating NoC verification in UVM. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2025, 44(1): 371–384. [doi: 10.1109/TCAD.2024.3430195]

[36] Reis S, Abreu R, d'Amorim M, Fortunato D. Leveraging practitioners' feedback to improve a security linter. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2023. 66. [doi: 10.1145/3551349.3560419]

[37] CHIPS Alliance. Verible. 2020. https://github.com/chipsalliance/verible

[38] Synopsys. SpyGlass Lint: Early design analysis. 2018. https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-lint.html

[39] Ryou Y, Joh S, Yang J, Kim S, Kim Y. Code understanding linter to detect variable misuse. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2023. 133. [doi: 10.1145/3551349.3559497]

[40] Cui X, Wu JZ, Ling X, Luo TY, Yang MT, Ou WX. Exploring large language models for analyzing open source license conflicts: How far are we? In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering Companion (ICSE-Companion). Ottawa: IEEE, 2025. 291–302. [doi: 10.1109/ICSE-Companion66252.2025.00083]

[41] Liu S, Fang WJ, Lu Y, Wang J, Zhang QJ, Zhang HC, Xie ZY. RTLCoder: Fully open-source and efficient LLM-assisted RTL code generation technique. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2025, 44(4): 1448–1461. [doi: 10.1109/TCAD.2024.3483089]

[42] Wang X, Wan GW, Wong SZ, Zhang L, Liu TY, Tian Q, Ye JM. ChatCPU: An agile CPU design and verification platform with LLM. In: Proc. of the 61st ACM/IEEE Design Automation Conf. (DAC). San Francisco: ACM, 2024. 212. [doi: 10.1145/3649329.3658493]

[43] Chen YX, Wu JZ, Ling X, Li CJ, Rui ZQ, Luo TY, Wu YJ. When large language models confront repository-level automatic program repair: How well they done? In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering: Companion. Lisbon: ACM, 2024. 459–471. [doi: 10.1145/3639478.3647633]

[44] Collini L, Ahmad B, Ah-kiow J, Karri R. MARVEL: Multi-agent RTL vulnerability extraction using large language models. arXiv:2505.11963, 2025.

[45] Ahmad B, Ah-Kiow J, Tan B, Karri R, Pearce H. FLAG: Finding line anomalies (in RTL code) with generative AI. ACM Trans. on Design Automation of Electronic Systems, 2025, 30(6): 103. [doi: 10.1145/3736411]

[46] Fan FH, Xia YJ, Kuang L. SecV: LLM-based secure Verilog generation with clue-guided exploration on hardware-CWE knowledge graph. In: Proc. of the 34th Int'l Joint Conf. on Artificial Intelligence. Montreal: ijcai.org, 2025. 8049–8057. [doi: 10.24963/ijcai.2025/895]

[47] Zhang KC, Li J, Li G, Shi XJ, Jin Z. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In: Proc. of the 62nd Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Bangkok: ACL, 2024. 13643–13658. [doi: 10.18653/v1/2024.acl-long.737]

[48] Zhao YJ, Zhang HJ, Huang HX, Yu ZM, Zhao JS. MAGE: A multi-agent engine for automated RTL code generation. In: Proc. of the 62nd ACM/IEEE Design Automation Conf. (DAC). San Francisco: IEEE, 2025. 1–7. [doi: 10.1109/DAC63849.2025.11133191]

[49] Ma BQ, Zhou YH, Wang ZY, Tian ZH. A LLMs-based method for threat intelligence information extraction. Journal of Cybersecurity, 2024, 2(2): 36–46 (in Chinese with English abstract). [doi: 10.20172/j.issn.2097-3136.240203]

[50] White J, Fu QC, Hays S, Sandborn M, Olea C, Gilbert H, Elnashar A, Spencer-Smith J, Schmidt DC. A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv:2302.11382, 2023.

[51] Teo S. How I won Singapore's GPT-4 prompt engineering competition. 2023. https://towardsdatascience.com/how-i-won-singapores-gpt-4-prompt-engineering-competition-34c195a93d41/

[52] Brown TB, Mann B, Ryder N, *et al*. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 159.

[53] Mu YY, Chen HX, Li HW. Advances in security and privacy-preserving techniques for large language models. Journal of Cybersecurity, 2024, 2(1): 40–49 (in Chinese with English abstract). [doi: 10.20172/j.issn.2097-3136.240103]

附中文参考文献

[10] 许鹏宇, 况博裕, 苏铠, 付安民. 基于大语言模型的自动代码修复综述. 计算机研究与发展, 2025, 62(8): 2040–2057. [doi: 10.7544/issn1000-1239.202440467]

[17] 王路桥, 周洋涛, 李青山, 王铭康, 徐子轩, 崔笛, 王璐, 罗懿行. 基于大语言模型的多智能体协作代码评审人推荐. 软件学报, 2025, 36(6): 2558–2575. http://www.jos.org.cn/1000-9825/7326.htm [doi: 10.13328/j.cnki.jos.007326]

[20] 芮志清, 梅瑶, 陈振哲, 吴敬征, 凌祥, 罗天悦, 武延军. SeChain: 基于国密算法的 RISC-V 安全启动机制设计与实现. 计算机研究与发展, 2024, 61(6): 1458–1475. [doi: 10.7544/issn1000-1239.202440088]

[22] 刘畅, 武延军, 吴敬征, 赵琛. RISC-V 指令集架构研究综述. 软件学报, 2021, 32(12): 3992–4024. http://www.jos.org.cn/1000-9825/6490.htm [doi: 10.13328/j.cnki.jos.006490]

[31] 詹博华, 吴志林. 芯片设计形式验证. 前瞻科技, 2023, 2(1): 23–32. [doi: 10.3981/j.issn.2097-0781.2023.01.002]

[49] 马冰琦, 周盈海, 王梓宇, 田志宏. 一种基于大语言模型的威胁情报信息抽取方法. 网络空间安全科学学报, 2024, 2(2): 36–46. [doi: 10.20172/j.issn.2097-3136.240203]

[53] 牟奕洋, 陈涵霄, 李洪伟. 大语言模型的安全与隐私保护技术研究进展. 网络空间安全科学学报, 2024, 2(1): 40–49. [doi: 10.20172/j.issn.2097-3136.240103]

作者简介

芮志清, 博士生, CCF 学生会员, 主要研究领域为系统安全, 物联网安全, RISC-V 安全.

凌祥, 博士, 副研究员, CCF 高级会员, 主要研究领域为软件安全, 人工智能安全.

曹方泽, 硕士生, 主要研究领域为系统安全, 漏洞挖掘, 操作系统安全, 软件供应链安全.

罗天悦, 高级工程师, 主要研究领域为操作系统安全分析, 代码漏洞挖掘, 人工智能安全.

吴敬征, 博士, 研究员, 博士生导师, CCF 杰出会员, 主要研究领域为开源软件供应链, 软件安全, 漏洞挖掘, 操作系统.