无论是extended Berkeley Packet Filter（eBPF，扩展的伯克利包过滤器）还是Web Assembly（wasm，网络汇编语言），两者在IoT[1]和Web[2-4]等领域得到了大量的应用。eBPF 和 wasm 两者作为新兴的轻量级代码沙盒技术，在实现上各有其相似和独特之处。

相似之处在于，eBPF 允许用户编写C或rust代码，通过编译工具链编译为bpf字节码，经由bpf系统调用加载到内核态验证，由 Just-In-Time（JIT，即时）编译器编译为最终的机器码等待特定事件触发执行[5]；执行时借由 eBPF 辅助函数来实现复杂逻辑[6]。 wasm 也支持如C，C++，rust，go，lua等多种高级编程语言，通过编译工具链将高级编程语言编写的代码编译为wasm字节码，经wasm运行时加载到用户态沙箱环境，由wasm解释器解释执行或JIT编译器、Ahead-of-Time（AoT，预先）编译器编译执行[1]6。执行时通过Web assembly System Interface（wasi，wasm系统调用接口）来与沙箱外的操作系统交互[1,7]。

独特之处在于，eBPF程序运行于内核态，且以寄存器虚拟机的形式执行指令。其异常捕获视系统所用的硬件指令集而定。而wasm运行于用户态，采用栈式虚拟机来执行指令以trap的方式来捕捉系统异常以告知操作系统终止沙盒内的程序[8]。

若是通过研究两者的特点以及在部署应用时普遍出现的问题，着重于综合比较研究两者在软件层面上已有的安全机制，较全面地整理前人所作的研究和优化策略，对于后续完善两者乃至在两者基础上构建一套更为安全可靠的沙盒机制，无论是对业界还是学术界，都会起到一定有益的帮助并提供科学、客观、有效的材料支撑。而这也正是本研究关切的所在。

1 eBPF现状

对于eBPF程序的执行过程，首先需要用户在相应的eBPF框架，如BCC Python等框架下编写，经过如LLVM/clang的编译工具链编译为eBPF字节码[6]。生成后的eBPF字节码再通过bpf()加载到内核态[6]，并创建eBPF映射表来维护同一eBPF程序的执行状态和eBPF 程序之间的状态分享[9]2。未经过运行的eBPF字节码会由eBPF静态验证器来实施检查[10-11]。检查内容包括穷举所有可能的执行情况，eBPF程序的内存访问情况，程序能否结束，循环次数，无效指令等[6,12]。验证程序不会对内核造成影响后，最后经由JIT编译器编译为对应ISA可执行的机器码[5]，并被加载到内核环境下的虚拟机中等待特定事件触发执行[13]。此外，eBPF还可以依靠Linux提供的Kprobe技术将程序插入到几乎任何内核函数所在的位置[6]。

eBPF的这种灵活性被用于加速特定的任务[11]，如网络包过滤[3-4]、网络流量监控[2]、文件数据存储[14]和FPGA接收网络数据流[15]等。因为eBPF运行于内核态上，eBPF也成为了潜在的攻击介质[16]。相应地就需要确保eBPF所加载执行的代码以及自身的处理逻辑不会破坏内核和其它程序的正常执行。 eBPF以性能和执行策略上的妥协来换取这一安全需求的实现。比如eBPF不支持浮点运算[17]，也不允许死循环的存在[6]，在执行用户态代码之前，还会调用自身的验证器检查代码是否符合规范。

即使有上述通过降低性能或缩减功能的等妥协来换取保证安全性的约束，李有霖[5, 62]和 Mohamed,Wang,Ravindran [18]表明，eBPF仍有三个主要的攻击面：验证器、辅

助函数和JIT编译器。特别地，Mohamed,Wang,Ravindran [18]汇总整理了截至2023年8月为止的eBPF的漏洞报告列表；指出在比重上，eBPF验证器占比最大，为44.4%；eBPF内核辅助函数占16.7%；eBPF分析字节码和转译执行的核心部分占11.1%； eBPF即时编译器占5.6%；其它（如以eBPF这一技术作为基础的其它应用，cilium）则占22.2%。而截至2024年8月，eBPF验证器就已经对eBPF整体贡献了超过半数的CVE[19]。

1.1 eBPF验证器存在的问题

就eBPF验证器而言，其实现逻辑规定eBPF程序的调用栈最大深度为512[6]30，且在加载前会检查eBPF字节码是否包含潜在的不安全操作和不可接受的性能开销[20]。据Lim,et al. [21]和Marcos A. M. Vieira,et al. [22, 6]指出， eBPF验证器对eBPF字节码有两步静态检查。第一步用深度优先搜索检查eBPF程序是否能构成directed acyclic graph(DAG, 有向无环图)，即是否没有包含不可到达的指令、无法跳出的循环以及非法跳转。包含则不通过此步骤的校验。第二步则按照第一步生成的有向无环图通过生成状态自动机来穷举和存储程序所有可能的路径状态，并要求经JIT编译后的指令数对于非特权程序最多不能超过4096条，特权程序不能超过100万条。

Lim,Han,Pasquier [23, 2]还指出， 自2019年3月Linux5.0发布一直到2023年4月发布Linux6.3为止，整个过程中eBPF验证器的代码行数从7306行增长到了17904行，以缓解自身设计和规范上的漏洞；但持续增加的代码规模以及逻辑复杂性，难以通过形式化验证整个验证器而消除验证器中存在的任何缺陷。此外，eBPF的验证器缺乏对循环的支持[24]，存在假阳性误报[19,24]以及李浩，等人 [20, 2,5,6]指出的假阴性，即验证器自身的漏洞被利用于恶意程序的绕过的问题。这些约束以及存在的问题限制了大型复杂程序在eBPF下的开发，使得开发人员不得不从验证器的角度开发[20]或重构[11]代码，将构成程序的整体代码拆解为许多小片零散的函数[25]，进而导致可编程性和执行性能的下降[26]，加重了开发者的负担。

其次，由于验证器造成 eBPF 在编程上的困难，又因为内核中存在着一些已编译过的、可为 eBPF 程序提供常用的功能的C函数能作为 eBPF 的辅助函数以节省开发者的精力和时间[5]，并方便eBPF程序和系统及其执行上下文之间交互[27]。辅助函数以白名单的形式提供给 eBPF 程序调用，不同类型的eBPF程序被允许调用不同的辅助函数集合[5,28]。

1.2 eBPF辅助函数存在的问题

Jia,et al. [29]指出，随着辅助函数的数量和代码复杂度的日益增加，潜在隐患和缺陷也随之被引入；截至2022年，已有249个的辅助函数，其逻辑复杂程度不一；有52.2%的辅助函数可以调用超过30个内核函数，34.2%的辅助函数可以调用超过500个内核函数。文献中衡量函数复杂度的标准是通过构建所有辅助函数的调用图，统计最终的调用节点数来判断。结果显示调用节点数在1到 10^3 以上均有分布。 Sahu,Williams [30, 3]指出，eBPF验证器无法进入这些辅助函数并使用给定参数分析；又由于eBPF程序经常使用这些内部可能执行复杂操作的辅助函数，它们对程序性能和安全性的影响同样不容忽视。另外据 Jia,et al. [29, 4]声称，不安全的eBPF辅助函数带来的问题仍没有被重视。

1.3 eBPF即时编译器存在的问题

eBPF即时编译器解释执行eBPF字节码。因eBPF即时编译器假设了自身将要执行的字节流均通过了验证器验证，于是在转译时没有做额外的安全检查[24]或者对内存漏洞的插桩验证[31]5。 Liu,et al. [32]的研究表明，eBPF字节码在转译过程缺少注入和劫持的防护，其解释器在执行时不会检测所执行的程序，其解释流可以被文中提出的“Tailcall Trampoline(尾调用蹦床)”攻击所劫持而用于执行任意的字节码，绕过现有内核代码完整性保护机制；出现这一问题的原因在于 eBPF 没有将 bpf_prog 数组限制在内核态而是通过让用户调用 sys_bpf(...) 而分配在用户态的堆上、解释器没有验证将要尾调用的函数是否遭受过篡改。Yuan,Besson,Talpin [33, 2]也提出， eBPF即时解释器或eBPF虚拟机的实现不当会引起例如内核态下任意代码执行的安全漏洞，从而破坏Linux内核完整性。

由上述依据，有关 eBPF 中三个主要攻击面和潜在攻击面的防护机制的整理以及可供创新之处，有待本研究后续整理深化。

2 WebAssembly

wasm是一种为增强Web浏览器性能和拓展性[34]而面向C，C++，rust和Go等高级编程语言设计的新型紧凑的[35-37] 中间编译表示规范[38-42]（或称字节码）。 wasm字节码通过如Emscripten[43]，cheerp[44] 和rustc等wasm前端编译器来生成[45]，因wasm字节码格式紧凑，空间占用小，其也被用于加速处理计算密集型任务[46]、部署于边缘计算和物联网领域[47]。因这种字节码并不能直接交由处理器译码执行，还需经过解释器或JIT或AoT等后端编译器转译为与当前运行环境下CPU指令集一致的机器码[1]6，wasm有跨OS和处理器架构的可移植性和可扩展性[1,39-40,42,47-48]。

wasm首先经过前端编译器编译为wasm形成中间的字节码表示，再交由wasm运行时执行[1]。 wasm运行时会根据wasm的二进制表示来划分到不同的内存节并解析成对应的指令[49]并指令严格按照规定的数量和类型，从栈上执行操作数[50]；其次，wasm设计之初即考虑安全[38]，采用了只提供四种基本数据类型(32/64位，整型/浮点型) 的静态强类型系统[8,34,38,51]、严格[52]且结构化[40,53]的控制流完整性验证并彻底废除了任意跳转指令[50]。

而庄骏杰，等人 [54]通过筛选调研了从2017年至2024年以来的44篇文献，还是明确地在高级语言支持、编译工具链、二进制表示和语言虚拟机这4个层面上归纳出了共计14种类型的安全漏洞，为后续研究提供切入的方向。

参考文献

[1] Yixuan Zhang, et al. Research on WebAssembly Runtimes: A Survey[Z]. 2024. arXiv: 2404.12621.

[2] C. Cassagnes, et al. The rise of eBPF for non-intrusive performance monitoring[C]. in: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium. 2020: 1-7.

[3] The Tcpdump Group. Tcpdump, a powerful command-line packet analyzer[EB/OL]. 2024. https://www.tcpdump.org/.

[4] M. A. M. Vieira, et al. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications[J]. ACM Comput. Surv., 2020, 53(1).

[5] 李有霖. 基于模糊测试的eBPF漏洞挖掘技术研究[D]. 四川成都: 电子科技大学, 2023.

[6] L. Rice. Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security[M]. First edition. Sebastopol, CA: O'Reilly Media, 2023.

[7] Yixuan Zhang, et al. Characterizing and Detecting WebAssembly Runtime Bugs[J]. ACM Transactions on Software Engineering and Methodology, 2024, 33(2): 1-29.

[8] WebAssembly Community, et al. WebAssembly Specification[Z]. Specification. 2024.

[9] T. A. Benson, et al. NetEdit: An Orchestration Platform for eBPF Network Functions at Scale [C]. in: Proceedings of the ACM SIGCOMM 2024 Conference. Sydney NSW Australia: ACM, 2024: 721-734.

[10] Yusheng Zheng, et al. Bpftime: Userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions[Z]. 2023. arXiv: 2311.07923.

[11] Hao Sun, et al. Validating the eBPF Verifier via State Embedding[C]. in: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). Santa Clara, CA: USENIX Association, 2024: 615-628.

[12] Y. Zhou, et al. Electrode: Accelerating Distributed Protocols with eBPF[C]. in: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). Boston, MA: USENIX Association, 2023: 1391-1407.

[13] Mao Jinsong, et al. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness[C]. in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. La Jolla CA USA: ACM, 2024: 639-653.

[14] Yuhong Zhong, et al. XRP: In-Kernel Storage Functions with eBPF[C]. in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 375-393.

[15] M. S. Brunella, et al. hXDP: Efficient Software Packet Processing on FPGA NICs[C]. in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020: 973-990.

[16] 张子君. Linux系统eBPF攻击建模及防护技术研究[D]. 浙江杭州: 浙江大学, 2024.

[17] A. Osaki, et al. Dynamic Fixed-Point Values in eBPF: A Case for Fully in-Kernel Anomaly Detection[C]. in: Aintec '24: Proceedings of the Asian Internet Engineering Conference 2024. New York, NY, USA: Association for Computing Machinery, 2024: 46-54.

[18] M. H. N. Mohamed, et al. Understanding the Security of Linux eBPF Subsystem[C]. in: Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems. Seoul Republic of Korea: ACM, 2023: 87-92.

[19] Peihua Zhang, et al. HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64[C]. in: 33rd USENIX Security Symposium (USENIX Security 24). Philadelphia, PA: USENIX Association, 2024: 163-180.

[20] 李浩, 等. 基于PKS硬件特性的eBPF内存隔离机制[J]. 软件学报, 2023, 34(12): 5921-5939.

[21] S. Y. Lim, et al. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions [Z]. 2024. arXiv: 2409.07508.

[22] Marcos A. M. Vieira, et al. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications[J]. ACM Computing Surveys, 2021, 53(1): 1-36.

[23] S. Y. Lim, et al. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing[C]. in: Proceedings of the 1st Workshop on eBPF and Kernel Extensions. New York NY USA: ACM, 2023: 42-48.

[24] E. Gershuni, et al. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions [C]. in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. Phoenix AZ USA: ACM, 2019: 1069-1084.

[25] Yoann Ghigoff, et al. BMC: Accelerating Memcached Using Safe in-Kernel Caching and Pre-Stack Processing[C]. in: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). USENIX Association, 2021: 487-501.

[26] H.-C. Kuo, et al. Verified Programs Can Party: Optimizing Kernel Extensions via Post-Verification Merging[C]. in: Proceedings of the Seventeenth European Conference on Computer Systems. Rennes France: ACM, 2022: 283-299.

[27] B. Gbadamosi, et al. The eBPF Runtime in the Linux Kernel[Z]. 2024. arXiv: 2410.00026.

[28] isovalent. eBPF Docs[EB/OL]. 2024 [2024-12-25]. https://github.com/isovalent/ebpf-docs .

[29] J. Jia, et al. Kernel Extension Verification Is Untenable[C]. in: Proceedings of the 19th Workshop on Hot Topics in Operating Systems. Providence RI USA: ACM, 2023: 150-157.

[30] R. Sahu, et al. Enabling BPF Runtime Policies for Better BPF Management[C]. in: Proceedings of the 1st Workshop on eBPF and Kernel Extensions. New York NY USA: ACM, 2023: 49-55.

[31] H. Sun, et al. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program[C]. in: Proceedings of the Nineteenth European Conference on Computer Systems. Athens Greece: ACM, 2024: 689-703.

[32] Q. Liu, et al. Interp-flow Hijacking: Launching Non-control Data Attack via Hijacking eBPF Interpretation Flow[C]. in: Computer Security – ESORICS 2024: 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16–20, 2024, Proceedings, Part III. Bydgoszcz, Poland: Springer-Verlag, 2024: 194-214.

[33] S. Yuan, et al. End-to-End Mechanized Proof of a JIT-accelerated eBPF Virtual Machine for IoT[C]. in: A. Gurfinkel, et al. Computer Aided Verification. Cham: Springer Nature Switzerland, 2024: 325-347.

[34] A. Haas, et al. Bringing the Web up to Speed with WebAssembly[C]. in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. Barcelona Spain: ACM, 2017: 185-200.

[35] Yutian Yan, et al. Understanding the Performance of Webassembly Applications[C]. in: Proceedings of the 21st ACM Internet Measurement Conference. Virtual Event: ACM, 2021: 533-549.

[36] B. L. Titzer. A Fast In-Place Interpreter for WebAssembly[Z]. 2022. arXiv: 2205.01183.

[37] P. Daniel, et al. Discovering Vulnerabilities in WebAssembly with Code Property Graphs [C]. in: 2019.

[38] WebAssembly Community Group. Introduction of WebAssembly[Z]. https://webassembly.github.io/spec/core/intro/introduction.html. 2024.

[39] D. Lehmann, et al. Wasabi: A Framework for Dynamically Analyzing WebAssembly[C]. in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Providence RI USA: ACM, 2019: 1045-1058.

[40] D. Lehmann, et al. Everything Old is New Again: Binary Security of WebAssembly[C]. in: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, 2020: 217-234.

[41] S. Bhansali, et al. A First Look at Code Obfuscation for WebAssembly[C]. in: Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks. San Antonio TX USA: ACM, 2022: 140-145.

[42] M. Waseem, et al. Issues and Their Causes in WebAssembly Applications: An Empirical Study[Z]. 2024. arXiv: 2311.00646.

[43] Emscripten Contributors. emscripten[Z]. https://emscripten.org/. 2015.

[44] Leaning Technologies. An Enterprise-Grade C++ Compiler For The Web[Z]. https://leaningtech.com/cheerp/. 2024.

[45] A. Romano, et al. An Empirical Study of Bugs in WebAssembly Compilers[C]. in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). Melbourne, Australia: IEEE, 2021: 42-54.

[46] S. Cao, et al. WASMixer: Binary Obfuscation for WebAssembly[Z]. 2023. arXiv: 2308.03123.

[47] P. P. Ray. An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions[J]. Future Internet, 2023, 15(8): 275.

[48] J. Bosamiya, et al. Provably-Safe Multilingual Software Sandboxing using WebAssembly [C]. in: 31st USENIX Security Symposium (USENIX Security 22). Boston, MA: USENIX Association, 2022: 1975-1992.

[49] skanehira. Writting a Wasm Runtime in Rust[EB/OL]. 2024. https://github.com/skanehira/writing-a-wasm-runtime-in-rust/.

[50] 张秀宏. WebAssembly原理与核心技术[M]. 北京: 机械工业出版社, 2020.

[51] Wenlong Zheng, et al. WASMDYPA: Effectively Detecting WebAssembly Bugs via Dynamic Program Analysis[C]. in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Rovaniemi, Finland: IEEE, 2024: 296-307.

[52] J. Ménétrey, et al. A Comprehensive Trusted Runtime for WebAssembly With Intel SGX [J]. IEEE Transactions on Dependable and Secure Computing, 2024, 21(4): 3562-3579.

[53] I. Bastys, et al. SecWasm: Information Flow Control for WebAssembly[G]. in: G. Singh, et al. Static Analysis: vol. 13790. Cham: Springer Nature Switzerland, 2022: 74-103.

[54] 庄骏杰, 等. WebAssembly安全研究综述[J]. 计算机研究与发展, 2024, 61(8): 1-27.

[55] J. Dejaeghere, et al. Comparing Security in eBPF and WebAssembly[C]. in: eBPF '23: Proceedings of the 1st Workshop on EBPF and Kernel Extensions. New York, NY, USA: Association for Computing Machinery, 2023: 35-41.

[56] A. Jangda, et al. Not so Fast: Analyzing the Performance of WebAssembly vs. Native Code [C]. in: 2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA: USENIX Association, 2019: 107-120.

[57] E. Johnson, et al. WaVe: A Verifiably Secure WebAssembly Sandboxing Runtime[C]. in: 2023 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, 2023: 2940-2955.

[58] S. Narayan, et al. Swivel: Hardening WebAssembly against Spectre[Z]. 2021.

[59] Yusheng Zheng, et al. Wasm-Bpf: Streamlining eBPF Deployment in Cloud Environments with WebAssembly[Z]. 2024. arXiv: 2408.04856.

[60] WebAssembly Community Group, et al. WebAssembly Specification[Z]. Specification. 2024.

[61] Yi He, et al. Cross Container Attacks: The Bewildered eBPF on Clouds[C]. in: 32nd USENIX Security Symposium (USENIX Security 23). Anaheim, CA: USENIX Association, 2023: 5971-5988.