处理器微体系结构安全研究综述

尹嘉伟[1,2,3,4], 李孟豪[1,2,3,4], 霍 玮[1,2,3,4]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 中国科学院网络测评技术重点实验室 北京 中国 100195
[3] 网络安全防护技术北京市重点实验室 北京 中国 100195
[4] 中国科学院大学网络空间安全学院 北京 中国 100049

摘要 在 CPU 指令流水线中, 为了提高计算机系统的执行效率而加入的 Cache、TLB 等缓存结构是不同进程共享的, 因此这些缓存以及相关执行单元在不同进程之间的共享在一定程度上打破了计算机系统中基于内存隔离实现的安全边界, 进而打破了计算机系统的机密性和完整性。*Spectre* 和 *Meltdown* 等漏洞的披露, 进一步说明了处理器微体系结构所采用的乱序执行、分支预测和推测执行等性能优化设计存在着严重的安全缺陷, 其潜在威胁将涉及到整个计算机行业的生态环境。然而, 对于微体系结构的安全分析, 到目前为止尚未形成较为成熟的研究框架。虽然当前针对操作系统内核及上层应用程序的漏洞检测和安全防护方面已经有较为成熟的方法和工具, 但这些方法和工具并不能直接应用于对微体系结构漏洞的安全检测之中。一旦微体系结构中出现了漏洞将导致其危害更加广泛并且难以修复。此外, 由于各个处理器厂商并没有公布微体系结构的实现细节, 对于微体系结构安全研究人员来说, 微体系结构仍然处于黑盒状态, 并且缺少进行辅助分析的工具。这也使得微体系结构的安全分析变得十分困难。因此本文从当前处理器微体系结构设计中存在的安全威胁入手, 分析了其在设计上导致漏洞产生的主要原因, 对现有处理器微体系结构的 7 种主流攻击方法进行了分类描述和总结, 分析对比现有的 10 种软硬件防护措施所采用的保护方法及实用效果, 并从微体系结构漏洞研究方法、漏洞防护及安全设计等方面, 进一步探讨了处理器微体系结构安全的研究方向和发展趋势。

关键词 处理器微体系结构安全; 微指令集漏洞; 信息泄露; 侧信道攻击; 防御技术;
中图法分类号 TP309.7 DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.07.02

Survey on Security Researches of Processor's Microarchitecture

YIN Jiawei[1,2,3,4], LI Menghao[1,2,3,4], HUO Wei[1,2,3,4]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing 100195, China
[3] Beijing Key Laboratory of Network Security and Protection Technology, Beijing 100195, China
[4] School of CyberSpace Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract In the instruction pipeline, cache structures such as Cache and TLB, which are added to improve the execution efficiency of computer systems, are shared by different processes. The sharing of these cache structures and related execution units between different processes breaks the security boundary implemented in computer systems based on memory isolation, which in turn breaks the confidentiality and integrity of entire computer systems. The disclosure of attacks on processor's micro-architecture such as *Spectre* and *Meltdown* indicates that the performance optimization techniques, such as out-of-order execution, branch prediction and speculative execution, that are used in current processors have some serious security flaws. They are capable to threat the entire computer ecosystem. Although there are many methods and tools for vulnerability detection and security protection of operating system kernel and user space applications, these methods and tools are not capable to be directly applied to detect the micro-architecture vulnerabilities which are hidden in the micro-architecture. Once a vulnerability occurs in a micro-architecture, it will be more dangerous and difficult to fix. In addition, because the implementation details of micro-architecture are not published by the processor vendors (e.g., Intel, AMD, and ARM), micro-architecture remains in a black-box state for micro-architecture security researchers. Moreover, there is a lack of tools and methods to assist in the analysis of micro-architecture. This also makes the security analysis of micro-architecture very difficult. Therefore, In this paper, we begin with the security threats in the current design of processor

通讯作者: 李孟豪, 博士, 助理研究员, Email: limenghao@iie.ac.cn。

本课题得到了中国国家自然科学基金(No. 61602470, No. 61702508, No. 61802394, No. U1836209, No. 62032010), 中国国家重点研究开发计划(No. 2016QY071405), 中国科学院战略重点研究计划(No. XDC02040100, No. XDC02030200, No. XDC02020200)的部分支持。

收稿日期: 2019-11-19; 修改日期: 2019-11-19; 定稿日期: 2022-05-11

micro-architecture to analyze the roots of the micro-architecture vulnerabilities, and summarize seven attack methods on the existing processor micro-architecture. We systematically illustrate 10 kinds of software and hardware defense mechanisms and summarize the effects of them. Besides, we further discuss the research and development trend of micro-architecture security from the vulnerability examination approaches, vulnerability protection methods and security designs.

Key words processor's micro-architecture security; micro-instruction set vulnerability; information leakage; side channel attack; defense methods

1 引言

一个完整的计算机系统主要由应用层软件, 操作系统, 硬件等几个部分组成。这些组成部分的任何环节出现漏洞, 都有可能破坏整个计算机系统的安全。因此, 针对整个计算机系统而言, 需要应用软件, 操作系统, 以及硬件这几个部分协同设计实现, 共同保证整个计算机系统的机密性, 完整性以及可用性。由于 Intel、AMD 以及 ARM 等厂商的处理器在微体系结构层面采用了相似的设计思路, 且这些厂商的处理器几乎覆盖了整个计算机行业, 因此微体系结构漏洞可能会威胁到整个计算机行业的安全。为了保证程序的正确以及安全运行, 计算机系统的设计以及实现人员在体系结构层面实现了诸如内存隔离、内存地址虚实转换、内存地址随机化以及内存加密等技术, 并以这些技术为基础, 针对进程及其相关数据实现了安全边界, 任何跨越安全边界的访问都会被拒绝。而在 CPU 指令流水线中, 为了提高计算机系统的执行效率, 并且解决处理器计算速度和访存速度不匹配的问题, CPU 体系设计人员在指令执行以及访存过程中加入了 Cache[1]、TLB 等缓存结构并引入了推测执行[2], 分支预测[3]等优化措施, 这些缓存结构、推测执行以及分支预测单元是不同进程共享的, 因此这些缓存以及相关执行单元在不同进程之间的共享在一定程度上打破了计算机系统中基于内存隔离实现的安全边界, 进而打破了计算机系统的机密性和完整性。而随着 Spectre[4]、Meltdown[5]等漏洞的披露, 印证了微体系结构漏洞的严重危害, 以及当前安全漏洞检测的研究体系中, 缺少对于微体系结构漏洞的检测和防护的相关技术。此外, 由于各个处理器厂商并没有公布微体系结构的实现细节, 对于微体系结构安全研究人员来说, 微体系结构仍然处于黑盒状态, 并且缺少进行辅助分析的工具。这也使得微体系结构的安全分析变得十分困难。

微体系结构是计算机体系结构的重要组成部分, 一旦其出现漏洞, 将会影响构建于其上的操作系统及应用程序的安全性。然而, 对于微体系结构的安全分析, 到目前为止尚未形成较为成熟的研究框架。虽然当前针对操作系统内核及上层应用程序的漏洞检测和安全防护方面已经有较为成熟的方法和工具, 但这些方法和工具并不能直接应用于对微体系结构漏洞的安全检测之中。一旦微体系结构中出现了漏洞将导致其危害更加广泛并且难以修复。

本文总结了当前处理器在微体系结构层面所面临的安全威胁及相应的防护技术, 并指出了微体系结构安全的后续研究方向, 其主要贡献如下:

(1) 分析了 CPU 指令流水线中乱序执行、分支预测以及缓存结构等优化措施对体系结构安全设计产生的安全威胁。本文较为全面的总结了处理器微体系结构的安全缺陷所带来的攻击方法。

(2) 本文从引入漏洞的优化措施角度, 对现有的微体系结构漏洞进行了分类分析, 详细的介绍了微体系结构漏洞的利用方法和缓解漏洞威胁的安全防护措施①, 并从硬件以及软件两个角度对各个防护措施进行了分类统计与对比分析; 本文较为全面的统计并对比了当前各类针对微体系结构漏洞的安全防护措施。

(3) 从微体系结构设计、操作系统设计、相关软件设计以及微体系结构漏洞挖掘方法这三个方面阐述了微体系结构安全未来的研究方向以及相关方法。总结创新处理器微体系结构安全研究的框架, 为后续研究提供较为实际的方法论指导。

本文结构: 首先对计算机微体系结构的基本架构及相关安全性设计进行介绍(第 2 章); 然后对微体系结构当前所面对的安全威胁和挑战进行总结, 包括常见攻击手段、硬件漏洞、以及防护绕过手段等(第 3 章); 随后针对已知的微体系结构安全威胁, 总结当前已有的防御技术, 包括软件防御及硬件防御技术, 并对比各种防御技术的实施效果(第 4 章); 分别

① 由于本文的讨论范围仅限于为提高 CPU 计算速度以及解决 CPU 计算能力和访存能力之间的差异而在 CPU 指令流水线中引入的优化措施以及缓存结构, 因此 NetCat[6]以及汪东升教授团队披露的电源管理机制等漏洞不在本文的考虑范围。

从设计原理和实现两方面, 对当前微体系结构安全研究进行讨论和未来研究方向(第 5 章); 最后进行总结(第 6 章)。

2 研究背景

本章节将介绍当前计算机微体系结构的基本设计以及微体系结构漏洞对整个计算机领域的影响。

2.1 微体系结构基本设计

计算机微体系结构刻画了指令集在 CPU 指令流水线中的执行方式, 包括分支预测、乱序执行等单元的内部交互, 以及多级高速缓存(Cache)、转换旁视缓冲区(TLB)等用于提高指令执行效率的缓存结构。在实现中, 微体系结构是对计算机上运行的指令集架构的进一步解析与处理, 保证程序指令在实际处理器上能够被正确读取、解析、执行和输出的一套完整方法。为了提高存储系统和处理器单元的使用效率, 并且消除处理器计算能力以及访存速度的不匹配的问题, 微体系结构在设计上采用了分级存储、乱序执行、分支预测以及推测执行等技术。本节将分别进行介绍。

2.1.1 分级存储

在体系结构中, 对数据进行处理的过程需要处理器和主存储系统系统配合来完成。其中, 存储系统主要负责提供处理器所需要的相应的数据和运算指令, 并将处理器计算生成结果进行记录。但是, 相较于运算速度增势迅猛的处理器来说, 主存储系统的输入输出速率并不能始终跟上处理器的运行频率。在处理器进行计算的过程中, 计算所需要的数据需要从内存中获取, 并且计算的结果需要写入到内存之中。因此, 较快的处理器运算与较慢的存储系统之间的数据传输速度差异是制约计算机运算效率的主要问题。

为了解决此问题, 处理器研发人员在设计中依次加入了多层级、小容量但具有更快读写速度的缓存结构作为处理器中运算单元与主存储系统之间的桥梁以减少处理器所需等待的时间。当前主流台式机、笔记本及服务器中均采用 3 级缓存结构来提升运算效率[7]。其缓存结构如图 1 所示。

多级缓存作为主存和处理器之间数据传递的桥梁纽带, 不断地将主存中的数据传递到更靠近处理器的缓存中, 以供处理器进行处理分析。在对主存中数据进行传递的过程中, 缓存会将主存分成若干定长的数据块(数据块又被称为行, 其长度一般保持在 64 到 128 字节之间[4])。这些数据块将被按需复制到各级缓存之中。在图 1 中, L1 级缓存位于分级存储系统的顶层, 能够直接与处理器交互, 需要具有能够匹配处理器处理数据速度的数据交换速度。但是, 由于成本受限, L1 级缓存的存储容量是分级存储中最小的。L2 级缓存位于分级存储的中间位置, 其数据交换速度低于 L1 级缓存但高于 L3 级缓存; 其存储容量也介于 L1 级缓存与 L3 级缓存之间。最后一级缓存也被称为(Last-level-Cache)LLC 级缓存(在图 1 中, LLC 指的是 L3)。在设计中, L3 级缓存被设定为能够被多个处理器核心所共享的形式。即在当前多核心的处理器微体系结构中, L3 即缓存中的数据能够被不同的处理器核心做共享, 如图 1 所示。而此共享缓存的设计方式给微体系结构的安全性带来的一定程度的隐患, 本文将在第 3.1 节进行详细讨论。

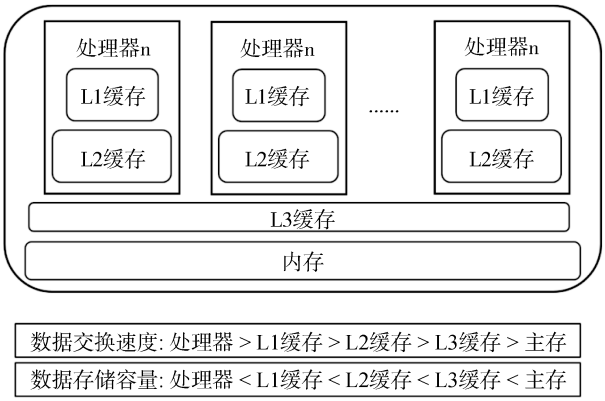


图 1 处理器 3 级缓存数据交换速度与容量相对关系
Figure 1 Processor Level 3 cache data exchange speed versus capacity

当处理器需要读取存储系统中数据时, 首先会检查 L1 缓存中是否存在所需数据。如果 L1 级缓存中存在处理器所需要的数据, 则直接将此数据传递到处理器的寄存器中, 否则, 程序会将数据请求传递给后续层级的缓存, 直到访问到主存。当完成数据读取后, 该数据会被暂存于缓存之中, 以备近期再次被访问。

2.1.2 乱序执行

传统的处理器在对程序中的指令进行处理的过程中, 是按照程序指令的线性顺序依次执行的。处理器在读取存储中数据的过程中可能需要访问数据交换速度较慢的主存设备, 而在此过程中, 处理器需要停止程序的执行, 并等待数据的读取, 直到所需要的数据到达之后才能继续工作。此等待过程会使得大量的处理器时钟周期处于闲置状态, 不能得到充分利用。

为了尽最大限度地发挥处理器的计算性能并且充分利用闲置的处理器时钟周期, 处理器在对指令进行运算处理的过程中不再机械式地线性执行程序

代码指令, 而是在处理器逻辑中加入了能够充分利用闲置处理器时钟周期的乱序执行(out of order execution)技术。乱序执行技术是由 Tomasulo[8]于 1976 年提出。其基本原理通过将运算指令解构为更小粒度的微指令(micro-operations, μOPs)集合, 并通过动态调度的方式来实现微指令的并行执行, 从而实现乱序执行。

乱序执行的实现如图 2 所示。首先, 处理器对当前要执行的指令进行解构处理, 将指令分解为多条微指令集合。这些将作为乱序执行的基础被传入重排序缓冲区。重排序缓冲区(reorder buffer)的作用是对微指令中所要使用的各个寄存器进行分配、重命名、以及释放等操作。此后, 经过处理的微指令集合将在调度器(即统一保留站)中进行统一调度, 以充分满足执行单元组中各个算术逻辑单元(arithmetic logic units)、地址生成单元(address generation units)的需要。在乱序执行过程中产生的异常, 不会立即触发异常处理程序, 并且异常指令之后的指令可以继续乱序执行, 只有在异常指令提交的时候该异常才会被触发。

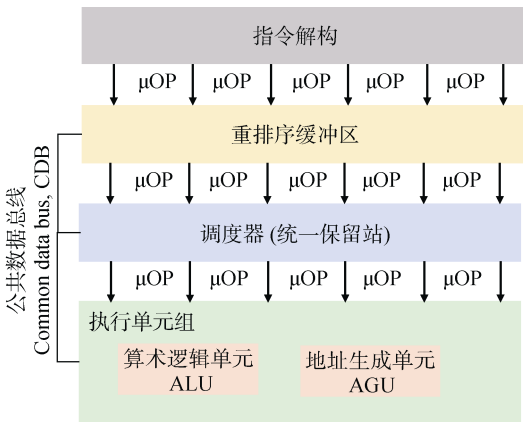


图 2　乱序执行实现原理

Figure 2　Principle of out of order execution

2.1.3　分支预测

分支预测技术作为能够进一步提高处理器运算速度的非线性执行技术, 目前已充分应用于主流处理器中。在执行过程中, 处理器利用分支预测技术对后续将要执行的指令进行合理猜测, 结合乱序执行技术预先执行猜测的指令, 并在分支预测正确的条件下, 直接输出该分支上的执行结果。如果分支预测不正确, 则将程序运行状态回滚到预测之前的正确状态, 重新进行处理即可。

当前已有的分支预测技术可以分为两大类: 一类是静态分支预测技术[7], 另一类是动态分支预测技术[9-10]。静态预测强调的是所预测的执行分支只能单调依赖于当前执行的指令, 而动态预测不仅能够结合当前执行指令的依赖关系, 还能够统计近期程序的执行轨迹用于更准确的进行分支预测。为了能够更精确地预测执行分支, 基于感知网络的神经分支预测技术也在不断发展并已经开始融入到实际的处理器微体系结构之中[11-12]。

在实现分支预测技术时, 处理器中会使用分支目标缓冲区(Branch Target Buffer, BTB)作为最近执行过的指令分支的目标地址的映射信息[13]。处理器可以利用 BTB 在指令解构之前预测未来将要执行的分支, 提高预测效率。即采用 BTB 能够在乱序执行之前, 将预测指令和当前运行指令同时进行解构形成混合微指令集合, 并利用乱序执行技术在处理器中进行并行运算, 同时得到当前指令及预测指令的输出信息, 以实现预测分支的超前执行。分支预测技术只能在同一个物理处理器内核中执行, 不能跨越不同内核进行共享, 因此, 在实现上, 分支预测只能应用于运行于同一个物理内核上的程序中[14]。

2.1.4　推测执行

在充分运用乱序执行、分支预测技术的前提下, 处理器能够完整实现对程序运行指令的推测执行(speculative execution)。在程序执行过程中, 处理器只能专注于当前执行的指令, 不能获取程序后续指令流中的指令。在实现中, 为了提高处理器的执行效率, 处理器会将当前运行程序结果的寄存器状态, 并基于分支预测技术预期程序执行路径中的后续执行指令, 并在处理器中采用乱序执行的方法预先计算出结果。当执行过程中遇到条件分支时, 处理器会根据已定义的预测规则[7, 9-12], 选取其认为执行可能性最高的分支进行推测执行。如果预测正确, 则将推测执行结果提交。这样既能减少处理器运行停驻的时间开销, 同时也能提高程序指令执行的效率。如果预测失败, 则处理器抛弃当前推测执行的结果, 并将保存当前运行结果的寄存器状态复原, 继续执行正确分支上的后续指令。

在现代处理器中, 推测执行有能力超前几百个指令进行预测[4]。目前唯一的限制在于处理器中重排序缓冲区的容量(如图 2 所示)。重排序缓冲区作为实现乱序执行重要的步骤, 约束着能够并行执行的微指令的数量上限。由于不同指令解构后所产生的微指令的数量不同, 这就导致每次能够进行乱序执行的指令数量会有较大的差别, 进而影响推测执行的能力。

2.2 微体系结构对于体系结构安全设计的影响

由于 Intel、AMD 以及 ARM 等主流处理器芯片厂商在微体系结构层面采用了相似的设计思路, 一旦处理器微体系结构的设计上存在安全隐患, 其威胁很有可能会蔓延到整个计算机行业。随着 Spectre 和 Meltdown 漏洞的披露, 使得安全研究人员更加清晰的认识到了微体系结构漏洞的安全威胁。在 Spectre[15]和 Meltdown[16]漏洞披露之前, 为了保证计算机系统的机密性, 完整性和一致性, 计算机设计以及安全研究人员均假设处理器的微体系结构是可信任的安全运算基础, 并在此基础上, 在体系结构层面实现了诸如内存隔离、内存地址虚实转换、内存地址随机化以及内存加密等技术, 这些技术有效的保证了程序的正确和安全运行。但是随着 Spectre 以及 Meltdown 漏洞的披露, 使得安全研究人员认识到, 微体系结构存在着巨大的安全威胁, 并且微体系结构的安全威胁比体系结构层面的安全威胁影响更广, 且更难修复。微体系结构位于整个计算机系统的底层, 在指令的执行过程中, CPU 指令流水线使用相同的执行单元执行不同进程的指令, 并将执行结果存储在 Cache、TLB、BTB 以及 PHT 等缓存结构中, 而这些缓存结构是运行在同一个 CPU 上所有进程所共享的, 因此微体系结构缓存的共享机制在一定程度上打破了以内存隔离、内存地址虚实转换、内存地址随机化以及内存加密等技术为基础实现的安全边界。

2.2.1 内存隔离和内存地址虚实转换

内存隔离(memory isolation)是为增强内存保护能力而开发的防护机制。传统的内存隔离技术通常采用分段(segmentation)和分页(paging)的方法来实现。

经典的基于分段的内存隔离技术(主要针对 x86 平台)主要采用由起始地址、空间大小和访问权限组成的段(segment)信息作为区分和隔离不同进程所使用内存区域的主要特征。其中, 访问权限是实现内存隔离的重要保障。随着 64 位体系结构系统的兴起, 指向安全内存区域的访问地址不再被存储于普通内存中。分段的隔离能力被削弱。为了保证 64 位系统的内存访问权限保护, 文献[17-18]通过将相关数据存储地址随机化实现对相应信息的隐藏, 以保证所访问只能进行段内访问, 确保不同进程内存之间的独立性。

除了基于分段的内存隔离技术之外, 经典的内存隔离还可通过分页技术实现。分页技术利用页表结构来实现从虚拟内存到物理内存的映射, 其中存储着映射关系和所需的访问权限信息。为保证不同进程之间的内存隔离, 操作系统会给每一个进程分配一个页表。在进程执行过程中, 操作系统将基于页表中的权限信息, 对相应的内存区域进行授权操作。

2.2.2 内存地址随机化

内存地址随机化(Address Space Layout Randomization, ASLR)能够有效缓解缓冲区溢出漏洞带来的安全威胁。在实现中, 通过将系统中的重要功能、服务、应用的内存区域地址随机化, 使得攻击者无法利用缓冲区溢出漏洞对上述内存区域进行越权访问或进行控制流劫持操作, 进而保护进程的安全运行。

2.2.3 内存加密

内存加密技术的目的是用于保护内存中数据和代码的机密性。虽然内存加密不能直接减缓或防御微体系结构设计中的侧信道漏洞问题, 但是通过对内存数据的机密性保护, 可以减少被窃取内存内容的可读性进而减少或者避免由攻击者所带来的损失。

依据加密所需的密钥的产生方式, 可以将内存加密方法分为三类: 基于 CPU 硬件生成密钥的内存加密方法, 基于操作系统计算密钥的内存加密方法, 以及基于专用加密处理器的内存加密方法[19]。

3 微体系结构所面对的安全威胁

3.1 安全威胁的根源

为了提高指令的执行速度, CPU 指令流水线在分支预测、推测执行等指令执行优化单元中加入了 BTB 以及 PHT 等缓存结构, 这些缓存结构极大的提升了 CPU 的指令执行效率, 并且为了消除 CPU 指令执行速度和 CPU 访存速度的差异, 微体系结构中还加入了诸如 TLB 以及 Cache 等缓存部件。但是由于这些优化执行单元以及缓存等部件是同一个物理 CPU 上运行的所有进程(包括高权限进程)所共享的, 前一个进程在这些共享部件中产生的数据, 会影响到之后进程的执行并且低权限进程可以将高权限进程中的数据加载到 Cache 中, 因此这些共享部件在微体系结构层面打破了进程之间基于地址以及权限隔离实现的安全边界, 进而打破了计算机系统得机密性和完整性。

3.2 传统侧信道攻击

侧信道[20](side channel)攻击是利用计算机中的时间、功率消耗、电磁辐射作为依据来获取计算机中的重要机密信息的攻击方法。对于处理器来说, 最重要的侧信道是其中存在的共享缓存。

虽然运行在同一个核心上的多个进程共享诸如 Cache、TLB 等缓存结构, 但是进程无法直接获取缓存中的数据, 因此需要通过 Flush-Reload[20-25]、Prime-Probe[26-27]等基于时间的侧信道方式[28]来推测缓存中的数据, 本部分将对上述针对微体系结构的侧信道攻击方式分别进行讨论。

3.2.1 Flush-Reload

Flush-Reload[29]利用不同进程之间 L3 Cache 的共享, 如果进程 A 先访问一个数据, CPU 会将该数据加载到 L3 Cache 中, 之后进程 B 再访问相同数据的时候, 会从 L3 Cache 中读取, 而不会再进行高延迟的内存读取操作, 进而减小内存操作的时间消耗。而安全研究人员发现通过感知内存读取和 L3 Cache 读取的时间差异, 可以对特定程序中的敏感数据进行泄露, 例如加密过程中的密文以及公私钥等, 到目前为止, 安全研究人员已经使用 Flush-Reload 侧信道方式对 GnuPG[21]以及 OpenSSL [30]等程序完成了攻击。

在 Flush-Reload 攻击过程中, 攻击者先将要监控的内存块从 CPU Cache 中驱逐出去(Flush), 然后对目标程序进行访存操作, 将要泄露的数据加载到 CPU Cache 中。随后攻击者重新加载监控的内存块并测量读取时间(Reload), 如果该内存块被目标程序访问过, 其对应的内存内容会被导入到处理器缓存中, 此时攻击者对该内存的访问时间将会缩短。通过测量访存的时间差异, 攻击者可以知道特定的内存块是否被目标程序读取过, 从而推测出目标程序进程内的数据, 完成所需信息的泄露, 具体攻击过程如图 3 所示。

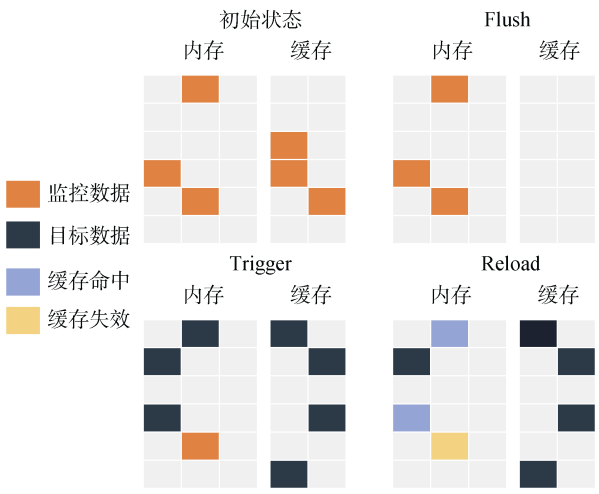


图 3 Flush-Reload 攻击过程

Figure 3 Flush-Reload attack process

Flush 阶段: 攻击者通过执行 cflush 指令等方式将 Cache 中原有内容清空

Trigger 阶段: 运行目标程序, 将目标程序所访问数据填充进 Cache

Reload 阶段: 访问所监控内存, 根据访问时间差异推测目标程序所访问数据。

3.2.2 Prime-Probes

在 Prime-Probe[31]攻击中, 首先要用特定的数据集完成 Cache 填充, 然后目标程序进行访存操作, 将目标数据加载进 Cache 中, 最后攻击者重新加载用于填充 Cache 的特定数据集, 访问被目标数据覆盖的特定数据的时间较长, 因此可以利用访存时间的差异确定目标数据, 具体攻击过程如图 4 所示。

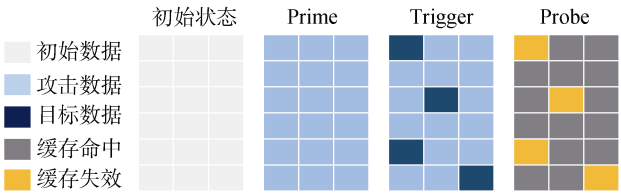


图 4 Prime-Probe 攻击过程

Figure 4 Prime-Probe attack process

Prime 阶段: 攻击者通过大量访存操作, 使用特定数据集完成 Cache 填充;

Trigger 阶段: 目标程序进行访存操作, 将目标数据填充进 Cache;

Probe 阶段: 攻击者访问用于填充 Cache 的特定数据集, 根据时间差异确定目标数据。

3.3 新型攻击技术

如前文所述, 为了提高程序运行的效率, CPU 指令流水线采用了较多的优化设计, 如针对指令执行采用的乱序执行、分支预测、推测执行等技术, 这些优化技术在提高程序运行效率的同时也引入了较多的安全漏洞, 例如分支预测、推测执行技术引入的 Spectre[4] 以及 SpectreRSB[32], 乱序执行引入的 Meltdown[5]、本部分将对上述优化设计所引入的安全漏洞分别进行讨论。

3.3.1 分支预测单元引入的漏洞

(1) Spectre

Spectre 是由基于 BTB 以及 PHT 等缓存结构实现的分支预测、推测执行等优化技术引入的微体系结构安全漏洞, 该漏洞是由于 BTB 以及 PHT 等缓存结构被运行于同一个物理 CPU 上的所有进程所共享导致的, 这些共享的缓存部件在微体系结构层面打破了以进程隔离为基础的安全边界。利用 Spectre 攻击, 可以使得处理器推测执行正常程序不会执行到的指令序列[33]。

程序执行过程中存在着大量的分支指令, CPU 执行到高延迟的分支指令时, 会通过分支预测单元预测分支判断的结果, 然后推测执行单元根据分支预测单元的预测结果, 跳转到指定分支进行推测执行。如果分支预测结果不正确, 则不将推测执行的结果提交到寄存器或者内存中, 但是不会清空推测执行过程残留在 Cache 内的数据, 根据 3.2 节所述, Cache 中的数据可以通过侧信道的方式进行读取。Spectre 漏洞利用推测执行过程会在 Cache 中残留数据这个特点, 先训练分支预测单元, 控制其分支预测结果, 进而让推测执行单元推测执行可以进行越界访问得分支指令, 进而将越界访问数据存入 Cache 中, 之后通过侧信道的方式从 Cache 中泄露越界访问数据。其具体指令模式如下:

```
if (x < array1_size) {
y = array2[array1[x] * 4096];
}
```

在上述代码中, 程序为了避免数组的越界访问, 使用了条件判断语句, 只有当数组索引小于数组元素个数的时候才进行数组的访问。此时如果变量 x 中的值是攻击者可控的数据, 攻击者可以在攻击的初始阶段先完成一定次数的非越界访问, 使得分支预测单元预测该分支指令的执行结果为真。之后, 攻击者提供一个大于数组元素个数的变量 x 值, 此时当 CPU 执行到该条件判断语句的时候, 分支预测单元预测分支结果为真, 推测执行单元会使用可以进行越界访问的 x 推测执行

$$y=array2[array1[x] * 4096];$$

语句。此时, 越界访问数据会被加载到 CPU Cache 中, 并且该数据是依赖于 $array1[x]$的, 之后 CPU 会发现分支预测单元产生了错误的预测, 进而抛弃推测执行产生的结果, 但是 CPU 并不会清除已经加载到 CPU Cache 中 *array2* 的内容, 因此攻击者可以在之后的指令中通过 Flush+Reload 等 Cache 侧信道方式完成泄露。

(2) SpectreRSB

SpectreRSB 通过返回栈缓存(Return Stack Buffer, RSB)结构利用 RET 指令推测执行可以完成越界访问的程序片段, 进而实现敏感信息泄露, 并且 SpectreRSB 可以绕过厂商发布的针对 Spectre 的防护措施[16]。

RSB 是用于预测返回指令(RET)返回地址的处理器结构, 当 CPU 执行到 CALL 指令时, 会将 CALL 指令的下一条指令的地址压入到 RSB 中, 之后执行到返回指令的时候就会从 RSB 中弹出最顶层的地址作为 RET 指令的预测结果。

RSB 有如下三个特点:

- CPU 执行到 RET 指令时, 会弹出 RSB 栈顶地址作为预测结果。
- 推测执行过程中 CPU 执行到 CALL 指令时, 会将 CALL 指令下一条指令地址放入 RSB 中, 且推测执行失败不会清空 RSB 中残留的数据。
- 进行进程切换时, RSB 中的内容不会被清空。

针对 RSB 的三个特点, 攻击者可以利用如下三种方式实现敏感信息泄露:

- **直接污染 RSB 中的返回地址**: 使用 pop 或者 jmp 来代替 ret 指令, 这样当函数返回的时候 RSB 中的返回地址并没有被弹出, 当执行到下一个返回指令的时候就会使用错误的地址进行预测, 从而达到分支注入的效果。
- **利用推测执行污染 RSB**: 在推测执行过程中, 如果遇到 call 指令, CPU 会将 call 指令的下一条指令地址压入到 RSB 中, 但是当推测执行失败的时候, CPU 不会清空 RSB 中刚才压入的返回地址, 因此攻击者可以利用推测执行来污染 RSB。
- **跨进程污染 RSB**: 由于在进行进程切换的时候 CPU 不会清空 RSB 中内容, 因此切换后的进程会使用之前进程产生的 RSB 返回地址, 攻击者可以利用这个特点实现跨进程 RSB 污染。

以图 5 所示的具体攻击样本为例:

```
1.  Function gadget()
2.  {
3.        push %rbp
4.        mov %rsp, %rbp
5.        pop %rdi
6.        pop %rdi
7.        pop %rdi
8.        nop
9.        pop %rbp
10.       clflush (%rsp)
11.       cupid
12.       retq
13. }
14.  Function speculative(char *secret_ptr)
15. {
16.       gadget();
17.       secret = *secret_ptr;
18.       temp &= Array[secret * 256];
19. }
20. Function main()
21. {
22.       speculative(secret_address)
23.       for(I = 1 to 256)
24.       {
25.           t1 = rdtscp();
26.           junk = Array[i * 256];
27.           t2 = rdtscp();
28.        }
29. }
```

图 5 ReturnRSB 攻击代码示例

Figure 5 The example of ReturnRSB

在代码片段中, main 函数会调用 speculative 函数,

CPU 会将 speculative 函数返回地址压入到 RSB 中, 随后 speculative 函数调用 gadget 函数, CPU 会将 gadget 函数返回地址压入到 RSB 中, 当程序执行到 12 行时, RSB 的状态及其返回栈状态如表 1 所示:

表 1　RSB 状态及其返回栈状态表

Table 1　RSB status and its return stack status table

RSB 状态	返回栈状态
gadget 返回地址	
Speculative 返回地址	Speculative 返回地址

此时, CPU 会使用 gadget 的返回地址作为预测结果, 从而推测执行 17 行的越界访问指令, 之后 CPU 发现预测结果错误, 会回退执行结果, 返回到正确的第 23 行继续执行, 但是没有清空 Cache 等共享部件, 因此攻击者可以通过侧信道的方式读取 Cache 中越界访问执行的执行结果, 从而完成信息泄露。

(3) BranchScope

为了保护程序的机密性以及完整性, 安全研究人员设计并实现了多种 TEE 技术[34-36]。SGX[37]是 Intel 基于 CPU 实现的 TEE 硬件防御技术, 该技术以 CPU 安全扩展为基础, 为每一个运行在 SGX 中的程序提供了一个安全的运行环境(Enclave)。SGX 的信任基只包含 CPU 以及 Enclave, 因此 SGX 可以防御来自操作系统、hypervisor、BIOS 以及 SMM 等特权攻击。

Enclave 有硬件预留的安全运行环境页缓存(Enclave Page Cache, EPC)内存, 任何非安全运行环境(Non-Enclave)对 EPC 的访问都会被 CPU 阻止。但是, 由于 SGX 的信任基是 CPU, 该防御技术无法防护来自微体系结构的攻击, 攻击者可以通过侧信道等攻击方式获取 Enclave 程序的数据以及代码。但是, 由于 SGX 的信任基是 CPU, 攻击者可以通过微体系结构漏洞泄露 SGX 中运行程序的数据, Dmitry Evtyushkin 等人提出的 BranchScope[38]以及 Guoxing Chen 等人提出的 SgxPectre[39]就是利用微体系结构漏洞实现了 SGX 程序的信息泄露。本部分将对上述两种绕过方法分别进行讨论。

如 2.1.3 节所述, 当 CPU 执行到分支指令的时候会使用分支预测单元产生一个分支预测结果。分支预测单元有两种预测模式, 一种是由程序计数器直接索引的 1-level 分支预测[40], 另一种是程序计数器结合最近执行过的分支结果对当前分支进行预测的 gshare-style 2-level 分支预测[41]。当前大多数的 CPU 分支预测器都同时采用了这两种预测方式, 具体如图 6 所示。

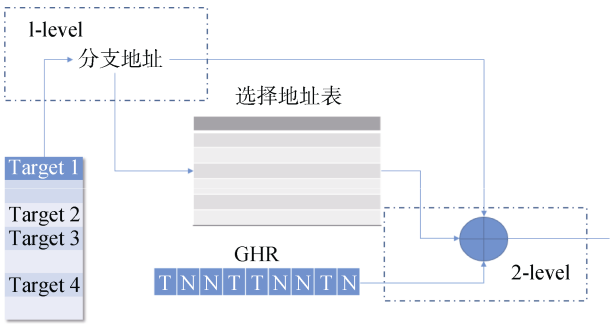


图 6　分支预测单元示例

Figure 6　The example of branch prediction

1-level 分支预测使用一个称为样式历史表(Pattern History Table, PHT)的结构来存储分支预测器之前的预测结果, 分支预测器最终会根据 PHT 中的预测记录来产生当前的预测结果。

在 Gshare-Style 2-level 分支预测模式中, 选择因子表(Selector Table)使用程序计数器作为索引, 并基于分支预测单元之前的分支预测结果为当前分支产生一个更优的结果。同时 Gshare-Style 还使用了全局历史寄存器(Global History Register, GHR)记录最近执行过的分支结果, Gshare-Style 结合程序计数器、Selector Table 以及 Global History Register 这三者的结果产生一个 PHT 的索引, 从而通过 PHT 得到分支预测结果。

PHT 中记录的是之前的分支预测结果, 并且 CPU 在进行进程切换的时候不会清空 PHT 中内容, 因此可以在攻击进程中利用 Prime-Probe 侧信道方式通过 PHT 来泄露 SGX 中程序的执行分支信息, 具体的攻击步骤如下:

Prime 阶段, 攻击者通过执行一系列特殊的分支指令使得 PHT 中的各项都处于一个攻击者已知的特定状态, 并且让分支预测器只使用 1-level 的 PHT 进行分支预测。在该阶段中, 此论文通过实验发现有两种情况分支预测单元会只使用 1-level 的分支预测模块: 一种是当第一次遇到一个分支指令的时候, 分支预测单元会只使用 1-level 的分支预测模块, 另一种是当 2-level 分支预测模块需要花费较长时间完成分支预测工作的时候, 分支预测单元会只使用 1-level 的分支预测模块。为了实现只使用 1-level 的分支预测模块, 作者使用了大量相互独立的分支指令, 并且为了让这些分支指令的地址可以覆盖到 PHT 中大多数项, 各个分支指令之间都插入了随机数量的 NOP 指令, 最后通过实验发现这种方式既可以避免分支预测单元使用 2-level 模块进行分支预测,

还可以将 PHT 中各项设置成特定的状态。

Target 阶段, 攻击者运行被攻击程序, 被攻击程序运行过程中会修改 PHT 状态。

Probe 阶段, 攻击者运行特定分支指令, 该分支指令所使用到的 PHT 项需要和被攻击者运行过程中修改的项一致。由于我们可以控制分支预测单元只使用 1-level 的分支预测模块对分支指令进行预测, 此时程序计数器就是 PHT 的直接索引, 因此只要使用与被攻击程序同样的虚拟地址就可以构造出满足条件的分支指令。之后攻击者可以通过侧信道的方式观察 PHT 中该项的变化情况, 从而推测被攻击程序的执行状态, 进而完成信息泄露。

(4) SgxSpectre

如 2.1.3 节所述, 为了加速分支预测的速度, 分支预测单元还采用了 BTB 的缓存结构, Guoxing Chen 等人提出的 SgxSpectre 的攻击方法就是利用 BTB 实现了跨进程的分支注入。

当 CPU 执行间接跳转、函数调用或者条件跳转等指令时, 该跳转的起始地址和目标地址将会被暂存在 BTB 中。这样在下一次相同的跳转或者调用被执行时, CPU 会从 BTB 中查询到相应的目标地址, 进而直接跳转到目标地址处进程投机执行。但是在进程切换过程中, CPU 并不会清空前一个进程产生的 BTB 信息。因此, SgxSpectre 通过 SGX 之外的进程实现 BTB 缓存污染, 在 BTB 中填充了特定的跳转项, 之后进程切换到 SGX 程序的时候, 会使用已经污染的 BTB 项作为分支预测的结果, 进而实现指定分支注入, 具体攻击方式如图 7 所示。

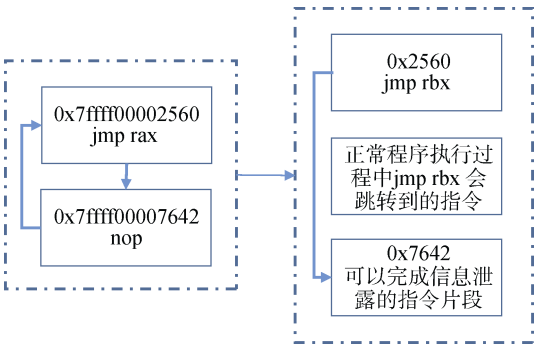


图 7 SgxSpectre 攻击示例

Figure 7 The example of SgxSpectre

为了节省空间, 许多英特尔处理器(如 Skylake)使用虚拟地址的低 32 位作为 BTB 条目的索引。因此如果需要完成的分支注入目标是 0x02560 到 0x07642 的跳转, 则可以通过 mmap 等方式申请一个 4GB 的内存空间, 然后在 0x7ffff00002560 处执行一个跳转到 0x7ffff00007642 的跳转指令, 此时 BTB 中会被填入 0x02560 到 0x07642 的跳转映射。之后, 当 SGX 程序进行 0x02560 地址跳转的时候, 就会跳转到 0x07642 这个特定的分支进行推测执行, 进而实现了 SGX 程序的分支注入。

3.3.2 乱序执行单元引入的漏洞

Meltdown 是由异常指令的乱序执行引入的微体系结构安全漏洞, 该漏洞的产生是因为, 异常指令可以在乱序执行过程中, 将程序本身访问不到的数据(“非法”数据)加载到 Cache 中, 并且之后的指令可以使用这些“非法”数据进行接下来的计算, 在异常指令提交, 异常产生之后, 攻击者在异常处理过程中通过 Cache 侧信道的方式恢复“非法”数据。利用 Meltdown 漏洞, 攻击者可以在非授权状态下获取其他进程或者云虚拟机中的敏感信息[5]。

程序执行过程中存在着大量高延迟指令, 当 CPU 执行单元执行到诸如访存指令等高延迟指令时, CPU 不会等待当前指令执行完毕后再执行后面的指令, 而会乱序执行当前指令后面的指令, 以提高 CPU 的运行效率。在乱序执行过程中, 异常只在该指令提交的时候才会产生。也就是说, 如果执行的指令发生异常, 异常指令之后的指令仍会使用异常指令的结果继续执行, 直到异常指令提交。当出现异常时, CPU 不会提交异常指令之后的所有指令的执行结果, 同时也不会清除乱序执行过程中残留在 CPU cache 中的信息, 因此攻击者可以利用侧信道的方式从 Cache 中获取敏感信息。

当代 CPU 采用基于页表的虚拟地址空间机制, 当进程需要访问内存时, CPU 中的内存管理单元(Memory Manage Unit)会通过页表查询将虚拟地址转换成物理地址, 并检查进程所具有的权限是否满足其读取需求, 如果满足则 CPU 使用物理地址从主存中取出所需数据, 并返回给流水线执行引擎。为了提高用户态到内核态的切换速度, 计算机系统将整个内核空间的页映射到每一个用户进程中, 因此当用户进程在乱序执行过程中尝试访问内核进程数据的时候, 该指令虽然会被标记为异常。然而, 乱序执行单元依旧可以访问到相应的内核数据。虽然该数据不会被提交到寄存器或者主存中, 但是仍会被放入 Cache 中, Meltdown 就是利用乱序执行过程中, 内存访问到异常时所产生这个时间窗口实现在用户进程中对内核数据的泄露。其具体过程如图 8 所示。

在此代码片段, 当执行到第 4 行访存指令的时候, CPU 为了充分利用流水线中空闲的执行资源, 在等待第 4 行指令执行完毕的过程中, 乱序执行单元

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

图 8 Meltdown 攻击代码示例
Figure 8 The example of Meltdown

会优先解码执行之后的指令。因此, 由于乱序执行的存在, 当第 4 行指令执行的时候, CPU 已经执行了之后的指令, 并且这些指令相关的微码已经被存储在保留站中, 当内核数据被加载到内存数据总线的时候, 这些依赖于该数据的微码就会继续执行, 当这些微码执行完毕, 会有序提交, 其运算结果也会被提交到寄存器或者内存中, 并且在提交过程中指令执行过程中产生的异常会被处理。因此当第 4 条加载内核数据的 MOV 指令提交时, CPU 会产生异常, 此时流水线会丢弃所有该指令之后的乱序执行结果, 但是不会清除 Cache 中残留的执行结果。如果在第 4 条指令执行到异常提交这个短暂的时间窗口内之后的指令正确执行, 那么攻击者就可以在之后的异常处理过程中通过侧信道的方式从 Cache 中获取内核数据, 完成信息泄露。

Meltdown 漏洞的产生原因是异常指令执行到指令提交、异常产生这个时间窗口内, CPU 可以获取到“非法数据”, 并使用“非法数据”继续乱序执行接下来的指令, 在异常指令提交的时候, CPU 会抛弃乱序执行结果, 但是不会清空乱序执行过程对 Cache 等微体系结构产生的影响, 攻击者可以在之后的异常处理过程中通过侧信道的方式恢复 Cache 中的数据。

上述攻击方法阐述了 Meltdown“熔断”用户和内核之间的安全边界的过程, 除此之外, 还可以“熔断”换页保护[42]、特殊寄存器保护[43]以及浮点寄存器保护[44]等安全边界。

4 微体系结构的安全防御技术

4.1 基于软件的防御技术

针对目前披露的微体系结构漏洞, 微软、谷歌、Intel 以及 AMD 等厂商都推出了相应的软件防御措施。

由于微体系结构漏洞利用需要攻击者能够通过脚本有效的感知指令执行时间的差异, 因此为了防止攻击者通过浏览器实施远程微体系结构漏洞利用, 微软、谷歌以及火狐等浏览器厂商降低了 JavaScript 脚本时间器的时间测量精度[45-47], 使得攻击者无法直接通过 JavaScript 计时器来完成侧信道信息泄露。随后, 谷歌又通过禁止 SharedArrayBuffer[45]以及 site-isolation[48]的方式进一步确保攻击者无法进行跨进程的信息泄露。

针对 Meltdown 漏洞, 采用了 KAISER[49-50]的系统可以有效的阻止攻击者通过 Meltdown 从用户空间泄露内核空间数据, 对于其他 Meltdown 变种[51], 微软也通过更新相应的微码进行了修复[52]。

Intel 和 AMD 利用 lfence 指令实现了分支指令的串行化执行[53], 避免使用分支预测单元进行分支预测, 这种方式虽然可以有效的防御 Spectre 类微体系结构漏洞, 但是也极大的降低了程序的执行效率。为了减小对程序执行效率的影响, 安全研究人员设计了 Retpoline[54]以及 SpectreCFI[55]等防御技术用于防御 Spectre 等微体系结构攻击, 本部分将对上述防御方式分别进行讨论。

4.1.1 Retpoline

谷歌提出了一种名为 retpoline 的技术, retpoline 使用返回指令替换间接跳转指令, 然后使用 RSB 将分支预测结果导向死循环代码, 而真正的返回地址被压入栈中, 当 ret 指令提交的时候就会返回到正确的执行流继续执行, 具体代码模式如表 2 所示。

表 2 Retpoline 间接指令替换保护模式
Table 2 The code pattern of Retpoline

间接跳转指令	retpoline 指令
jmp r11	call set_up_target; capture_spec: pause; jmp capture_spec; set_up_target: mov rsp, r11; ret;

Call set_up_target 语句会将 pause 语句地址压入 RSB 中, 然后执行 mov 指令, 同时执行 ret 指令的分支预测, 而 ret 指令的分支预测是通过 RSB 来实现的, 而此时的 RSB 中对应 Call set_up_target 的入口(entry)是刚才压入的 RSB 的 pause 指令, 因此推测执行单元会执行 capture_spec 循环, 直到 set_up_target 分支真正执行。

4.1.2 SpectreCFI

控制流完整性(Control Flow Integrity CFI)[56-57]用于防御间接跳转地址覆盖、函数指针地址覆盖以

及返回地址覆盖等基于控制流的攻击。CFI 要求程序执行路径必须和预先计算得到的程序控制流图中的路径一致[56]。因此 CFI 会为程序生成控制流图, 并为控制流图中的每一个节点分配唯一的标签(Label), 之后通过标签比对来决定跳转是否正确, 并且针对 ret 指令, CFI 维护了一个名为影子调用栈(Shadow Call Stack, SCS)的内存结构, 当 ret 指令执行的时候 CFI 会对比 SCS 中的返回地址和实际堆栈中的返回地址, 如果一致则正确返回, 如果不一致则抛出 CFI 异常。

在 SpectreCFI 中, 研究者针对 SpectreBTB 设计了如图 9 所示的 CFI 防御方法。

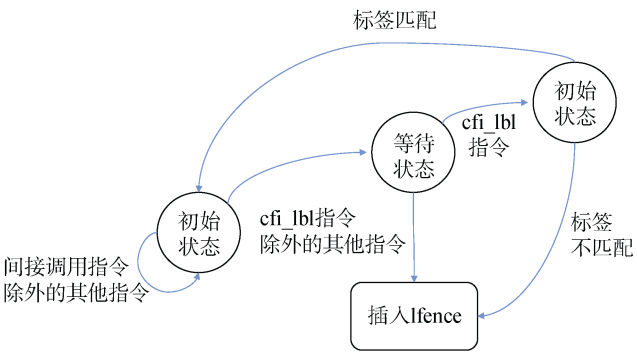


图 9 SpectreCFI 状态机

Figure 9 The state machine of SpectreCFI

在每一个 call/jmp 指令之后加入 cfi_lbl 指令, 然后对比分支预测单元预测的结果分支的 CFI Label 和 cfi_lbl 指令的 CFI Label, 如果两者一致则继续推测执行, 如果不一致或者 call/jmp 指令的下一条指令不是 cfi_lbl 指令, 则在 call/jmp 指令之后加入 lfence 指令, 进而阻止可能发生的分支注入。

针对 SpectreRSB, 研究者设计了名为 RSB/SCS 的结构, 当遇到 call 指令的时候, CPU 会将 call 指令的下一条指令的地址加入 RSB/SCS 结构中, 当解码到 ret 指令的时候, 会从 RSB/SCS 中弹出顶端的地址作为分支预测的结果, 最后当 ret 指令提交的时候, SpectreCFI 会对比分支预测的结果与传统堆栈中的返回地址, 如果一致则提交, 如果不一致则抛出一个 CFI 异常。

4.2 硬件防御技术

如第 3 章所述, Spectre、SgxSpectre、BranchScope、SpectreRSB 等攻击方式都是利用分支预测单元实现的分支注入, 因此为了抵御 Spectre 等漏洞攻击, 安全研究人员设计了诸如 The Indirect Branch Predictor Barrier (IBPB)[58]、Single Thread Indirect Branch Prediction (STIBP)[59-60]、Indirect Branch Restricted Speculation(IBRS)[61-62]、SafeSpec[63]以及 InvisiSpec[64]等防御方式, 本部分将对上述防御方式分别进行讨论并给出防御效果对比。

4.2.1 Intel/AMD 防御

Intel 和 AMD 所采用的硬件防御技术如下:

IBPB: 在 IBPB 模式下, CPU 会保证前一个进程产生的分支预测结果不会影响到之后进程的分支预测, 该防护措施针对的指令有: 间接跳转(jmp)、间接函数调用(call)以及返回(ret)指令。

STIBP: 在 STIBP 模式下, CPU 会阻止两个线程(sibling threads)共享分支预测器, 进而避免攻击者跨线程执行分支注入。

IBRS: 在 IBRS 模式下, 低权限进程产生的分支预测结果不会影响高权限进程的分支预测结果, 因此 IBRS 实现了不同权限级别之间的分支预测隔离, 但是 IBRS 不能实现同一个权限级别的进程的分支预测隔离。

4.2.2 SafeSpec

Spectre 类攻击利用推测执行的回退过程会在诸如 Cache 以及 TLB 等缓存结构中残留数据的特点, 通过侧信道的方式完成残留数据的泄漏。因此 KhaledN.Khasawneh 等人设计了名为 SafeSpec 的模型, 该模型将分支预测和推测执行过程中产生的 Cache 以及 TLB 等缓存数据存储在一个影子结构中, 当分支预测失败的时候, 该影子结构中所有的缓存数据将被清空, 通过这种方式可以在不关闭分支预测以及推测执行的情况下防止 Spectre 类攻击的发生, 具体的实现过程如图 10 所示。

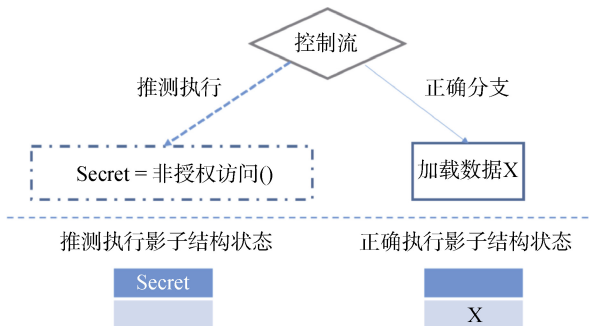


图 10 SafeSpec 影子结构转换图

Figure 10 The speculative shadow structure of SafeSpec

遇到分支指令(上图中的 Control Flow)时, CPU 会进行推测执行, 在推测执行过程中, CPU 会将访存过程中访问到的数据放入推测执行影子结构(Speculative Shadow Structure)中而不是 CPU Cache

中, 推测执行失败状态回退时, 推测执行影子结构中存储的访存结果将被清空, 并存入正确的分支执行结果, 之后正确的分支执行结果将被提交到 Cache 以及内存或者寄存器中。虽然 SafeSpec 可以有效的阻止攻击者通过 Cache 以及 TLB 侧信道的方式进行信息泄露, 但是无法针对类似 DRAM[65]缓存等共享部件进行防护。

4.2.3 InvisiSpec

针对分支预测与推测执行过程中产生的 Cache 侧信道泄露攻击, Esmaeil Mohammadian Koruyeh 等人设计了 InvisiSpec。该方法使用名为预测缓冲区(Speculative Buffer, SB)的结构存储推测执行过程中访问到的数据, 而不会将被访问数据直接加载到 Cache 中。如果分支预测失败, SB 中的内容会被清除, 如果分支预测成功, 则会将推测执行过程中访问到的数据加载到 Cache 中。为了保证数据一致性, InvisiSpec 会比较 SB 中的内容与 Cache 中该内容对应的 Cache line 中最新的值, 如果不一致则说明推测执行过程中投机读取的值已经被其他核所修改, 此时 InvisiSpec 会回退所有推测执行结果, 具体过程如图 11 所示。

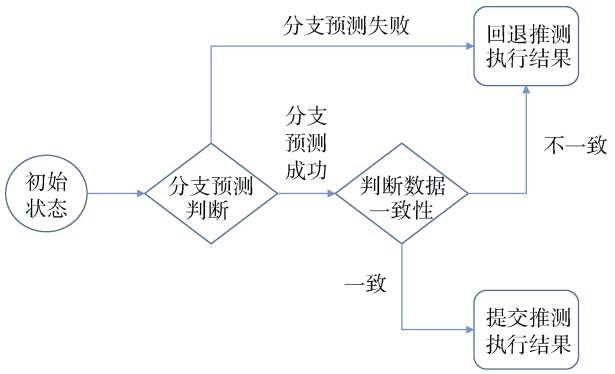


图 11 InvisiSpec 流程图

Figure 11 The workflow of InvisiSpec

在 InvisiSpec 中, 当遇到分支指令, CPU 会根据分支预测单元的预测结果进行推测执行, 在这个过程中 CPU 不会将访问到的数据加载到 Cache 中, 而会在分支预测成功, 且 SB 中内容和 Cache 内容一致的情况下提交推测执行结果到内存、寄存器以及 Cache 中, 因此 InvisiSpec 在推测执行回退的过程中不会在 Cache 中残留数据, 使得攻击者无法通过侧信道的方式完成信息泄露。

4.3 防御效果

在本节中, 我们从防御类别角度对上述软件以及硬件防护措施进行对比分析。

Spectre 是由于 CPU 在推测执行过程中执行了本不应该执行的指令序列, 这些指令序列的执行会对 Cache、TLB 等缓存结构中的数据产生影响, 攻击者可以通过测信道的方式从 Cache 等缓存结构中完成数据泄露, 因此 SafeSpec, InvisiSpec、IBRS、STIBP 以及 IBPB 等为了消除某种测信道泄露途径而设计的防护措施只能针对特定的 Spectre 类漏洞有效, 并不能有效的防护所有分支预测单元引入的漏洞, 而 SpectreCFI 这种通过检查程序真实执行路径和预先计算得到的程序控制流图中的路径一致的方法进行防御的防御措施可以对 Spectre 类攻击实现有效的防御。

Meltdown 是由于乱序执行过程中, 异常指令产生的异常只有在异常指令提交的时候才会被提交, 而在异常产生到异常提交这个时间窗口异常指令可以将程序本身不能访问到的数据加载到 cache 中, 而目前已有的 Meltdown 变种都是在乱序执行过程中通过权限控制等特定异常访问到程序本身访问不到的数据, 由于不同异常索要保护的对象不同, 因此目前已有的 Meltdown 防护都是 CPU 厂商或者操作系统厂商针对不同的变种开发的特定的防护措施。

5 讨论与展望

当代计算机系统支持多进程并行运行的模式,

表 3 防御技术有效性分析

Table 3 Defensive technology effectiveness analysis

漏洞引入单元	攻击方法	IBRS	STIBP	IBPB	SafeSpec	InvisiSpec	Retpoline	SpectreCFI	KAISER	取消浮点寄存器延迟存储
分支预测	Spectre	√	√		√	√	√	√		
	SpectreRSB			√				√		
	BranchScope							√		
	SGXSpectre	√	√		√	√		√		
乱序执行	权限检查熔断								√	
	浮点寄存器保护熔断									√[66]

在该模式下, 多个进程可能会共享同一个物理核,因此为了在共享计算机硬件的前提下实现计算机之中进程的分隔, 当代 CPU 采用了内存隔离机制, 内存隔离机制的实现需要操作系统以及微体系结构协同实现, 无论哪一个环节出现漏洞都有可能会破坏整个安全隔离。但是应用于 CPU 指令流水线中的优化措施以及各种共享的硬件缓存部件的设计之初是为了提高计算机系统的运行效率的, 在一定程度上忽略了对于安全的考虑。因此到目前为止在 CPU 指令流水线中出现的漏洞大都出现于各个优化单元。攻击者可以利用这些漏洞完成信息泄露的一个共同原因是 Cache 等共享缓存部件中的信息在指令回退以及进程切换过程中是不会被清除的。这些共享缓存部件在一定程度上打破了计算机系统中多进程的安全隔离机制, 攻击者可以通过侧信道的方式, 利用这些共享部件完成跨进程的信息泄露。

到目前为止, Intel 等厂商没有公布体系结构和微体系结构的实现细节, 因此对于漏洞研究人员来说, 整个体系结构和微体系结构还是一个黑盒。到目前为止, 尚未有系统化的漏洞挖掘方法以及辅助硬件逆向的工具, 这使得体系结构和微体系结构的漏洞研究人员面临着极大的挑战, 如何对体系结构和微体系结构实施有效的逆向分析工作, 以及如何对微体系结构进行系统化的漏洞挖掘都是值得深入探索的问题。

对于微体系结构的漏洞研究人员来说, 实现相关微体系结构逆向工具、对各个优化执行单元进行逆向并充分理解其执行过程、以及在微体系结构层面设计并实现有效且实用的漏洞防护措施是未来的主要研究方向, 具体如下:

(1) 在处理器微体系结构设计中, 性能和安全需求不可兼得。而现有的设计模式均是采取重性能、轻安全的设计思路, 更加导致了微体系结构漏洞在处理器中普遍存在的现象。因此, 在处理器微体系结构设计过程中, 有必要加入对应用场景的安全性需求的考虑。在安全要求较高的应用场景中, 可以通过进程切换过程中清空 CPU 指令流水线中可能会导致信息泄露的高速缓存, 或者在 BTB、PHT 等缓存结构中加入进程标签检查等机制在微体系结构层面实现隔离机制。同时, 在性能需求较高的场景中, 可以减少微体系结构层面上的防护, 在操作系统以及相关软件中加入类似 Retpoline 的防护技术, 进而在保证系统运行效率的基础上, 保证系统的相对安全。

(2) 微体系结构漏洞带来的威胁由于无法直接为现有的处理器进行硬件修复, 为了缓解处理器微体系结构漏洞所带来的威胁, 需要在操作系统内核中、相关软件以及编译器中加入相应的软件防护机制, 以加强安全。针对乱序执行单元引入的楼的漏洞, 可以在操作系统中优化异常处理机制, 使得攻击者无法通过异常处理过程完成侧信道信息泄露; 针对分支预测引入的漏洞, 可以通过在结合相关软件以及编译器在可能发生分支注入的地方加入 lfence 等方式来防止攻击者进行分支注入。

(3) 针对各个优化执行单元在微体系结构层面所产生的影响, 通过时间等侧信道方式, 设计和实现微体系结构数据读取工具, 用于感知优化执行过程对微体系结构状态产生的影响, 实现针对优化过程的微体系结构逆向工具。

(4) 通过微体系结构逆向工具对优化执行单元进行逆向, 了解优化执行单元的执行机制, 以及优化执行过程中所采用的共享缓存部件。对优化执行单元中的共享缓存部件进行监控, 查看前一个进程的优化执行过程在共享部件中产生的数据, 是否会影响之后进程的优化执行过程, 进而观察共享部件是否会打破体系结构层面上的安全边界。

(5) 针对各个优化执行单元的执行机制, 设计并实现微体系结构防护策略, 在保证安全的前提下尽可能降低防护措施对指令执行效率的影响。

6 总结

本文介绍了微体系结构漏洞及其防御措施, 并对各个防护措施进行了对比分析, 在分析过程中发现, Cache 等共享缓存部件在微体系结构层面打破了计算机系统一些原有的隔离机制, 使得攻击者可以利用侧信道等方式完成信息泄露, 而当前的安全防护措施都会在一定程度上影响 CPU 的执行效率。因此, 在保持运行效率的基础上提升微体系结构安全性, 就成为了安全研究人员以及微体系结构设计人员所关注的热点问题。在今后的工作中, 我们将继续深入研究微体系结构中的各个优化单元以及共享部件的功能特性, 深化了解微体系结构的内在安全逻辑, 提升微体系结构的安全性。

参考文献

[1] Maurice C, Weber M, Schwarz M, et al. Hello from the other Side: SSH over Robust Cache Covert Channels in the Cloud[C]. *Proceedings 2017 Network and Distributed System Security Symposium*, 2017: 8-11.

[2] ARM LIMITED. Vulnerability of Speculative Processors to Cache Timing Side Channel Mechanism, 2018.

[3] INTEL. Intel Analysis of Speculative Execution Side Channels, July 2018. Revision 4.0.

[4] Kocher P, Horn J, Fogh A, et al. Spectre Attacks: Exploiting Speculative Execution[C]. *2019 IEEE Symposium on Security and Privacy*, 2019: 1-19.

[5] M. Lipp, M. Schwarz, D. Gruss, et al, Meltdown: Reading Kernel Memory from User Space[C]. *USENIX Security Symposium*, 2018: 973-990.

[6] Kurth M, Gras B, Andriesse D, et al. NetCAT: Practical Cache Attacks from the Network[C]. *2020 IEEE Symposium on Security and Privacy*, 2020: 20-38.

[7] John L. Hennessy and David A. Patterson, Computer Architecture A Quantitative Approach [M]. *Morgan Kaufmann publications, the 6th edition*, 2019.

[8] Tomasulo R M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units[J]. *IBM Journal of Research and Development*, 1967, 11(1): 25-33.

[9] Dynamic Branch Prediction, Oregon State University, http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/, Jun, 2019.

[10] CHENG. The schemes and performances of dynamic branch predictors [J]. *Technical report, Berkeley Wireless Research Centre*, 2000.

[11] Jimenez D A, Lin C. Dynamic Branch Prediction with Perceptrons[C]. *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001: 197-206.

[12] Teran E, Wang Z, Jiménez D A. Perceptron Learning for Reuse Prediction[C]. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016: 1-12.

[13] S. Lee, M. Shih, P. Gera, et al, Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing[C]. in *USENIX Security Symposium*, 557-574, 2017.

[14] Ge Q, Yarom Y, Cock D, et al. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware[J]. *Journal of Cryptographic Engineering*, 2018, 8(1): 1-27.

[15] Q3 2018 Speculative Execution Side Channel Update. Intel. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html 2018.

[16] Pizlo, F. What Spectre and Meltdown mean for WebKit, Jan. 2018.

[17] Dang T H Y, Maniatis P, Wagner D. The Performance Cost of Shadow Stacks and Stack Canaries[C]. *The 10th ACM Symposium on Information*, *Computer and Communications Security*, 2015: 555-566.

[18] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, et al, Code-Pointer Integrity[M]. in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 81-116, 2014.

[19] Henson M, Taylor S. Memory Encryption[J]. *ACM Computing Surveys*, 2014, 46(4): 1-26.5

[20] Gruss, D., Spreitzer, R., and Mangard, S. “Cache template attacks: Automating attacks on inclusive last-level caches” [C]. In *24th USENIX Security Symposium*, 897-912, 2015.

[21] Yarom, Y., and Falkner, K. “FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack,” [C] In *USENIX Security Symposium*, 719-732, 2014.

[22] Gruss D, Maurice C, Fogh A, et al. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR[C]. *The 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 368-379.

[23] Irazoqui G, Inci M S, Eisenbarth T, et al. Wait a Minute! a Fast, Cross-VM Attack on AES[C]. *Research in Attacks*, *Intrusions and Defenses*, 2014: 299-319.

[24] Lipp, M., Gruss, D., Spreitzer, R., et al. Armageddon: Cache attacks on mobile devices[C]. *In 25th USENIX Security Symposium*, 2016, 549-564.

[25] Schwarz M, Lipp M, Gruss D, et al. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks[C]. *Proceedings 2018 Network and Distributed System Security Symposium*, 2018: 15.

[26] Osvik, Dag Arne, Adi Shamir, et al. Cache attacks and countermeasures: the case of AES[C]. In *Cryptographers’ track at the RSA conference. Springer, Berlin, Heidelberg*, 1-20, 2006.

[27] Liu F F, Yarom Y, Ge Q, et al. Last-Level Cache Side-Channel Attacks are Practical[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 605-622.

[28] Kocher P C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems[C]. *Advances in Cryptology — CRYPTO'96*, 1996: 104-113.

[29] Yarom, Y., and Benger, N. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. In *IACR Cryptology ePrint Archive*, 2014, 140.

[30] Benger N M, Pol J, Smart N P, et al. “Ooh Aah. Just a Little Bit”: A Small Amount of Side Channel can Go a Long Way[C]. *Cryptographic Hardware and Embedded Systems – CHES 2014*, 2014: 75-92.

[31] Kiriansky V, Waldspurger C. Speculative Buffer Overflows: Attacks and Defenses[J] *arXiv preprint arXiv*:1807.03757, 2018.

[32] Koruyeh, E. M., Khasawneh, K. N., Song, C., et al. Spectre returns! speculation attacks using the return stack buffer[C]. *USENIX Workshop on Offensive Technologies*, 2018, 3.

[33] Carruth, C., https://reviews.llvm.org/D41723 Jan. 2018.

[34] Chen X X, Garfinkel T, Lewis E C, et al. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems[C]. *The 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII*, 2008: 2-13.

[35] Cheng Y Q, Ding X H, Deng R H. Efficient Virtualization-Based Application Protection Against Untrusted Operating System[C]. *The 10th ACM Symposium on Information*, *Computer and Communications Security*, 2015: 345-356.

[36] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z., and et al. Inktag: Secure applications on an untrusted operating system[C]. *ACM SIGPLAN Notices*, 265-278, 2013.

[37] INTEL. Intel Software Guard Extensions (Intel SGX), 2016.

[38] Evtyushkin D, Riley R, Abu-Ghazaleh N C A E, et al. BranchScope[J]. *ACM SIGPLAN Notices*, 2018, 53(2): 693-707.

[39] Chen G X, Chen S C, Xiao Y, et al. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution[C]. *2019 IEEE European Symposium on Security and Privacy*, 2019:

142-157.

[40] Smith J E. A Study of Branch Prediction Strategies[C]. *25 years of the international symposia on Computer architecture* (*selected papers*) - *ISCA '98*, 1998: 135-148.

[41] Yeh T Y, Patt Y N. Two-Level Adaptive Training Branch Prediction[C]. *The 24th annual international symposium on Microarchitecture - MICRO 24*, 1991: 51-61.

[42] Intel. Resources and Response to Side Channel L1 Terminal Fault, Aug. 2018.

[43] Intel. Q2 2018 Speculative Execution Side Channel Update, May 2018.

[44] Stecklina J, Prescher T. LazyFP: Leaking FPU Register State Using Microarchitectural Side-Channels[J] *arXiv preprint arXiv*:1806.07480, 2018.

[45] The Chromium Projects. Actions required to mitigate Speculative Side Channel Attack techniques, 2018.

[46] Wagner, Luke, Mitigations landing for new class of timing attack. Retrieved 1, 2018.

[47] Microsoft, Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer, 2018.

[48] Smith, B, Enable SharedArrayBuffer by default on non-android, 2018.

[49] Gruss D, Lipp M, Schwarz M, et al. KASLR is Dead: Long Live KASLR[C]. *Engineering Secure Software and Systems*, 2017: 161-176.

[50] Ionescu, A. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). https://twitter.com/aionescu/status/930412525111296000 2017

[51] Weisse, O., Van Bulck, J., Minkin, M., et al. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution, 2018.

[52] van Bulck J, Minkin M, Weisse O, et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution[C]. *The 27th USENIX Conference on Security Symposium*, 2018: 991-1008.

[53] AMD. Software Techniques for Managing Speculation on AMD Processors, Revison 7.10.18, 2018.

[54] Retpoline: a software construct for preventing branch-target- injection. Turner, Paul. https://support. google. com/faqs/answer/7625886 2018.

[55] Koruyeh E M, Haji Amin Shirazi S, Khasawneh K N, et al. SpecCFI: Mitigating Spectre Attacks Using CFI Informed Speculation[C]. *2020 IEEE Symposium on Security and Privacy*, 2020: 39-53.

[56] Abadi M, Budiu M H, Erlingsson Ú, et al. Control-Flow Integrity Principles, Implementations, and Applications[J]. *ACM Transactions on Information and System Security*, 2009, 13(1): 1-40.

[57] Future of pax. https://pax.grsecurity.net/docs/pax-future.txt, 2002.

[58] Deep Dive: Indirect Branch Predictor Barrier. Intel. https:// software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-predictor-barrier.

[59] Larabel, M. Bisected, The Unfortunate Reason Linux 4.20 Is Running Slower, 2018.

[60] AMD. AMD64 Technology indirect branch control extension, Revison 4.10.18, 2018.

[61] Tkachenkot, V. 20-30% Performance Hit from the Spectre Bug Fix on Ubuntu, 2018.

[62] Intel. Speculative Execution Side Channel Mitigations, Revision 3.0, 2018.

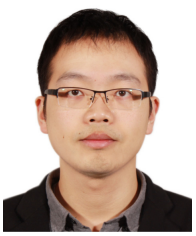
[63] Khasawneh K N, Koruyeh E M, Song C Y, et al. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation[C]. *2019 56th ACM/IEEE Design Automation Conference*, 2019: 1-6.

[64] Yan M J, Choi J, Skarlatos D, et al. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy[C]. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018: 428-441.

[65] Pessl P, Gruss D, Maurice C, et al. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks[EB/OL]. 2015: arXiv: 1511.08756[cs.CR]. https://arxiv.org/abs/1511.08756.

[66] Microsoft Guidance for Lazy FP State Restore https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/ADV180016 2018.

尹嘉伟 2017 年在吉林大学获得学士学位。现在中国科学院信息工程研究所攻读博士学位。主要研究方向为固件以及微体系结构安全。研究兴趣包括: 漏洞挖掘与利用、程序分析。Email: yinjiawei@iie.ac.cn



李孟豪 博士, 助理研究员。2017 年毕业于中国科学院信息工程研究所获得博士学位。主要研究方向为软件漏洞挖掘和安全测评、固件安全分析、软件相似性分析等。Email: limenghao@iie.ac.cn

霍玮 博士, 研究员、博士生导师, 中国科学院青年创新促进会成员。2010 年在中国科学院计算技术研究所获得博士学位。主要研究领域包括软件漏洞挖掘、利用和安全评测、基于大数据及知识图谱的软件安全分析、信息系统安全分析等。Email: huowei@iie.ac.cn