

# x86 处理器向量条件访存指令安全脆弱性分析

李丹萍 朱子元 史 岗 孟 丹

(中国科学院信息工程研究所 北京 100085)  
(中国科学院大学网络空间安全学院 北京 100049)

**摘 要** 单指令多数数据流(Single Instruction stream, Multiple Data streams, SIMD)是一种利用数据级并行提高处理器性能的技术,旨在利用多个处理器并行执行同一条指令增加数据处理的吞吐量.随着大数据、人工智能等技术的兴起,人们对数据并行化处理的需求不断提高,这使得 SIMD 技术愈发重要.为了支持 SIMD 技术,Intel 和 AMD 等 x86 处理器厂商从 1996 年开始在其处理器中陆续引入了 MMX(MultiMedia Extensions)、SSE(Streaming SIMD Extensions)、AVX(Advanced Vector eXtensions)等 SIMD 指令集扩展.通过调用 SIMD 指令,程序员能够无需理解 SIMD 技术的硬件层实现细节就方便地使用它的功能.然而,随着熔断、幽灵等处理器硬件漏洞的发现,人们逐渐认识到并行优化技术是一柄双刃剑,它在提高性能的同时也能带来安全风险.本文聚焦于 x86 SIMD 指令集扩展中的 VMASKMOV 指令,对它的安全脆弱性进行了分析.本文的主要贡献如下:(1)利用时间戳计数器等技术对 VMASKMOV 指令进行了微架构逆向工程,首次发现 VMASKMOV 指令与内存页管理和 CPU Fill Buffer 等安全风险的相关性;(2)披露了一个新的处理器漏洞 EvilMask,它广泛存在于 Intel 和 AMD 处理器上,并提出了 3 个 EvilMask 攻击原语:VMASKMOVL+Time(MAP)、VMASKMOVVS+Time(XD)和 VMASKMOVL+MDS,可用于实施去地址空间布局随机化攻击和进程数据窃取攻击;(3)给出了 2 个 EvilMask 概念验证示例(Proof-of-Concept, PoC)验证了 EvilMask 对真实世界的信息安全危害;(4)讨论了针对 EvilMask 的防御方案,指出最根本的解决方法是在硬件层面上重新实现 VMASKMOV 指令,并给出了初步的实现方案.

**关键词** 处理器安全;单指令多数数据流(SIMD);微体系结构侧信道攻击;VMASKMOV 指令;地址空间布局随机化(ASLR)

中图法分类号 TP309 DOI号 10.11897/SP.J.1016.2024.00525

## Security Vulnerability Analysis of the Vector Conditional Memory Instruction on x86 Processors

LI Dan-Ping ZHU Zi-Yuan SHI Gang MENG Dan

(Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085)

(School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049)

**Abstract** Single Instruction stream, Multiple Data streams (SIMD) is a technology that exploits data-level parallelism to improve processor performance. It aims to take advantage of multiple processors to execute the same instruction in parallel to increase data processing throughput. With the rapid rise of big data and artificial intelligence, the demand for data-parallel processing continues to increase, making SIMD technology increasingly important. To support SIMD technology, x86 processor manufacturers such as Intel and AMD have introduced SIMD instruction set extensions such as MMX (MultiMedia eXtensions), SSE (Streaming SIMD Extensions), and AVX (Advanced Vector eXtensions) in their processors since 1996. By calling SIMD instructions, programmers can easily use the SIMD feature without needing to understand the implementation details of

收稿日期:2023-05-12;在线发布日期:2023-12-27. 本课题得到中国科学院战略性先导科技专项(XDC02010400)资助. 李丹萍, 博士研究生, 助理研究员, 主要研究方向为处理器安全和微体系结构侧信道攻击. E-mail: lidanping@iie. ac. cn. 朱子元(通信作者), 博士, 正高级工程师, 主要研究领域为芯片安全和计算机体系结构. E-mail: zhuziyuan@iie. ac. cn. 史 岗, 博士, 正高级工程师, 主要研究领域为计算机体系结构和嵌入式安全. 孟 丹, 博士, 研究员, 主要研究领域为计算机体系结构、云计算及网络与系统安全.

SIMD at the hardware level. However, with the discovery of processor hardware vulnerabilities such as Meltdown and Spectre, people gradually realize that employing parallel optimization technology at the processor microarchitecture level is a double-edged sword that can bring security risks while improving performance. This paper focuses on the VMASKMOV instruction that implemented in the x86 SIMD instruction set extension and conducts an in-depth analysis of its security vulnerabilities. The main contributions of this paper are as follows: (1) the microarchitecture implementation details of the VMASKMOV instruction are studied by experiments with timestamp counters, hardware performance counters, Microarchitectural Data Sampling (MDS) techniques, and the instruction characteristics are summarized based on experimental results: ① by measuring the execution time of the VMASKMOV instruction, it is possible to determine whether the target address is mapped or not and the status of most of the page attribute flags on the page it is located on; ② even if the mask bit is zero, VMASKMOV (load) will copy all 128 or 256 bit data at the target address to a temporary storage, causing the masked data to move, and if this operation is performed on some Intel processors, the masked data will pass through the Fill Buffer and can be sampled using MDS technology; (2) based on experimental results, a new processor vulnerability named EvilMask is proposed, which is widely present on both Intel and AMD processors, then three attack primitives of EvilMask (VMASKMOVL + Time (MAP), VMASKMOVS + Time (XD), and VMASKMOVL + MDS) are presented, which can be used to implement de address space randomization attacks and process data leakage attack; (3) two Proof-of-Concept (PoC) examples are provided to demonstrate EvilMask's information security risks to the real world: ① using VMASKMOVL + Time (MAP) and VMASKMOVS + Time (XD) to break the Kernel Address Space Layout Randomization (KASLR) by de randomizing the kernel base address, physically mapped base addresses, and kernel module address successfully on Intel Core i5-6200U, Intel Core i7-6700, AMD Ryzen 7 3700X, and AMD Ryzen 5 5650 processors, and ② using VMASKMOVL + MDS to leak data in Linux kernel on an Intel Core i5-6200U processor; (4) the countermeasures for EvilMask are discussed and this paper points out that the most fundamental solution is to re implement the VMASKMOV instruction at the hardware level, then gives a preliminary implementation.

**Keywords** processor security; Single Instruction stream, Multiple Data streams (SIMD); microarchitectural side-channel attacks; VMASKMOV instruction; Address Space Layout Randomization (ASLR)

## 1 引 言

单指令流多数据流 (Single Instruction stream, Multiple Data streams, SIMD) 是一种利用数据级并行 (Data-Level Parallelism) 提高处理器性能的技术, 旨在利用多个处理器并行执行同一条指令增加数据处理的吞吐量<sup>[1]</sup>. SIMD 技术适用于需要对大量数据执行相同操作的场景, 比如多媒体声像处理. 随着大数据、人工智能等技术的兴起, 人们对数据并行化处理的需求不断提高, 这使得 SIMD 技术愈发重要. 为了支持 SIMD 技术, Intel 公司从 1996 年开始陆续引入了 MMX (MultiMedia eXtensions)、SSE

(Streaming SIMD Extensions)、AVX (Advanced Vector eXtensions) 等指令集扩展<sup>[2]</sup>, AMD 公司也从 1996 年开始逐步引入了类似的指令集扩展<sup>[3]</sup>. 通过调用这些 SIMD 指令, 程序员无需理解技术的底层硬件实现细节就能方便地使用处理器提供的 SIMD 功能.

计算机体系结构常被分为体系结构级和微体系结构级两个抽象层级: 指令集处于体系结构级, 它规定了处理器的功能行为, 微体系结构级是指令集的逻辑实现, 它规定了处理器的性能行为<sup>[4]</sup>. 为了不断提高处理器性能, 处理器设计人员常在微体系结构级引入各种并行优化技术, 如挖掘指令级并行 (Instruction-Level Parallelism) 的乱序执行和分支预

测技术、挖掘线程级并行( Thread-Level Parallelism) 的共享内存技术等。然而,随着熔断(Meltdown)<sup>[5]</sup>、幽灵(Spectre)<sup>[6]</sup>等微体系结构漏洞的发现,人们逐渐认识到在微体系结构级引入并行优化技术是一柄双刃剑,它在提高性能的同时也能带来安全风险。本文的研究工作表明,与熔断、幽灵漏洞类似,由于缺乏安全考虑,SIMD 技术也能带来安全风险。

本文对 x86 SIMD 指令集扩展中的 VMASKMOV 指令<sup>[2-3]</sup>进行了研究,在此基础上发现了一个新的处理器漏洞 EvilMask。Intel 手册<sup>[2]</sup>定义 VMASKMOV 指令为“条件 SIMD 打包加载和存储(Conditional SIMD Packed Loads and Stores)指令”,该指令会将所访问的 128 位或 256 位数据中的全部或部分打包后从或向内存中搬移,根据指令编码中的屏蔽位来决定这 128 位或 256 位中的哪些数据发生移动。本文的研究工作表明,以 Intel 和 AMD 为代表的 x86 处理器普遍受到 EvilMask 漏洞影响。

### 1.1 本文贡献和创新点

(1)首次披露了一个新的处理器漏洞 EvilMask,它利用了 x86 SIMD 指令集扩展中的 VMASKMOV 指令,可实施去地址空间布局随机化攻击和进程数据窃取攻击;提出了 3 个 EvilMask 攻击原语: VMASKMOVL+Time(MAP)、VMASKMOVS+Time(XD)和 VMASKMOVL+MDS。

(2)利用时间戳计数器等技术对向量指令进行了定量和定性的逆向工程实验,首次发现 VMASKMOV 指令与内存页管理和 CPU Fill Buffer 等安全风险的相关性。

(3)给出了 2 个 EvilMask 漏洞的概念验证示例(Proof-Of-Concept,POC),验证了 EvilMask 对真实世界的信息安全危害:①利用 VMASKMOVL+Time(MAP)和 VMASKMOVS+Time(XD)攻击原语成功攻破了 Intel 和 AMD 处理器上的 KASLR;②利用 VMASKMOVL+MDS 攻击原语成功窃取 Linux 内核中的数据。

### 1.2 本文组织结构

第 2 节提供了理解本文所需的背景知识和相关工作;第 3 节借助时间戳计数器等技术对 VMASKMOV 指令进行了逆向工程,这是得以发现 EvilMask 的基础;第 4 节介绍了 EvilMask 漏洞,提出了 3 个攻击原语;第 5 节给出了 2 个基于 EvilMask 的概念验证示例,用以证明 EvilMask 的有效性;第 6 节讨论了针对 EvilMask 的防御方案,给出了初步的硬件防御方案;第 7 节总结全文。

## 2 背景知识和相关工作

### 2.1 地址空间布局随机化

缓冲区溢出、堆溢出、整数溢出等内存漏洞<sup>[7]</sup>可用于实施控制流劫持、任意代码执行等攻击。为防御这类漏洞,计算机系统引入了 NX-bit、W $\oplus$ R、SMEP(Supervisor Mode Execution Prevention)、SMAP(Supervisor Mode Access Prevention)等防御技术。然而,return-to-libc 和 ROP(Return-Oriented Programming)等更高级的内存漏洞<sup>[7]</sup>能够通过重定位劫持的控制流绕过这些防御技术。于是, Linux PaX 组织于 2003 年提出了针对 ROP 等内存漏洞的地址空间布局随机化(Address Space Layout Randomization, ASLR)技术<sup>①</sup>。ASLR 目前已被部署在 Windows、Linux 和 Mac OS 等主流操作系统上<sup>[8]</sup>。ASLR 的主要思想是:每当一个用户态进程被加载进内存时,随机化该用户态进程的地址空间布局(用户态 ASLR);每当系统重启时,随机化内核地址空间布局(内核态 ASLR, Kernel ASLR, 简称 KASLR)。为了提高性能,大部分操作系统在实现 KASLR 时采用了粗粒度而非细粒度的随机化,即只随机化基地址,这使得破解 KASLR 成为可能。此外,研究人员发现,通过分析 KASLR 在具体操作系统上的实现方式,可以进一步增加破解 KASLR 的成功率<sup>②</sup>。此前的研究工作<sup>[9-12]</sup>表明,破解 KASLR 所需的两个关键的信息是:目标地址所在的地址区间和地址对齐单位。表 1 总结了 Linux 4.x 内核中 KASLR 的实现方式<sup>[11-12]</sup>;以内核基地址为例,它一般落在 0xffffffff81000000~0xffffffffbe000000 这个地址区间中,以 2 MB 为单位对齐,因此有 488(约等于 2<sup>9</sup>)种可能性<sup>③</sup>。

表 1 Linux 4.x 中 KASLR 的实现方式<sup>[11-12]</sup>

关键地址	地址区间	对齐单位	熵
内核基地址	0xffffffff81000000~ 0xffffffffbe000000	2 MB	9
内核模块地址	0xffffffffc0001000~ 0xffffffffc0400000	4 KB	10
物理直接映射基地址	0xffff888000000000~ 0xffff888000000000	1 GB	16

① PaX Team. Address Space Layout Randomization. <http://pax.grsecurity.net/docs/aslr.txt>, 2003, 3, 15

② Schwarz M, Canella C, Giner L, et al. Store-to-heap forwarding: Leaking data on meltdown-resistant CPUs (updated and extended version). <http://arxiv.org/abs/1905.05725>, 2019, 5, 14

③ 计算方式:(0xffffffffbe000000-0xffffffff81000000) $\div$ (2 $\times$ 1024 $\times$ 1024)=488 $\approx$ 2<sup>9</sup>,由此算得熵为 9(即 2<sup>9</sup>中的 9),熵越大,表明可能性越多,猜中难度越大。

## 2.2 处理器微体系结构攻击

早在 1996 年, Kocher<sup>[13]</sup> 就提出可以利用处理器微体系结构中的高速缓存(Cache)构建时间侧信道攻击来破解密码算法. 此后, 研究人员陆续提出 Prime+Probe<sup>[14]</sup>、Evict+Time<sup>[14]</sup>、Flush+Reload<sup>[15-16]</sup>、Evict+Reload<sup>[17]</sup>、Flush+Flush<sup>[18]</sup> 等缓存侧信道攻击. 经过多年发展, 缓存侧信道攻击经历了从只能利用第一级缓存<sup>[19-20]</sup> 到也能利用最后一级缓存<sup>[15-17]</sup>; 从只能进行核内(In-Core)攻击<sup>[20]</sup> 到也能进行跨核(Cross-Core)攻击<sup>[21-23]</sup>; 从只适用于包含式(Inclusive)缓存<sup>[15-16]</sup> 到也适用于专有(Exclusive)或非包含式(Non-Inclusive)缓存<sup>[24]</sup>; 从只适用于 x86 处理器<sup>[14-18, 22]</sup> 到也适用于其他处理器<sup>[25]</sup> 的发展过程. 除了高速缓存, 研究人员还提出了基于分支预测单元(Branch Prediction Unit, BPU)<sup>[26]</sup>、转址旁路缓冲(Translation Look-aside Buffer, TLB)<sup>[27]</sup>、执行单元端口(Execution Unit Ports)<sup>[28]</sup> 等的侧信道攻击. 近年来, 处理器微体系结构安全领域的另一个研究热点是瞬态执行攻击(Transient Execution Attacks)<sup>[29]</sup>, 它区别于传统的微体系结构侧信道攻击, 可以直接泄露数据而非元数据<sup>[30]</sup>①, 因此更加强. 瞬态执行攻击的思想是: 现代处理器在微体系结构级引入的推测执行和乱序执行等机制可以导致指令超前执行, 这使得处理器处于一种“瞬时状态”, 此后, 若超前执行的指令被证明不该被执行(如发生了越权访问), 处理器会回滚到指令执行前的状态以保证程序执行的正确性, 然而, 攻击者已经通过隐蔽通道把“瞬时状态”下的指令执行结果进行了保存和传输, 从而导致信息泄露. 2018 年初, Kocher 等人<sup>[6]</sup> 披露了第一个瞬态执行漏洞——幽灵(Spectre)(分支目标注入, Branch Target Injection, CVE-2017-5715), 它利用了微体系结构中的分支预测技术, 攻击者在对分支预测器进行训练后, 能够瞬态地控制程序的执行路径, 这允许攻击者突破软件隔离边界访问未授权的数据. 如果目标程序中存在特定的代码片段(gadget), 幽灵漏洞还可被用于任意代码执行攻击. 同在 2018 年初, Lipp 等人<sup>[5]</sup> 披露了另一个重要的瞬态执行漏洞——熔断(Meltdown)(恶意数据缓存加载, Rogue Data Cache Load, CVE-2017-5754), 作为一个系统级漏洞, 熔断漏洞不仅利用了微体系结构中的指令乱序执行技术, 还利用了操作系统中用户态和内核态空间隔离的不彻底, 这允许处于用户态的攻击者能够越权访问内核态的数据. 此后, 更多的瞬态执行漏洞被陆续披露<sup>[31-33]</sup>. 2019 年, Intel 公

司披露其处理器存在微体系结构数据采样(Microarchitectural Data Sampling, MDS)漏洞<sup>②</sup><sup>[34-38]</sup>, 这类漏洞利用了微体系结构中的各种小容量缓冲队列(如 Fill Buffer、Store Buffer、Load Port 等). 由于 Intel 处理器在实现这些缓冲队列时没有采取严格的隔离机制, 导致攻击者能够采样到存储其中的旧数据, 引发安全问题.

下面重点介绍 3 类与本文相关的处理器微体系结构攻击.

(1) MDS 攻击<sup>[34-38]</sup>. 3 个主要的 MDS 攻击 MLPDS (Microarchitectural Load Port Data Sampling, CVE-2018-12127)、MSBDS (Microarchitectural Store Buffer Data Sampling, CVE-2018-12126)、MFBDS (Microarchitectural Fill Buffer Data Sampling, CVE-2018-12130) 可以分别泄露存储在 Intel 处理器 Load Port、Store Buffer、Fill Buffer 中的数据. 本文主要关注 Fill Buffer, 如图 1 所示, 它充当了第一级数据缓存(L1D Cache)和第二级缓存(L2 Cache)之间的数据传输桥梁, 可以看作是在这两级缓存中又增加了一级小容量缓存: 当处理器访问的数据未在 L1D Cache 中找到时, 它会去查找 Fill Buffer, 若找到, 则直接返回数据, 若未找到, 则继续访问层级更高的缓存或内存, 此时返回的数据会被填充到 Fill Buffer 中, 之后再访问该数据时, 就能在 Fill Buffer 中找到. 攻击者可以通过 ZombieLoad<sup>[34]</sup> 等方法采样到 Fill Buffer 中的数据.

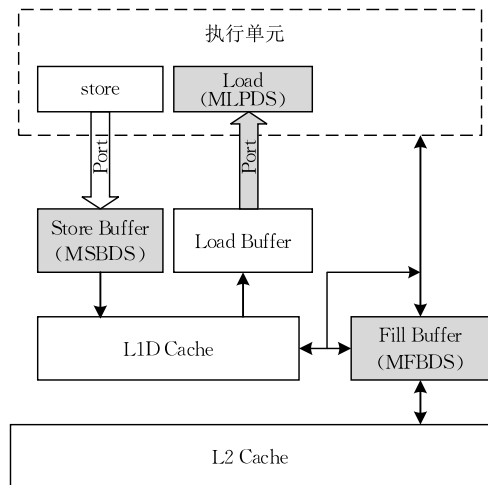


图 1 MDS 攻击和其利用的微体系结构部件<sup>[34-35]</sup>

- ① 研究人员把从侧信道提取出的直接信息称为元数据(metadata, 如访问时间), 把根据元数据推断出的间接信息称为数据(data, 如密钥).
- ② Intel. Microarchitectural Data Sampling. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html>, 2021, 3, 11

(2) Flush+Reload 缓存侧信道攻击<sup>[15-16]</sup>. 实施 Flush+Reload 攻击有 3 个前提条件: ① 攻击者和受害者进程所在的处理器核共享最后一级缓存; ② 攻击者和受害者的进程中存在共享内存(如共享库); ③ 最后一级缓存中的缓存行能够通过虚拟地址被刷新到内存中(如使用 x86 处理器中的 CLFLUSH 指令). Flush+Reload 攻击包含 3 个攻击步骤: ① 选择一个目标地址并将它所在的缓存行刷新到内存中(即“Flush”); ② 等待受害者进程执行; ③ 测量再次访问该目标地址所花费的时间(即“Reload”). 于是, 根据步骤③测得的时间, 攻击者能够推断受害者进程是否在步骤②访问过目标地址: 若花费的时间短, 则访问过; 若花费的时间长, 则未访问过.

(3) 传统的破解 ASLR 的攻击方法通常依赖软件漏洞来减少 ASLR 的随机化熵, 因此, 只要及时消除已知的软件漏洞就可以防御它们. 然而, 近年来, 研究人员发现还可以利用微体系结构侧信道攻击来破解 ASLR<sup>[9-12, 39-42]</sup>, 表 2 对此进行了总结, 这类攻击一般包含 3 个步骤: ① 对目标地址区间的各个地址遍历执行某个特定操作; ② 采集对各个地址执行该特定操作花费的时间(或功耗<sup>[42]</sup>); ③ 根据步骤②采集到的时间(或功耗<sup>[42]</sup>), 攻击者能够分析出地址空间布局, 从而破解 ASLR. 因为不依赖软件漏洞, 这类攻击通常较难防御. 当然, 它们也存在一些局限性: ① 由于依赖硬件特性, 这类攻击往往只对某种或某几种处理器有效, 比如表 2 中的攻击普遍只对 Intel 处理器有效; ② 这类攻击一般需要攻击者对涉及的微体系结构部件的实现细节有深入了解, 而出于商业目的, 这些信息一般不对外公开, 因此往往需要攻击者对所涉及部件进行逆向工程<sup>①</sup>, 比如 Jump Over ASLR 攻击<sup>[40]</sup>就需要攻击者对目标处理器的分支目标缓冲(Branch Target Buffer, BTB)有充分了解; ③ 部分攻击依赖特殊的硬件条件, 比如 Drk 攻击<sup>[9]</sup>就依赖 Intel TSX (Transactional Synchronization eXtensions) 技术. 此前的研究工作中, 与本文最相关的是 Gruss 等人提出的 Prefetch+Time<sup>[42]</sup>攻击, 该攻击利用了 x86 处理器中的 PREFETCH 指令在访问非法地址时不抛出异常并且访问已映射地址和未映射地址时花费的时间不同的特性, 可用于攻击 AMD 处理器上的 KASLR. Gruss 等人<sup>[43]</sup>对 2017 年及此前的这类攻击<sup>[9-10, 39-40]</sup>进行了总结, 认为这类攻击的主要成因是内核地址空间和用户地址空间隔离得不彻底, 并提出了 KAISER(Kernel Address Isolation to have Side channels Efficiently Removed)来加强两者的隔离, 此

后, 为了防御熔断漏洞, KAISER 被部署到 Linux 和 Windows 操作系统上<sup>②</sup>. 几乎与此同时, Gens 等人<sup>[44]</sup>提出了与 KAISER 思想接近的 LAZARUS 来防御此类漏洞, 它能防御 KAISER 无法防御的 Jump Over ASLR 攻击<sup>[40]</sup>. 随后, Canella 等人<sup>[11]</sup>指出 KAISER 和 LAZARUS 并不能完全防御这类漏洞, 提出了更具一般性的 FLARE(Fake Load Address Response)防御方法.

表 2 利用微体系结构侧信道攻击破解 ASLR

攻击	逆向工程	处理器	特殊条件
Double Page Fault <sup>[39]</sup>	无需	Intel	中断处理
Jump Over ASLR <sup>[40]</sup>	BTB	Intel	无
Drk <sup>[9]</sup>	无需	Intel	Intel TSX
Evict+Prefetch <sup>[10]</sup>	无需	Intel	Cache eviction
AnC <sup>[41]</sup>	MMU	Intel、AMD、ARM	Cache eviction
Data Bounce <sup>[11]</sup>	无需	Intel	Flush+Reload
EchoLoad <sup>[11]</sup>	无需	Intel	Flush+Reload
TagBleed <sup>[12]</sup>	Tagged TLB	Intel	A confused deputy attack
Prefetch+Time <sup>[42]</sup>	无需	AMD	无
Prefetch+Power <sup>[42]</sup>	无需	AMD	无

## 2.3 x86 处理器内存管理

现代处理器普遍采用虚拟内存技术管理内存. 整个虚拟地址空间被划分成固定大小的页, 每个虚拟内存页被映射到一个物理内存页. 处理器采用页表来保存虚拟地址和物理地址之间的映射关系, 由于每个进程拥有各自独立的映射关系, 这就使得它们拥有各自独立的虚拟地址空间, 从而保证了进程隔离. Linux 操作系统中的 mmap 系统调用用来在当前进程的虚拟地址空间中开辟一块可用的地址空间, 并为它建立映射关系. 如图 2 所示, Intel 处理器采用多级页表管理内存映射关系<sup>[2]</sup>, 其中 CR3 寄存器存储了当前进程的第一级页表的物理基地址, 将它与虚拟地址中与其对应的 9 位偏移量相加, 就得到访问下一级页表的物理地址, 最终, 将最后一级页表中存储的物理页号与虚拟地址中与其对应的 12 位偏移量相加就得到了该虚拟地址对应的物理地址. 处理器这种访问多级页表的过程被称作走表(Page Walk). 由于页表存储在内存中, 访问速度慢, 为了加速走表过程, 现代处理器一般使用 TLB 缓存常用的页表数据. 最后一级页表除了存储物理

① 研究人员将通过工程实验来分析处理器微体系结构实现细节的过程称作逆向工程(Reverse Engineering)<sup>[26]</sup>.

② KAISER 在 Linux 上被称作 KPTI(Kernel Page Table Isolation); 在 Windows 上被称作 KVA(Kernel Virtual Address shadow). 由于熔断漏洞也是利用了内核地址空间和用户地址空间隔离的不彻底, 因此 KAISER 对熔断漏洞也是有效的.

页号,还存储了页属性标志位,如图 3 所示,其中一个重要的标志位是 P,它记录了所指向的物理页是否在内存中,若为 0,表明未在内存中,此时访问该页会产生一个缺页异常(Page Fault)。

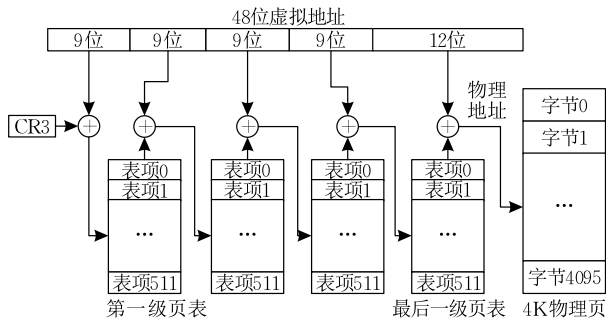


图 2 Intel 处理器中的地址转换

P	RW	US	A	D	XD	物理页号	...
P(Present)	RW(Read/Write)	US(User/Supervisor)	A(Accessed)	D(Dirty)	XD(Execute Disable)		
: 是否在内存中,1表示是,0表示否		: 是否可写,1表示可写,0表示只读		: 访问模式,1表示可在普通用户模式下被访问,0表示只能在超级用户模式下被访问			
: 是否被访问过,1表示是,0表示否		: 是否被写过,1表示是,0表示否		: 是否可执行,0表示是,1表示否			

图 3 页属性标志位

如前所述,为了保证进程隔离,各个进程理应拥有各自独立的映射关系.然而,x86 硬件要求在每个用户地址空间中保留相当一部分内核地址映射关系以方便上下文切换<sup>[43]</sup>.如此一来,当发生系统调用等需要从用户进程切换到内核进程的情况时,硬件只需切换到内核模式而不需要改变地址空间映射.此外,如图 4 所示,许多操作系统会将部分或全部物理内存直接映射到内核地址空间中的某处,这被称作物理直接映射(Physical Direct Maps).物理直接映射基地址对普通用户不可见,知道它的值可实施 ret2dir 攻击<sup>[10]</sup>.

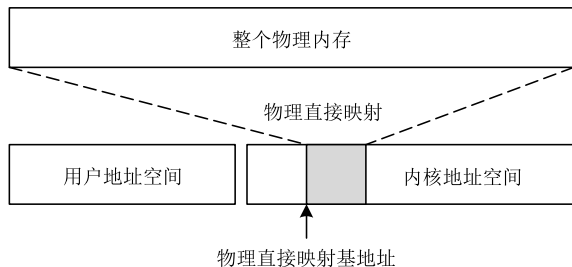


图 4 物理直接映射

## 2.4 微码辅助

微码(Microcode)指的是包含不止 4 条微操作(Micro-ops)的指令<sup>[34]</sup>,它可以实现比普通指令更复

杂的功能,比起普通指令,它的执行速度更慢.x86 处理器通过微码来扩展和修改处理器的行为<sup>[34]</sup>,比如 Intel SGX(Software Guard Extensions)技术就是通过微码实现的<sup>①</sup>.微码辅助(Microcode Assist)指的是利用微码协助处理某些不频繁使用或者复杂的操作,比如浮点运算和 VMASKMOV 指令都涉及微码辅助。

此前也有研究工作对 VMASKMOV 指令的安全性进行了讨论.Ragab 等人<sup>[45]</sup>指出,当在 Intel 处理器上使用 VMASKMOV 指令访问被屏蔽的非法地址时,处理器不会抛出异常,但会引起机器清扫(Machine Clear)和微码辅助.然而,Ragab 等人还指出 VMASKMOV 指令不存在可捕捉的瞬态执行窗口,因此无法利用它实施瞬态执行攻击.Ragab 等人的研究工作主要关注了 VMASKMOV 指令在引起机器清扫方面的情况,并没有对 VMASKMOV 指令特性进行深入研究。

## 3 面向安全分析的 VMASKMOV 指令微架构逆向工程

VMASKMOV 指令是 x86 处理器在 SIMD 指令集扩展中引入的一条向量操作指令,可以对 128 位(VMASKMOVPS)或 256 位(VMASKMOVDPD)的数据进行条件打包和移动操作,此处的“条件”指的是这条指令能够通过屏蔽位来决定对这 128 位或 256 位中的哪些数据进行打包和移动,对应屏蔽位为 1 的数据进行打包和移动,对应屏蔽位为 0 的数据不发生打包和移动<sup>[2]</sup>.如图 5(a)所示,当被用作 mask load 操作时(后文简称 VMASKMOV(load)),VMASKMOV 指令从内存中取数据,根据屏蔽寄存器 xmm1 中的屏蔽位(0100)决定把内存中‘1’对应位置的数据‘B’取到目的寄存器 xmm0;如图 5(b)所示,当被用作 mask store 操作时(后文简称 VMASKMOV(store)),VMASKMOV 指令往内存中存数据,它根据屏蔽寄存器 xmm1 中的屏蔽位(0100)决定把源寄存器 xmm0 中‘1’对应位置的数据‘W’存到内存.注意,图 5 中的访问地址 addr 落在一个非法内存页中,一般而言,使用 load 和 store 访问非法地址会抛出异常并终止程序执行,但 x86 手册<sup>[2,46]</sup>规定,当使用 VMASKMOV 指令访问非法

① Intel. Xucode: An Innovative Technology for Implementing Complex Instruction Flows. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html> 2021, 5, 6

地址且其对应的屏蔽位是 0 时不抛出异常,因此图 5 中(a)和(b)两种情况都不会抛出异常。

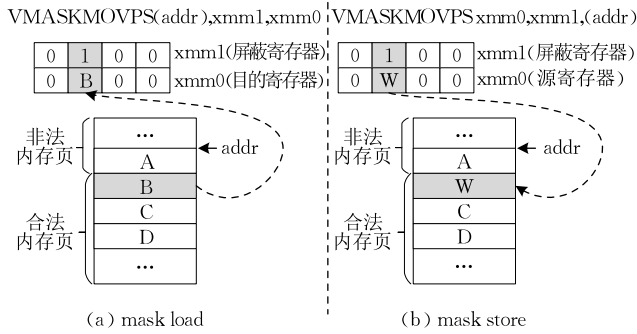


图 5 VMASKMOV 指令功能

Intel 手册<sup>[46]</sup>在介绍 VMASKMOV 指令时强调:VMASKMOV(load)访问非法地址时,即使该地址对应的屏蔽位为 0,也会导致一个长达几百个时钟周期的微码辅助。Costan 等人指出 VMASKMOV 指令中的微码辅助的作用是对正在访问的所有地址及其屏蔽位进行检查,当发现正在访问一个非法地址且它对应的屏蔽位是 1 时,抛出异常并终止程序执行<sup>①</sup>。Agner 也有类似的观点,他在 AMD Jaguar 处理器上观察到当所有屏蔽位都为 0 时,VMASKMOV 指令的执行时间超过 300 个时钟周期<sup>②</sup>。考虑到上述文献对 VMASKMOV 指令的描述都较为简略,本文首次结合了时间戳计数器(Time Stamp Counter, TSC)、硬件性能计数器(Hardware Performance

Counters, HPC)、MDS 等技术对 VMASKMOV 指令进行了逆向工程实验,EvilMask 漏洞正是基于这些实验结果提出的。

### 3.1 基于时间戳计数器(TSC)的逆向工程实验

本节通过测量 VMASKMOV 指令在不同处理器上访问不同页属性标志位的内存页所花费的时钟周期来研究指令的特性。x86 处理器中的 TSC 存储了自处理器启动以来所经历的时钟周期数,该值可在用户态下使用 x86 处理器提供的 RDTSC 指令读取。

在设计本节实验时,首先,考虑到不同处理器实现的 VMASKMOV 指令不同,为了对比分析,选取了多款不同型号的 x86 处理器进行实验,表 3 列举了其中有代表性的几款处理器(涵盖 Skylake、Ivy Bridge、Tiger Lake、Zen2、Zen3 这些常见的微架构,以及真实环境和虚拟机环境)。其次,考虑到 VMASKMOV 指令访问属性不同的内存页时其行为会有所不同(例如图 5 中的合法和非法内存页,访问它们会产生不同的执行结果),因此将内存页属性在图 5 的“合法”和“非法”的基础上进一步细分,测试了 6 种可能影响 VMASKMOV 指令执行的时钟周期的内存页属性标志位(P、RW、US、A、D、XD),见表 3,除此之外,还测试了 VMASKMOV 指令在访问已映射和未映射内存页时的情况。为了比较分析,本节还对 PREFETCH 指令进行了实验。

表 3 VMASKMOV (load)、VMASKMOV (store)、PREFETCH 指令在不同情况下执行花费的时钟周期

处理器	操作	是否映射		P		RW		US		A		D		XD	
		否	是	0	1	0	1	0	1	0	1	0	1	0	1
Intel Core i7-6700 (Skylake)	load	886	449	727	113	111	111	594	111	887	112	112	111	112	111
	store	878	428	670	426	425	108	587	108	909	425	528	109	110	425
	prefetch	468	111	315	105	105	105	244	105	401	106	105	105	105	105
Intel Core i5-6200U (Skylake)	load	438	167	276	96	96	96	216	96	203	96	96	96	96	96
	store	487	227	492	226	227	98	268	98	248	226	226	98	98	227
	prefetch	358	93	181	92	92	92	146	92	134	92	92	92	92	92
Intel Core i5-3320M (Ivy Bridge)	load	785	445	573	183	184	183	503	181	480	179	202	177	176	178
	store	179	180	181	182	183	184	184	183	184	181	180	180	178	178
	prefetch	576	184	351	187	188	188	261	186	260	184	182	537	180	181
Intel Core i7-1160G7 (Tiger Lake)	load	356	240	286	83	83	85	276	87	283	83	83	85	85	83
	store	240	114	159	111	111	106	150	109	159	111	111	107	107	111
	prefetch	161	69	99	67	67	67	67	68	99	67	67	67	67	67
AMD Ryzen 7 3700X (Zen 2)	load	551	378	536	259	260	265	515	267	389	260	262	267	269	260
	store	359	287	302	278	276	278	275	278	278	277	278	278	280	277
	prefetch	381	272	327	259	260	268	291	266	367	260	261	265	267	260
AMD Ryzen 5 5600X (Zen 3)	load	556	525	560	240	244	241	522	243	340	239	241	241	240	247
	store	247	247	248	247	248	244	244	244	245	248	245	245	245	249
	prefetch	329	236	315	236	238	238	263	237	332	235	238	238	238	237

注:load 表示 VMASKMOV (load),store 表示 VMASKMOV (store);标灰的项表示当页属性标志位不同时,指令执行存在较明显的时钟周期差异。

① Costan V, Devadas S. Intel SGX explained. <https://eprint.iacr.org/2016/086>, 2017, 2, 21

② Agner F. The microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. <https://agner.org/optimize/microarchitecture.pdf>, 2023, 3, 26

本节实验用到的伪代码见源代码 1(部分代码参考了文献[9]),以下对它作简要说明。

(1) `vmalloc` 函数用来分配一段内核地址空间, `set0_A` 函数用来将页属性标志位 `A` 设置为 0. 这两个操作是内核级操作,我们将这些功能封装在一个内核设备文件中,用户进程通过 `ioctl` 接口调用。

(2) 未映射地址和已映射地址. 如源代码 1 第 18 行所示,根据表 1 得知 `0xfffffffbf000000` 未被映射到内核地址空间中,因此使用它作为未映射地址;如源代码 1 第 19 行所示,因为 `/proc/kallsyms` 中的地址都已经被映射到内核地址空间中,本文从 `/proc/kallsyms` 中选择一个地址作为已映射地址。

(3) 使用 `mmap` 系统调用来开辟一段已映射地址空间和设置 `RW` 及 `XD` 标志位;使用 `read_page` 函数读访问当前的内存页(设置 `P`、`A` 标志位为 1);使用 `write_page` 函数写访问当前的内存页(设置 `P`、`A`、`D` 标志位为 1)。

**源代码 1.** 基于时间戳计数器(TSC)的逆向工程实验。

```

1. char * create_addr (void) {
2.     char * addr=mmap (NULL, 4096,
3.         PROT_READ|PROT_WRITE|PROT_EXEC,
4.         MAP_PRIVATE |MAP_ANONYMOUS,-1,0);
5.     return addr;
6. }
7. char global_variable;
8. void read_page(char * addr) {
9.     int * tmp=(int *) addr;
10.    global_variable=tmp[0];
11. }
12. void write_page(char * addr) {
13.    int * tmp=(int *) addr;
14.    for (int i=0; i<0x400; i++)
15.        tmp[i]=0x12345678;
16. }
17. /* 未映射地址和已映射地址 */
18. char * unmapped=0xfffffffbf0000000;
19. char * mapped=0xfffffff81800060;
20. /* 未在内存中地址和已在内存中地址 */
21. char * p0=create_addr();
22. char * p1=create_addr();
23. read_page(p1); //读访问将 p1 加载到内存中
24. /* 只读地址和可写地址 */
25. char * rw0=mmap (NULL, 0x1000,
26.     PROT_READ|PROT_EXEC,
27.     MAP_PRIVATE |MAP_ANONYMOUS,-1,0);
28. read_page(rw0);

```

```

29. char * rw1=create_addr();
30. write_page(rw1);
31. /* 内核地址和用户地址 */
32. char * us0=vmalloc(4096); //内核地址
33. char * us1=malloc(4096); //用户地址
34. /* 未访问地址和已访问地址 */
35. char * a0=create_addr();
36. read_page(a0);
37. set0_A(a0); //设置 A 标志位为 0
38. char * a1=create_addr();
39. read_page(a1);
40. /* 未写地址和已写地址 */
41. char * d0=create_addr();
42. read_page(d0);
43. char * d1=create_addr();
44. write_page(d1);
45. /* 可执行地址和不可执行地址 */
46. char * xd0=create_addr();
47. write_page(xd0);
48. char * xd1=mmap (NULL, 0x1000,
49.     PROT_READ|PROT_WRITE,
50.     MAP_PRIVATE |MAP_ANONYMOUS,-1,0);
51. read_page(xd1);
52. /* 设置待测地址 */
53. addr=unmapped;
54. /* 设置屏蔽寄存器为 0 */
55. vxorps xmm1, xmm1, xmm1
56. /* 测试 VMASKMOV(load)执行的时钟周期 */
57. time1=rdtsc(); //读取当前 TSC 值
58. vmaskmovps(addr), xmm1, xmm0
59. time2=rdtsc(); //读取当前 TSC 值
60. clock_cycle=time2-time1; //指令执行时钟周期
61. /* 测试 VMASKMOV(store)执行的时钟周期 */
62. time1=rdtsc(); //读取当前 TSC 值
63. vmaskmovps xmm0, xmm1, (addr)
64. time2=rdtsc(); //读取当前 TSC 值
65. clock_cycle=time2-time1; //指令执行时钟周期
66. /* 测试 PREFETCH 执行的时钟周期 */
67. time1=rdtsc(); //读取当前 TSC 值
68. prefetchnta(addr)
69. prefetcht2(addr)
70. time2=rdtsc(); //读取当前 TSC 值
71. clock_cycle=time2-time1; //指令执行时钟周期

```

本文对源代码 1 中的每种地址测试了 1000 次,取平均值后的结果见表 3,实验现象和分析如下。

(1) 在几乎所有已测处理器上, `VMASKMOV` (`load`)和 `PREFETCH` 的实验结果基本一致(访问未映射、页属性标志位为 `P=0`、`US=0`、`A=0` 的地



址花费的时钟周期要大于访问已映射、页属性标志位为  $P=1$ 、 $US=1$ 、 $A=1$  的地址)。因此,当屏蔽位为 0 时,VMASKMOV (load) 和 PREFETCH 在这些处理器上的实现方法类似。

(2) VMASKMOV (store) 的实验结果与 VMASKMOV (load) 和 PREFETCH 有区别:在 Intel Skylake 处理器上,它能区分是否映射以及几乎所有的页属性标志位 ( $P$ 、 $RW$ 、 $US$ 、 $D$ 、 $XD$ ) 的状态,在其他处理器上,它只能区分是否映射以及少数页属性标志位的状态。此外,在 Intel Ivy Bridge 和 AMD Zen 3 处理器上,当访问属性不同的内存页时,VMASKMOV (store) 花费的时钟周期几乎没有变化,可见在这两类处理器上 VMASKMOV (store) 进行了重新设计:当屏蔽位为 0 时,VMASKMOV (store) 什么也不做。

(3) 已映射地址和未映射地址。对大部分已测处理器,使用 VMASKMOV (load)、VMASKMOV (store)、PREFETCH 访问未映射地址花费的时钟周期都要明显大于访问已映射地址,分析原因为:即使屏蔽位全为 0,指令也会对所访问地址的合法性逐一检查,这意味着处理器需要访问 TLB 或页表来确定所访问的各个地址是否合法,当映射关系已经建立时,处理器一般能从 TLB 中快速访问到页表数据,反之,处理器只能从内存中访问到页表数据。

(4) 未在内存中地址和已在内存中地址。对大部分已测处理器,使用 VMASKMOV (load)、VMASKMOV (store)、PREFETCH 访问未在内存中地址花费的时钟周期都要明显大于访问已在内存中地址。

(5) 只读地址和可写地址、未写地址和已写地址。对 Intel Skylake 处理器,使用 VMASKMOV (store) 访问只读地址、未写地址花费的时钟周期要明显大于访问可写地址、已写地址。

(6) 可执行地址和不可执行地址。对 Intel Skylake 处理器,使用 VMASKMOV (store) 访问不可执行地址花费的时钟周期要明显大于访问可执行地址。

(7) 未访问地址和已访问地址。对大部分已测处理器,使用 VMASKMOV (load) 和 PREFETCH 访问未访问地址花费的时钟周期要明显大于访问已访问地址。

综上所述,VMASKMOV 指令能区分当前地址是否被映射以及其所在页的大部分页属性标志位的状态。Gruss 等人提出的 Prefetch + Time<sup>[42]</sup> 是与本文最相关的研究工作,这来自于 PREFETCH 指令与 VMASKMOV (load) 指令的相似性:这两条指令都能区分地址是否被映射以及  $P$ 、 $US$ 、 $A$  页属性标志位的状态。我们认为 EvilMask 比起 Prefetch + Time 的优势之一体现在 VMASKMOV (store) 指

令上:在部分 Intel 处理器上,它能区分 PREFETCH 指令无法区分的  $RW$ 、 $D$ 、 $XD$  页属性标志位的状态。

### 3.2 基于硬件性能计数器(HPC)的逆向工程实验

本节利用 HPC 分析 VMASKMOV 指令引起表 3 现象的原因。本节实验在一台搭载了 Intel Core i5-6200U (Skylake) 处理器的计算机上进行。选取了 6 种性能事件,见表 4,事件的详细含义见 Intel 官网<sup>①</sup>,覆盖微码辅助和几种与访存操作密切相关的事件,其中 OTHER\_ASSISTS\_ANY 可直接反映指令执行过程中是否触发了微码辅助。本节使用 likwid-perfctr 工具<sup>②</sup>对指令执行过程中性能事件的发生次数进行统计,见源代码 2,实验结果见表 5。

表 4 本节实验中用到的性能事件

性能事件	含义	本文简写
OTHER_ASSISTS_ANY	除 FP 外的其他微码辅助发生次数	ASSIST
DTLB_LOAD_MISSES. MISS_CAUSES_A_WALK	因 DTLB 缺失引起走表的 load 指令条数	DTLBLE
DTLB_STORE_MISSES. MISS_CAUSES_A_WALK	因 DTLB 缺失引起走表的 store 指令条数	DTLBS
ITLB_MISSES. MISS_CAUSES_A_WALK	因 ITLB 缺失引起走表的指令条数	ITLB
ICACHE_64B. IFTAG_ALL	从 ICACHE 中取指令的次数	ICACHE
L1D_PEND_MISS. PENDING_CYCLES_ANY	发生 L1D 缺失的周期数	L1D

#### 源代码 2. 基于硬件性能计数器(HPC)的逆向工程实验。

```

1.  vxorps xmm1, xmm1, xmm1 //设置屏蔽寄存器
                                xmm1 为 0
2.  LIKWID_MARKER_INIT
3.  LIKWID_MARKER_THREADINIT
4.  LIKWID_MARKER_START("test");
5.  for (int i=0; i<1000000; i++) { //测试 1000000 次
6.      /* VMASKMOV (load) 测试 */
7.      VMASKMOVPS (addr), xmm1, xmm0
8.      /* VMASKMOV (store) 测试 */
9.      VMASKMOVPS xmm0, xmm1, (addr)
10.     /* PREFETCH 测试 */
11.     PREFETCHNTA (addr)
12.     PREFETCHT2 (addr)
13. }
14. LIKWID_MARKER_STOP("test");
15. LIKWID_MARKER_CLOSE;

```

① Intel. Skylake Client Events. <https://perfmon-events.intel.com>, 2023, 4, 25

② RRZE-HPC. likwid. <https://github.com/RRZE-HPC/likwid>, 2023, 4, 26

表 5 基于硬件性能计数器(HPC)的逆向工程实验结果

操作	标志位	标志位值	ASSIST	DTLBL	DTLBS	ITLB	ICACHE	L1D
load	是否映射	否	1000000	3999918	2	49	135579100	1761971
		是	1000000	25	5	43	63792540	898937
	P	0	1000000	4000190	10	61	109700100	1524780
		1	0	4	0	8	1003545	1832
	RW	0	0	2	0	4	1002872	1856
		1	0	2	0	3	1002794	1776
	US	0	1000000	30	1	45	57372730	2874769
		1	0	1	0	2	1002840	2370
	A	0	1000000	2000619	1	33	85616460	1849011
		1	0	2	10	0	1002699	1755
	D	0	0	2	0	4	1002852	2200
		1	0	3	0	2	1002886	2311
	XD	0	0	4	0	5	1003444	1455
		1	0	3	0	5	1005658	2423
store	是否映射	否	1000000	52	3999733	89	131415400	2195273
		是	1000000	42	4	57	69006440	1504138
	P	0	1000000	50	3999737	77	120518400	784688
		1	1000000	43	3	68	61047050	869553
	RW	0	1000000	41	4	51	59164300	678204
		1	1000000	30	3	45	61132770	507234
	US	0	1000000	32	2	42	60718800	535692
		1	0	2	0	23	1003263	2161
	A	0	1000000	53	2001725	48	87785450	1618817
		1	1000000	28	2	45	61277800	471843
	D	0	1000000	20	5	30	60236480	427910
		1	1000000	19	3	37	61221420	426445
	XD	0	1000000	35	3	58	61262950	481157
		1	1000000	20	8	48	60770080	2217138
prefetch	是否映射	否	0	2019639	6	44	2299133	631494
		是	0	1	0	22	2003257	302991
	P	0	0	2020267	4	47	1161277	679413
		1	0	3	0	6	1003193	2015
	RW	0	0	3	0	13	1002728	1878
		1	0	2	0	14	1002782	1748
	US	0	0	4	0	4	1002601	2425
		1	0	1	0	13	1002660	1505
	A	0	0	2023286	6	60	1175140	710777
		1	0	4	0	4	1003401	1897
	D	0	0	3	0	8	1003334	2000
		1	0	4	0	9	1002758	1426
	XD	0	0	3	0	7	1002879	2435
		1	0	2	0	20	1002714	2452

注:标灰的项表示页属性标志位不同时,该性能事件发生次数存在较明显差异。

实验结果分析如下:

(1) 使用 VMASKMOV (load)、VMASKMOV (store)、PREFETCH 访问未映射地址花费的时钟周期要明显大于访问已映射地址,这主要与 DTLB 有关:访问未映射地址发生的 DTLB 缺失次数要明显多于访问已映射地址。

(2) 使用 VMASKMOV (load)、VMASKMOV (store)、PREFETCH 访问未在内存中地址花费的时钟周期要明显大于访问已在内存中地址,这主要与 DTLB 有关:访问未在内存中地址发生的 DTLB 缺失次数要明显多于访问已在内存中地址。此外,访

问未在内存中地址会触发缺页异常,系统处理该异常也会消耗时间。

(3) 使用 VMASKMOV (load) 和 PREFETCH 访问未访问地址花费的时钟周期要明显大于访问已访问地址,这主要与 DTLB 有关:访问未访问地址发生的 DTLB 缺失次数要明显多于访问已访问地址。此外,虽然表 3 在 Intel Core i5-6200U (Skylake) 处理器上使用 VMASKMOV (store) 并不能明显区分已访问地址和未访问地址,但在表 5 中也观察到使用它访问未访问地址时发生的 DTLB 缺失次数要明显多于访问已访问地址。

(4) 使用 VMASKMOV (load)、VMASKMOV (store) 访问内核地址花费的时钟周期要明显大于访问本进程的用户地址. 这主要与是否引起微码辅助有关: 访问内核地址会引起微码辅助, 而访问本进程的用户地址不会, 由前文可知, 微码辅助会花费较多的时钟周期.

(5) 使用 VMASKMOV (store) 访问不可执行地址花费的时钟周期要大于访问可执行地址, 分析与 L1D 相关, 访问不可执行地址发生的 L1D 缺失次数要大于访问可执行地址.

以下两种情况表 5 中的实验结果无法解释, 留作未来的研究工作.

(1) 使用 PREFETCH 访问内核地址花费的时钟周期要明显大于访问本进程的用户地址. 初步分析如下: 虽然 PREFETCH 在访问非法地址时不产生异常和引起微码辅助, 但这种情况比起访问合法地址更复杂, 系统在处理这种情况时所花费的时间也会更长.

(2) 在 Intel Skylake 处理器上, 使用 VMASKMOV (store) 访问只读地址、未写地址花费的时钟周期要明显大于访问可写地址、已写地址. 初步分析如下: Intel Skylake 处理器采用了某些优化机制来优化 store 操作, 使得使用 VMASKMOV (store) 访问此前写过的地址会比访问此前没有写过的地址速度更快.

### 3.3 基于 MDS 的逆向工程实验

Intel 公开的两项专利<sup>[47-48]</sup>对 VMASKMOV 指令的执行流程进行了介绍, 如图 6 所示, 这揭示了 VMASKMOV 指令的一个特性: VMASKMOV (load) 会把目标地址所在的全部 128 位或 256 位数据拷贝到临时存储器中(它可以是一个对体系结构层不可

见的寄存器), 然后再去做屏蔽操作. 换言之, 在微体系结构层面, 被屏蔽数据也会发生移动. 本节设计实验验证 VMASKMOV 指令在 Intel 处理器上是否确实采用了文献[47-48]所述的实现方式.

考虑到当屏蔽位为 0 时, 被屏蔽数据是否发生移动在体系结构层面不可见, 本节结合 MDS 采样技术观察被屏蔽数据在微体系结构层面中的移动, 如图 6 所示, 如果被屏蔽数据发生了移动, 那么它一定会经过 Fill Buffer 并能通过 MFBDS 攻击观察到. 本节实验用到的伪代码见源代码 3.

#### 源代码 3. 基于 MDS 的逆向工程实验.

```

1. /* 被屏蔽数据是本进程用户地址空间中的数据 */
2. char * addr = malloc(4096);
3. memset(addr, 'U', 4096);
4. /* 被屏蔽数据是内核地址空间中的数据 */
5. char * addr = vmalloc(4096);
6. kmemset(addr, 'K', 4096);
7. /* 设置屏蔽寄存器 xmm1 为 0 */
8. vxorps xmm1, xmm1, xmm1
9. /* 使用 VMASKMOV 指令移动被屏蔽数据 */
10. vmaskmovps(addr, xmm1, xmm0)
11.
12. /* MFBDS, 详见参考文献[33] */
13. char * target = NULL;
14. memory_access(mem + 4096 * target[0]);
15. for (int i = 0; i < 256; i++)
16.   if (flush_reload((char *)mem + 4096 * i))
17.     /* 打印 Fill Buffer 中的数据,
18.        若 'U' 或 'K' 被打印,
19.        表明被屏蔽数据发生了移动 */
20.     printf("%c\n", i);

```

本节的实验在一台搭载了 Intel Core i5-3320M (Ivy Bridge) 和一台搭载了 Intel Core i5-6200U (Skylake) 处理器的计算机上进行, 这两款处理器都受 MFBDS 影响, 在这两台机器上观察到的实验现象一致: (1) 当被屏蔽数据是本进程用户地址空间中的数据时, 能够通过 MFBDS 采样到它(源代码 3 中的 'U' 字符被打印); (2) 当被屏蔽数据是内核地址空间中的数据时, 无法通过 MFBDS 采样到它(源代码 3 中的 'K' 字符未被打印). 实验现象(1)表明: VMASKMOV 指令在 Intel 处理器上确实采用了文献[47-48]所述的实现方式, 即 VMASKMOV (load) 确实会把目标地址所在的全部 128 位或 256 位数据拷贝到一个临时存储器中, 然后再去做屏蔽操作. 实验现象(2)表明: 当被屏蔽数据是内核地址空间中的数据(或当前进程无权访问的数据)时, 被屏蔽数据

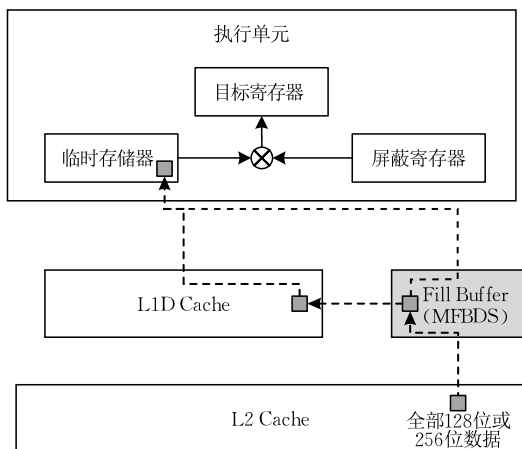


图 6 被屏蔽数据的移动过程(虚线箭头)

不会经过 Fill Buffer. 结合专利<sup>[47-48]</sup>提供的信息, 分析原因如下: 被屏蔽数据在移动的过程中会进行地址合法性检查, 当处理器发现正在访问的是非法地址时, 会终止将该地址中的数据移动到临时存储器中.

### 3.4 实验小结

对第 3.1、3.2、3.3 节的实验结果总结如下: (1) 通过测量 VMASKMOV 指令的执行时间, 可以判断目标地址是否被映射及其所在页的大部分页属性标志位 (P、RW、US、A、D、XD) 的状态, 这主要与 DTLB 是否发生缺失以及是否引起了微码辅助有关; (2) 即使屏蔽位为 0, VMASKMOV (load) 也会把目标地址所在的全部 128 位或 256 位数据拷贝到临时存储器中, 导致被屏蔽数据发生移动, 如果在某些 Intel 处理器上执行此操作, 被屏蔽数据将经过 Fill Buffer, 并可利用 MDS 技术采样到.

## 4 EvilMask

本文提出 EvilMask, 一个利用 VMASKMOV 指令实施侧信道攻击的新漏洞, 并提出 3 个 EvilMask 攻击原语: VMASKMOVL + Time (MAP)、VMASKMOVS + Time (XD) 和 VMASKMOVL + MDS.

### 4.1 VMASKMOVL + Time (MAP)

VMASKMOVL + Time (MAP) 攻击原语适用的威胁模型为: 假设受害机是一台搭载了 Intel 或 AMD 处理器的计算机, 且支持 VMASKMOV 指令, 该指令普通用户也有调用权限; 受害机开启了 KASLR, 关闭了 KPTI/KVA; 攻击者在受害机上能以普通用户身份运行程序. 攻击者利用此攻击原语能获取本该只有超级用户才有权限获取的内核基地址等地址空间布局信息, 从而攻破 KASLR.

VMASKMOVL + Time (MAP) 攻击原语如代码段 1 所示, 攻击者使用 VMASKMOV (load) 遍历访问目标地址, 根据指令执行花费的时钟周期, 区分目标地址是已映射地址还是未映射地址, 从而推断出地址空间布局信息, 破解 KASLR. 它主要基于以下观察: 使用 VMASKMOV (load) 访问未映射地址花费的时钟周期要明显大于访问已映射地址. 前人的研究工作<sup>[9]</sup>表明, 根据内核地址是否映射, 可定位内核基地址、物理直接映射基地址、内核模块等 (这些地址因为是已映射地址, 花费的时钟周期更少).

**代码段 1.** VMASKMOVL + Time (MAP) 攻

击原语.

```
vxorps xmm1, xmm1, xmm1
t1=timer(); //获取当前时钟周期,如使用RDTSC指令
vmaskmovps (addr), xmm1, xmm0
t2=timer();
clock_cycle=t2-t1;
```

### 4.2 VMASKMOVS + Time (XD)

VMASKMOVS + Time (XD) 攻击原语适用的威胁模型为: 假设受害机是一台搭载了 Intel 或 AMD 处理器的计算机, 支持 VMASKMOV 指令, 该指令普通用户也有调用权限; 受害机开启了 KASLR; 攻击者能够在受害机上以普通用户身份运行程序. 攻击者利用此攻击原语能获取本该只有超级用户才有权限获取的内核模块地址布局信息, 从而攻破 KASLR.

VMASKMOVS + Time (XD) 攻击原语如代码段 2 所示, 攻击者使用 VMASKMOV (store) 遍历访问目标地址, 根据指令执行花费的时钟周期, 区分目标地址是可执行地址还是不可执行地址, 从而推断出内核模块地址布局信息, 破解 KASLR. 它主要基于以下观察: 使用 VMASKMOV (store) 访问不可执行地址花费的时钟周期要明显大于访问可执行地址. 前人的研究工作<sup>[9]</sup>表明, 根据内核地址是否可执行, 可定位具有唯一大小的内核模块的名称和位置 (内核模块的名称和其所对应的位置信息只有超级用户才有权限访问). 如图 7 所示, 在 Linux 中, 各内核模块一般被一段未映射地址分隔开, 内核模块本身则开始于一个可执行的代码段 (如 .text), 随后再跟着一个不可执行的数据段 (如 .bss 和 .rodata). 通过获知可执行与不可执行段的位置, 攻击者可以获知一个内核模块的起始和结束位置, 并且算得该内核模块的大小. 之后攻击者通过非特权命令 lsmod 获知内核中所有内核模块的名称和其所对应的大小信息 (如图 8 所示), 将这与利用 VMASKMOVS + Time (XD) 攻击原语获知的内核模块的大小和位置信息进行比对, 就可以定位出具有唯一大小的内核模块的名称和其所对应的位置信息, 从而使得 KASLR 无效.

**代码段 2.** VMASKMOVS + Time (XD) 攻击原语.

```
vxorps xmm1, xmm1, xmm1
t1=timer(); //获取当前时钟周期,如使用RDTSC指令
vmaskmovps xmm0, xmm1, (addr)
t2=timer();
clock_cycle=t2-t1;
```

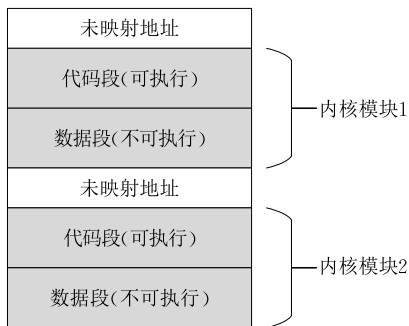


图 7 Linux 中内核模块地址布局规律

```
hack@ubuntu:~$ lsmod
Module                Size  Used by
vmw_vsock_vmci_transport 28672  1
vsock                 36864  2 vmw_vsock_vmci_transport
crct10dif_pclmul     16384  0
crc32_pclmul         16384  0
snd_ens1371          28672  2
snd_ac97_codec       131072 1 snd_ens1371
gameport             16384  1 snd_ens1371
ac97_bus              16384  1 snd_ac97_codec
vmw_balloon          20480  0
snd_pcm               106496 2 snd_ac97_codec,snd_ens1371
aesni_intel          167936 0
aes_x86_64           20480  1 aesni_intel
lrw                   16384  1 aesni_intel
```

图 8 Linux 4.4.0 中的内核模块名称和大小

### 4.3 VMASKMOVL+MDS

VMASKMOVL+MDS 攻击原语适用的威胁模型为:假设受害机是一台搭载了 Intel 处理器的计算机,且该处理器支持 VMASKMOV 指令,该指令普通用户也有调用权限;该处理器受 MDS 漏洞影响且受害机关闭了 MDS 漏洞补丁;受害者和攻击者都有权限在受害机上运行程序;受害者进程中存在 VMASKMOVL gadget 或者攻击者有能力向受害者进程注入 VMASKMOVL gadget;攻击者能够控制 VMASKMOVL gadget 反复访问受害者进程中的敏感数据;受害者进程受软件约束不能直接访问该敏感数据(即通过普通的 load 操作访问该敏感数据报软件错误),但直接访问它不报硬件错误(即通过普通的 load 操作访问该敏感数据不报硬件错误).攻击者利用此攻击原语能突破软件隔离(如沙箱)窃取受害者进程中的敏感数据.

3.3 节实验表明:即使屏蔽位为 0,VMASKMOV (load) 也会把被屏蔽数据拷贝到临时存储器中,且被拷贝数据经过 Fill Buffer. 基于此,本文提出了 VMASKMOVL+MDS 攻击原语,见代码段 3:受害者进程中存在一个 VMASKMOVL gadget,攻击者通过控制该 gadget 执行将受害者进程中的数据泄露到 Fill Buffer 中,最后在自己的进程中采样 Fill Buffer 中的数据. 源代码 4 展示了如何使用 VMASKMOVL+MDS 攻击原语突破软件隔离窃取数据:(1)受害者

进程中存在一个 VMASKMOVL gadget(5~9 行),它使用 VMASKMOV (load) 访问受害者进程中地址为 addr 的数据;(2)攻击者控制 addr 使得受害者进程访问敏感数据 secret,这就导致 secret 被泄露到 Fill Buffer 中;(3)攻击者使用 MFBDS 采样 Fill Buffer,窃取到 secret 的值.

#### 代码段 3. VMASKMOVL+MDS 攻击原语.

```
/* ----- 受害者进程 ----- */
/* VMASKMOVL gadget */
v xorps xmm1, xmm1, xmm1
vmaskmovps (addr), xmm1, xmm0
```

```
/* ----- 攻击者进程 ----- */
/* 使用 MFBDS 采样 Fill Buffer 中的数据 */
.....
```

#### 源代码 4. VMASKMOVL+MDS 攻击原语应用举例.

```
1. /* ----- 受害者进程 ----- */
2. /* 假设受害者进程直接访问 secret 报软件错误,
   但直接访问它不报硬件错误 */
3. char * secret="SSSSSSSSSS";
4. /* 假设 addr 被攻击者控制 */
5. addr=&secret;
6. /* 攻击者注入 VMASKMOVL gadget */
7. v xorps xmm1, xmm1, xmm1 //屏蔽寄存器为 0
8. /* 利用 VMASKMOV 指令访问受害者进程中的
   任意数据.屏蔽位为 0,因此不报软件错误 */
9. vmaskmovps (addr), xmm1, xmm1
10.
11. /* ----- 攻击者进程 ----- */
12. /* 使用 MFBDS 采样 Fill Buffer 中的数据 */
13. char * target=NULL;
14. memory_access (mem+4096 * target[0]);
15. for (int i=0; i<256; i++)
16.   if (flush_reload ((char *) mem+4096 * i))
17.     printf ("%c\n", i); //打印采样到的数据
```

### 4.4 EvilMask 小结

表 6 将 EvilMask 与其他针对 ASLR 的微体系结构侧信道攻击进行了比较:(1)以往的攻击大部分只泄露目标地址是否被映射(MAP)的信息,EvilMask 还能泄露目标地址是否可执行(XD)的信息;(2)以往的攻击大部分只针对某一家处理器芯片厂商研制的处理器,EvilMask 对 Intel 和 AMD 这两家 x86 芯片厂商的处理器都有效;(3)以往的攻击大部分依赖一些特殊条件,一旦目标处理器不满足这些特殊条件,攻击就无法进行,EvilMask 没有严苛的特殊条件,攻击步骤简单,容易部署.因此,比起以往

的针对 ASLR 的微体系结构侧信道攻击, EvilMask 具有泄露内容多、攻击范围广、部署实施简单的特点. 除此之外, 本文还创造性地将 VMASKMOV 指令和 MDS 漏洞结合, 提出了 VMASKMOVL + MDS 攻击原语, 进一步加深了 EvilMask 的危害性.

表 6 EvilMask 与其他微体系结构侧信道攻击对比

攻击	泄露内容	处理器	特殊条件
Double Page Fault <sup>[39]</sup>	MAP	Intel	中断处理
Jump Over ASLR <sup>[40]</sup>	MAP	Intel	逆向 BTB
Drk <sup>[9]</sup>	MAP, XD	Intel	Intel TSX
Evict + Prefetch <sup>[10]</sup>	MAP	Intel	cache eviction
AnC <sup>[41]</sup>	MAP	Intel, AMD, ARM	cache eviction
Data Bounce <sup>[11]</sup>	MAP	Intel	Flush + Reload
EchoLoad <sup>[11]</sup>	MAP	Intel	Flush + Reload
TagBleed <sup>[12]</sup>	MAP	Intel	a confused deputy attack
Prefetch + Time <sup>[42]</sup>	MAP	AMD	无
Prefetch + Power <sup>[42]</sup>	MAP	AMD	无
EvilMask	MAP, XD	Intel, AMD	无

注: MAP 表示能区分是否可映射, XD 表示能区分是否可执行.

VMASKMOV 指令还有很多其他可深入挖掘的特性, 比如根据表 3 的实验结果, 攻击者可用它来分析内核地址 P、RW、A 和 D 标志位的状态, 从而推断内核的活动情况; 此外, VMASKMOV + Time 攻击原语中的 Time 还可替换成 HPC, 设计出 VMASKMOV + HPC (DTLB) 这样不依赖 RDTSC 指令的攻击原语. 以上留作未来的研究工作.

## 5 安全风险的概念验证示例 (POC)

### 5.1 去地址空间布局随机化

本节利用 VMASKMOVL + Time (MAP) 和 VMASKMOVS + Time (XD) 攻击原语对 Intel 和 AMD 处理器上的 KASLR 进行了攻击. 包含以下 3 个子攻击.

(1) 去随机化内核基地址. 根据表 1, 在 Linux 内核中, 内核基地址一般位于地址区间 0xffffffff81000000 ~ 0xffffffffbe000000, 以 2 MB 为单位对齐, 有 488 种可能性. 本节使用 VMASKMOVL + Time (MAP) 对这 488 种可能性进行遍历, 成功找到了 Intel 和 AMD 处理器上的内核基地址, 实验结果见表 7. 比如在 Intel Core i7-6700 处理器上, 未经优化的攻击代码在 1.6 s 内完成攻击, 并猜测内核基地址为 0xffffffff97000000, 通过和 /proc/kallsyms 文件中的 startup\_64 进行比对, 发现攻击成功.

(2) 去随机化物理直接映射基地址. 根据表 1, 在 Linux 内核中, 物理直接映射基地址一般位于地址区间 0xffff880000000000 ~ 0xffffc88000000000, 以 1 GB 为单位对齐, 因此有 65 536 种可能性. 本节使用 VMASKMOVL + Time (MAP) 对这 65 536 种可能性进行遍历, 成功找到了 Intel 和 AMD 处理器上的物理直接映射基地址, 实验结果见表 7. 比如在 AMD Ryzen 7 3700X 处理器上, 未经优化的攻击代码在 2 s 内完成攻击, 并猜测物理直接映射基地址为 0xffff888000000000, 通过和真实的物理直接映射基地址进行比对, 发现攻击成功.

表 7 使用 VMASKMOVL + Time (MAP) 泄露内核基地址和物理直接映射基地址 (关闭 KPTI 补丁的情况下)

处理器	内核基地址	物理直接映射基地址
Intel Core i5-6200U	✓	✓
Intel Core i7-6700	✓	✓
AMD Ryzen 7 3700X	✓	✓
AMD Ryzen 5 5650	✓	✓

注: ✓ 表示攻击成功.

(3) 去随机化内核模块地址. 根据表 1, 在 Linux 内核中, 内核模块地址具有这些特点: 一般位于地址区间 0xffffffffc0001000 ~ 0xffffffffc0400000, 以 4 KB 为单位对齐; 模块开始于一段可执行地址, 结束于一段不可执行地址. 本节使用 VMASKMOVS + Time (XD) 猜测 Intel 和 AMD 处理器上的内核模块分布情况, 实验结果见表 8. 比如在 Intel Core i5-6200U 处理器上, 系统中一共存在 74 个内核模块, 共检测出 24 个具有唯一大小的内核模块, 和 /proc/modules 文件中的内核模块情况一致, 表明攻击成功. AMD Ryzen 7 3700X 处理器上的实验结果表明这一型号的处理器的处理器暂未受 VMASKMOVS + Time (XD) 影响.

表 8 使用 VMASKMOVS + Time (XD) 泄露内核模块地址

处理器	操作系统	泄露情况
Intel Core i5-6200U	Linux 4.15.0	✓ (24/74)
AMD Ryzen 7 3700X	Linux 4.15.0	✗

注: ✓ 表示攻击成功, ✗ 表示攻击失败.

### 5.2 窃取进程数据

本节演示如何利用 VMASKMOVL + MDS 攻击原语越权窃取受害者进程数据: 往 Linux 内核中注入一个攻击者可控制的 VMASKMOVL gadget, 攻击者控制该 gadget 执行, 并将内核进程中的数据泄露到 Fill Buffer 中, 之后通过 MDS 采样 Fill Buffer 中的数据.

向内核中注入的 VMASKMOVL gadget 如下:

```
void VMASKMOVL_gadget (char * addr) (
asm volatile (
“vxorps %%xmm1, %%xmm1, %%xmm1”
“vmaskmovps (%[addr]), %%xmm1, %%xmm0”
“mfence”
:
: “r” (addr)
: “memory”
);)
```

之后在内核中分配一段长度为 4096 的空间,向其中写入固定字符‘K’:

```
char * addr = vmalloc (4096);
memset (addr, ‘K’, 4096);
```

随后,攻击者控制受害者进程(内核)反复执行 VMASKMOVL gadget,访问包含固定字符‘K’的内核空间:

```
while (1) {
for (int i=0; i<4096; i+=64)
VMASKMOVL_gadget (addr+i);
}
```

最后,攻击者在其本地进程内通过 MFBDS 采样 Fill Buffer 中的数据(见源代码 4)。

我们在一台搭载了 Intel Core i5-6200U 处理器的机器上做实验,成功观察到字符‘K’被泄露。

注意,本节仅是演示 VMASKMOVL+MDS 的可用性,当前的 Linux 内核未见受到 VMASKMOVL+MDS 影响,这是因为截止目前本文尚未发现可以向 Linux 内核中注入 VMASKMOVL gadget 的手段。但是,随着 SIMD 技术的普及,未来的 Linux 内核很可能会支持这一点(比如在 Linux eBPF<sup>①</sup> 中增加对 VMASKMOV 指令的支持)。当然,除 Linux 内核外的其他已经支持 VMASKMOV 指令的应用软件也受 VMASKMOVL+MDS 的影响。更多的攻击场景留作未来研究工作。

## 6 防御方案

### 6.1 硬件防御方案

正如 Agner<sup>②</sup> 所说,VMASKMOV 指令存在设计缺陷:当屏蔽位都为 0 时,由于触发微码辅助,VMASKMOV 指令的执行时钟周期长达 300 个,而此时指令本该什么也不做。本文同意 Agner 的观点,毕竟这种设计既无益于性能也无益于安全。本文认为,只有从硬件上修改 VMASKMOV 指令的实现方式,才能从根本上防御 EvilMask。本节提出一种

VMASKMOV 指令的备选实现方案,通过消除微码辅助防御 EvilMask,如图 9 所示:当被用作 mask load 或 mask store 操作时,并不把目标地址或源寄存器中的全部 128 位或 256 位数据都拷贝到临时存储器和 Fill Buffer 中,而是根据屏蔽寄存器的值只访问未被屏蔽的地址和拷贝未被屏蔽的数据。在这种实现方式下,只有未被屏蔽的数据才会经过 Fill Buffer,因此可以很好地防御 VMASKMOVL+MDS。此外,当屏蔽位为 0 时,处理器不再需要访问被屏蔽地址,也就不涉及对 DTLB 和 Cache 的访问,因此可以很好地防御 VMASKMOVL+Time (MAP) 和 VMASKMOVS+Time (XD)。而且,当屏蔽位为 0 时,由于无需引入微码辅助去检查被屏蔽地址的合法性,VMASKMOV 指令的执行时钟周期也会更短,有利于性能。当然,这种实现方式会增加硬件的复杂性,由于拷贝操作总是要等屏蔽操作完成之后才进行,当屏蔽位不为 0 时,还会影响指令执行的性能。此外,本文认为在当前受影响的处理器上更改 VMASKMOV 指令的实现方式并不现实,期待在新一代的 x86 处理器中看到改变。

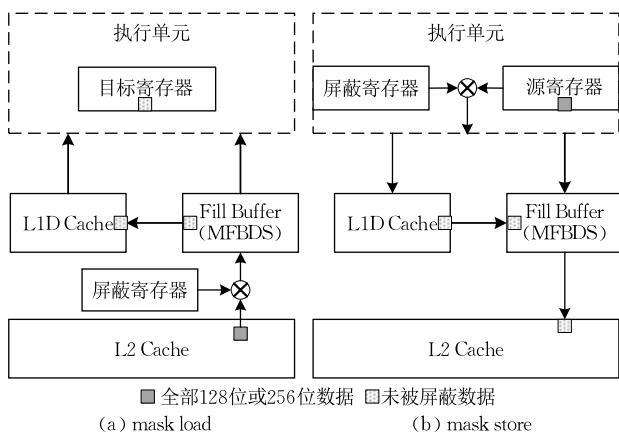


图 9 重新设计的 VMASKMOV 指令

部分 Intel 处理器已经能够在硬件上防御 MDS 攻击,由于 VMASKMOVL+MDS 依赖于 MDS 攻击来采样 Fill Buffer 中的数据,因此这些处理器暂未受 VMASKMOVL+MDS 的影响。

### 6.2 软件防御方案

目前已有一些软件防御方案可用来暂时防御 EvilMask,列举如下。

① Linux Team. BPF Documentation. <https://www.kernel.org/doc/html/latest/bpf/index.html>, 2023, 4, 27  
② Agner F. The microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. <https://agner.org/optimize/microarchitecture.pdf>, 2023, 3, 26

### 6.2.1 KAISER 和 LAZARUS

KAISER 和 LAZARUS 可用来防御 VMASKMOVL+Time(MAP). KAISER 由 Gruss 等人<sup>[43]</sup>提出,一开始用来防御 Evict+Prefetch<sup>[10]</sup>等针对 KASLR 的微体系结构侧信道攻击,后来又被用来防御危害更大的熔断漏洞,目前已被广泛部署在主流操作系统中<sup>①②</sup>. KAISER 的主要思想是:隔离用户地址空间和内核地址空间的地址映射关系.如 2.3 节所述,为了方便上下文切换,有相当一部分内核地址被映射到用户地址空间中,KAISER 的作用是尽量少地把内核地址映射到用户地址空间中. KAISER 虽然可用来缓解 VMASKMOVL+Time(MAP),但它存在以下问题:(1)KAISER 只提供了基本的保护,某些关键的内核地址(比如蹦床位置, trampoline locations<sup>[11]</sup>)依旧会被映射到用户地址空间中,如表 9 所示,在 KAISER 开启的情况下,使用 EvilMask 仍能成功获取到 Intel 处理器上作为蹦床位置之一的 \_\_entry\_text\_start 的地址;(2)在部分不受熔断漏洞影响的处理器上(比如,几乎所有的 AMD 处理器和部分最新的 Intel 处理器),KAISER 是默认关闭的.

表 9 在开启 KAISER 的 Intel 处理器上泄露 \_\_entry\_text\_start 的地址

处理器	目标地址	泄露情况
Intel Core i5-3320M		✓
Intel Core i7-1160G7	__entry_text_start	✓
Intel Core i5-6200U		✓

注:✓表示攻击成功.

LAZARUS 由 Gens 等人<sup>[43]</sup>提出,主要思想和 KAISER 类似,它能防御 KAISER 无法防御的 Jump Over ASLR 攻击<sup>[40]</sup>,但存在和 KAISER 类似的问题.

### 6.2.2 FLARE

FLARE 可用来防御 VMASKMOVL+Time(MAP)和 VMASKMOVS+Time(XD). FLARE 由 Canella 等人<sup>[11]</sup>提出,可弥补 KAISER 和 LAZARUS 的不足.它的主要思想是:为整个内核地址空间的地址建立映射关系,以使得攻击者无法根据内核地址的映射状态推断关键地址的位置(因为每个地址都是已映射地址),从而彻底消除蹦床位置等引入的残余攻击面,可很好地防御 VMASKMOVL+Time(MAP).此外,FLARE 还能消除访问可执行地址和不可执行地址的时间差,因此也能防御 VMASKMOVS+Time(XD).遗憾的是,FLARE 并未正式部署在任何主流操作系统上.

### 6.2.3 MDS 软件补丁

MDS 软件补丁<sup>③</sup>可用于防御 VMASKMOVL+MDS,主要思想是:在受害者进程中,每次使用 VMASKMOV(load)读取敏感数据后立即使用 Intel 提供的 VERW 指令或软件序列(Software Sequences)往 Fill Buffer 中填充无用数据,以覆盖被拷贝进 Fill Buffer 中的敏感数据,这样,攻击者此后采样到的就是无用数据.但是,我们认为这种方法并没有消除数据泄露的源头,即敏感数据依旧会被拷贝到 Fill Buffer 中,如果攻击者能在无用数据填充之前进行采样,这种防御方法就会失效.另外,如果受害者进程也被攻击者所控制,攻击者可以向受害者进程中注入不带 MDS 软件补丁的 VMASKMOVL gadget,也会使这种防御方法失效.

## 7 结 论

VMASKMOV 指令是 x86 处理器在 SIMD 指令集扩展中引入的一条指令,可用来对 128 位或 256 位的数据进行条件打包和移动操作,本文对这条指令的特性和安全性进行了较为深入的研究.首先,借助硬件性能计数器等技术对 VMASKMOV 指令进行了逆向工程并总结了指令特性.之后,基于指令特性指出 VMASKMOV 指令存在安全风险,并披露了一个新的处理器漏洞 EvilMask.之后提出了 3 个 EvilMask 攻击原语:VMASKMOVL+Time(MAP)、VMASKMOVS+Time(XD)和 VMASKMOVL+MDS,可用来实施去地址空间布局随机化攻击和进程数据窃取攻击.随后给出了 2 个概念验证示例(POC)用来验证 EvilMask 对真实世界的信息安全危害.最后,讨论了针对 EvilMask 的防御方案,指出最根本的解决方法是在处理器微体系结构层面重新实现 VMASKMOV 指令,并给出了初步的实现方案.

VMASKMOV 指令以及类似的指令并行优化技术在提升处理器性能方面发挥了重要作用.然而,本文的研究工作表明,这一效率优先的设计思想也

- ① Thomas G. x86/KPTI: Kernel Page Table Isolation (was KAISER). <https://lkml.org/lkml/2017/12/4/709>, 2017, 12, 4
- ② swiat. KVA shadow: Mitigating Meltdown on windows. <https://msrc.microsoft.com/blog/2018/03/kva-shadow-mitigating-meltdown-on-windows>, 2018, 3, 23
- ③ Intel. Microarchitectural Data Sampling. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html>, 2021, 3, 11




易引入安全风险。处理器安全是网络安全和数据安全的基础,在对安全性要求较高的应用场景,安全因素应放在首要位置,优先考虑,在安全目标达成的前提下,均衡考虑性能、功耗与成本。

## 参 考 文 献

- [1] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach. 5th Edition. Singapore: Elsevier Pte Ltd, 2012
- [2] Intel. Intel 64 and IA-32 Architectures Software Developer Manual. USA: Intel, Technical Report: 767375, 2023
- [3] AMD. AMD64 Architecture Programmer's Manual. USA: AMD, Technical Report: 40332, 2023
- [4] Shen J P, Lipasti M H. Modern Processor Design: Fundamentals of Superscalar Processors. USA: Waveland Press, 2013
- [5] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading kernel memory from user space//Proceedings of the USENIX Security Symposium. New York, USA, 2018: 973-990
- [6] Kocher P, Horn J, Fogh A, et al. Spectre attacks: Exploiting speculative execution//Proceedings of the IEEE Symposium on Security and Privacy. New York, USA, 2019: 1-19
- [7] Szekeres L, Payer M, Wei T, et al. SoK: Eternal war in memory//Proceedings of the IEEE Symposium on Security and Privacy. Berkeley, USA, 2013: 48-62
- [8] Shacham H, Page M, Pfaff B, et al. On the effectiveness of address-space randomization//Proceedings of the ACM Conference on Computer and Communications Security. Berkeley, USA, 2004: 298-307
- [9] Jang Y, Lee S, Kim T. Breaking kernel address space layout randomization with Intel TSX//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria, 2016: 380-392
- [10] Gruss D, Maurice C, Fogh A, et al. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria, 2016: 368-379
- [11] Canella C, Schwarz M, Haubenwallner M, et al. KASLR: Break it, fix it, repeat//Proceedings of the ACM Asia Conference on Computer and Communications Security. Taipei, China, 2020: 481-493
- [12] Koschel J, Giuffrida C, Bos H, et al. TagBleed: Breaking KASLR the isolated kernel address space using tagged TLBs//Proceedings of the IEEE European Symposium on Security and Privacy. Genoa, Italy, 2020: 309-321
- [13] Kocher P. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems//Proceedings of the Annual International Cryptology Conference. Berlin, Germany, 1996: 104-113
- [14] Osvik D A, Shamir A, Tromer E. Cache attacks and countermeasures: The case of AES//Proceedings of the Cryptographers' Track at the RSA Conference. Berlin, Germany, 2006: 1-20
- [15] Yarom Y, Falkner K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack//Proceedings of the USENIX Security Symposium. San Diego, USA, 2014: 719-732
- [16] Liu Fang-Fei, Yarom Y, Ge Qian, et al. Last-level cache side-channel attacks are practical//Proceedings of the IEEE Symposium on Security and Privacy. San Jose, USA, 2015: 605-622
- [17] Gruss D, Spreitzer R, Mangard S. Cache template attacks: Automating attacks on inclusive last-level caches//Proceedings of the USENIX Security Symposium. Washington, USA, 2015: 897-912
- [18] Gruss D, Maurice C, Wagner K, et al. Flush + Flush: A fast and stealthy cache attack//Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Cham, 2016: 279-299
- [19] Gullasch D, Bangerter E, Krenn S. Cache games — bringing access-based cache attacks on AES to practice//Proceedings of the IEEE Symposium on Security and Privacy. Oakland, USA, 2011: 490-505
- [20] Aciicmez O. Yet another MicroArchitectural attack: Exploiting I-cache//Proceedings of the ACM Workshop on Computer Security Architecture. New York, USA, 2007: 11-18
- [21] Apecechea G I, Eisenbarth T, Sunar B. S\$A: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES//Proceedings of the IEEE Symposium on Security and Privacy. San Jose, USA, 2015: 591-604
- [22] Irazoqui G, Eisenbarth T, Sunar B. Cross processor cache attacks//Proceedings of the Asia Conference on Computer and Communication Security (ASIA CCS). Xi'an, China, 2016: 353-364
- [23] Irazoqui G, et al. Wait a minute! A fast, cross-VM attack on AES//Proceedings of the International Workshop on Recent Advances in Intrusion Detection. Gothenburg, Sweden, 2014: 299-319
- [24] Yan M, Sprabery R, Gopireddy B, et al. Attack directories, not caches: Side channel attacks in a non-inclusive world//Proceedings of the IEEE Symposium on Security and Privacy. San Francisco, USA, 2019: 888-904
- [25] Lipp M, Gruss D, Spreitzer R, et al. ARMageddon: Cache attacks on mobile devices//Proceedings of the USENIX Security Symposium. Austin, USA, 2016: 549-564
- [26] Aciicmez O, Koç Ç K, Seifert J-P. Predicting Secret Keys Via Branch Prediction//Proceedings of the Cryptographers' track at the RSA conference on Topics in Cryptology. San Francisco, USA, 2007: 225-242
- [27] Gras B, Razavi K, Bos H, et al. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks//Proceedings of the USENIX Conference on Security Symposium.

- Baltimore, USA, 2018; 955-972
- [28] Aldaya A C, Brumley B B, Hassan S U, et al. Port contention for fun and profit//Proceedings of the IEEE Symposium on Security and Privacy. San Francisco, USA, 2019; 870-887
- [29] Canella C, Van Bulck J, Schwarz M, et al. A systematic evaluation of transient execution attacks and defenses//Proceedings of the USENIX Security Symposium. Santa Clara, USA, 2019; 249-266
- [30] Gruss D. Transient-Execution Attacks [Habilitation Thesis]. Graz University of Technology, Australia, 2020
- [31] Van Bulck J, Minkin M, Weisse O, et al. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution//Proceedings of the USENIX Security Symposium. Baltimore, USA, 2018; 991-1008
- [32] Koruyeh E M, Khasawneh K N, Song C, et al. Spectre returns! Speculation attacks using the return stack buffer//Proceedings of the USENIX Workshop on Offensive Technologies. Baltimore, USA, 2018; 3-14
- [33] Maisuradze G, Rossow C. Ret2spec: Speculative Execution using Return Stack Buffers//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. Toronto, Canada, 2018; 2109-2122
- [34] Schwarz M, Lipp M, Moghimi D, et al. ZombieLoad: Cross-privilege-boundary data sampling//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. London, UK, 2019; 753-768
- [35] van Schaik S, Milburn A, Österlund S, et al. RIDL: Rogue in-flight data load//Proceedings of the IEEE Symposium on Security and Privacy. San Francisco, USA, 2019; 88-105
- [36] Canella C, Genkin D, Giner L, et al. Fallout: Leaking data on meltdown-resistant CPUs//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. London, UK, 2019; 769-784
- [37] Moghimi D, Lipp M, Sunar B, et al. Medusa: Microarchitectural data leakage via automated attack synthesis//Proceedings of the USENIX Security Symposium. Online, 2020; 1427-1444
- [38] Ragab H, Milburn A, Razavi K, et al. CrossTalk: Speculative data leaks across cores are real//Proceedings of the IEEE Symposium on Security and Privacy. San Francisco, USA, 2021; 1852-1867
- [39] Hund R, Willems C, Holz T. Practical timing side channel attacks against kernel space ASLR//Proceedings of the IEEE Symposium on Security and Privacy. Berkeley, USA, 2013; 191-205
- [40] Evtvushkin D, Ponomarev D, Abu-Ghazaleh N. Jump over ASLR: Attacking branch predictors to bypass ASLR//Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture. Taipei, China, 2016; 1-13
- [41] Gras B, Razavi K, Bosman E, et al. ASLR on the line: Practical cache attacks on the MMU//Proceedings of the Annual Network and Distributed System Security Symposium (NDSS). San Diego, USA, 2017; 17-26
- [42] Lipp M, Gruss D, Schwarz M. AMD prefetch attacks through power and time//Proceedings of the USENIX Security Symposium. Boston, USA, 2022; 643-660
- [43] Gruss D, Lipp M, Schwarz M, et al. KASLR is dead; Long live KASLR//Proceedings of the Engineering Secure Software and Systems (ESSoS). Bonn, Germany, 2017; 161-176
- [44] Gens D, Arias O, Sullivan D, et al. LAZARUS: Practical side-channel resilient kernel-space randomization//Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses. Atlanta, USA, 2017; 238-258
- [45] Ragab H, Barberis E, Bos H, et al. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks//Proceedings of the USENIX Security Symposium. Online, 2021; 1451-1468
- [46] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. USA: Intel, Technical Report; 779559, 2023
- [47] Orenstien D, Sperber Z, Valentine R, et al. Instructions and Logic to Perform Mask Load and Store Operations. USA, Application Number: 13793529, 2013, 7, 25
- [48] Orenstien D, Sperber Z, Valentine R, et al. Instructions and Operations to Perform Mask Load and Store Operations as Sequential or One-at-a-time Operations after Exceptions and for Un-Cacheable Type Memory[Patent]. USA, Patent Number: 10120684, 2018, 11, 6



**LI Dan-Ping**, Ph.D. candidate, associate researcher. Her main research interests include processor security and microarchitectural side-channel attacks.

**ZHU Zi-Yuan**, Ph.D., professor. His main research interests include chip security and computer architecture.

**SHI Gang**, Ph.D., professor. His main research interests include computer architecture and embedded system security.

**MENG Dan**, Ph.D., professor. His main research interests include computer architecture, cloud computing, network and system security.

## Background

Processor security is one of the research hotspots in the field of chip security. Currently, a large amount of research works have shown that there are security vulnerabilities in processors such as cache side-channel attacks and transient execution attacks. These vulnerabilities mainly exploit parallel optimization technologies introduced at the processor micro-architecture level.

Single Instruction stream, Multiple Data streams (SIMD) is a parallel optimization technology that utilizes data-level parallelism to improve processor performance which has been widely used in modern processors such as Intel and AMD processors. To support SIMD technology, mainstream processor manufacturers have introduced SIMD instruction set extensions such as MMX (MultiMedia eXtensions), SSE (Streaming SIMD Extensions), and AVX (Advanced Vector eXtensions) into their processors and have implemented SIMD technology at the microarchitecture level. However, there is a lack of in-depth research on the security of SIMD technology.

This paper focuses on the VMASKMOV instruction in the SIMD instruction set extension on x86 processors and conducts an in-depth research on its characteristics and security with reverse engineering experiments. This paper analyzes the implementation details of the VMASKMOV instruction with hardware performance counters and other techniques and summarizes the characteristics of this instruction based on experimental results. Then this paper proposes a new processor

microarchitectural attack named EvilMask. It also proposes three EvilMask attack primitives: VMASKMOVL + Time (MAP), VMASKMOVS + Time (XD), and VMASKMOVL + MDS, which can be used to implement de-address space randomization attacks and process data leakage attacks. This paper also gives two attack examples based on EvilMask: (1) using VMASKMOVL + Time (MAP) and VMASKMOVS + Time (XD) to break the KALSR (Kernel Address Layout Space Randomization) on both Intel and AMD processors; (2) injecting the VMASKMOVL gadget into the Linux kernel and using VMASKMOVL + MDS to leak the data in kernel process. At the end of this paper, it discusses the countermeasures for EvilMask and points out that the most fundamental solution is to re-implement the VMASKMOV instruction at the hardware level, then gives a preliminary implementation. This paper emphasizes that in the post Meltdown and Spectre era, there are still unexplored security vulnerabilities at the processor microarchitecture level and the security of instruction parallel technology is a new research topic worthy of in-depth research.

This work is supported by the Strategic Priority Research Program of Chinese Academy of Sciences (No. XDC02010400). This project aims to evaluate the security of existing computer architectures and promote the proposal for more secure computer architectures.