计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330049
61(12): 3027−3053, 2024

WebAssembly 安全综述

庄骏杰　胡　霜　华保健　汪　炀　潘志中
（中国科学技术大学软件学院　合肥　230026）
（中国科学技术大学苏州高等研究院　苏州　215123）
（zhuangjj@mail.ustc.edu.cn）

Survey of WebAssembly Security

Zhuang Junjie, Hu Shuang, Hua Baojian, Wang Yang, and Pan Zhizhong
（*School of Software Engineering*, *University of Science and Technology of China*, *Hefei* 230026）
（*Suzhou Institute for Advanced Research*, *University of Science and Technology of China*, *Suzhou* 215123）

Abstract　WebAssembly is an emerging binary instruction set architecture and code distribution format, providing a unified compiling target for high-level programming languages. Due to its design advantages such as efficiency, security and portability, WebAssembly has already been widely used in the Web and non-Web scenarios, and it is becoming one of the most promising platform-independent language runtimes. Although WebAssembly provides a variety of advanced features to guarantee its security, existing studies have demonstrated that WebAssembly still has unique attack surfaces leading to serious security issues. These security issues pose challenges to the security of whole ecosystems based on WebAssembly. Therefore, it is critical to study the security issues of WebAssembly and their mitigations. To the best of our knowledge, we present the first study of WebAssembly security, based on 42 published research papers in this area. First, we systematically analyze and summarize key security features of WebAssembly. Second, we propose the first four-layer threat model for WebAssembly: threats from high-level programming languages, compilation toolchains, binary files, and WebAssembly virtual machines. Third, we propose a taxonomy to classify current research efforts into four categories: empirical security study, vulnerability detection and exploitation, security enhancements, and formal semantics and verifications. Finally, we point out potential challenges to be addressed in this field, as well as five future research directions to be explored.

Key words　WebAssembly; language security; vulnerability detection and exploitation; security enhancement; formal verification

摘　要　WebAssembly 是一种新兴的二进制指令集体系结构与代码分发格式，旨在为高级程序语言提供统一且架构无关的编译目标. 由于其安全、高效与可移植等先进特性，WebAssembly 在 Web 领域与非 Web 领域均得到了广泛应用，正在成为最有前景的跨平台公共语言标准之一. 尽管 WebAssembly 提供了多种先进特性以保证安全性，然而，已有研究表明，WebAssembly 仍然存在特有的攻击面从而导致安全问题，这些安全问题直接影响到基于 WebAssembly 的整个软件系统生态. 因此，对 WebAssembly 安全问题的产生机理、现有解决方案以及亟待解决的科学问题展开系统研究尤为重要. 基于 WebAssembly 安全研究领域已经公开发表的 42 篇研究论文，对 WebAssembly 安全的相关研究进行了系统研究、分析、归纳和总结：首

收稿日期：2023-01-30；**修回日期**：2023-08-16
基金项目：国家自然科学基金项目（62072427, 12227901）；中国科学院稳定支持基础研究领域青年团队计划项目（YSBR-005）；中国科学技术大学学术带头人培养项目
This work was supported by the National Natural Science Foundation of China (62072427, 12227901), the Project of Stable Support for Youth Team in Basic Research Field, Chinese Academy of Sciences (YSBR-005), and the Academic Leaders Cultivation Program, University of Science and Technology of China.

先,研究分析了 WebAssembly 的核心安全特性,并在此基础上首次提出了 WebAssembly 的 4 层安全威胁模型,包括高级语言支持、编译工具链、二进制表示和语言虚拟机,并对每一层的安全威胁和攻击面进行了详细讨论;其次,提出了 WebAssembly 安全研究的分类学,将已有研究划分为安全实证研究、漏洞检测与利用、安全增强、形式语义与程序验证 4 个热点研究方向,并对这 4 个方向分别进行了综述、分析和总结;最后,指出了该领域待解决的科学问题,并展望了 5 个潜在的研究方向.

关键词 WebAssembly;语言安全;漏洞检测与利用;安全增强;形式化验证

中图法分类号 TP312

软件系统是现代信息基础设施的重要部分,已经渗透到现代社会的方方面面.软件系统的安全性与可靠性,不但保证了整个信息基础设施的稳定运行,更是关系到国计民生的重要课题[1].近年来,随着万物互联时代的到来,物联网[2]、区块链[3-4]、边缘计算[5]、函数即服务(function as a service, FAAS)[6]等复杂多样的新型计算场景不断涌现,这些场景对高安全、高效率和可移植的通用二进制代码格式提出了更加迫切的需求,而传统的 x86 和 A-RM 等本地二进制格式、Java 字节码[7]和. Net[8]等中间代码格式,由于其安全脆弱性、执行效率低或平台强依赖强等内在技术局限性,已经无法适应快速发展的新型计算场景的需求.

WebAssembly(Wasm[9])是为了适应万物互联时代日益复杂多样的程序运行场景而提出的一种安全、高效、可移植的二进制指令集体系结构和代码分发格式.首先,Wasm 通过在设计中引入强类型系统[10]、软件故障隔离[11]、安全控制流[12]、线性内存[13]等多种安全语言特性,保证了程序运行的安全性.其次,Wasm 采用基于栈式虚拟机的抽象指令集,并在设计中兼顾了空间占用与执行效率,使其能够充分利用各种平台上的硬件功能,实现了接近原生代码的高执行效率.最后,Wasm 是一种与高级语言和具体执行平台都无关的指令格式,并且依靠其丰富的语言生态,以及多种编译工具链与虚拟机的支持,Wasm 程序可以快速部署到各种运行环境中,从而实现了良好的可移植性.

因其安全、高效、可移植的先进特性,Wasm 已经在 Web 领域与非 Web 领域都得到了广泛应用.首先,在 Web 领域,Wasm 已经成为继 HTML, CSS, JavaScript 后的第 4 个 Web 语言标准规范[14],并且已经得到所有主流浏览器的支持[15].其次,在非 Web 领域,随着 WebAssembly 系统接口(WebAssembly system interface, WASI)[16]技术的出现,Wasm 程序可以通过调用本地函数直接与底层操作系统交互,这使得 Wasm 程序能够脱离浏览器,直接独立地运行在主机、云或者边缘计算环境中[17-21].伴随着 Wasm 生态系统和应用领域的快速发展,Wasm 有望在将来成为最重要的通用二进制代码格式之一.

虽然 Wasm 的重要设计目标是通过引入多种安全特性来保证和实现安全性,但是,已有关于 Wasm 安全的研究成果表明,Wasm 的新特性也带来了全新的安全风险与攻击面,进而导致了新的安全问题[22-24].由于这些安全问题遍布于 Wasm 生态系统的各个层面,并且该研究领域具有一定的前瞻性和新颖性,所以目前尚未有研究工作对 Wasm 安全的相关问题与研究进展进行系统的梳理、分析及总结,也尚未指出当前还存在的亟待解决的科学问题,并讨论未来可能的研究方向.

本文的研究目标是通过对 Wasm 安全研究领域的最新研究进展进行系统的研究、梳理、讨论和总结,提炼重点研究方向,提出关键研究问题,并探讨未来可能的研究方向和机会,从而为 Wasm 生态安全以及二进制安全研究领域的研究者提供有价值的参考.

为了对该领域的研究工作进行系统地调研、分析和总结,我们首先按照 4 个步骤收集并筛选自 2017 年以来(Wasm 于 2017 年正式对外发布)正式公开发表的研究文献:1)使用 Google 学术搜索引擎,以及 ACM, IEEE, Springer, CNKI 等论文数据库;2)检索关键字,包括英文搜索引擎中的“WebAssembly security”和中文搜索引擎中的“WebAssembly 安全”等;3)检索从 2017 年至今的全部文献;4)对按照 1~3 步骤检索得到的论文进行人工筛选和复核,筛选重要的研究论文,并总结该领域活跃的研究方向.最终,我们通过筛选得到了共计 42 篇论文.

根据对文献的调研与分析结果,结合对 Wasm 安全特性的研究与分析,本文提出了 Wasm 安全威胁模型,系统总结了 Wasm 可能受到的 14 类重要安全威胁,并将这些安全威胁系统划分为高级语言支持、编译工具链、二进制表示和语言虚拟机 4 个层面,且对

每一层面的安全威胁进行了详细讨论. 此外, 本文将 Wasm 安全研究工作总结归纳为 4 个重要研究方向: 1)安全实证研究, 主要采用实证研究方法, 针对高级语言支持、编译工具链以及二进制表示 3 个方面展开研究; 2)Wasm 漏洞检测与利用研究, 主要采用程序分析、模糊测试、深度学习等技术, 研究对 Wasm 安全漏洞进行静态或动态检测的有效技术, 以及对 Wasm 漏洞的利用方法; 3)Wasm 安全增强研究, 主要使用代码混淆、访问控制、硬件增强等技术, 对 Wasm 程序及运行时进行安全增强; 4)Wasm 形式语义与程序验证研究, 主要使用形式语义以及程序验证技术, 对 Wasm 进行形式化语义描述以及程序属性证明. 本文对这 4 个方面的研究都进行了综述、深入分析和总结, 指出了现有研究的不足和该领域亟待解决的科学问题, 并展望了 5 个潜在的研究方向.

本文系统研究和总结 Wasm 安全相关研究进展, 主要贡献有 3 点:

1)系统研究和总结了 Wasm 的核心安全特性, 并首次提出了包括高级语言支持、编译工具链、二进制表示和语言虚拟机的 4 层安全威胁模型;

2)提出了 Wasm 安全研究的分类学, 将已有研究划分为安全实证研究、漏洞检测与利用、安全增强、形式语义与程序验证 4 个热点方向, 并对每个方向都进行了深入分析和总结;

3)指出了该领域亟待解决的重要科学问题, 并展望了 5 个潜在的研究方向.

1 Wasm 概述

Wasm 是一种新兴的二进制指令集体系结构和代码分发格式, 本节对其进行概述: 首先介绍 Wasm 的历史、概况和生态系统; 然后总结和分析 Wasm 的主要安全特性; 最后对已有的 Wasm 安全研究工作进行概述和分类.

1.1 Wasm 历史与概况

Wasm 是一种新兴的、仍在快速演进和发展的二进制指令集体系结构和和代码分发格式. 2015 年, Google 和 Mozilla 团队正式提出 Wasm 的设计[25], 通过吸收借鉴 NaCl[26] 的沙箱执行和 Asm.js[27] 的类型化等先进设计理念和实践经验, 给 Wasm 确立了安全、高效、可移植的主要设计目标. 2017 年, Firefox, Chrome, Safari, Microsoft Edge 这 4 大主流浏览器在 Wasm 的标准上达成共识, Wasm 成为浏览器上的事实标准[15]. 2018 年, Wasm 核心规范 1.0 版本正式发布[28], 对 Wasm 进行了完整的形式化定义. 2019 年, W3C 正式宣布 Wasm 成为官方 Web 标准, 这标志着 Wasm 成为继 HTML, CSS, JavaScript 之后的第 4 种 Web 语言[14]. 同年, WASI[29] 的标准化工作正式启动, 旨在为 Wasm 在 Web 领域之外的应用定义一种安全的官方标准. 2022 年, Wasm 核心规范 2.0 版本草案正式发布[30], 该草案对 Wasm 进行了类型和指令扩展, 旨在进一步增强 Wasm 的表达能力和提高其执行效率.

由于 Wasm 兼具安全、高效、可移植等先进特性, 已经在 Web 领域和非 Web 领域都得到了广泛应用. 首先, 在 Web 领域[31], 目前所有主流浏览器都已经支持 Wasm[32], 并且, 随着 Wasm 官方标准[30] 的完善, Wasm 在 Web 领域内的应用已日趋完善和成熟. 其次, 在非 Web 领域, 随着 WASI[16] 等新特性的引入, Wasm 作为一种独立于高级语言以及硬件平台的通用二进制指令格式, 已经被广泛应用到无服务器云计算[33-36]、物联网和嵌入式设备[20]、区块链[37-38]、边缘计算[17,19,21]、机器学习[39]、游戏引擎[40] 等新型计算场景中. 可以预见, 随着 WASI 相关标准的不断完善和增强, Wasm 将会在更多领域得到广泛应用.

1.2 Wasm 生态系统

随着 Wasm 核心规范的不断完善和在应用领域的不断拓展, Wasm 已经构建了一个完整而强大的生态系统. 本文认为, Wasm 的生态系统可划分为 4 个层面: 高级语言支持、编译工具链、二进制表示以及语言虚拟机. 图 1 给出了 Wasm 生态系统 4 个层面的划分, 以及每个层面中应用较为广泛和典型的技术.

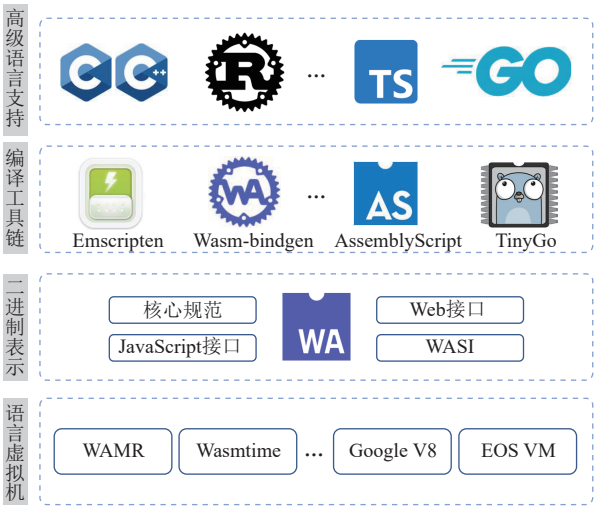

Fig. 1 Overview of Wasm ecosystem

图 1 Wasm 生态系统概况

1.2.1 高级语言支持

Wasm 作为一种通用编译目标, 正在得到越来越

多高级语言的支持. 尽管 Wasm 发布的时间不长, 但目前已经成熟稳定地支持 C/C++, Rust, Go, Python, TypeScript 等 10 余种高级语言[41], 这些语言涵盖了系统编程、数据科学、云计算、人工智能和 Web 等广泛的领域. Wasm 不仅为这些语言提供了通用编译目标, 其先进特性也进一步增强了高级语言的功能. 例如, 对于 Python 程序, 开发者可以将热点模块编译为 Wasm 来运行, 从而利用 Wasm 高效率的特性来显著提升 Python 应用程序的执行效率[42].

但另一方面, 不安全的高级语言(如 C/C++等)很容易导致程序出现安全漏洞[43-44]; 而安全的高级语言, 也可能包含不安全的语言子集(如 Rust 中的 unsafe 机制[45])或包括不安全的外部函数接口(如 Python 的 Python/C 接口[46])等. 高级语言层的这些安全漏洞或脆弱性, 会进一步传播到 Wasm 生态系统的其余各层, 威胁到整个 Wasm 生态系统的安全性.

1.2.2 编译工具链

编译工具链将各种高级语言编写的高层源程序, 高效正确地编译为等价的底层 Wasm 目标程序. 目前, 支持 C/C++, Rust, TypeScript, Go 等高级语言到 Wasm 的编译工具链 Emscripten[47], Rustc/Wasm-bin-dgen[48-49], AssemblyScript[50], TinyGo/LLVM[51] 等已经得到了广泛应用, 并且仍在快速迭代中.

但另一方面, 由于高级语言与 Wasm 在类型系统、内存模型、并发编程机制等方面有显著差异[10-13], 因此将高级语言程序编译为安全、高效、等价的 Wasm 程序是编译工具链设计面临的一大挑战. 不正确或包含漏洞的 Wasm 编译工具链不仅可能改变编译后目标程序的语义, 甚至可能引入安全漏洞[22], 对 Wasm 生态系统安全造成威胁.

1.2.3 二进制表示

Wasm 的二进制表示规定了指令集体系结构定义和二进制格式规范. Wasm 的二进制表示主要包括 4 个组成部分: Wasm 核心规范[30,52]、Wasm Web 接口[53]、Wasm JavaScript 接口[54] 和 Wasm 系统接口 WASI[16]. Wasm 的二进制表示充分吸收借鉴了底层安全语言设计的理论成果和实践经验[7,26-27], 在规范和接口定义中大量使用了严格的形式化理论和方法[55], 这为 Wasm 实现其安全性的设计目标奠定了坚实基础.

但另一方面, 为追求极致运行性能, Wasm 在二进制表示中舍弃了许多重要的安全防御机制. 例如, Wasm 不提供金丝雀等栈溢出检查安全机制, 这可能进一步导致缓冲区溢出漏洞[24]. Wasm 二进制表示中重要安全机制的缺失, 给二进制程序带来了易被利用的攻击面, 进而给整个 Wasm 生态系统带来严重的安全威胁[43-44].

1.2.4 语言虚拟机

作为一种虚拟指令集体系结构, Wasm 程序运行在 Wasm 语言虚拟机上. Wasm 虚拟机包括字节码编译、内存管理、接口实现等重要组件, 负责执行 Wasm 程序并和执行环境交互. Web 领域的 Wasm 虚拟机, 如 Google V8[56], SpiderMonkey[57] 等, 已经实现了对 Wasm 的完整支持. 而随着 WASI 接口规范的推出和完善, 在区块链、物联网等非 Web 领域也涌现出一系列的 Wasm 虚拟机实现[58], 包括 WasmEdge[21], EOS VM[59], WAMR[60], Wasmtime[61], Wasmer[62] 等. 这些语言虚拟机, 对于 Wasm 生态系统起到了基础性支撑作用.

但另一方面, Wasm 虚拟机也存在易被利用的攻击面. 首先, 为追求极致执行性能, 许多广泛使用的 Wasm 虚拟机用 C/C++等不安全的语言实现, 这容易导致虚拟机实现本身出现安全漏洞[63]. 其次, Wasm 虚拟机的实现还可能包括逻辑错误和漏洞, 如机器码生成错误[64]、沙箱逃逸[65] 等, 这些错误和漏洞也对 Wasm 生态系统构成了安全威胁.

1.3 Wasm 核心安全特性

Wasm 作为底层字节码指令格式, 充分吸收借鉴了安全语言设计[7,66-67] 及软件安全领域[68] 的最新研究成果, 引入了一系列核心安全特性, 在保证高执行效率的前提下实现了语言安全性. 这些核心安全特性包括强类型系统、安全控制流、软件故障隔离、线性内存等. 本节对 Wasm 的核心安全特性进行提炼和总结.

1.3.1 强类型系统

Wasm 的初始设计[55] 就引入了静态强类型系统, 并证明了该类型系统的类型安全性[69]. 和已有类型系统相关研究相比, Wasm 的强类型系统具有 4 个方面的技术特点和优势: 1)Wasm 的类型系统更加简单, 这使得其安全性更容易保证和证明. 例如, 和 Java 字节码[7] 等包括类和接口等特性的复杂类型系统相比, Wasm 只有 4 种基本类型, 即 32 位和 64 位整数(i32, i64)以及 32 位和 64 位浮点数(f32, f64)[10]. 高级语言中的复杂类型都在编译阶段被编译为这类基本类型. 2)Wasm 中的操作数栈具有确定的静态类型[13], 类型检查的过程不涉及复杂耗时的类型推断[70], 因此类型检查过程更加高效. 而已有研究, 如类型化汇编语言(typed assembly language, TAL)[66] 或携带证明的代码(proof-carrying code, PCC)[67] 等, 都需要维护变量的类型环境, 类型检查过程相对复杂且低效. 3)Wasm 的强类型系统强调在静态时完成程序的安全性检查,

减少了非必要的运行时检查. 因此, 与其前身 Asm.js 的弱类型系统[27] 相比, Wasm 的强类型系统设计有助于在不牺牲安全性的前提下, 进一步提高程序的执行性能. 而且, Wasm 类型系统相对简洁的设计, 令其更容易支撑各种不同抽象层级高级语言的编译[47-51].

1.3.2 安全的控制流

Wasm 引入了 2 个技术来实现控制流安全性: 结构化控制流和控制流完整性检查.

1)结构化控制流. Wasm 的控制流是结构化的[71], 即引入了 loop 循环语句和 if 条件语句, 而没有其他底层语言[72] 中常见的 goto 等非结构化控制流语句. Wasm 的结构化控制流特性, 保证其能够阻止经典的控制流攻击[24], 最终实现控制流的安全性. 例如: 经典的 Shellcode 注入[73] 攻击通过对二进制程序进行代码注入, 从而使被攻击程序执行任意的恶意指令, 而 Wasm 结构化控制流的设计, 使得攻击者即使注入了恶意代码, 也无法令控制流跳转到该代码, 从而实现有效的安全防御.

2)控制流完整性检查. Wasm 控制流完整性检查, 包括直接函数调用检查、函数返回检查以及间接函数调用检查 3 个方面[74]. 首先, Wasm 直接函数调用检查通过函数索引空间[55] 中的索引来指定被调用函数并进行函数签名校验, 如果函数签名不匹配, 则该调用会触发校验异常并终止调用; 其次, Wasm 函数返回检查通过使用托管内存的调用栈, 来保护函数返回地址; 最后, Wasm 间接函数调用检查通过在运行前对函数进行类型检查, 保证了基于类型的粗粒度的控制流完整性.

1.3.3 软件故障隔离

软件故障隔离(software fault isolation, SFI)[68] 将存储、读取和跳转等指令沙箱化到独立的内存段, 从而安全地运行不受信任的代码. Wasm 在设计中采用了软件故障隔离技术, 将每个模块都运行在沙箱环境中, Wasm 代码与外部环境交互只能通过特定的 WASI 接口进行[16]. 本文认为, Wasm 采用基于沙箱的软件故障隔离技术, 同时实现了安全性和高执行性能. 首先, 该技术实现了沙箱内的 Wasm 模块之间、沙箱和浏览器等宿主环境之间的隔离, 有效提高了安全性; 其次, Wasm 沙箱都运行在同一进程中, 有效降低了沙箱启动和切换的时间开销. Wasm 在安全和性能这 2 方面的优势, 对于其在物联网[20] 和 FAAS[18] 等领域的成功应用, 起到了关键的支撑作用.

1.3.4 线性内存

Wasm 的内存模型包括托管内存和线性(非托管)内存 2 部分. 托管内存由 Wasm 虚拟机直接管理, 包括局部变量表、全局变量表等; 而 Wasm 线性内存为专门存储 Wasm 程序中的变量开辟的一段按字节寻址的连续存储空间, 其由辅助栈、堆和静态数据等组成[30]. 作为示例, 图 2 给出了一个由 C 程序编译得到的 Wasm 程序, 以及该 Wasm 程序在 Wasm 虚拟机中执行时的内存模型. 首先, Wasm 虚拟机在托管内存为 Wasm 程序建立全局变量表, 其中存储了全局变量

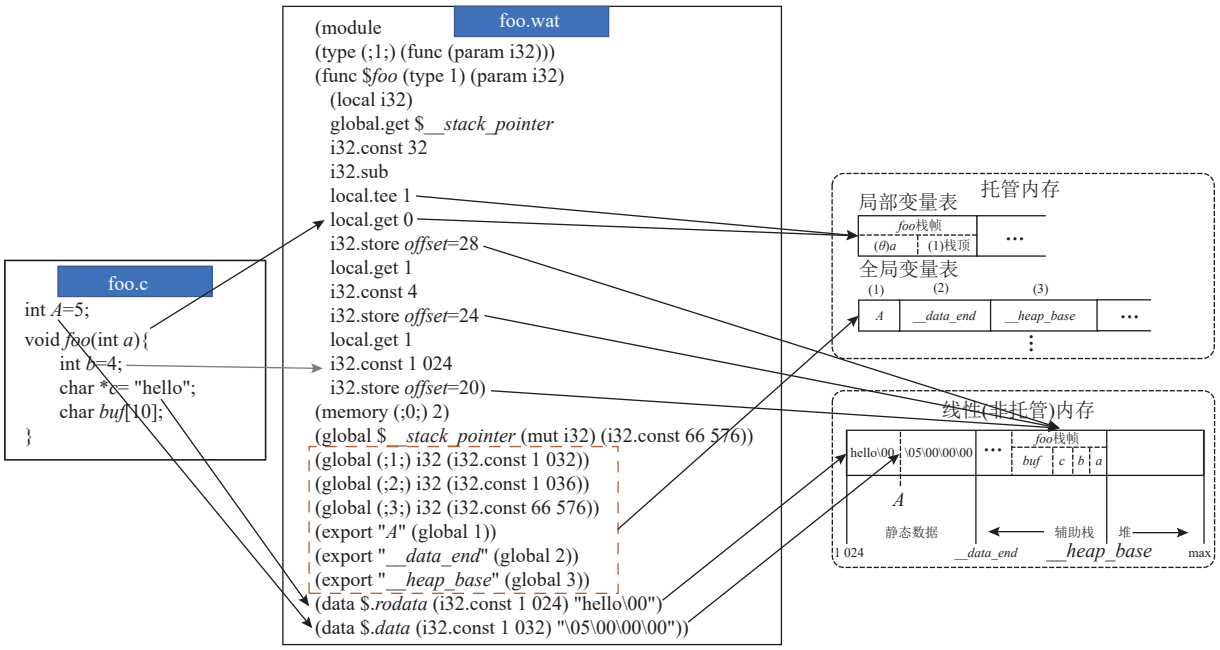

Fig. 2 Program and memory model of Wasm

图 2 Wasm 程序与内存模型

的地址, 全局变量自身, 如变量 *A* 的值则存储在线性内存中; 其次, 当函数被调用时, 虚拟机在托管内存的局部变量表中给被调用的函数, 如函数 *foo* 建立栈帧, 栈帧包含了函数的参数, 如参数 *a*、函数返回地址等信息; 同时, Wasm 虚拟机在线性内存中的辅助栈中为被调用的函数也被分配 1 个栈帧, 如函数 *foo* 的栈帧, 该栈帧上存储了函数的所有局部变量, 如变量 *b*, *c*, *buf* 等, 特别地, 函数参数也被拷贝到辅助栈的栈帧上, 如参数 *a*. 总之, Wasm 通过对每个函数引入 2 个栈帧, 将函数的返回地址和函数局部变量分别放置在托管内存栈帧和线性内存栈帧上, 从而实现了二者的分离.

Wasm 线性内存的独特设计, 实现了沙箱隔离、越界检查以及索引请求数据等 3 方面的安全保证[55]. 首先, Wasm 程序的线性内存是沙箱隔离的, 它与其他 Wasm 程序的托管内存以及 Wasm 虚拟机的堆栈、操作数栈等数据结构相互独立, 即 Wasm 程序不能破坏其执行环境、不能跳转到内存的其他任意位置或执行其他未定义的访存行为; 其次, Wasm 会动态检查所有内存访问, 确保所有访问都不会越界(即不能超过 max); 最后, Wasm 程序不能直接访问线性内存中的数据, 只能通过虚拟机向托管内存请求数据地址索引, 再根据索引获取 Wasm 线性内存中的数据, 这使得攻击者无法通过内存地址直接对线性内存中的数据进行修改.

1.4 研究框架

尽管 Wasm 引入了一系列安全特性, 以期实现其安全性的设计目标, 但是已有研究结果表明, Wasm 仍然存在许多安全漏洞, 而且这些安全漏洞来自 Wasm 生态系统的各个层面, Wasm 安全研究已经成为当前的研究热点. 本文基于对该领域已有研究的全面调研、深入分析和总结, 提出了 Wasm 的安全威胁模型和对现有研究的分类方法学. 首先, 本文将 Wasm 安全威胁模型分为 4 层, 即高级语言支持、编译工具链、二进制表示和语言虚拟机, 主要包括 14 类安全漏洞和攻击面. 其次, 本文将已有 Wasm 安全研究总结凝练成 4 个大的研究方向: 1)安全实证研究; 2)漏洞检测与利用研究; 3)安全增强研究; 4)形式语义与程序验证研究.

这 4 个研究方向是 Wasm 安全研究的不同方面, 同时它们又具有非常紧密的内在联系. Wasm 安全实证研究的结果对于其他研究方向, 包括漏洞检测与利用、安全增强和程序验证, 都具有重要指导意义; 根据 Wasm 安全实证研究的结果, 可以提出有效的漏洞检测技术来发现并修复软件系统中的安全漏洞. 对于已知的安全漏洞, 对应的安全增强技术可以阻止漏洞的发生, 从而增强软件系统的安全性. 形式语义与程序验证技术通过严格的数学方法和证明来验证 Wasm 软件系统的安全属性或功能正确性, 从而为软件系统提供更强的安全保证.

按照上述研究方向分类, 表 1 给出了对已发表文献的分类统计. 通过对表中的数据进行分析得出: 漏洞检测与利用研究的占比最高, 达到 52.27%, 而安全实证研究、安全增强研究和形式语义与程序验证研究的占比分别为 9.09%, 20.46%, 18.18%. 在接下来的各节, 本文将分别对 Wasm 安全威胁模型以及上述 4 个研究方向进行综述、深入分析和总结, 指出现有研究的不足, 并提出潜在的科学问题以及未来重要研究方向.

Table 1 Classification Statistics of Wasm Security Research

表 1 Wasm 安全研究分类统计

研究方向	篇数	占比/%
安全实证	4	9.52
漏洞检测与利用	21	50.00
安全增强	9	21.43
形式语义与程序验证	8	19.05

2 Wasm 安全威胁模型

Wasm 的新特性以及丰富的生态系统, 为其引入了新的安全威胁和更大的攻击面. 基于本文提出的 Wasm 生态系统划分, Wasm 的安全威胁模型也包括高级语言支持、编译工具链、二进制表示和语言虚拟机 4 个层面. 本节将详细分析这 4 个层面, 共计 14 类主要安全漏洞. 详细的安全威胁模型和具体漏洞分布如表 2 所示.

2.1 高级语言支持

高级语言支持是 Wasm 生态中的重要组成部分, 高级语言的不安全特性带来的安全问题, 也会为 Wasm 引入安全威胁. 本文认为, 高级语言支持对 Wasm 构成的安全威胁主要有 2 个来源: 高级语言本身的脆弱性和代码漏洞. 本文将典型的由高级语言支持导致的安全漏洞划分为 4 类: 越界访问漏洞、格式化字符串漏洞、整型溢出漏洞和释放后使用漏洞.

1)越界访问漏洞. 如果程序对内存的读写超过了合法的范围, 则会导致越界访问漏洞. 由于 C/C++

Table 2 Security Threat Model of Wasm

表 2 Wasm 安全威胁模型

威胁层面	安全漏洞	根因分析
高级语言支持	越界访问、格式化字符串、整型溢出、释放后使用	由高级语言的不安全特性引入，编译后得到的 Wasm 程序包含易被利用的漏洞.
编译工具链	内存分配器缺陷、类型转换错误、危险库函数	编译工具链无法对高级语言与 Wasm 的差异进行正确转换.
二进制表示	非托管栈溢出、非托管栈上的缓冲区溢出、堆元数据损坏、间接调用重定向	Wasm 线性内存的必要安全检查缺失.
语言虚拟机	主机环境注入、侧信道攻击、沙箱逃逸	Wasm 虚拟机或即时编译器的安全检查缺失或实现错误.

缺乏必要的边界检查机制，极易出现由越界访问漏洞导致的安全问题. 图 3 展示了一个经典的越界访问漏洞，由于第 4 行缺少了边界检查，不安全的函数 *strcpy* 可以写越过数组 *b* 的边界，而将数据写入数组 *a* 中，并进而覆盖其他内存的数据. 而 Wasm 对 C/C++ 的支持，使得 C/C++程序中的越界访问漏洞成为 Wasm 生态中的一个严重攻击面[24]. 攻击者可以利用越界访问漏洞，实现对 Wasm 非托管栈上的缓冲区溢出攻击，导致线性内存中的重要数据被覆盖. 越界访问漏洞的严重性主要体现在攻击者可以利用它作为起点，最终在 Wasm 中实现一系列端到端的高级攻击，如跨站脚本攻击、远程代码执行和任意文件写入等.

```
1 void foo(){
2   char a[4]="AAAA";
3   char b[4]="BBBB";
4   strcpy(b, "ABCDEFGHIJK");
5 }
```

Fig. 3 An example of out-of-bounds access vulnerabilities

图 3 越界访问漏洞例子

2)格式化字符串漏洞. 如果攻击者能够控制格式化字符串的输入，则容易导致格式化字符串漏洞. 该漏洞可导致敏感信息泄露或被篡改、缓冲区溢出等安全问题. 高级语言程序中存在的格式化字符串漏洞，会导致 Wasm 线性内存中的数据发生泄露[23]. 并且，由于 Wasm 的线性内存缺少必要的内存保护机制[23]，格式化字符串漏洞会导致任意内存读写，从而引发更严重的后果.

3)整型溢出漏洞. 当整型运算的结果超过整型数的可表示范围，会导致整型溢出漏洞. 整型溢出除了影响运算结果本身的正确性外，还会进一步导致内存越界访问等其他错误[75]，引发安全漏洞. 由于目前最新的 Wasm 的标准规范[30] 仍然缺少对整数运算的溢出检查保护机制，因此，高级语言程序中存在的整型溢出漏洞，在编译得到的 Wasm 程序中仍然存在，而 Wasm 程序中的整型溢出漏洞会进一步导致数据覆盖、缓冲区溢出等安全问题[23].

4)释放后使用漏洞. 若程序访问一个已经被释放的指针，则会导致指针的释放后使用漏洞. 由于 Wasm 缺少了对指针的有效性检查，因此在高级语言程序中存在的指针释放后使用漏洞在 Wasm 中依旧存在[76]. 一旦释放后使用漏洞被攻击者利用，则会引起重要数据破坏、内存块合并、任意地址写入等安全问题.

2.2 编译工具链

Wasm 生态中丰富的编译工具链，实现了对多种高级语言的有效支持. 但是，Wasm 本身的新特性及其与高级语言间的语义差异性，给 Wasm 编译工具链带来了独特挑战，并引入了新的安全问题[22]. 本文认为，编译工具链给 Wasm 生态带来的安全漏洞和威胁主要包括 3 个方面：内存分配器缺陷、类型转换错误、危险库函数.

1)内存分配器缺陷. 由于 Wasm 的运行环境没有提供默认的内存分配器，因此 Wasm 编译工具链提供了特定的内存分配器支持. 但这类内存分配器的实现缺陷或漏洞，给 Wasm 带来了攻击面和安全威胁. 例如，Emscripten 编译器基于 dlmalloc 内存分配器[77] 设计并实现了 emmalloc 内存分配器，但 emmalloc 对数据边界检查的缺失，导致如果一个堆块的数据发生溢出，会直接影响其相邻块的堆块[24]，并引发内存安全漏洞[78].

2)类型转换错误. Wasm 编译器需要将源语言类型编译为 Wasm 支持的类型，而编译过程中引入的类型转换错误将导致安全漏洞和攻击面. 例如，JavaScript 本身并不支持 64 位整型数，为了让 Wasm 程序调用 JavaScript 的库，Emscripten 编译器需要将 Wasm 中的 1 个 64 位整型数转换为 JavaScript 中的 2 个 32 位整型数. 如图 4[22] 所示，Emscripten 为了绕过浏览器沙盒的文件读取限制，而提供了一个使用 JavaScript 实现的文件系统库 FS，若在 *fseek*() 的执行路径中涉及到多个 JavaScript 模块代码，尽管第 1 个模块中导出函数的类型是 64 位整数，但中间的 JavaScript 函数不能

```
1  int main(){
2    FILE* file_ = std::fopen("input.txt","rb");
3    if(file_){
4      std::fseek(file_, 01, SEEK_END);
5      std::fclose(file_);
6    }
7    return 0;
8  }
```

Fig. 4 An example of type coversion error

图 4 类型转换错误例子

接受该类型的参数, 从而导致类型冲突错误[22].

3)危险库函数. 由于 Wasm 和高级语言特性间的差异性, 编译工具链需要模拟提供一些特殊的库函数[79]来实现 Wasm 不支持的操作, 而这些库函数中的漏洞给 Wasm 带来了安全威胁. Wasm 引入库函数中的安全漏洞包括隐式依赖、错误路径解析、数据截断等. 此外, 目前的 Wasm 的官方标准中还不支持多线程并发机制[15], 因此, 为支持多线程而引入线程库也容易导致并发安全漏洞[22].

2.3 二进制表示

虽然 Wasm 引入了许多新特性来保证安全性, 但对这些新特性的不正确使用会导致安全漏洞和攻击面的出现. 本文认为, 在二进制表示层面, Wasm 安全漏洞主要包括 4 类: 非托管栈溢出、非托管栈上的缓冲区溢出、堆元数据损坏以及间接调用重定向.

1)非托管栈溢出. 如图 5 所示, Wasm 利用线性内存中的辅助栈(也称为非托管栈)[30]完成函数调用, 辅助栈的主要作用是存储 Wasm 函数中非基本数据类型的数据(即非 32 位或 64 位的整型和浮点型数据). 辅助栈的空间从 *_heap_base* 开始, 向低地址增长, 一直到 *_data_end* 结束. 若非托管栈的地址空间占用超过可允许的最大范围(即 *Top* < *_data_end*), 则非托管栈发生溢出. 攻击者一般可通过 2 种方式造成非托管栈的溢出: 一是控制栈分配变量的大小, 导致其超过栈帧的最大允许范围; 二是利用函数的过多或无限递归, 导致非托管栈的空间用尽, 进而发生溢出. 非托管栈溢出会覆盖栈外的全局数据或堆等其他数据区中的数据, 这可以进一步导致堆溢出、关键数据被覆盖等其他安全漏洞[24].

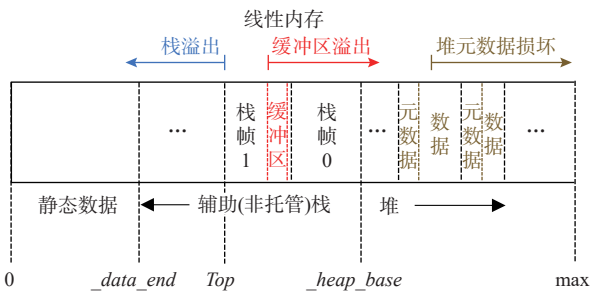

Fig. 5 The attack surface faced by Wasm linear memory

图 5 Wasm 线性内存面临的攻击面

2)非托管栈上的缓冲区溢出. 如图 5 所示, Wasm 将函数中的缓冲区放置在非托管栈的栈帧上(假设在栈帧 1 上), 若缓冲区发生溢出, 则会覆盖本栈帧(即栈帧 1)或相邻栈帧(即栈帧 0)中的数据, 甚至会覆盖除非托管栈外的其他内存区域中的数据[24]. 和传统 C/C++程序中的缓冲区溢出相比, 对 Wasm 非托管栈上的缓冲区溢出进行检测和防护存在 2 个技术挑战: 首先, 由于 Wasm 缺少栈金丝雀[80]等栈保护措施, 使得检测非托管栈上的缓冲区溢出较为困难; 其次, Wasm 的线性内存缺少读写权限的细粒度保护, 因此难以利用传统的栈不可执行等防护技术[81].

3)堆元数据损坏. Wasm 的内存分配器会在线性内存的堆中执行堆空间分配, 分配的堆元数据中包括使用位、块大小等元信息. 如图 5 所示, Wasm 堆元数据损坏是指堆上的元数据可能会因为数据的溢出而被更改或破坏, 导致元数据错误或失效. 由于 Wasm 缺少堆元数据块间的边界检查, 使得攻击者可以通过内存拷贝导致的数据溢出, 损坏堆的元数据, 进而实现任意地址写入等攻击[24].

4)间接调用重定向. Wasm 引入了间接调用函数索引表[30], 以支持高级语言中的函数指针以及虚函数等特性, 其典型结构如图 6 所示[24]. Wasm 中的间接调用重定向攻击是指攻击者可以通过覆盖线性内存中对应存储的索引号数据, 使得间接调用指令 call_indirect 指向攻击者设定的恶意函数. 例如, 在图 6 中, 索引号 *offset*=32 所对应线性内存的数据由 0 被覆盖成了 3, 则其指向的函数索引表由第 0 个表项变成了第 3 个表项, 则被调用的函数从 func1 重定向到了 func2. 尽管 Wasm 会对间接调用做类型检查, 但是如果被重定向到的函数参数类型和原函数一致, 则会发生控制流劫持攻击, 并导致执行恶意代码片段[24].

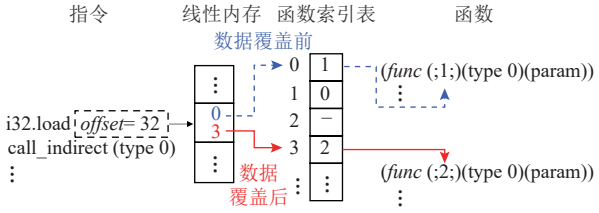

Fig. 6 Indirect call redirection

图 6 间接调用重定向

2.4 语言虚拟机

Wasm 在语言虚拟机层面的安全威胁主要来自 2 个方面: 一是攻击者通过利用 Wasm 程序中的漏洞实

现对主机环境的注入攻击；二是攻击者利用硬件与操作系统的漏洞，对 Wasm 虚拟机自身进行攻击. 本文认为，在语言虚拟机层面的 Wasm 安全漏洞主要包括 3 类：主机环境注入[23-24]、侧信道攻击[82]和沙箱逃逸[83].

1）主机环境注入. 由于 Wasm 可以通过 JavaScript 接口或 WASI[16]与浏览器或主机环境进行交互，因此，攻击者可以利用 Wasm 模块与主机环境间的接口存在的安全漏洞来实现对主机环境的注入攻击. 例如，攻击者可以通过调用 JavaScript 的函数 *eval*[84]，实现对 Wasm 主机环境的代码注入[24].

2）侧信道攻击. 攻击者通过分析加密算法所需的执行时间，来破坏加密系统完整性和可靠性的恶意攻击称为定时侧信道攻击[82]. 如果 Wasm 虚拟机中缺少对程序的恒定时间执行的限制，则攻击者能够通过对 Wasm 虚拟机进行定时侧信道攻击以获取 Wasm 程序中的加密信息，造成重要信息的泄露.

3）沙箱逃逸. 攻击者可以利用特定的漏洞，使得 Wasm 程序能够脱离沙箱的限制，从而实现沙箱逃逸. 攻击者可以实现的典型沙箱逃逸是利用幽灵攻击[83]，通过混淆硬件控制流预测的组件（条件分支预测、分支目标缓冲区等）来对沙箱边界外的代码进行推测执行访问. 同时，由于 Wasm 的每个模块都独立运行在同一个进程的沙箱之中，因此 Wasm 正在设计中的多线程特性[15]也可能引发沙箱逃逸相关的安全问题.

2.5 本节小结

本节介绍了 Wasm 的安全威胁模型，该模型完整涵盖了 Wasm 生态系统的 4 个层面，即高级语言支持、编译工具链、二进制表示以及语言虚拟机；详细分析了这 4 个层面中共计 14 种典型安全漏洞. 通过研究和建立 Wasm 生态系统的安全威胁模型，研究者可以深入理解 Wasm 生态系统的脆弱点和安全风险，从而为系统开展安全实证、漏洞检测和安全增强等研究奠定基础.

3 Wasm 安全实证研究

实证研究强调在观察和实验的经验事实上，通过经验观察的数据和实验研究的手段，来揭示一般结论、归纳事物的本质属性和发展规律. 对于 Wasm 这种新型的底层二进制代码格式，安全实证研究是 Wasm 安全研究的一个重要方向，通过对 Wasm 安全机理的理解和把握，分析漏洞形成的根因和潜在攻击面，从而为研究有效的漏洞检测与利用、安全增强、程序验证等技术奠定基础.

本文将现有 Wasm 安全实证研究分为 3 类：对高级语言支持、对编译工具链以及对二进制表示的研究，并对相关工作从研究内容、数据集、分析方法和主要结论等方面进行对比和总结，结果如表 3 所示. 本节将讨论 Wasm 安全实证研究的研究进展、已有成果和未来研究方向.

Table 3 Summary and Analysis of Empirical Studies on Wasm Security

表 3 Wasm 安全实证研究工作总结分析

研究内容	研究工作	研究数据集	分析方法	开源	主要结论
高级语言支持	Stiévenart 等人[43]	4 469 个具有缓冲区溢出漏洞的 C 程序	定量+定性	√	相同 C 程序对应的 x86 程序和 Wasm 程序的运行结果可能不同，其主要原因在于 x86 和 Wasm 在标准库的实现、所提供的安全保护机制和运行环境上存在差异.
	Stiévenart 等人[44]	17 802 个存在漏洞的 C 程序	定量+定性	√	
编译工具链	Romano 等人[22]	Emscripten 的 146 个 Wasm 相关漏洞报告	定量+定性	√	编译工具链中的漏洞主要来自高级语言与 Wasm 的同步机制差异、数据类型不兼容、内存模型差异等方面.
二进制表示	Hilbig 等人[85]	来自包管理器和实时网站的 8 461 个 Wasm 二进制文件	定量+定性	√	Wasm 二进制表示层面的安全威胁主要来自不安全的高级语言特性、非托管栈的错误使用、不安全的外部接口等.

3.1 对高级语言支持的实证研究

目前，Wasm 已经成熟稳定地支持 C/C++, Rust, Go 等 10 余种高级语言. 通过实证研究可以了解高级语言程序中的漏洞对 Wasm 程序的影响，以及从高级语言编译到 Wasm 的过程中存在的安全威胁. 因此，实证研究的结果对于 Wasm 安全特性的不断完善和基础设施的不断演进具有重要指导意义.

为了研究高级语言程序中的漏洞对 Wasm 程序的影响，Stiévenart 等人[43]将 4 469 个存在缓冲区溢出漏洞的 C 程序，分别编译为 x86 程序和 Wasm 程序，并对它们的运行结果进行对比. 实验结果表明：在 4 469 个 C 程序中，有 1 088 个程序对应的 x86 程序和 Wasm 程序的运行结果存在差异，即存在缓冲区溢出漏洞的 x86 程序在运行时会崩溃，而对应的 Wasm 程序仍

会正常执行. 该研究还指出, 导致运行结果差异的根本原因在于 Wasm 缺少栈金丝雀[80]等安全保护机制.

为了进行更全面和深入的分析, Stiévenart 等人[44]又对上述研究进行了扩展, 采用了更大的数据集, 覆盖了更多漏洞类型, 并对运行结果差异的根因进行了深入分析. Stiévenart 等人[44]从 Juliet 测试套件[86]中选取了 17 802 个 C 程序, 并比较其对应的 x86 程序和 Wasm 程序的运行结果. 实验结果表明: 测试集中有 4 911 个 C 程序的运行结果不同, 导致运行结果差异的原因可归纳为 3 类: 1)x86 程序使用的标准库与 Wasm 通过 WASI[16]使用的标准库之间存在差异; 2)Wasm 缺少 x86 中的栈金丝雀等安全保护机制; 3)x86 和 Wasm 的运行环境差异. 但是, 该研究没有将不同漏洞类型对运行结果的影响进行区分, 也没有从漏洞产生机理层面分析高级语言程序中的漏洞对 Wasm 程序的影响.

3.2 对编译工具链的实证研究

Wasm 编译工具链需要处理不同高级语言与 Wasm 在类型系统、内存模型等方面的巨大差异[10,13]. Wasm 编译工具链的错误不仅可能改变源程序的语义, 还可能引入安全漏洞, 对 Wasm 生态系统造成安全威胁. 因此, 对 Wasm 编译工具链的实证研究有助于深入理解编译工具链的错误, 从而帮助编译器开发人员确定开发和测试工作的重点, 促进 Wasm 编译工具链的完善和迭代.

为了分析 Wasm 编译工具链中漏洞的根因和影响, Romano 等人[22]首先对编译工具链 Emscripten 的 146 个 Wasm 相关漏洞报告进行了深入的定性分析, 将导致这些漏洞的根因总结为 9 类: 同步机制差异、数据类型不兼容、内存模型差异、其他基础设施错误、模拟本地环境错误、Web API 支持错误、跨语言优化错误、虚拟机实现差异和不支持的原语. Romano 等人[22]还对 AssemblyScript, Emscripten, Rustc/Wasm-Bindgen 三个编译工具链中已报告的 1 054 个漏洞进行了定量分析研究. 研究结果表明: 部分错误报告没有包含关键信息, 这导致调试和修复的时间开销增大; 43% 的漏洞会导致编译后的 Wasm 程序出现运行时错误, 这些错误难以定位和修复.

本文认为, 上述研究的主要不足在于深入定性分析所采用的研究目标和数据集规模较小, 容易产生过拟合问题. 目前支持不同高级语言的 Wasm 编译工具链已有 10 余种[41], 而上述研究[22]只定性分析了面向 C/C++的编译工具链 Emscripten 中的 146 个漏洞报告. 并且, 在分析得到的漏洞根因中, 同步机制差异、内存模型差异、不支持的原语等根因与 C/C++的具体特性强相关, 所以该研究提出的漏洞分类不适用于其他编译工具链和其他高级语言.

3.3 对二进制表示的实证研究

Wasm 二进制表示的主要设计目标是在保证运行效率的前提下实现安全性, 因此, 对 Wasm 二进制表示的安全实证研究对于 Wasm 二进制安全特性的设计与演进具有重要指导意义.

为了探索 Wasm 二进制表示层面的潜在安全威胁, Hilbig 等人[85]从高级语言、安全属性等方面, 对来自包管理器和实时网站的 8 461 个 Wasm 二进制文件进行了实证研究. 该研究主要有 5 点重要发现: 1)64.2% 的 Wasm 二进制文件是从 C/C++这类内存不安全的高级语言编译而来的, 这类语言中的安全漏洞容易对 Wasm 二进制表示安全性构成威胁; 2)约 80.0% 的二进制文件是通过 LLVM 工具链[87]编译而来, 因此, 如果 LLVM 工具链对 Wasm 提供栈金丝雀这类安全保护机制, 将有效提升 Wasm 二进制表示乃至整个 Wasm 生态系统的安全性; 3)65.0% 的二进制文件使用了非托管栈[24], 攻击者可能会通过非托管栈对 Wasm 程序进行栈溢出等攻击; 4)38.6% 的二进制文件使用了自定义的内存分配器, 引入了针对内存分配器的潜在攻击面; 5)21.2% 的二进制文件从主机环境导入了有潜在风险的接口, 这可能会导致主机环境注入攻击.

本文认为, 上述研究的局限性主要体现在 2 个方面: 一是所使用数据集的覆盖面窄, 仅来自包管理器和实时网站等 Web 领域; 二是没有对潜在的安全威胁提出可行的解决方案. 鉴于 Wasm 已经在物联网、边缘计算、区块链等非 Web 领域得到了广泛应用, 对 Wasm 二进制表示在这些领域面临的特有安全威胁进行深入的实证研究, 将有助于推动 Wasm 在非 Web 领域的应用和发展.

3.4 本节小结

Wasm 安全实证研究可以揭示 Wasm 威胁模型中各层面临的安全问题及其产生机理, 其结果对于研究有效的漏洞检测技术、安全增强技术和程序验证技术具有重要指导意义. 目前 Wasm 安全实证研究主要针对高级语言支持、编译工具链和二进制表示, 并且初步获得了有价值的发现. 但是, 相关研究都存在数据集覆盖面窄、结论不具通用性等问题. 本文认为, 随着 Wasm 的应用场景越来越广泛, 针对 Wasm 的安全实证研究应该覆盖包括语言虚拟机在内的完整生态系统, 同时还应该对 Wasm 语言特性与安全威

胁的内在联系进行深入研究，提出对语言特性的最佳安全实践.

4 Wasm 漏洞检测与利用研究

漏洞检测与利用是软件安全的重要研究领域，也是 Wasm 安全研究的重要方向. 目前已有大量针对 Wasm 安全漏洞的检测与利用研究，本节将对这一领域的研究成果和进展进行全面梳理和总结，深入讨论相关研究的局限性和未来研究方向.

4.1 漏洞检测

漏洞检测针对需要检测的程序和安全性质，采用程序分析、模糊测试等技术，来检测程序可能的状态和行为是否满足安全规范. 漏洞检测是提高软件安全性的重要手段，也是 Wasm 安全领域的重要研究方向.

Wasm 作为一种底层二进制代码格式，与高级语言在语法、内存模型等方面有着显著不同，这使得传统的针对高级语言的漏洞检测技术和工具无法直接用于 Wasm，这给 Wasm 漏洞检测研究提出了新的挑战. 本文对 Wasm 漏洞检测已有研究进行了全面、深入地分析，并按照检测技术将现有研究分为静态检测、动态检测和混合检测三大类，并对其中的代表性工作从检测技术、程序表示、漏洞类型、准确率、是否开源等方面进行了总结和对比，结果如表 4 所示.

Table 4 Summary and Analysis of Research Work on Wasm Vulnerability Detection

表 4 Wasm 漏洞检测研究工作总结分析

检测方法	主要技术	研究工作	具体技术	程序表示	漏洞类型	准确率/%	开源
静态检测	程序分析	Wassail[88]	信息流分析+污点分析	控制流图		64	√
		Wasmati[89-90]	数据流分析	代码属性图	内存安全漏洞	92.6	√
		VeriWasm[91]	抽象解释	控制流图	内存安全漏洞	100	√
		MinerRay[92]	控制流分析+语义分析	控制流图	加密劫持	99.3	√
		WANA[93]	符号执行+执行信息分析	二进制程序	智能合约安全	100	√
	深度学习	MINOS[94]	深度学习	二进制程序	加密劫持	99.0	×
动态检测	模糊测试	WAFL[95]	模糊测试+虚拟机快照	二进制程序			√
		Fuzzm[96]	灰盒模糊测试+金丝雀插桩	二进制程序	内存安全漏洞		√
		WASAI[97]	混合模糊测试+符号执行	二进制程序	虚假转账	96.6	√
		WASMAFL[98]	灰盒模糊测试+分层变异算法	虚拟机源程序			×
		WasmFuzzer[99]	模糊测试+字节码变异算法	虚拟机源程序			×
	运行时特征分析	MineThrottle[100]	重复执行指令序列分析	二进制程序	加密劫持	69.9	×
		CoinSpy[101]	CPU、内存分析+深度学习	二进制程序	加密劫持	100	×
		MineSweeper[102]	缓存行为分析+加密原语分析	二进制程序	加密劫持	100	√
	污点分析	Szanto 等人[103]	运行时污点追踪	二进制程序		100	×
		TaintAssembly[104]	运行时污点追踪	二进制程序	跨语言安全		√
混合检测		Wasabi[105]	函数调用插桩+运行时分析	二进制程序			√
		EVulHunter[106]	控制流分析+运行时检查	控制流图	虚假转账	86.0	√

4.1.1 静态检测

静态检测是指在不运行 Wasm 程序的前提下，对程序进行抽象和建模，再通过分析程序的属性来完成漏洞检测. 本文将 Wasm 静态检测研究总结为 2 类：基于程序分析的静态检测和基于深度学习的静态检测，并对相关工作进行归纳总结和对比.

1）基于程序分析的静态检测

程序分析技术通过扫描程序或中间表示、对程序的运行流程进行抽象、构建程序状态的模型、推导程序可能的行为来获取程序的特征与属性，以检查程序的安全性或正确性. 现有基于程序分析的 Wasm 静态检测研究主要使用了控制流分析、数据流分析、信息流分析以及符号执行等技术.

Stiévenart 等人[88] 提出了一个针对 Wasm 程序的信息流分析算法，该算法首先基于控制流图和调用图为 Wasm 程序的每个函数生成摘要，摘要描述了函数的参数、返回值和全局变量间的信息流信息，然后通过不动点算法获得 Wasm 全程序的信息流近似. 该

研究实现了自动化的信息流分析原型系统 Wassail. 实验结果表明: Wassail 在 56.13 s 内完成了对 196 157 行程序的分析, 平均精度为 64%. 本文认为, 虽然该算法可以进行过程间分析, 但是其本质还是一个粗粒度的信息流分析算法, 因为函数摘要信息中没有包含所有函数执行期间的信息流, 将导致一定程度的精度损失.

代码属性图[107]是一种同时包含了抽象语法树、控制流图和程序依赖图的数据结构. Lopes 等人[89-90]基于代码属性图, 提出了针对 Wasm 的漏洞检测框架 Wasmati, 其架构如图 7 所示. 首先生成 Wasm 二进制程序的代码属性图, 然后通过查询规范语言遍历该代码属性图来检测程序中的安全漏洞. 该研究还对 Wasm 的代码属性图进行了形式化定义, 实现了 4 种查询规范语言来完成对 10 种常见内存安全漏洞的查询. 实验结果表明: 在已知漏洞集上, Wasmati 的准确率达到 92.6%; 同时, Wasmati 还在真实的 Wasm 二进制文件中发现了潜在的漏洞. 但是, Wasmati 只能分析单个 Wasm 模块, 而无法检测出由多模块交互和多语言交互产生的漏洞. 此外, Wasmati 只能检测释放后使用、缓冲区溢出等具有通用错误模式的漏洞, 难以检测特定场景下的漏洞.

为了检测 Wasm 程序中的内存安全错误, Johnson 等人[91]提出了一个基于抽象解释的漏洞检测框架 VeriWasm. VeriWasm 通过对由 Wasm 编译得到的 x86-64 程序进行控制流分析来检测 Wasm 程序是否满足控制流安全、线性内存隔离、栈帧完整性和栈隔离等内存安全性质. 该研究还利用 Coq 定理证明器证明了 VeriWasm 的可靠性. 在 119 个 Wasm 二进制模块上的实验结果表明: VeriWasm 可以有效检测出内存安全错误, 每个模块的平均检测时间为 8.6 s, 没有假阳性. 本文认为, VeriWasm 仍然存在 3 点不足: 首先, 它的检测结果依赖未经验证的反汇编器; 其次, 它采用的控制流分析算法难以扩展到大型的复杂函数, 因此 VeriWasm 无法用于具有低延迟需求的实时应用; 最后, 分析算法特定于 Lucet 编译器[108], 无法扩展到 Wasm 生态系统.

文献 [88–91] 都是针对 Wasm 二进制程序的通用程序分析框架. 针对 Wasm 在浏览器、区块链等特定应用场景中所表现出的独特错误模式, Romano 等人[92]提出了一个跨 Wasm 和 JavaScript 的静态程序分析算法, 用于检测浏览器中的加密劫持攻击. 该算法首先将同时包含 Wasm 和 JavaScript 的程序统一转换为 Wasm 程序分析, 判断程序是否具有与加密劫持攻击相匹配的语义. 该研究实现了检测工具 MinerRay, 对 120 万个网站的实验结果表明, 该工具成功检测出 901 个遭受加密劫持攻击的网站. 本文认为, MinerRay 主要存在 2 点不足: 一是工具的鲁棒性不够高, MinerRay 对于编码、混淆[109]或加密后的程序无效; 二是工具可扩展性较低, 算法使用的自定义中间表示只支持 Wasm 和 JavaScript, 如果要支持新的语言, 需要改变或重新设计中间表示.

智能合约漏洞检测通常针对特定平台, 但是, 主流智能合约平台 EOSIO 和以太坊对 Wasm 的支持为跨平台漏洞检测提供了可能性. WANA[93]是一个基于符号执行技术的跨平台智能合约漏洞检测框架, 该框架首先将 EOSIO 和以太坊智能合约编译为 Wasm 程序, 然后基于对 Wasm 程序的符号执行完成漏洞检测. WANA 还通过设置循环上限来缓解路径爆炸问题, 从而提高执行效率. 实验结果表明, WANA 在 3 964 个合约中成功检测出了 411 个漏洞, 平均检测时间为 0.21 s. 本文认为, WANA 的局限性主要体现在 2 个方面: 首先, WANA 适用范围有限, 它只支持 Wasm 核心规范 1.0 的一个子集, 不支持目前最新的规范

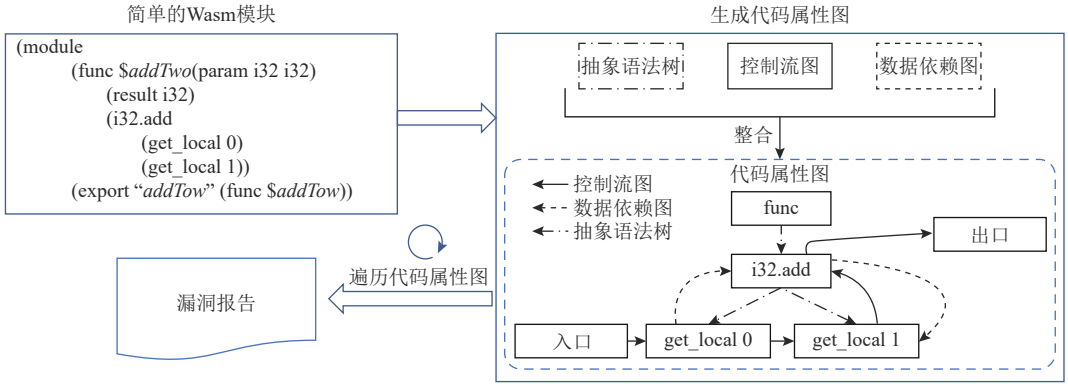

Fig. 7 Architecture diagram of Wasmati

图 7 Wasmati 架构图

2.0; 其次, WANA 分析范围有限, 它只能分析单个模块, 如果模块中存在对其他 API 函数或模块外函数的调用, WANA 只会根据返回值类型生成一个随机值, 进而导致误报.

2)基于深度学习的静态检测

近年来, 深度学习技术的快速发展和广泛应用为 Wasm 静态检测研究注入了新的活力. 基于深度学习的静态检测借助深度学习提供的数据挖掘能力, 挖掘 Wasm 二进制代码蕴含的各种信息, 包括控制流信息、数据流信息、依赖信息等, 并使用深度学习模型进行漏洞检测.

MINOS[94] 是一个基于深度学习的轻量级加密劫持检测系统. MINOS 首先将 Wasm 二进制程序转换为灰度图像表示, 然后利用训练好的基于卷积神经网络[110] 的 Wasm 程序分类器对灰度图像表示进行分类, 从而识别出存在加密劫持漏洞的 Wasm 二进制程序. 实验结果表明: MINOS 从 675 个网站中成功检测出 67 个存在加密劫持的恶意网站, 准确率达到 98.97%, 平均检测时间为 25.9 ms. 但是, MINOS 只能检测针对浏览器端的加密劫持攻击, 对于其他常见 Wasm 安全漏洞的深度学习检测模型的研究, 是一个亟待探索的研究方向.

Wasm 静态检测已有研究主要基于程序分析技术和深度学习技术, 并且大多针对 Wasm 在区块链领域特有的虚假转账、加密劫持等安全威胁, 以及由高级语言引入的常见内存安全漏洞. 而实际上, Wasm 面临的安全威胁遍布整个 Wasm 生态系统. 本文在对已有静态检测研究进行深入分析后认为, 未来有 3 个研究方向值得进一步探索: 1)对更多 Wasm 安全漏洞类型的检测; 2)对利用了 WASI 标准的程序, 包括跨 Wasm 和高级语言的混合程序检测技术的研究; 3)将程序分析和深度学习相结合的研究, 即利用程序分析技术挖掘 Wasm 程序的显式特征, 使用深度学习挖掘程序的隐式特征, 将这 2 种特征相结合, 形成互补, 为程序的漏洞检测提供更加强有力的支撑.

4.1.2 动态检测

动态检测是通过为目标程序提供特定输入, 观察程序运行时的行为通过分析程序的执行结果、异常行为和崩溃等信息, 从而判断程序中是否存在漏洞. 对 Wasm 安全漏洞的动态检测研究主要使用了 3 类技术: 模糊测试、运行时特征分析和污点分析.

1)模糊测试

模糊测试是一种重要的动态漏洞检测技术, 其核心思想是通过为程序提供大量测试用例, 在程序执行过程中监控异常行为, 从而发现程序漏洞. 模糊测试技术已经被用于 Wasm 漏洞检测研究. 本文将基于模糊测试的 Wasm 漏洞检测研究总结为 2 类: 针对 Wasm 二进制程序的模糊测试和针对 Wasm 虚拟机的模糊测试, 并按此分类对现有工作进行梳理和对比.

①对 Wasm 二进制程序的模糊测试. WAFL[95] 是第 1 个针对 Wasm 二进制程序的模糊测试研究. WAFL 利用模糊测试工具 AFL++[111] 为目标程序生成输入, 然后通过修改 Wasm 虚拟机来记录路径覆盖信息, 最后通过虚拟机快照来保存内存状态, 从而快速恢复虚拟机状态, 提高执行效率. 实验结果表明, WAFL 的执行效率超过了 x86-64 原生二进制程序, 执行 1 次程序的平均耗时为 55 μs. 本文认为, 尽管 WAFL 通过虚拟机快照提升了执行效率, 但是该研究仍然存在 2 点不足: 一是 WAFL 的实现复杂, 需要修改 Wasm 虚拟机, 这限制了 WAFL 在不同虚拟机上的普适性; 二是该研究没有在真实的 Wasm 应用程序上进行大规模实验, 因此没有表明 WAFL 在真实的 Wasm 二进制程序上进行漏洞检测的有效性.

Fuzzm[96] 也是一个基于模糊测试的 Wasm 漏洞检测框架, 不同于 WAFL, Fuzzm 主要检测 Wasm 内存安全漏洞. 由于 Wasm 不提供栈金丝雀等对缓冲区溢出的检查机制, Fuzzm 首先向 Wasm 二进制程序中插入栈金丝雀, 然后基于模糊测试工具 AFL++ [111] 为目标二进制程序生成测试用例, 再使用 Wasm 虚拟机执行插桩后的代码. 在真实的 Wasm 二进制文件上的实验结果表明, Fuzzm 在 24 h 内覆盖了 1 232 条路径, 触发了 40 个不同的程序崩溃, 其中 50.0% 的崩溃由插入的栈金丝雀导致; 插桩后程序的平均执行时间为原来的 1.05~1.06 倍.

文献 [95–96] 都是针对 Wasm 二进制程序的通用漏洞检测框架. Chen 等人[97] 提出了一个针对 Wasm 智能合约的模糊测试框架 Wasai. Wasai 首先在 Wasm 二进制程序上进行插桩来记录程序的执行路径, 然后使用 EOSVM 模拟器对程序进行符号执行, 并通过构建函数模型和跳过冗余路径来缓解符号执行的路径爆炸问题. Wasai 在已部署的 991 个 Wasm 智能合约上进行了实验, 结果表明: 超过 70% 的智能合约存在安全风险. 但是, Wasai 为了获得了更大的吞吐量, 限制了约束求解器的资源, 进而影响了漏洞检测的准确率.

本文认为, 对 Wasm 二进制程序的模糊测试研究都采用了类似的技术路线, 即先基于现有模糊测试

工具生成大量测试用例，再使用 Wasm 虚拟机来执行目标程序并进行漏洞检测. 但是，已有的相关研究都存在计算资源需求大、检测效率低等不足. 未来有 2 个可能的研究方向：一是通过针对具体漏洞设计插桩算法，来进一步提高检测效率；二是通过自动生成漏洞利用，来降低为定位漏洞而进行的手动分析程序异常的成本.

②对 Wasm 虚拟机的模糊测试. 林敏等人[98]提出了对 EOS 的 Wasm 虚拟机进行模糊测试的方案 WASMAFL. 如图 8 所示，该方案提出了分层变异算法，用于生成合法测试用例，算法首先通过对 Wasm 文件的分解、对模块间的组合和修改来进行高级结构层面的变异，然后通过改变初始化数据、打乱指令序列来进行代码段层面的变异，使得测试用例能够覆盖更多的路径. WASMAFL 是基于 AFL 实现的 Wasm 模糊测试工具，实验结果表明：WASMAFL 在 24 h 内覆盖了 3 153 条程序执行路径，触发了 258 个程序崩溃，成功发现了 2 种段类型错误.

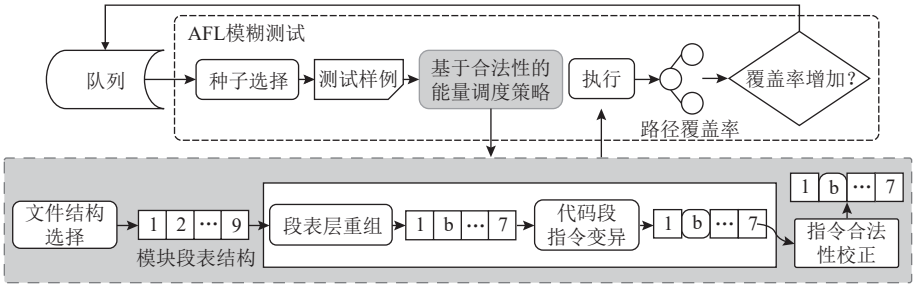

Fig. 8 Architecture diagram of WASMAFL

图 8 WASMAFL 架构图

WasmFuzzer[99]也是一个针对 Wasm 虚拟机的模糊测试框架. WasmFuzzer 提供了 1 组变异算子，用于在不同粒度上系统地变异 Wasm 模块，还设计了一种自适应的变异策略，可以为不同的 Wasm 虚拟机寻找最佳变异算子. 实验结果表明，WasmFuzzer 在 3 个 Wasm 虚拟机上共计触发了 235 个崩溃，在路径覆盖率和触发程序崩溃方面都优于 AFL.

现有针对 Wasm 虚拟机的模糊测试研究的重点都集中在测试用例生成阶段. 有效的测试用例不仅需要具有良好的脆弱性导向，还需要具有高路径覆盖率，才能有效提高漏洞检测的针对性和效率. 本文认为，未来一个重要的研究方向是将模糊测试与深度学习技术紧密结合，先使用 Wasm 程序训练深度学习模型，再利用模型来指导高质量的测试用例生成，缓解模糊测试中常见的路径爆炸和盲目性问题.

2)运行时特征分析

在区块链领域，加密劫持是指攻击者在未经所有者授权的情况下，利用其设备的计算能力来挖掘加密货币. 由于 Wasm 的高执行效率以及浏览器缺少对恶意 Wasm 代码的防护能力，近年来，Wasm 已经被广泛用于加密劫持攻击，严重影响 Wasm 生态安全[112]. 运行时特征分析是一种通过分析 Wasm 程序的运行时行为，提取与安全漏洞相关的运行时特征，从而进行漏洞检测的技术. 目前已经有许多基于运行时特征分析的 Wasm 加密劫持检测研究.

Bian 等人[101]提出通过对用于加密劫持的 Wasm 程序的执行序列进行分析，确定可能存在加密劫持攻击的指令子序列模式，再通过模式匹配进行检测的技术. 并开发了一个基于该技术的运行时特征分析工具 MineThrottle，对正在进行加密货币挖掘的 Wasm 程序进行实时安全检测. MineThrottle 在 Alexa 排名前 100 万的网站中成功检测出 109 个存在加密劫持的网站. 在这个方向的工作中，CoinSpy[101]是一个结合 CPU、内存、网络行为分析和深度学习的 Wasm 加密劫持检测框架；而 MineSweeper[102]则通过分辨 Wasm 程序的加密原语和运行时的缓存行为来检测加密劫持.

3)污点分析

污点分析技术通过对程序的敏感数据进行标记，并且跟踪被标记数据在程序执行过程中的传播，从而实现精确的数据流分析，来发现软件系统中存在的安全问题. 污点分析技术已经被用于 Wasm 漏洞检测.

Szanto 等人[103]设计并实现了一个 Wasm 污点追踪系统，该系统通过为内存中的每个变量设置一个污点标记，来追踪敏感数据在程序运行时的流动. 此外，该系统还引入了间接污点来表示局部变量之间敏感信息的隐式流动. 实验结果表明：该污点追踪系统引入的额外时间和内存开销是线性有界的. 但是，该系统只支持 Wasm 的一个子集，不支持浮点数计算、

导入等常见 Wasm 操作，因此，该系统在实际生产环境中的可用性有待进一步研究.

文献 [103] 只能对 Wasm 程序进行污点追踪，而在实际应用中 Wasm 程序常常需要与 JavaScript 程序交互. 虽然 Wasm 的沙箱运行环境提供了安全保证，但是 Wasm 与 JavaScript 的交互仍然会引入安全威胁[24]. TaintAssembly[104] 是一个检测跨 Wasm 和 JavaScript 的不安全行为的污点追踪系统. TaintAssembly 通过为 Wasm 程序中的每个值增加一个污点标记来追踪 Wasm 和 JavaScript 之间的数据流. 实验结果表明，TaintAssembly 增加了 5%~12% 的运行时开销. 本文认为，TaintAssembly 仍存在 3 方面的局限性：首先，它的污点传播语义不完备，不包括比较操作符等常见操作；其次，污点追踪过程依赖于随机数生成，这使其不适用于计算密集型场景；最后，它没有为 Wasm 的线性内存实现有效的污点追踪.

4.1.3 混合检测

混合检测是一种将静态检测和动态检测相结合的漏洞检测技术，也被用于 Wasm 漏洞检测.

Wasabi[105] 是第 1 个基于混合检测的通用 Wasm 漏洞检测框架. 如图 9 所示，Wasabi 首先进行静态插桩，将特定分析函数插桩到 Wasm 二进制程序；然后在程序运行时进行动态分析. Wasabi 实现了 8 种分析，包括基本块分析、内存访问跟踪、调用图分析和污点分析等. 在计算密集型程序测试集和真实的 Web 应用程序上的实验结果表明：Wasabi 引起的代码膨胀率为 1%~742%，运行时开销增加了 1.02~163 倍. 本文认为，Wasabi 主要有 2 点不足：一是最大运行时开销较大；二是不能进行 Wasm 和 JavaScript 的跨语言分析.

EVulHunter[106] 是一个基于程序分析和运行时检查的 EOSIO 智能合约虚假转账漏洞检测框架. 它首先解析 EOSIO 合约的 Wasm 程序生成控制流图，然后基于控制流图和预定义的漏洞模式进行程序分析来检测合约中的虚假转账漏洞. 为了进行更精确的分析，它还需要与 Wasm 虚拟机交互来获取运行时信息. 实验结果表明：EVulHunter 的检测准确率达到 86%，且检测时间随合约规模的增大线性增长. 但该研究仍然具有 3 方面的局限性：首先，EVulHunter 不具普适性，目前只能针对某一类漏洞，而无法适用于其他类型的漏洞；其次，系统没有在大量的真实合约上进行实验，无法证明 EVulHunter 在实际的生产环境中的可用性；最后，系统的运行时检查带来了可观的额外时间开销.

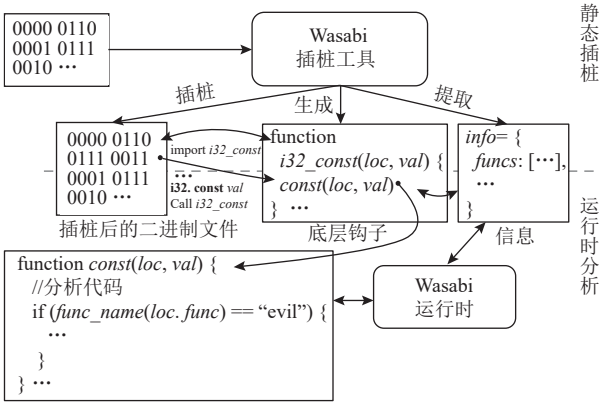

Fig. 9 Architecture diagram of Wasabi

图 9 Wasabi 架构图

4.2 漏洞利用

漏洞利用是指通过分析程序的漏洞信息，绕过系统的安全防护机制，生成漏洞利用代码来获取目标计算机系统的控制权限，最终达到实现软件的非预期功能的目的.

尽管 Wasm 引入了沙箱隔离等特性来保证安全性，但是由于 Wasm 缺少栈金丝雀、内存权限防护等漏洞缓解机制，Wasm 程序中仍然可能会出现缓冲区溢出、栈溢出等漏洞，攻击者可以利用这些漏洞来实现攻击. 本节将对现有 Wasm 漏洞利用研究进行梳理和总结.

McFadden 等人[23] 对主流 Wasm 编译工具链 Emscripten 的漏洞缓解措施进行了研究，研究结果表明，Emscripten 为 Wasm 程序提供了控制流完整性检查、堆保护等安全机制来缓解常见的控制流劫持等攻击. 该研究进一步总结了 Wasm 中仍然可能存在的 3 类安全漏洞，包括整数溢出、格式化字符串攻击和基于栈的缓冲区溢出. 该研究还利用 Wasm 的间接函数调用、缓冲区溢出漏洞实现了服务端的远程代码执行攻击和浏览器端的跨站脚本攻击.

Lehmann 等人[24] 对 Wasm 二进制安全进行了深入研究，研究结果表明，由于 Wasm 的线性内存缺少页保护标记、栈金丝雀等安全防护机制，所以在 Wasm 二进制表示层面可能存在缓冲区溢出、栈溢出和堆元数据损坏等漏洞. 在此基础上，该研究提出了 3 种攻击原语，包括任意内存写入、重写安全相关数据以及通过转移控制流和操作主机环境，来触发恶意行为. 进一步，该研究还利用这些攻击原语在不同主机平台上实现了 3 种端到端攻击：浏览器上的跨站脚本攻击、Node.js 上的远程代码执行攻击和虚拟机上的任意文件写入攻击. 该研究证明由于 Wasm 中缺少对常见漏洞的缓解措施，攻击者可以利用漏洞

实现对 Wasm 的真实攻击.

漏洞利用是安全研究的重要方向,通过理解漏洞被利用的根因,可以制定相应的安全防护方案,提高软件安全性. 现有研究表明,在 Wasm 的编译工具链和二进制表示层面都存在可以被利用的漏洞,并且已有相关研究利用这些漏洞在 Wasm 的不同应用领域实现了攻击. 本文认为,一个重要的未来研究方向是 Wasm 自动化漏洞利用研究. 通过自动生成漏洞利用代码和补丁,实现对 Wasm 软件系统的自动漏洞修复和积极防御,缩短漏洞的生命周期,增强软件系统的安全性.

4.3 本节小结

漏洞检测与利用是软件安全领域的重要研究方向,也是 Wasm 安全研究中最活跃的方向,研究工作已经取得积极进展. 但是现有漏洞检测研究大多针对区块链领域的加密劫持和虚假转账等漏洞,而对于第 2 节中 Wasm 安全威胁模型涉及的重要安全漏洞,目前缺乏全面有效的检测技术. 同时,目前对于 Wasm 漏洞利用的研究局限于人工进行漏洞分析与利用,没有自动化漏洞利用相关研究. 随着 Wasm 应用领域的不断拓展,以及软件复杂性、漏洞类型与数量的不断增加,漏洞检测与利用也面临更大的挑战,因此,Wasm 漏洞检测与利用是一个非常重要但仍亟待探索的重要研究方向.

5 Wasm 安全增强研究

安全增强是安全研究的重要组成部分,对于程序中的安全漏洞,安全增强研究利用相关实证研究和漏洞检测研究的结果,通过在软件或系统中引入特定安全机制,增强软件系统的防御功能,防止或阻断漏洞的发生.

按照安全增强技术的应用时机,本文将已有的 Wasm 安全增强研究分为静态安全增强和动态安全增强两大类. 表 5 分别列出了其中的重要研究工作,对每项研究工作,本文从关键技术、运行时开销、能够应对的安全威胁、是否开源可用 4 个维度进行了全面对比.

Table 5 Summary and Analysis of Research Work on Wasm Security Enhancement

表 5 Wasm 安全增强研究工作总结分析

类别	研究工作	关键技术	运行时开销/%	应对的安全威胁	开源	研究进展总结
静态安全增强	CROW[113]	代码多样化		代码破解	√	Wasm 静态安全增强主要通过对 Wasm 核心特性和硬件辅助 2 方面设计安全策略来实现.
	Swivel[114]	控制流—致性+硬件防护	3.3~240.2	幽灵攻击	√	
	MS-Wasm[115]	段式内存 + ARM MTE	35~60	内存安全问题	×	
	Vassena[116]	内存标记		内存安全问题	×	
动态安全增强	Aerogel[117]	访问控制	18.8~45.9	访问控制缺失	√	Wasm 动态安全增强主要通过利用运行时和硬件的安全特性来实现.
	SELWasm[118]	自检、加解密、延迟加载	3.45	代码破解	×	
	TWINE[119]	Intel SGX	0.9~426.0	执行环境不可信	√	
	WATZ[120]	ARM TrustZone	0.02~5	执行环境不可信	√	
	VeriZero[121]	零成本软件故障隔离	22.5~25	上下文转换错误	√	

5.1 静态安全增强

目前 Wasm 静态安全增强研究主要用到了 3 种技术: 1)通过对 Wasm 二进制程序直接进行改写和变形,以达到代码多样化的目的; 2)通过对 Wasm 编译工具链的改写,在编译生成的代码中插桩保护代码,保证编译得到的 Wasm 程序的控制流安全以及实现软件错误隔离; 3)通过使用硬件防护技术,保证 Wasm 程序的控制流安全及内存安全.

在 Web 应用场景中,Wasm 程序以二进制形式分发到用户的浏览器中,因此,若攻击者发现了 Wasm 程序中的漏洞,则可以向下载了该二进制程序的所有用户发起攻击. 为解决这一攻击和防御间的不对称性挑战,Arteaga 等人[113]提出了 CROW 系统,用代码多样化技术[122]来对代码进行静态变形,以实现对基于同一源代码分发的 Wasm 二进制代码的独特性目标, CROW 的工作流如图 10 所示. CROW 基于 LLVM 框架[87]实现,依靠 Souper[123]对 LLVM 字节码进行变形,并使用 Z3 约束求解器[124]保证变形的语义等价性. 在 303 个 C 程序以及在 libsodium[125]上的实验结果表明: CROW 能够有效对 239 个(79%)程序生成变体. 本文认为,该研究主要有 3 个方面的不足: 首先, CROW 只能用于源代码而无法直接用于 Wasm 二进制代码; 其次, CROW 只对基于 LLVM 编译工具链的 C/C++程序有效,而无法用于其他语言和

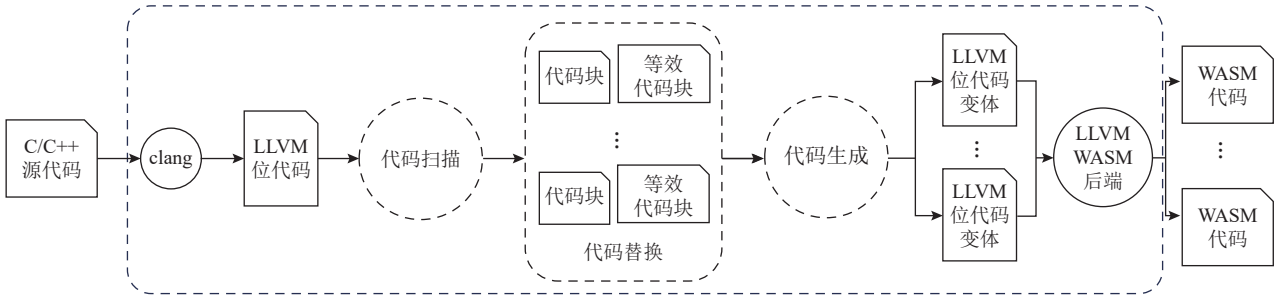

Fig. 10 Workflow of CROW

图 10 CROW 的工作流

编译工具链；最后，CROW 无法在运行时对程序进行增强，因此难以用于诸如 Chrome V8 TurboFan[126] 等主流的即时编译器中.

为了抵御针对 Wasm 的幽灵攻击[83]，Narayan 等人[116] 提出了一个基于编译器技术的静态安全增强框架 Swivel. Swivel 引入了 2 种防护技术：Swivel-SFI 和 Swivel-CET. Swivel-SFI 使用了控制流一致性（control flow integrity, CFI）检查和软件错误隔离技术，即检查每条直接或间接跳转指令时，都只能跳转到基本块的开头. Swivel-CET 使用 Intel 第 11 代 CPU 开始引入的 2 个最新的硬件防护机制：控制流强制技术（control enforcement technology, CET）和内存保护秘钥（memory protection key, MPK），CET 用基于硬件的影子栈保证返回地址不被改写；MPK 将内存划分为附加了不同保护秘钥的分区，这些分区具有不同的读写访问权限. 文献 [116] 修改了 Lucet 编译器[108] 中的 Wasm 到 x86 的代码生成器，并进行了实验. 实验结果表明：Swivel 的运行时开销介于 3.3%~240.2% 之间. 本文认为，该工作主要有 3 点不足：1）该工作需要重写 Wasm 到具体二进制代码的编译器实现，以便引入基于软件和硬件的防御，工作量相对较大；2）硬件防护机制导致了可观的运行时开销，难以适用于性能敏感的应用场景；3）硬件防御技术需要使用 Intel 最新的硬件平台（即 11 代之后的最新 CPU），因此无法实现对老硬件的向后兼容，也无法直接适用于非 Intel 架构的硬件.

由于 Wasm 缺乏数组边界检查等关键安全机制，导致 Wasm 程序易出现缓冲区等攻击面；并且，由于最新的基于硬件防护的机制，如 Intel 的内存保护秘钥和 ARM 的内存标记扩展（memory tagging extension, MTE）等，都需要使用数组的边界大小等信息，而 Wasm 中数组边界信息的缺失，使得相关研究很难利用这些最新的硬件安全防护机制. 为此，Disselkoen 等人[115] 提出了 MS-Wasm，即内存安全的 Wasm，其对标准 Wasm 的内存模型进行了扩展，引入了段内存的概念. 本质上，段内存标记了指针的基地址和界限等额外的元信息，从而使得 Wasm 虚拟机能够在运行时利用 ARM 的 MTE 等硬件机制，对指针的有效性进行安全检查. 和基于纯软件的防护技术[127-129] 相比，MS-Wasm 不但保证了内存安全性，而且具有更低的运行时开销. 但本文认为，MS-Wasm 的不足之处主要有 2 点：1）只在最新的具有特定硬件防护机制（如 ARM）的平台上有效，不具有普适性；2）对 Wasm 的标准进行了扩展和修改，失去了和 Wasm 标准的兼容性，并且会进一步导致 Wasm 生态系统的碎片化.

受 MS-Wasm 设计的启发，Vassena 等人[116] 提出一个以 MS-Wasm 为编译目标的可信编译器设计方案和思路. 但是，该研究并未给出具体的实现和实验结果，因此，无法评估该工作在实际中的有效性和运行开销.

本文认为，Wasm 静态安全增强研究在 2 个方面亟待深入探索：1）对 Wasm 二进制通用表示的研究和构建，即研究针对 Wasm 的一套面向二进制的中间表示基础设施，这将极大方便在二进制层直接对 Wasm 进行操作和分析，给 Wasm 的静态安全增强提供保障；已有的在 Java 字节码[130-131] 等方向的研究为这一方向的探索提供了可行的思路；2）对 Wasm 编译工具链的安全增强和扩展，即扩展和增强目前的 Wasm 编译工具链[41]，加入对控制流完整性、内存安全等特性的自动代码插桩的支持.

5.2 动态安全增强

目前 Wasm 动态安全增强的相关研究主要利用 2 种技术：1）通过在 Wasm 虚拟机中增加核心安全机制，如访问控制、代码加解密等，来增强 Wasm 虚拟机的安全防护能力；2）利用硬件的可信执行特性，如 Intel 的 SGX[132] 技术等，为 Wasm 提供可信执行环境[133].

尽管 Wasm 提供了进程内的沙箱隔离，但是没有提供细粒度的访问控制机制. 针对这一问题，Liu 等

人[117]对物联网外围设备的访问控制这一问题进行了研究, 提出了运行时访问控制框架 Aerogel. Aerogel 利用 Wasm 沙箱为物联网外围设备提供隔离, 同时, 基于领域专用语言和自定义配置文件的方式, 对这些外围设备的功耗、处理器时间、内存消耗等进行控制. 在 MCU 开发板(nRF52840)上, 基于 QEMU 模拟器[134]进行的实验表明: Aerogel 带来 0.19%~1.04%的额外执行时间开销以及 18.8%~45.9% 的额外能耗开销. 本文认为, Aerogel 在基于 Wasm 构建物联网设备的访问控制机制方面进行了有益的探索, 但其较高的能耗消耗使其难以应用到边缘计算等应用场景中的低电量设备上; 同时, Aerogel 缺少了针对侧信道攻击的安全防御, 无法防御针对内存的侧信道攻击[24].

为解决 Wasm 代码在 Web 上分发时的代码保护和对抗逆向问题, Sun 等人[118]提出了运行时防护框架 SELWasm, 其包含 3 部分的核心机制: 运行时执行环境检查、运行时代码加解密以及延迟加载. 本文认为, SELWasm 的主要不足有 2 点: 1)其加解密功能依赖于 JavaScript 实现, 和浏览器形成了紧耦合, 因此难以应用到独立的缺少 JavaScript 支持的 Wasm 虚拟机中; 2)Wasm 代码的加解密功能以及环境自检等功能的加入, 显著增大了 Wasm 代码的规模(文献 [118] 中报告了最多 4 倍的增长), 进而降低了执行效率.

为了支持分布式系统中的可信代码执行, Ménétrey 等人[120]提出了 TWINE, TWINE 利用 Intel SGX [132]机制提供的可信执行能力[133], 为 Wasm 虚拟机提供了可信执行环境. TWINE 依赖于 WAMR 虚拟机[60], 并通过 WASI 接口实现了与安全文件系统的交互, 其架构如图 11 所示. 实验结果表明: 基于 TWINE 实现的 SQLite[135], 与 SGX-LKL[136]中实现的 SQLite 平均性能开销接近. 本文认为, TWINE 仍有 3 点不足: 首先, 由于 Intel SGX 缺少了对侧信道攻击的防护[133], 因此 TWINE 上运行的 Wasm 程序仍可能遭受侧信道攻击[82]; 其次, 由于 Intel SGX 技术的加密特性带来的性能开销, TWINE 的运行速度相比于 WAMR 虚拟机有一定差距; 最后, TWINE 的内存消耗可观(在部分测例上超过了 80 MiB), 这使得其难以用于物联网或边缘计算的小微型设备上.

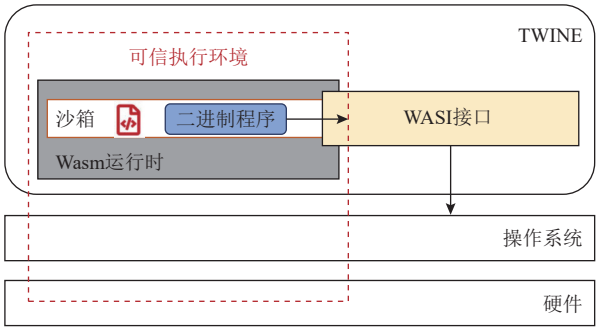

Fig. 11 Architecture diagram of TWINE

图 11 TWINE 架构图

为了实现程序的安全远程执行, Ménétrey 等人[120]基于 ARM TrustZone[137]可信执行环境, 设计并实现了轻量级的可信 Wasm 虚拟机 WATZ. WATZ 不仅利用 ARM TrustZone 提供了安全、高效的 Wasm 运行时环境, 还集成了远程证明系统[138], 可以在 Wasm 程序执行前进行验证和优化, 从而在远程执行中为共享密钥等机密数据提供安全保证. 实验结果表明: 在物联网设备上, WATZ 与 Wasm 虚拟机 WAMR[60]的性能开销接近. 本文认为, 虽然 WATZ 具有的高执行效率、低内存占用的特性, 但 ARM TrustZone 缺乏了对易失性内存数据的保护, 因此也可能会遭受来自硬件的攻击[83]或侧信道攻击[82].

Wasm 采用软件故障隔离技术来将 Wasm 程序运行于沙箱环境中, 但 Wasm 程序与主机环境交互时的上下文转换是易错的且开销较大. 为了实现安全的零成本软件故障隔离[68], Kolosick 等人[121]提出了一个安全的零成本上下文转换模型, 该模型通过良好的控制流来保证机器状态转换的机密性和完整性, 确保程序返回到真实的调用点, 同时对零成本转换条件进行了精确的形式化定义. 该研究通过修改 Wasm 编译器和虚拟机实现了一个零成本验证器 VeriZero, 可以有效检查 Wasm 程序是否满足模型的要求. 实验结果显示: VeriZero 将 Firefox 的图像解码和字体渲染速度分别提升了 29.7% 和 10%. 但是, 该研究仍然存在局限性: VeriZero 不支持用户定义的可变函数参数, 也不支持 JIT 编译.

本文认为, Wasm 动态安全增强研究还可以在 2 个方向加强探索: 1)除了已有研究使用的 SGX 和 TrustZone 等技术外, 还可以将其他提供可信执行环境的硬件特性, 如 AMD SME[139], RISC-V Keystone[140]等用于 Wasm 安全增强; 2)研究针对侧信道攻击的动态安全增强技术.

5.3 本节小结

安全增强是提升 Wasm 程序安全性的重要手段, 也是 Wasm 安全研究的重要目标. 已有研究利用代码多样化、代码插桩等技术, 实现对 Wasm 程序的静态安全增强; 通过向 Wasm 虚拟机增加安全防护机制以及利用硬件的可信执行环境等特性, 实现对 Wasm 程序的动态增强.

6 Wasm 形式语义与程序验证研究

Wasm 形式语义与程序验证研究是 Wasm 安全领域的热点方向. 形式语义与程序验证研究和 Wasm 语言安全研究的其他方向具有紧密联系: 对于 Wasm 中可能存在的安全漏洞和缺陷, 可以通过为其定义严格的形式语义来证明 Wasm 语言设计的可靠性; 而对于经过漏洞检测和安全增强的 Wasm 软件系统, 可以利用程序验证技术来进一步严格证明其安全性或功能正确性.

本节将对现有形式语义和程序验证相关工作从发表时间、验证技术、验证工具、被验证的性质、实验数据集和是否开源等方面进行分析对比, 结果如表 6 所示. 本节将对 2 类研究分别进行梳理和总结, 并讨论现有工作的不足以及未来可能的研究方向.

6.1 Wasm 形式语义研究

形式语义[149]是以数学为方法和工具, 形式化地定义和解释程序设计语言语义的学科. Wasm 形式语义研究可以通过对 Wasm 语义的形式化定义指导语言设计, 为 Wasm 程序验证奠定基础. 本节将按照时间顺序和相关研究之间的承接关系, 梳理并总结 Wasm 形式语义研究及其发展脉络.

Haas 等人[55]进行了 Wasm 形式语义的奠基性研究, 该研究讨论了 Wasm 的设计理念, 并形式化定义了 Wasm 的类型系统和操作语义. 该研究还从执行时间和二进制代码大小 2 个方面将 Wasm 代码和原生代码进行对比. 结果表明: Wasm 代码的执行时间与原生代码的差距在 10% 以内; Wasm 代码大小平均是 x86-64 代码大小的 85.3%. 但是, 该工作存在 2 点不足: 1)形式语义模型不完备, 例如, 模型中没有讨论 Wasm 与主机环境的交互; 2)没有对 Wasm 类型系统的可靠性进行严格的机械化证明.

为了解决上述研究的局限性, Watt 等人[141]提出了完整的 Wasm 形式化语义模型, 并证明了 Wasm 类型系统的可靠性. 研究结果表明, Wasm 最初官方规范里的类型系统并不可靠, 可能导致异常传播、函数返回错误以及主机函数与 Wasm 程序交互崩溃 3 类严重错误. 这些设计缺陷已经报告给 Wasm 工作组并被成功修复. 本文认为, 随着 Wasm 语言设计的不断演进, Wasm 规范中增加的零成本异常、并发机制、单指令多数据流指令等新特性, 对 Wasm 形式化模型的扩展和安全性证明提出了新的需求和挑战, 是一个重要的未来研究方向.

随后, Watt 等人[143]又提出了第 1 个针对 Wasm 的程序逻辑 Wasm Logic, 该研究基于 Wasm 的栈式抽象机器操作语义和强类型系统设计了一种新的断言语法, 并且通过扩展分离逻辑三元组和证明规则来证明 Wasm 程序的控制流安全. 本文认为, 该研究仍然存在 2 点局限性: 1)Wasm Logic 只能处理单个模块, 对于多个模块的处理需要修改程序逻辑; 2)Wasm Logic 只支持一阶 Wasm 程序推理, 因此无法处理跨语言交互等需要高阶推理的操作.

为了将 Wasm 用于安全加密算法实现, Watt 等人[144]提出了对 Wasm 的扩展系统 CT-Wasm. CT-Wasm 扩展了 Wasm 的类型系统和语义, 通过把密钥等安全数据与普通数据在类型级别上进行严格区分, 来抵御侧信道攻击, 保证信息流安全. 实验结果表明, CT-Wasm 可以有效防止信息泄露, 并且额外性能开销低于 1%. 本文认为, 该工作主要存在 2 点不足: 1)实际使用中, 需要先验证 CT-Wasm 程序的安全性, 再重写为 Wasm 来执行, 这导致 CT-Wasm 在实际使用中比较复杂; 2)Wasm 虚拟机的优化策略可能会破坏 CT-Wasm 的安全保证.

Table 6 Summary and Analysis of Research Work on Wasm Formal Semantics and Program Verification

表 6 Wasm 形式语义与程序验证研究工作总结分析

研究类别	研究工作	发表时间	验证技术	验证工具	被验证的性质	实验数据集	开源
形式语义	Haas 等人[55]	2017-06			执行时间	PolyBenchC	√
	Watt[141]	2018-01	演绎推理	Isabelle[142]	类型系统可靠性		√
	Wasm Logic[143]	2018-11	一阶逻辑推理	Isabelle	控制流安全	WebAssembly B 树库	×
	CT-Wasm[144]	2019-01	演绎推理	Isabelle	信息流安全	加密算法库	√
	Watt 等人[145]	2019-10	约束求解		SC-DRF		×
程序验证	Sjölén[146]	2020-09	关系符号执行	Z3	恒定时间安全	Salsa20	√
	Vivienne[147]	2021-09	关系符号执行	Z3，CVC4	恒定时间安全	加密算法库	√
	WASP[148]	2022-06	混合执行	Z3	功能正确性	B 树库、加密库	×

为了支持并发程序到的 Wasm 的正确编译, Watt 等人[145]对 Wasm 的形式化模型进行了并发扩展. 首先, 该研究定义了 Wasm 并发扩展的形式语义, 包括线程、原子操作、引用和可变表等; 其次, 该研究提出了一个弱内存模型, 并证明了内存模型对于无数据竞争程序的顺序一致性(sequentially consistent for data race free, SC-DRF)[150]. 本文认为, 该研究提出的 Wasm 并发模型是基于共享内存的, 对于其他并发模型的 Wasm 扩展将是一个重要的未来研究方向, 尤其是, 对于边缘计算领域的通信顺序进程(communicating sequential processes, CSP)[151]的 Wasm 并发扩展, 是一个亟待研究的问题.

现有 Wasm 形式语义研究致力于对 Wasm 的语义进行抽象和严格定义, 并进行安全性的严格证明, 从而为 Wasm 程序验证和分析奠定基础. 随着 Wasm 核心规范的不断扩展[15], 对 Wasm 新特性的形式语义研究将会成为一个重要的未来研究方向, 有助于进一步推动 Wasm 在各领域的广泛应用, 促进 Wasm 生态系统的繁荣发展.

6.2 Wasm 程序验证研究

程序验证[152]基于严格数学方法, 验证程序是否符合预期的设计属性和安全规范. 程序验证在形式语义和形式规约的基础上, 将程序的分析和验证问题转化为逻辑推理问题或形式模型的判定问题, 利用定理证明器[142,153]或者约束求解器[124]来进行验证. 本节将按照时间顺序梳理并总结程序验证相关工作.

Sjölén[146]开展了基于关系符号执行技术[154]的 Wasm 程序验证研究. 该研究实现了一个 Wasm 关系符号解释器, 并定义了解释器的形式语义. 实验结果表明: 解释器可以在 59 s 内完成对流加密算法 Salsa20[155]的恒定时间安全性验证. 本文认为, 该研究所实现的解释器难以成为通用的 Wasm 程序验证工具, 其局限性主要体现在 3 个方面: 1)解释器只能验证恒定时间安全性, 无法验证其他 Wasm 安全性质或功能正确性; 2)解释器能够处理的 Wasm 语法不完备, 需要对 Wasm 的类型、函数调用等进行简化或移除; 3)所采用的实验数据集规模较小且结构简单, 因此实验结果只能代表解释器在最理想情况下的表现.

随后, Tsoupidi 等人[147]也提出了一个基于关系符号执行的 Wasm 恒定时间安全验证框架 Vivienne, 其架构如图 12 所示. Vivienne 接收 Wasm 模块、安全策略和入口函数作为输入, 经过关系符号执行模块生成逻辑公式, 并调用 SMT 求解器对逻辑公式进行求解, 最后输出验证结果. 此外, Vivienne 会尝试对循环生成不变量. 在 57 个真实的加密算法库上的实验结果表明: Vivienne 能够在不重构代码的前提下, 有效验证 Wasm 程序是否满足恒定时间安全, 成功率达到 96% 且没有假阳性.

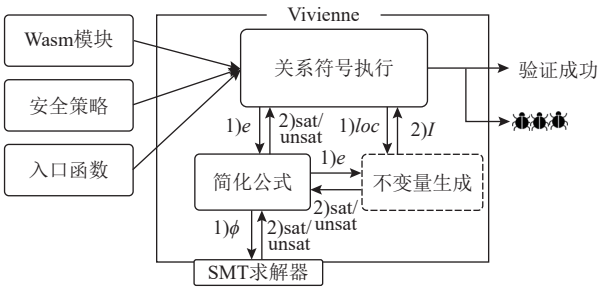

Fig. 12 Architecture diagram of Vivienne

图 12 Vivienne 架构图

文献 [146–147] 都只能验证 Wasm 程序的恒定时间安全. 为研究通用的 Wasm 符号执行框架, Marques 等人[148]将符号执行与具体执行相结合, 设计并实现了一个 Wasm 混合执行框架 WASP. 对于带有功能标注的 Wasm 程序, WASP 按 3 个步骤进行混合执行: 首先, WASP 解析 Wasm 二进制文件, 生成抽象语法树传递给混合执行解释器; 其次, 混合解释器执行程序指令, 记录具体执行和符号执行状态, 并成相应的路径条件; 最后, 混合执行结束时, WASP 调用 Z3 求解器进行命题求解和程序验证.

现有 Wasm 程序验证主要基于符号执行技术来验证恒定时间安全和功能正确性. 本文认为, 在该领域有 2 个重要的未来研究方向: 1)将定理证明、模型检测和抽象解释等其他程序验证技术用于构建面向 Wasm 的通用程序验证框架; 2)通过将多技术深度融合提出 Wasm 自动化程序验证工具, 例如将插值技术和可满足性模理论结合、将抽象解释和模型检测结合等, 在现有技术的基础上提出可配置的程序验证框架, 并在框架中集成多种验证技术和求解器, 从而提高框架的可扩展性和验证能力.

6.3 本节小结

形式语义与程序验证研究是严格证明 Wasm 语言设计的可靠性, 以及证明基于 Wasm 构建的软件系统满足特定安全规范的重要基础. 本文认为, 这个研究领域中有 2 个重要的未来研究方向: 1)随着 Wasm 规范的演进和新特性的增加, 对新规范和语言的新特性展开形式语义的研究; 2)采用多种验证技术融合, 进一步开展对 Wasm 程序的自动或半自动的程序验证研究. 随着 Wasm 的快速发展以及应用领域的快速扩展, 这 2 个方向的研究都亟待开展.

7 未来研究方向展望

随着 Wasm 的持续演进，及其应用领域的不断扩展，Wasm 安全仍将是未来的热点研究方向. 通过系统梳理、分析并总结已有研究，本文认为，该研究领域还有 5 个重要的未来研究方向：对 Wasm 生态系统的安全实证研究；通用 Wasm 中间表示和程序分析框架的研究和构建；对 Wasm 程序漏洞的自动修复研究；对 Wasm 线性内存的安全增强研究；Wasm 自动化程序验证研究.

1）对 Wasm 生态系统的安全实证研究. 安全实证研究可以为 Wasm 漏洞检测、安全增强等提供经验数据，对于 Wasm 安全研究的开展具有指导意义. 但是，现有实证研究只覆盖了高级语言支持[43-44]、编译工具链[22]及二进制表示[85] 3 个层面，且相关研究都存在数据集覆盖面窄、结论不通用等局限性. 因此，对包括语言虚拟机在内的 Wasm 全生态系统的全面、深入的安全实证研究是一个重要的未来研究方向. 未来针对 Wasm 实证研究不仅应该包括语言虚拟机在内的全生态系统，还应该覆盖物联网、边缘计算等 Wasm 的典型应用领域，同时对 Wasm 的语言设计与安全威胁的内在联系进行深入探索，提出 Wasm 的最佳安全实践，指导 Wasm 的持续演化.

2）通用 Wasm 中间表示和程序分析框架的研究和构建. 在 Wasm 安全研究中，通用的中间表示和程序分析框架起到关键作用，它们不但可以为漏洞检测研究提供基础、为 Wasm 动态和静态分析框架的构建提供核心能力，还可以为研究成果的持续积累和演进提供必要的基础设施支撑. 但已有研究尚未建立统一的中间表示和程序分析基础设施. 最近，Wasmati[90] 提出了一种融合抽象语法树、控制流图和程序依赖图的中间表示，这为 Wasm 程序分析基础框架的研究提供了新的可行思路. 但是，研究和构建面向 Wasm 的通用中间表示和程序分析基础设施，仍是一个非常有挑战性但亟待解决的研究问题.

3）对 Wasm 程序漏洞的自动修复研究. 研究表明[156]，在现代软件开发中，漏洞修复成本占开发过程总成本的 50%~70%. 因此，对程序漏洞的自动化修复研究，对于保证软件质量、提高软件开发效率有积极意义. 但是，目前并没有针对 Wasm 软件漏洞的自动修复研究. 一种可行的解决方案是借鉴已有漏洞修复理论和技术（如针对 Java 字节码[72,157] 的修复技术以及变异测试[158] 等），来研究和构建针对 Wasm 程序漏洞的自动修复理论和技术，有效提高漏洞修复的效率和降低漏洞修复成本.

4）对 Wasm 线性内存的安全增强研究. Wasm 线性内存的独特设计，实现了沙箱隔离和越界检查等安全保证，但由于 Wasm 缺少对线性内存自身的安全检查，可能导致新的安全漏洞，引入潜在的攻击面. 第 2 节的安全威胁模型中，许多安全漏洞都与线性内存的安全保护机制缺失相关，因此，有效的线性内存安全增强理论和技术将会极大提升 Wasm 软件系统的安全性，促进 Wasm 在各领域的应用及发展. 对线性内存的安全增强可以从 2 个层面进行：一是在编译器层面控制内存的分配与布局，同时为 Wasm 程序插入必要的安全检查指令[24]；二是在虚拟机层面添加越界检查机制[105].

5）Wasm 自动化程序验证研究. 对于面向 Wasm 的自动化程序验证研究，目前只有 WASP[148] 尝试基于符号执行来验证 Wasm 程序的功能正确性. 基于近年来自动化程序验证器领域最新研究进展（如 Dafny[159]，Why3[160]，VeriFast[161] 等），通过定制从 Wasm 到特定程序验证器所需输入的专用编译器，从而可以有效地将已有的自动化程序验证器用于 Wasm 程序验证，这是一个亟待探索的重要研究方向.

8 结 语

WebAssembly 作为最新一代安全、高效、可移植的二进制指令集体系结构和代码分发格式，正在成为最有前景的跨平台公共语言标准之一. 随着 WebAssembly 的快速发展和越来越广泛的应用，WebAssembly 安全已经成为近年来的热点研究领域. 本文首先总结了 WebAssembly 的核心安全特性，并提出了 WebAssembly 安全威胁模型；随后，本文提出了 WebAssembly 安全研究的分类学，将已有工作分为安全实证研究、漏洞检测与利用、安全增强以及形式语义与程序验证 4 类，并分别对这 4 类研究进行了综述和总结，分析了不足和亟待开展的工作. 最后，本文对 WebAssembly 安全的未来研究方向进行了展望，指出了 5 个潜在的研究方向，以期为相关领域的研究者提供有价值的参考.

作者贡献声明：庄骏杰和胡霜共同完成文献调研、内容整理和论文撰写；华保健和汪炀对论文选题、

研究展开和论文撰写提供了指导;潘志中参与了本文部分内容研究和论文撰写.庄骏杰和胡霜为共同第一作者,华保健和汪炀为共同通信作者.

参 考 文 献

[1] Yang Ting, Zhang Jiayuan, Huang Zaiqi, et al. Survey of industrial control systems security[J]. Journal of Computer Research and Development, 2022, 59(5): 1035−1053 (in Chinese)
(杨婷, 张嘉元, 黄在起, 等. 工业控制系统安全综述[J]. 计算机研究与发展, 2022, 59(5): 1035−1053)

[2] Madakam S, Lake V, Lake V, et al. Internet of things (IoT): A literature review[J]. Journal of Computer and Communications, 2015, 3(3): 164−173

[3] Zheng Zibin, Xie Shaoan, Dai Hongning, et al. Blockchain challenges and opportunities: A survey[J]. Internaltional Journal of Web and Grid Services, 2018, 14(4): 352−375

[4] Monrat A A, Schelén O, Andersson K. A survey of blockchain from the perspectives of applications, challenges, and opportunities[J]. IEEE Access, 2019, 7: 117134−117151

[5] Varghese B, Wang Nan, Barbhuiya S, et al. Challenges and opportunities in edge computing[C]//Proc of the 2016 IEEE Int Conf on Smart Cloud (SmartCloud). Piscataway, NJ: IEEE, 2016: 20−26

[6] Lynn T, Rosati P, Lejeune A, et al. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms[C]//Proc of the 7th IEEE Int Conf on Cloud Computing Technology and Science (CloudCom). Piscataway, NJ: IEEE, 2017: 162−169

[7] Leroy X. Java bytecode verification: Algorithms and formalizations [J]. Journal of Automated Reasoning, 2003, 30(3): 235−269

[8] Microsoft. NET[EB/OL]. [2022-11-07]. https://dotnet.microsoft.com/zh-cn/

[9] W3C Community Group. WebAssembly[EB/OL]. [2022-11-07]. https://webassembly.org/

[10] W3C Community Group. WebAssembly types syntax[EB/OL]. (2023-06-27)[2023-06-27]. https://webassembly.github.io/spec/core/syntax/types.html

[11] W3C Community Group. WebAssembly security document[EB/OL]. [2022-11-07]. https://www.wasm.com.cn/docs/security/

[12] W3C Community Group. WebAssembly execution[EB/OL]. (2023-06-27)[2023-06-27]. https://webassembly.github.io/spec/core/exec/index.html

[13] W3C Community Group. WebAssembly structure[EB/OL]. (2023-06-27)[2023-06-27]. https://webassembly.github.io/spec/core/syntax/index.html

[14] W3C. World Wide Web consortium (W3C) brings a new language tothe Web as WebAssembly becomes a W3C recommendation [EB/OL]. (2019-12-05)[2022-11-07]. https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

[15] W3C Community Group. WebAssembly roadmap[EB/OL]. [2022-11-07]. https://webassembly.org/roadmap/

[16] ByteCode Alliance. WASI: The WebAssembly system interface[EB/OL]. [2022-11-07]. https://wasi.dev/

[17] CNCF. Wasmcloud: Why stop at the edge[EB/OL]. [2022-11-07]. https://wasmcloud.com/

[18] Secondstate. The second state functions[EB/OL]. [2022-11-07]. https://www.secondstate.io/faas/

[19] Fastly. Faster, simpler, and more secure serverless code[EB/OL]. [2022-11-07]. https://www.fastly.com/products/edge-compute

[20] Gurdeep Singh R, Scholliers C. WARDuino: A dynamic WebAssembly virtual machine for programming microcontrollers[C]//Proc of the 16th ACM SIGPLAN Int Conf on Managed Programming Languages and Runtimes. New York: ACM, 2019: 27−36

[21] CNCF. WasmEdge bring the cloud-native and serverless application paradigms to edge computing[EB/OL]. [2022-11-07]. https://wasmedge.org/

[22] Romano A, Liu Xinyue, Kwon Y, et al. An empirical study of bugs in WebAssembly compilers[C]//Proc of the 36th IEEE/ACM Int Conf on Automated Software Engineering (ASE). Piscataway, NJ: IEEE, 2021: 42−54

[23] McFadden B, Lukasiewicz T, Dileo J, et al. Security chasms of Wasm[EB/OL]. (2018-08-03)[2022-11-07]. https://git.edik.cn/book/awesome-wasm-zh/raw/commit/e046f91804fb5deb95affb52d6348de92c5bd99c/spec/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf

[24] Lehmann D, Kinder J, Pradel M. Everything old is new again: Binary security of WebAssembly[C]//Proc of the 29th USENIX Security Symp (USENIX Security 20). Berkeley, CA: USENIX Association, 2020: 217−234

[25] Mozilla. Going public launch bug[EB/OL]. (2015-06-12)[2022-11-7]. https://github.com/WebAssembly/design/issues/150

[26] Yee B, Sehr D, Dardyk G, et al. Native client: A sandbox for portable, untrusted x86 native code[J]. Communications of the ACM, 2010, 53(1): 91−99

[27] Mozilla. Asm. js: An extraordinarily optimizable, low-level subset of JavaScript[EB/OL]. [2022-11-07]. http://asmjs.org/

[28] W3C Community Group. WebAssembly core specification W3C recommendation, 2019[EB/OL]. (2019-12-05)[2022-11-07]. https://www.w3.org/TR/wasm-core-1/

[29] Clark L. Standardizing WASI: A system interface to run WebAssembly outside the web[EB/OL]. (2019-03-27)[2022-11-07]. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[30] W3C. WebAssembly core specification 2.0[EB/OL]. (2019-06-27)[2023-06-27]. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[31] Attrapadung N, Hanaoka G, Mitsunari S, et al. Efficient two-level homomorphic encryption in prime-order bilinear groups and a fast implementation in webassembly[C]//Proc of the 13th on Asia Conf on Computer and Communications Security. New York: ACM, 2018:

685−697

[32] Deveria A, Schoors L. Can I use WebAssembly[EB/OL]. (2023-06-11)[2023-06-11]. https://caniuse.com/?search=WebAssembly

[33] Hall A, Ramachandran U. An execution model for serverless functions at the edge[C]//Proc of the 5th Int Conf on Internet of Things Design and Implementation. New York: ACM, 2019: 225−236

[34] Hickey P. Edge programming with rust and WebAssembly[EB/OL]. (2018-12-12)[2022-11-07]. https://www.fastly.com/blog/edge-programming-rust-web-assembly

[35] Hickey P. Lucet: Takes WebAssembly beyond the browser fastly [EB/OL]. (2019-03-28)[2022-11-07]. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[36] Varda K. WebAssembly on cloudflare workers[EB/OL]. (2018-10-01)[2022-11-07]. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/

[37] Block. One. EOSIO: Fast, flexible, and forward-driven[EB/OL]. [2022-11-07]. https://eos.io/

[38] McCallum T. Diving into Ethereum's virtual machine (EVM): The future of Ewasm[EB/OL]. (2019-10-21)[2022-11-07]. https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy

[39] Brown A, Sun Mingqiu. A proposed WebAssembly system interface API for machine learning[EB/OL]. [2022-11-07]. https://github.com/WebAssembly/wasi-nn

[40] Stack. Viry3D: A game engine that supports Wasm[EB/OL]. [2022-11-07]. http://www.viry3d.com/

[41] Akinyemi S. Awesome WebAssembly languages[EB/OL]. [2022-11-07]. https://github.com/appcypher/awesome-wasm-langs

[42] Peny P. MicroPython and WebAssembly (Wasm)[EB/OL]. [2022-11-07]. https://github.com/pmp-p/micropython-ports-wasm

[43] Stiévenart Q, De Roover C, Ghafari M. The security risk of lacking compiler protection in WebAssembly[C]//Proc of the 21st IEEE Int Conf on Software Quality, Reliability and Security (QRS). Piscataway, NJ: IEEE, 2021: 132−139

[44] Stiévenart Q, De Roover C, Ghafari M. Security risks of porting C programs to webassembly[C]//Proc of the 37th ACM/SIGAPP Symp on Applied Computing. New York: ACM, 2022: 1713−1722

[45] Astrauskas V, Matheja C, Poli F, et al. How do programmers use unsafe rust?[J]. Proceedings of the ACM on Programming Languages, 2020, 4 (OOPSLA): 1−27

[46] Van Rossum G. Python/C API reference manual[EB/OL]. (2006-09-19)[2022-11-07]. http://sephounet.free.fr/pdf/python/api.pdf

[47] Zakai A, Dawborn T, Shawabkeh M, et al. Emscripten: C/C++ compiler for Wasm[EB/OL]. [2022-11-07]. https://emscripten.org/

[48] Rust Foundation. The Rust programming language[EB/OL]. [2022-11-07]. https://github.com/rust-lang/rust

[49] Rust and WebAssembly Working Group. Wasm-bindgen: Facilitating high-level interactions between Wasm modules and JavaScript[EB/OL]. [2022-11-07]. https://github.com/rustwasm/wasm-bindgen

[50] Near. AssemblyScript: TypeScript complier for Wasm[EB/OL]. [2022-11-07]. https://github.com/AssemblyScript/assemblyscript

[51] van Laethem A, Esteban D, Evans Ron, et al. TinyGo: Go compiler for small places[EB/OL]. [2022-11-07]. https://github.com/tinygo-org/tinygo

[52] W3C Community Group. WebAssembly text format[EB/OL]. (2023-06-27)[2023-06-27]. https://webassembly.github.io/spec/core/text/index.html

[53] W3C. WebAssembly Web API[EB/OL]. (2022-04-19)[2022-11-07]. https://www.w3.org/TR/wasm-web-api-2/

[54] W3C Community Group. Understanding the JS API[EB/OL]. [2022-11-07]. https://webassembly.org/getting-started/js-api/

[55] Haas A, Rossberg A, Schuff D L, et al. Bringing the Web up to speed with WebAssembly[C]//Proc of the 38th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2017: 185−200

[56] Google. What is V8[EB/OL]. [2022-11-07]. https://v8.dev/

[57] Mozilla. SpiderMonkey: Mozilla's JavaScript and WebAssembly engine[EB/OL]. [2022-11-07]. https://spidermonkey.dev/

[58] Akinyemi S. Awesome WebAssembly runtimes[EB/OL]. [2022-11-07]. https://github.com/appcypher/awesome-wasm-runtimes

[59] Block. One. EOS VM: EOSIO[EB/OL]. [2022-11-07]. https://eos.io/for-developers/build/eos-vm/

[60] Intel. WebAssembly micro runtime[EB/OL]. [2022-11-07]. https://github.com/bytecodealliance/wasm-micro-runtime

[61] Bytecode Alliance. Wasmtime: A standalone runtime for WebAssembly[EB/OL]. [2022-11-07]. https://github.com/bytecodealliance/wasmtime

[62] Akbary S, Chaudry W, Chevalier S, et al. Run any code on any client with WebAssembly and Wasmer[EB/OL]. [2022-11-07]. https://wasmer.io/

[63] D'Antras A. Double free vulnerability in wasmtime[EB/OL]. (2021-12-04)[2022-11-07]. https://github.com/bytecodealliance/wasmtime/pull/3582

[64] Crichton A. x64: Incorrect codegen for f32x4. abs v128. not[EB/OL]. (2021-09-11)[2022-11-07]. https://github.com/bytecodealliance/wasmtime/issues/3327

[65] McCaskey M. Sandbox excape in wasmer[EB/OL]. (2020-10-24)[2022-11-07]. https://github.com/wasmerio/wasmer/issues/1759

[66] Morrisett G, Walker D, Crary K, et al. From system F to typed assembly language[J]. ACM Transactions on Programming Languages and Systems, 1999, 21(3): 527−568

[67] Necula G C. Proof-carrying code[C]//Proc of the 24th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 1997: 106−119

[68] Wahbe R, Lucco S, Anderson T E, et al. Efficient software-based fault isolation[C]//Proc of the 14th ACM Symp on Operating Systems Principles. New York: ACM, 1993: 203−216

[69] Pierce B C. Types and Programming Languages[M]. Cambridge, MA: MIT press, 2002: 1−13

[70] Damas L, Milner R. Principal type-schemes for functional

programs[C]//Proc of the 9th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 1982: 207−212

[71] Kozen D, Tseng W L D. The Böhm–Jacopini theorem is false, propositionally[C]//Proc of the 9th Int Conf on Mathematics of Program Construction. Berlin: Springer, 2008: 177−192

[72] Lindholm T, Yellin F, Bracha G, et al. The Java virtual machine specification[EB/OL]. (2013-02-28)[2023-06-27]. https://docs.oracle.com/javase/specs/jvms/se7/html/

[73] Arce I. The Shellcode generation[J]. IEEE Security & Privacy, 2004, 2(5): 72−76

[74] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Transactions on Information and System Security, 2009, 13(1): 1−40

[75] Zhang Chao, Wang Tielei, Wei Tao, et al. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time[C]//Proc of the 15th European Symp on Research in Computer Security. Berlin: Springer, 2010: 71−86

[76] John B. Memory safety: Old vulnerabilities become new with WebAssembly[EB/OL]. (2018-12-06)[2022-11-07]. https://www.forcepoint.com/blog/x-labs/new-whitepaper-memory-safety-old-vulnerabilities-become-new-webassembly

[77] Lea D. A memory allocator[EB/OL]. (2000-04-04)[2022-11-07]. https://gee.cs.oswego.edu/dl/html/malloc.html

[78] Chen Xiaoquan, Xue Rui. Cause, exploitation and mitigation of program vulnerability—C and C++ language as an example[J]. Journal of Cyber Security, 2017, 2(4): 41−56 (in Chinese)
(陈小全, 薛锐. 程序漏洞: 原因、利用与缓解——以 C 和 C++语言为例[J]. 信息安全学报, 2017, 2(4): 41−56)

[79] Zakai A, Dawborn T, Shawabkeh M, et al. Emscripten file system API[EB/OL][2022-11-07]https://emscripten.org/docs/api_reference/Filesystem-API.html

[80] Cowan C, Wagle F, Pu C, et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade[C]//Proc of the 2000 DARPA Information Survivability Conf and Exposition. Piscataway, NJ: IEEE, 2000: 119−129

[81] Gisbert H M, Ripoll I. On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows[C]//Proc of the 13th IEEE Int Symp on Network Computing and Applications. Piscataway, NJ: IEEE, 2014: 145−152

[82] Dhem J F, Koeune F, Leroux P A, et al. A practical implementation of the timing attack[C]//Proc of the 3rd Int Conf on Smart Card Research and Advanced Applications. Berlin: Springer, 1998: 167−182

[83] Kocher P, Horn J, Fogh A, et al. Spectre attacks: Exploiting speculative execution[J]. Communications of the ACM, 2020, 63(7): 93−101

[84] Richards G, Hammer C, Burg B, et al. The eval that men do[C]//Proc of the 25th European Conf on Object-Oriented Programming. Berlin: Springer, 2011: 52−78

[85] Hilbig A, Lehmann D, Pradel M. An empirical study of real-world webassembly binaries: Security, languages, use cases[C]//Proc of the 30th Web Conf 2021. New York: ACM, 2021: 2696−2708

[86] NSA Center. Juliet C/C++ 1.3[EB/OL]. (2017-10-01)[2022-11-07]. https://samate.nist.gov/SARD/test-suites/112

[87] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Proc of the 2004 Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2004: 75−86

[88] Stiévenart Q, De Roover C. Compositional information flow analysis for WebAssembly programs[C]//Proc of the 20th IEEE Int Working Conf on Source Code Analysis and Manipulation (SCAM). Piscataway, NJ: IEEE, 2020: 13−24

[89] Lopes P. Discovering vulnerabilities in WebAssembly with code property graphs[EB/OL]. [2022-11-07]. https://www.inesc-id.pt/publications/17400/pdf/

[90] Brito T, Lopes P, Santos N, et al. Wasmati: An efficient static vulnerability scanner for WebAssembly[J]. Computers & Security, 2022, 118: 102745

[91] Johnson E, Thien D, Alhessi Y, et al. Доверяй, но проверяй: SFI safety for native-compiled Wasm[C]//Proc of the 28th Network and Distributed System Security Symp (NDSS). Reston, VA: The Internet Society, 2021

[92] Romano A, Zheng Y, Wang W. MinerRay: Semantics-aware analysis for ever-evolving cryptojacking detection[C]//Proc of the 35th IEEE/ACM Int Conf on Automated Software Engineering (ASE). Piscataway, NJ: IEEE, 2020: 1129−1140

[93] Wang Dong, Jiang Bo, Chan W K. WANA: Symbolic execution of Wasm bytecode for cross-platform smart contract vulnerability detection[J]. arXiv preprint, arXiv: 2007.15510, 2020

[94] Naseem F N, Aris A, Babun L, et al. MINOS: A lightweight real-time cryptojacking detection system[C]//Proc of the 28th Network and Distributed System Security Symp (NDSS). Reston, VA: The Internet Society, 2021

[95] Haßler K, Maier D. WAFL: Binary-only WebAssembly fuzzing with fast snapshots[C]//Proc of the 5th Reversing and Offensive-oriented Trends Symp. New York: ACM 2021: 23−30

[96] Lehmann D, Torp M T, Pradel M. Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of WebAssembly [J]. arXiv preprint, arXiv: 2110.15433, 2021

[97] Chen Weimin, Sun Zihan, Wang Haoyu, et al. WASAI: Uncovering vulnerabilities in Wasm smart contracts[C]//Proc of the 31st ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2022: 703−715

[98] Lin Min, Zhang Chao. Fuzzing scheme for WebAssembly virtual machine[J]. Network Security Technology & Application, 2020, 2(6): 15−18 (in Chinese)
(林敏, 张超. 针对 WebAssembly 虚拟机的模糊测试方案[J]. 网络安全技术与应用, 2020, 2(6): 15−18)

[99] Jiang Bo, Li Zichao, Huang Yuhe, et al. WasmFuzzer: A fuzzer for WebAssembly virtual machines[C]//Proc of the 34th Int Conf on Software Engineering and Knowledge Engineering (SEKE 2022).

Pittsburgh, Pennsylvania: KSIR Virtual Conf Center, 2022: 537−542

[100] Bian Weikang, Meng Wei, Zhang Mingxue. MineThrottle: Defending against Wasm in-browser cryptojacking[C]//Proc of the 29th ACM Web Conf 2020. New York: ACM, 2020: 3112−3118

[101] Kelton C, Balasubramanian A, Raghavendra R, et al. Browser-based deep behavioral detection of web cryptomining with CoinSpy[C]// Proc of the 2020 Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb). Reston, VA: The Internet Society, 2020. 2020: 1−12

[102] Konoth R K, Vineti E, Moonsamy V, et al. MineSweeper: An in-depth look into drive-by cryptocurrency mining and its defense[C]//Proc of the 25th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 1714−1730

[103] Szanto A, Tamm T, Pagnoni A. Taint tracking for WebAssembly[J]. arXiv preprint, arXiv: 1807.08349, 2018

[104] Fu W, Lin R, Inge D. TaintAssembly: Taint-based information flow control tracking for WebAssembly[J]. arXiv preprint, arXiv: 1802.01050, 2018

[105] Lehmann D, Pradel M. Wasabi: A framework for dynamically analyzing WebAssembly[C]//Proc of the 24th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2019: 1045−1058

[106] Quan Lijin, Wu Lei, Wang Haoyu. EVulHunter: Detecting fake transfer vulnerabilities for EOSIO's smart contracts at WebAssembly-level[J]. arXiv preprint, arXiv: 1906.10362, 2019

[107] Yamaguchi F, Golde N, Arp D, et al. Modeling and discovering vulnerabilities with code property graphs[C]//Proc of the 35th IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2014: 590−604

[108] Bytecode Alliance. Lucet: A native WebAssembly compiler and runtime[EB/OL]. [2022-11-07]. https://github.com/bytecodealliance/lucet

[109] Dan. Javascript obfuscate and encoder[EB/OL]. [2022-11-07]. https://www.cleancss.com/javascript-obfuscate/index.php.

[110] Gu Jiuxiang, Wang Zhenhua, Kuen J, et al. Recent advances in convolutional neural networks[J]. Pattern Recognition, 2018, 77: 354−377

[111] Fioraldi A, Maier D, Eißfeldt H, et al. AFL++: Combining incremental steps of fuzzing research[C]//Proc of the 14th USENIX Workshop on Offensive Technologies (WOOT 20). Berkeley, CA: USENIX Association 2020

[112] Musch M, Wressnegger C, Johns M, et al. New kid on the Web: A study on the prevalence of WebAssembly in the wild[C]//Proc of the 16th Int Conf on Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin: Springer, 2019: 23−42

[113] Cabrera Arteaga J, Floros O, Vera Perez O, et al. CROW: Code diversification for Webassembly[C]//Proc of the 2021 Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb), Reston, VA, USA: The Internet Society, 2021

[114] Narayan S, Disselkoen C, Moghimi D, et al. Swivel: Hardening WebAssembly against Spectre[C]//Proc of the 30th USENIX Security Symp (USENIX Security 21). Berkeley, CA: USENIX Association, 2021: 1433−1450

[115] Disselkoen C, Renner J, Watt C, et al. Position paper: Progressive memory safety for WebAssembly[C]//Proc of the 8th Int Workshop on Hardware and Architectural Support for Security and Privacy. New York: ACM, 2019: 1−8

[116] Vassena M, Patrignani M. Memory safety preservation for WebAssembly[C/OL]//Proc of the 47th ACM SIGPLAN Symp on Principles of Programming Languages. New York: ACM, 2020[2022-11-07]. https://people.cispa.io/marco.vassena/publications_files/sc-wasm.pdf

[117] Liu Renju, Garcia L, Srivastava M. Aerogel: Lightweight access control framework for WebAssembly-based bare-metal IoT devices[C]//Proc of the 6th IEEE/ACM Symp on Edge Computing (SEC). Piscataway, NJ: IEEE, 2021: 94−105

[118] Sun Jian, Cao Dingyuan, Liu Ximing, et al. SELWasm: A code protection mechanism for WebAssembly[C]//Proc of the 2019 IEEE Int Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). Piscataway, NJ: IEEE, 2019: 1099−1106

[119] Ménétrey J, Pasin M, Felber P, et al. TWINE: An embedded trusted runtime for WebAssembly[C]//Proc of the 37th IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2021: 205−216

[120] Ménétrey J, Pasin M, Felber P, et al. WATZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone[J]. arXiv preprint, arXiv: 2206.08722, 2022

[121] Kolosick M, Narayan S, Johnson E, et al. Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI[J]. Proceedings of the ACM on Programming Languages, 2022, 6(POPL): 1−30

[122] Jacob M, Jakubowski M H, Naldurg P, et al. The superdiversifier: Peephole individualization for software protection[C]//Proc of the 3rd Int Workshop on Security. Berlin: Springer, 2008: 100−120

[123] Sasnauskas R, Chen Y, Collingbourne P, et al. Souper: A synthesizing superoptimizer[J]. arXiv preprint, arXiv: 1711.04422, 2017

[124] Moura L, Bjørner N. Z3: An efficient SMT solver[C]//Proc of the 14th Int Conf on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2008: 337−340

[125] Denis F, Bernstein D J, Percival C, et al. The sodium cryptography library[EB/OL]. [2022-11-07]. https://download.libsodium.org/doc/

[126] Google. TurboFan: A V8's optimizing compilers[EB/OL]. [2022-11-07]. https://v8.dev/docs/turbofan

[127] Akritidis P, Costa M, Castro M, et al. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors[C]//Proc of the 18th USENIX Security Symp. Berkeley, CA: USENIX Association, 2009

[128] Nagarakatte S, Zhao Jianzhou, Martin M M K, et al. SoftBound: Highly compatible and complete spatial memory safety for

C[C]//Proc of the 30th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2009: 245−258

[129] Van Der Kouwe E, Nigade V, Giuffrida C. Dangsan: Scalable use-after-free detection[C]//Proc of the 12th European Conf on Computer Systems. New York: ACM, 2017: 405−419

[130] OW2 consortium. ASM: An all purpose Java bytecode manipulation and analysis framework[EB/OL]. (2022-10-02)[2022-11-07]. https://asm.ow2.io/

[131] Vallée-Rai R, Co P, Gagnon E, et al. Soot: A Java bytecode optimization framework[C]//Proc of the 1999 Conf of the Centre for Advanced Studies on Collaborative Research. Indianapolis, Indiana: IBM, 1999

[132] Intel. What is Intel SGX[EB/OL]. [2023-06-27]. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html

[133] Sabt M, Achemlal M, Bouabdallah A. Trusted execution environment: What it is, and what it is not[C]//Proc of the 14th IEEE Trustcom/BigDataSE/ISPA. Piscataway, NJ: IEEE, 2015: 57−64

[134] Bellard F. QEMU, a fast and portable dynamic translator[C]//Proc of the FREENIX Track: 2005 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2005: 41−46

[135] SQLite Consortium. SQLite home page[EB/OL]. (2022-09-29)[2022-11-07]. https://www.sqlite.org/index.html

[136] Priebe C, Muthukumaran D, Lind J, et al. SGX-LKL: Securing the host OS interface for trusted execution[J]. arXiv preprint, arXiv: 1908.11143, 2019

[137] Pinto S, Santos N. Demystifying ARM TrustZone: A comprehensive survey[J]. ACM Computing Surveys, 2019, 51(6): 1−36

[138] Coker G, Guttman J, Loscocco P, et al. Principles of remote attestation[J]. International Journal of Information Security, 2011, 10(2): 63−81

[139] Kaplan D, Powell J, Woller T. AMD memory encryption[EB/OL]. (2021-10-18)[2022-11-07]. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf

[140] Lee D, Kohlbrenner D, Shinde S, et al. Keystone: An open framework for architecting trusted execution environments[C]//Proc of the 15th European Conf on Computer Systems. New York: ACM, 2020: 1−16

[141] Watt C. Mechanising and verifying the WebAssembly specification[C]//Proc of the 7th ACM SIGPLAN Int Conf on Certified Programs and Proofs. New York: ACM, 2018: 53−65

[142] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A Proof Assistant for Higher-Order Logic[M]. Berlin: Springer, 2002

[143] Watt C, Maksimović P, Krishnaswami N R, et al. A program logic for first-order encapsulated WebAssembly[C]//Proc of the 33rd European Conf on Object-Oriented Programming. Berlin: Springer, 2019

[144] Watt C, Renner J, Popescu N, et al. CT-Wasm: Type-driven secure cryptography for the web ecosystem[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 1−29

[145] Watt C, Rossberg A, Pichon-Pharabod J. Weakening WebAssembly[J]. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1−28

[146] Sjölén J. Relational symbolic execution in WebAssembly[D]. Stockholm, Sweden: KTH, School of Electrical Engineering and Computer Science (EECS), 2020

[147] Tsoupidi R M, Balliu M, Baudry B. Vivienne: Relational verification of cryptographic implementations in WebAssembly[C]//Proc of the 6th IEEE Secure Development Conf (SecDev). Piscataway, NJ: IEEE, 2021: 94−102

[148] Marques F, Fragoso Santos J, Santos N, et al. Concolic execution for WebAssembly[C]//Proc of the 36th European Conf on Object-Oriented Programming (ECOOP 2022). Berlin: Springer, 2022

[149] Cann R. Formal Semantics: An Introduction[M]. Cambridge, UK: Cambridge University Press, 1993

[150] Adve S V, Gharachorloo K. Shared memory consistency models: A tutorial[J]. Computer, 1996, 29(12): 66−76

[151] Hoare C A R. Communicating sequential processes[J]. Communications of the ACM, 1978, 21(8): 666−677

[152] Fetzer J H. Program verification: The very idea[J]. Communications of the ACM, 1988, 31(9): 1048−1063

[153] Sozeau M, Bertot Y, Dénès M, et al. The Coq proof assistant[EB/OL]. [2023-06-27]. https://coq.inria.fr/

[154] Farina G P, Chong S, Gaboardi M. Relational symbolic execution[C]//Proc of the 21st Int Symp on Principles and Practice of Declarative Programming. New York: ACM, 2019: 1−14

[155] Bernstein D J. The Salsa20 Family of Stream Ciphers[M]//New Stream Cipher Designs. Berlin: Springer, 2008: 84−97

[156] Zhang Yun, Liu Jiakun, Xia Xin, et al. Research progress on software bug localization technology based on information retrieval[J]. Journal of Software, 2020, 31(8): 2432−2452 (in Chinese)
(张芸, 刘佳琨, 夏鑫, 等. 基于信息检索的软件缺陷定位技术研究进展[J]. 软件学报, 2020, 31(8): 2432−2452)

[157] Ghanbari A, Benton S, Zhang Lingming. Practical program repair via bytecode mutation[C]//Proc of the 28th ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2019: 19−30

[158] Jia Yue, Harman M. An analysis and survey of the development of mutation testing[J]. IEEE Transactions on Software Engineering, 2010, 37(5): 649−678

[159] Microsoft. Dafny[EB/OL]. [2022-11-07]. https://dafny.org/

[160] Toccata. Why3: Where programs meet provers[EB/OL]. [2022-11-07]. http://why3.lri.fr/

[161] Jacobs B, Smans J, Piessens F, et al. VeriFast: A research prototype of a tool for modular formal verification[EB/OL]. (2021-04-21)[2022-11-07]. https://github.com/verifast/verifast

Zhuang Junjie, born in 1998. Master candidate. His main research interests include programming languages, and cyber and information security.
庄骏杰, 1998 年生. 硕士研究生. 主要研究方向为程序设计语言、网络与信息安全.

Hu Shuang, born in 1995. Master candidate. Her main research interest includes programming language and compiler.

胡　霜, 1995 年生. 硕士研究生. 主要研究方向为程序设计语言与编译器.

Hua Baojian, born in 1979. PhD, lecturer. Member of CCF. His main research interests include programming languages and cyber and information security. (bjhua@ustc.edu.cn)

华保健, 1979 年生. 博士, 讲师. CCF 会员. 主要研究方向为程序设计语言、网络与信息安全.

Wang Yang, born in 1980. PhD, associate professor. Senior member of CCF. His main research interests include spatiotemporal data mining and its interdisciplinary studies, wireless sensor network and vehicular network, and distributed systems. (angyan@ustc.edu.cn)

汪　炀, 1980 年生. 博士, 副教授. CCF 高级会员. 主要研究方向为时空数据挖掘与交叉研究、无线传感网与车联网、分布式系统.

Pan Zhizhong, born in 1988. Master. His main research interests include programming languages security and blockchain security.

潘志中, 1988 年生. 硕士. 主要研究方向为程序语言安全、区块链安全.