简译版

Meltdown 攻击

非官方中文译文·安天技术公益翻译组 译注

文档信息			
原文名称	Meltdown		
原文作者	Moritz Lipp, Thomas Prescher 等人	**原文发布日期**	2018 年 1 月
作者简介	Moritz Lipp 是格拉茨技术大学信息安全学院的博士生。 https://twitter.com/mlqxyz		
原文发布单位	格拉茨技术大学，Cyberus 技术有限公司等		
原文出处	https://meltdownattack.com/meltdown.pdf		
译者	安天技术公益翻译组	**校对者**	VMware 解决方案架构师臧铁军，安天微电子与嵌入式安全研发中心，安天安全研究与应急处理中心
免责声明	• 本译文译者为安天实验室工程师，本文系出自个人兴趣在业余时间所译，本文原文来自互联网的公共方式，译者力图忠于所获得之电子版本进行翻译，但受翻译水平和技术水平所限，不能完全保证译文完全与原文含义一致，同时对所获得原文是否存在臆造、或者是否与其原始版本一致未进行可靠性验证和评价。 • 本译文对应原文所有观点亦不受本译文中任何打字、排版、印刷或翻译错误的影响。译者与安天实验室不对译文及原文中包含或引用的信息的真实性、准确性、可靠性、或完整性提供任何明示或暗示的保证。译者与安天实验室亦对原文和译文的任何内容不承担任何责任。翻译本文的行为不代表译者和安天实验室对原文立场持有任何立场和态度。 • 译者与安天实验室均与原作者与原始发布者没有联系，亦未获得相关的版权授权，鉴于译者及安天实验室出于学习参考之目的翻译本文，而无出版、发售译文等任何商业利益意图，因此亦不对任何可能因此导致的版权问题承担责任。 • 本文为安天内部参考文献，主要用于安天实验室内部进行外语和技术学习使用，亦向中国大陆境内的网络安全领域的研究人士进行有限分享。望尊重译者的劳动和意愿，不得以任何方式修改本译文。译者和安天实验室并未授权任何人士和第三方二次分享本译文，因此第三方对本译文的全部或者部分所做的分享、传播、报道、张贴行为，及所带来的后果与译者和安天实验室无关。本译文亦不得用于任何商业目的，基于上述问题产生的法律责任，译者与安天实验室一律不予承担。		

译者小序

鉴于 Meltdown（熔断）漏洞的严重性和漏洞的复杂性足以动摇全球云计算基础设施的根基，为了让管理部门和客户深入了解该漏洞的机理和威胁，安天公益翻译组，在安天微电子与嵌入式安全研发中心和安天安全研究与应急处理中心的支持下，针对披露该漏洞的关键论文“Meltdown”进行了翻译。鉴于文章篇目较长、威胁形势非常紧急，我们通过不足 10 个小时完成的翻译肯定有很多错误，希望大家予以指出，我们会不断补充完善。勘误交流地址，傲气安天 BBS 公益翻译板块，详见：
http://bbs.antiy.cn/forum.php?mod=viewthread&tid=77670

鸣谢

感谢 VMware 解决方案架构师臧铁军连夜协助校对本译文。

Meltdown 攻击

Moritz Lipp[1]，Michael Schwarz[1]，Daniel Gruss[1]，Thomas Prescher[2]，Werner Haas[2]，
Stefan Mangard[1]，Paul Kocher[3]，Daniel Genkin[4]，Yuval Yarom[5]，Mike Hamburg[6]

1 格拉茨技术大学

2 Cyberus 技术有限公司

3 独立研究人员

4 宾夕法尼亚大学和马里兰大学

5 阿德莱德大学和 Data61

6 加密研究部门 Rambus

摘要

计算机系统的安全性从根本上依赖于内存隔离，例如，内核地址范围被标记为不可访问，并且受到访问保护。在这篇文章中，我们将介绍 Meltdown 攻击。Meltdown 利用现代处理器乱序执行的副作用来读取任意内核内存位置，包括个人数据和密码。乱序执行是不可或缺的性能特征，并且存在于各种现代处理器中。该攻击独立于操作系统，并不依赖于任何软件漏洞。Meltdown 破坏了地址空间隔离和半虚拟化环境所提供的所有安全假设，从而打破了建立在此基础上的每个安全机制。在受影响的系统上，Meltdown 使攻击者可以在没有任何许可或权限的情况下读取同一系统中的其他进程或同一主机上的其它虚拟机的内存，从而影响数百万客户以及几乎所有 PC 用户。我们证明，KASIS[8]的 KAISER 防御机制具有阻止 Meltdown 的重要（歪打正着）副作用。我们强调必须立即部署 KAISER，以防止大规模的信息泄露。

1 简介

内存隔离是当今操作系统的主要安全特征之一。操作系统确保用户应用程序不能访问彼此的内存，并防止用户应用程序读取或写入内核内存。这种隔离是计算环境的基石，允许在个人设备上运行多个应用程序，或者在云中的单台计算机上执行多个用户的进程。

在现代处理器上，内核和用户进程之间的隔离通常由处理器的一个监视位（supervisor bit）来实现，该监视位定义内核的内存页面是否可以被访问。其基本思想是，只有在进入内核代码时才能设置该位，切换到用户进程时该位被清除。该硬件特征允许操作系统将内核映射到每个进程的地址空间，并且从用户进程到内核具有非常高效的转换，例如用于中断处理。因此，在实践中，从用户进程切换到内核时不会改变内存映射。

在本文中，我们将介绍 Meltdown 攻击。Meltdown 是一种新型的攻击方法，可以完全突破内存隔离，为任何用户程序提供一个简单的方法来读取它在其中执行的机器的

整个内核内存，包括映射到内核区域的所有物理内存。Meltdown 不利用任何软件漏洞，即它适用于所有主要的操作系统。相反，Meltdown 利用了大多数现代处理器（例如 2010 年后的英特尔微体系结构和其他厂商的其他 CPU）上的侧信道信息。

侧信道攻击通常需要了解目标应用程序的精确信息，而且是为获取泄漏机密信息量身定做的，但是 Meltdown 允许能够在存在漏洞的处理器上运行代码的恶意访问者获得整个内核地址空间，包括任何映射的物理内存。 Meltdown 如此简单而强大的根本原因是乱序执行造成的副作用。

乱序执行是当今处理器的一个重要的性能特征，以便克服繁忙的执行单元的延迟，例如，内存读取单元需要等待从内存读取数据。现代处理器并没有延迟执行，而是乱序地执行操作，即它们预见性地将后续操作安排到处理器的空闲执行单元。然而，这样的操作通常会有不必要的副作用，例如，时序差异[28，35，11]可能会泄漏来自顺序执行和乱序执行的信息。

从安全角度来看，一个发现特别重要：乱序，存在漏洞的 CPU 允许非特权进程将数据从特权（内核或物理）地址加载到临时 CPU 寄存器中。而且，CPU 甚至根据该寄存器值进一步执行计算，例如，根据寄存器值访问数组。如果某个指令不应该被执行，处理器就会通过简单地丢弃内存查找的结果（例如，修改的寄存器状态）来确保正确的程序执行。因此，在体系结构层面（例如，处理器如何执行计算的抽象定义），不会出现安全问题。

然而，我们发现乱序内存查找会影响缓存，而缓存又可以通过缓存侧信道进行检测。结果，通过在乱序执行流中读取特权内存，攻击者就能把整个内核内存转储出来，并利用微结构的隐蔽通道在这种微妙的状态下把数据传送到外部。在隐蔽通道的接收端，寄存器值被重构。因此，在微体系结构层面（例如，实际的硬件实现），存在可利用的安全问题。

Meltdown 打破了 CPU 的内存隔离能力给出的所有安全假设。我们评估了该攻击对现代台式机和笔记本电脑，以及云中的服务器的影响。Meltdown 允许非特权进程读取映射到内核地址空间的数据，包括 Linux 和 OS X 的整个物理内存以及 Windows 的大部分物理内存。这可能包括其他进程和内核的物理内存；在内核共享沙箱解决方案（例如 Docker，LXC）、Xen 超虚拟模式以及其他共址的情况下，包括内核（或虚拟化管理程序）的内存。虽然性能在很大程度上取决于特定的机器，例如处理器速度，TLB 和缓存大小以及 DRAM 速度，但是我们可以以高达 503KB/s 的速率转储内核和物理内存。因此，大量的系统受到影响。

KAISER[8] 最初是为了防止针对 KASLR 的侧信道攻击，歪打正着地阻止了 Meltdown 攻击。我们的评估显示，KAISER 可以在很大程度上阻止 Meltdown 攻击。因此，我们强调，立即在所有操作系统上部署 KAISER 至关重要。幸运的是，三大操作系统（Windows，Linux 和 OS X）都部署了 KAISER 变种，并将很快推出补丁。

Meltdown 在几个方面与 Spectre 攻击[19]截然不同，Spectre 需要根据受害者进程

的软件环境定制攻击，能够更广泛地应用于CPU，而且 KAISER 无法对其进行缓解。

贡献。本文的贡献如下：

1.我们介绍了一种新的、功能强大的、基于软件的侧信道——乱序执行。

2.我们展示了如何将乱序执行与微体系结构隐蔽通道相结合，在这种微妙的状态下把数据传送到外部的接收器。

3.我们介绍了将乱序执行与异常处理程序或 TSX 结合起来的端到端攻击，该攻击能够读取笔记本电脑、台式机和公共云计算机上的任意物理内存，无需任何许可或权限。

4.我们评估了 Meltdown 的性能以及 KAISER 对它的影响。

大纲。本文的其余部分结构如下：在第 2 章中，我们描述了乱序执行导致的基本问题。在第 3 章中，我们提供了一个示例，说明了 Meltdown 利用的侧信道。在第 4 章中，我们描述了整个 Meltdown 攻击的构建块。在第 5 章中，我们介绍了 Meltdown 攻击。在第 6 章中，我们评估了 Meltdown 攻击对几种系统的影响。在第 7 章中，我们讨论了基于软件的 KAISER 对策的效果，并提出了硬件解决方案。在第 8 章中，我们讨论相关的工作，并在第 9 章中总结全文。

2 背景

在本章中，我们介绍乱序执行、地址转换和缓存攻击的背景。

2.1 乱序执行

乱序执行是一种优化技术，它最大限度地利用 CPU 内核的所有执行单元。CPU 不是严格按照顺序执行处理指令的，只要需要的资源可用，就会立即执行它们。当前操作的执行单元被占用时，其他执行单元可以向前排。因此，只要结果遵循架构定义，指令就可以并行运行。

在实践中，支持乱序执行的 CPU 会在弄清楚是否允许之前推测性地执行处理器的乱序逻辑处理指令。在本文中，我们对推测性执行给出一个更具限制性的意思，它指的是分支之后的指令序列，并且使用乱序执行这个术语来指代在处理器已经提交所有先前指令的结果之前执行操作的任何方式。

在 1967 年，Tomasulo [33]开发了一种算法[33]，使指令的动态调度能够实现乱序执行。Tomasulo [33]引入了一个统一的预留站，允许 CPU 使用已经计算的数据值，而不是将其存储到寄存器并重新读取。预留站重命名寄存器，以允许在相同物理寄存器上运行的指令使用最后一个逻辑寄存器来解决写后读（RAW），读后写（WAR）和写后写（WAW）危险。此外，预留单元通过通用数据总线（CDB）连接所有的执行单元。如果操作数不可用，则预留单元可以在 CDB 上监听，直到可用，然后直接开始执行该指令。

在英特尔架构上，流水线由前端、执行引擎（后端）和内存子系统[14]组成。x86 指令由前端从内存中读取并解码为连续发送到执行引擎的微指令（μOP）。乱序执行在执行引擎中执行，如图 1 所示。重组缓存器负责寄存器分配，寄存器重命名和确认执行。此外，像数据移动消除或清零操作识别等优化可以由重组缓存器直接完成。微指令

被转发到统一预留站，将连接到执行单元的出口端口上的操作排队。每个执行单元可以执行不同的任务，如 ALU 操作、AES 操作、地址生成单元（AGU）或内存加载和存储。AGU 以及加载和存储执行单元直接连接到内存子系统以处理其请求。

由于 CPU 通常不运行线性指令流，因此它们具有分支预测单元，用于获得接下来将执行哪条指令的有根据的猜测。分支预测器在实际评估条件之前尝试确定分支的哪个方向。如果预测是正确的，那么在这条路上的指令并没有任何依赖关系可以被提前执行，并立即使用它们的结果。如果预测不正确，重组缓存器允许回滚，即通过清除重组缓存器并重新初始化统一预留站来回滚。

预测分支的各种方法：对于静态分支预测[12]，分支的结果完全基于指令本身。动态分支预测[2]在运行时收集统计数据来预测结果。一级分支预测使用 1 位或 2 位计数器记录分支的最后结果[21]。现代处理器通常使用两级自适应预测器[36]，记忆最后 n 个结果的历史记录以定期预测循环模式。最近，有人提出了使用神经分支预测的思想[34,18,32]，并将其整合到 CPU 架构中[3]。

2.2 地址空间

为了将进程彼此隔离，CPU 支持将虚拟地址空间，并负责将虚拟地址转换为物理地址。虚拟地址空间被分成一组页面，可以通过多级页面转换表逐一映射到物理内存。转换表定义了实际的虚拟到物理映射，以及用于强制执行特权检查的保护特性，例如可读、可写、可执行和用户可访问。当前使用的转换表保存在特殊的 CPU 寄存器中。在每个上下文切换中，操作系统将该寄存器更新为下一个进程的转换表地址，以便实现每个进程的虚拟地址空间。因此，每个进程只能引用属于自己虚拟地址空间的数据。每个虚拟地址空间本身被分割成用户部分和内核部分。

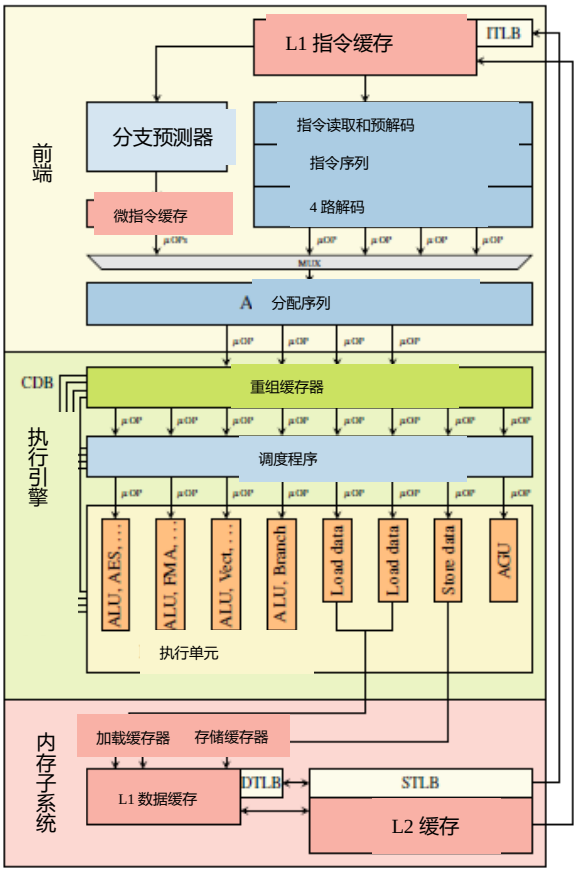

图 1：简单介绍 Intel Skylake 微体系结构的一个内核。指令被解码成微指令，并由执行单元乱序执行。

虽然用户地址空间可以被正在运行的应用程序访问，但只有当 CPU 以特权模式运行时才能访问内核地址空间。这是由禁用相应转换表的用户可访问属性的操作系统强制执行的。内核地址空间不仅为内核自己的使用而映射内存，还需要在用户页面上执行操作，例如用数据填充它们。因此，整个物理内存通常映射到内核中。在 Linux 和

OS X 系统上，这是通过直接物理映射完成的，即将整个物理内存直接映射到预定义的虚拟地址（参见图 2）。

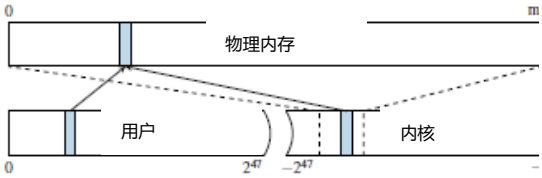

图 2: 物理内存以一定的偏移量直接映射到内核中。用户空间可访问的物理地址（蓝色）也直接映射到内核空间。

Windows 不使用直接物理映射，而是维护多个所谓的分页池、非分页池和系统缓存。这些池是内核地址空间中的虚拟内存区域，将物理页面映射到虚拟地址，或者保留在内存中（非分页池），或者可以从内存中移除，因为副本已经存储在磁盘上（分页池）。系统缓存还包含所有文件支持页面的映射。结合起来，这些内存池通常会将大部分物理内存映射到每个进程的内核地址空间。

内存损坏漏洞的利用通常需要了解特定数据的地址。为了阻止这种攻击，引入了地址空间布局随机化（ASLR）以及非可执行堆栈和 stack canaries 。为了保护内核，KASLR 随机化每次启动时驱动程序所在位置的偏移量，使得攻击变得更加困难，因为攻击者需要猜测内核数据结构的位置。但是，侧信道攻击可以检测到内核数据结构的确切位置[9，13，17]，或者在 JavaScript 中对 ASLR 进行去随机化[6]。软件漏洞和这些地址的知识的组合可以导致特权代码的执行。

2.3 缓存攻击

为了加速内存访问和地址转换，CPU 包含一些称为缓存的小内存缓冲区，用于存储常用数据。CPU 缓存通过在较小和较快的内部内存中缓存常用数据来隐藏慢速内存访问延迟。现代 CPU 具有多个级别的缓存，这些缓存或者是其内核专用的，或者是在它们之间共享的。地址空间转换表也存储在内存中，也被缓存在常规缓存中。

缓存侧信道攻击利用缓存引入的时序差异。之前，研究人员已经提出并演示了不同的缓存攻击技术，包括 Evict + Time [28]，Prime + Probe [28,29]和 Flush + Reload [35]。Flush+Reload（刷新+重新加载）攻击的颗粒度可以达到单个缓存线级别。这些攻击利用共享的最后一级缓存。攻击者经常使用 clflush 指令刷新目标内存位置。通过测量重新加载数据所需的时间，攻击者可以确定数据是否由另一个进程同时加载到缓存中。Flush + Reload 攻击已被用于各种计算的攻击，例如加密算法[35，16，1]，Web 服务器函数调用[37]，用户输入[11，23，31]和内核寻址信息[9]。

一个特殊用例是隐蔽通道。在这里，攻击者控制导致副作用的部分和测量副作用的部分。这可以用来将信息从一个安全域泄漏到另一个安全域，同时绕过架构层或以上的任何边界。Prime + Probe 和 Flush + Reload 都被用于高性能的隐蔽通道[24，26，10]。

3 简单示例

在本节中，我们将从一个示例开始，以一个简单的代码片段，说明乱序执行会改变微体系结构状态，从而成为泄漏信息的方式。尽管很简单，但是它将成为第 4 节和第 5 节的基础，在那里我们展示了如何利用这种状态改变来进行攻击。

清单 1 显示了一个简单的代码片段，首先引发一个（未处理的）异常，然后访问一个数组。异常的属性是控制流不会继续执行在异常之后的代码，而是跳转到操作系统中的异常处理程序。

```
raise_exception();
// the line below is never reached
access(probe_array[data * 4096]);
```

清单 1：一个演示乱序执行副作用的示例。

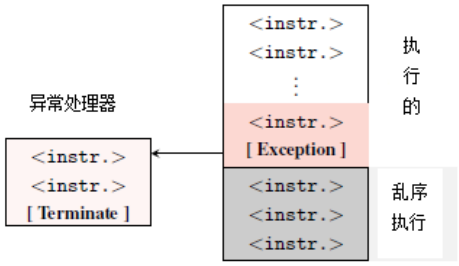

图 3：如果执行的指令导致异常，将控制流转移到异常处理程序，则不能再执行后续的指令。由于乱序执行，后续的指令可能已经部分执行了，但还没有执行完成。但是，执行的体系结构影响将会被丢弃。

不管这个异常是由于内存访问（例如：通过访问一个无效的地址）或是由于任何其他 CPU 异常（例如：一个除以 0 的运算）而引起的，控制流将在内核中继续，而不是用户空间的下一个指令。

因此，理论上说我们的简单示例程序不会访问数组，因为异常会立即陷入到内核并终止应用程序。但是，由于乱序执行，因为后续指令不存在对异常的依赖，CPU 可能已经执行了紧接着的后续指令。这在图 3 中进行了说明。由于异常情况，乱序执行的指令不会执行完成，因此不会在架构层面产生可见的影响。

尽管乱序执行的指令对寄存器或内存没有任何可见的体系结构影响，但是它们具有微体系结构的副作用。在乱序执行期间，引用的内存被读取到寄存器，并且也被缓存在 cache 中。如果乱序执行的结果必须被丢弃，不会有内容被提交到寄存器和存储器。然而，缓存的内容仍保存在 cache 中。我们可以利用一种微体系结构的侧信道攻击，例如 Flush + Reload [35]来检测特定的内存位置是否被缓存，以使该微体系结构状态可见[译者注：已经读入 Cache 的内存（缓存过的），访问速度和未读入 Cache 的不同，因此可以用读取速度测试作为一种侧信道]。还有其他的侧信道也可以用来检测是否缓存了特定的内存位置，包括 Prime + Probe [28,24,26]，Evict + Reload [23]或 Flush + Flush [10]。但是，由于 Flush + Reload 是最准确的已知 cache 侧信道，并且易于实现，所以本例中我们不考虑任何其他侧信道。

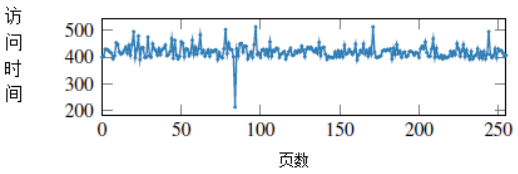

图 4：即使在乱序执行期间只访问一个内存位置，它仍然会被缓存。遍历 256 页的探针阵列，显示了一个缓存命中，正好在乱序执行期间访问的页面上。

这个示例中，当乱序执行内存访问时，将根据 data 的值访问 cache 的不同部分。由于 data 乘以了 4096，对 probe_array 的数据访问将以 4 kB 分散在数组中（假设 probe_array 为 1 B 数据类型）。因此，有一个从 data 到内存页的单射映射，也就是说，没有两个不同的 data 值可以导致访问

同一页面。因此，如果一个页面的缓存行被缓存，我们就知道 data 的值。由于预取不会跨越页面边界访问数据，因此分散到不同的页面避免了由于预取造成的误报[14]。

图 4 显示了在执行乱序代码段（data = 84）之后，Flush + Reload 度量遍历所有页面的结果。尽管由于异常，数组访问不应该发生，但是我们可以清楚地看到，被访问的索引被缓存了。遍历所有页面（例如，在异常处理程序中）仅显示第 84 页的缓存命中。

这表明，即使是从未实际执行的指令，也会改变 CPU 的微体系结构状态。第 4 章将修改这个示例，不读取数值，而是泄漏一个不可访问的秘密。

4 攻击的构建块

第 3 节中的示例表明，乱序执行的副作用可以修改微体系结构状态以泄漏信息。虽然代码片段揭示了传递给缓存端通道的数据值，但我们想要展示如何利用这种技术来泄露不可访问的秘密。在本节中，我们想概括和讨论必要的构建块，以便利用乱序执行攻击。

攻击者的目标是保存在物理内存中的某个秘密值。注意，寄存器内容也在任务切换时被存储在存储器中，即它们也被存储在物理存储器中。如第 2.2 节所述，每个进程的地址空间通常包括整个用户空间以及整个内核空间，这个空间通常也映射了所有物理内存（正在使用中）。但是，这些内存区域只能在特权模式下访问(参见第 2.2 节)。

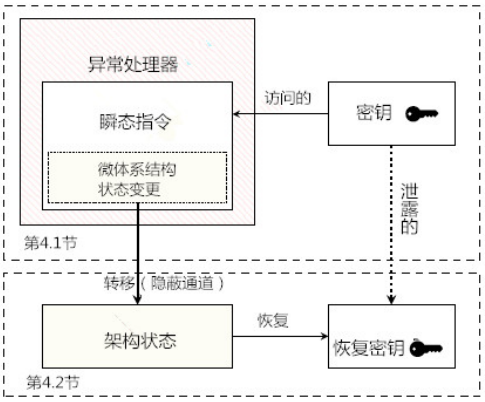

图 5：Meltdown 攻击利用异常处理或阻止手段（例如：TSX）来运行一系列临时指令[译者注：也就是不会执行完成（retired）的指令]。这些临时指令获取（持续的）秘密值，并基于该秘密值改变处理器的微体系结构状态。这构成了微体系结构隐蔽通道（译者注：即侧信道）的发送部分。接收方读取微体系结构状态，使其构建和恢复秘密价值。

在这项工作中，我们通过绕过特权模式隔离的方法来展示如何泄漏秘密数据，使攻击者能够对整个内核空间（包括映射的任何物理内存，包括任何其他进程和内核的物理内存）进行完全读取访问。注意，Kocher 等人[19]提出一种与之不同的方法，称为“幽灵攻击”，它诱骗执行指令以泄露信息，使受害者的进程被授权访问。因此，幽灵攻击缺少 Meltdown 的特权升级能力，需要针对受害者进程的软件环境进行定制，但更广泛地适用于支持推测性执行的 CPU，并且不会被 KAISER 阻止。

完整的 Meltdown 攻击由两个构建块组成，如图 5 所示。Meltdown 的第一个构建块是让 CPU 执行一条或多条在执行路径中永远不会发生的指令。在示例中（参见第 3 节），这是对一个数组的访问，通常不会

执行，因为前面的指令总是引发异常。我们把这样被乱序执行的指令称为临时指令，留下（侧信道）可见的影响。此外，我们称任何包含至少一个临时指令的指令序列为临时指令序列。

为了利用临时指令进行攻击，临时指令序列必须利用一个攻击者想要泄漏的秘密值。第 4.1 节描述构建块来运行一个临时指令序列，并依赖于一个秘密值。

Meltdown 的第二个构建块是将临时指令序列的微体系结构副作用转换为架构状态，以进一步处理泄露的秘密。因此，第 4.2 节中描述的第二构建块描述了使用隐蔽通道将微体系结构的副作用转移到架构状态的构建块。

4.1 执行临时指令

Meltdown 的第一个构建块就是执行临时指令。临时指令基本上都会发生，因为 CPU 在当前指令前连续运行，以最小化所需延迟，从而使性能最大化(参见第 2.1 节)。如果临时指令的操作取决于秘密值，则会引入可利用的侧信道。我们关注的是在攻击者的进程中映射的地址，即用户可访问的用户空间地址以及用户无法访问的内核空间地址。请注意，针对在另一个进程的上下文(即地址空间)中执行的代码的攻击是可能的[19]，但是超出了本文的范围，因为所有物理内存（包括其他进程的内存）都可以通过内核地址空间读取。

访问用户不可访问的内存分页（例如内核页面）会触发一个通常会终止应用程序的异常。如果攻击者以用户不可访问的地址为目标，则攻击者必须处理这个异常。我们提出了两种方法：在异常处理中，我们在执行临时指令序列后有效地捕获异常，或者在异常阻止的情况下，我们可以防止异常发生，并且在执行临时指令序列后重定向控制流。下面我们将详细讨论这些方法。

异常处理。一种简单的方法是在访问终止进程的无效内存位置之前，使用攻击应用程序，并且只访问子进程中无效的内存位置。CPU 在崩溃之前执行子进程中的临时指令序列。然后父进程可以通过观察微体系结构状态（例如通过侧信道）来恢复秘密。

还可以安装一个信号处理程序，如果某个异常发生，将执行该处理程序，在这个特殊的情况下是一个分段错误。这使得攻击者能够发出指令序列并防止应用程序崩溃，从而减少了支出，因为不需要创建新的进程。

异常阻止。处理异常的另一种方法是首先防止它们被提出。事务性内存允许将内存访问分组到一个看似原子操作中，如果出现错误，则允许回滚到以前的状态。如果在事务中发生异常，则架构状态将被重置，程序继续运行而不会中断。

此外，由于分支预测错误，推测执行可能不会出现在执行的代码路径上。根据前面的条件分支的指令可以进行推测。因此，无效存储器访问被放置在仅在先前分支条件评估为真时才执行的推测指令序列内。通过确保条件在执行的代码路径中从不计算为真，我们可以抑制发生的异常，因为内存访问只是推测性地执行。这种技术可能需要对分支预测器进行复杂的训练。Kocher 等人[19]在与 Meltdown 不同的方法中试图实现该方法，因为这一结构经常可以在其他进程

的代码中找到。

4.2 构建隐蔽通道

Meltdown 的第二个构建块是将由临时指令序列更改为一个架构状态(参见图 5)的微体系结构状态的转移。临时指令序列可以看作微体系结构隐蔽通道的发送端。隐蔽通道的接收端接收微体系结构状态变化，并从状态推导出秘密。请注意，接收器不是临时指令序列的一部分，它可以是不同的线程，甚至是不同的进程，例如 fork-and-crash 方法中的父进程。

我们利用来自缓存攻击的技术，因为缓存状态是一种微体系结构状态，可以使用各种技术可靠地在 CPU 和计算机宏观体系结构状态中反映出来[28,35,10]。具体而言，我们利用 Flush + Reload[35]，因为它能构建一个快速且低噪的隐蔽通道。 因此，临时指令序列（参见 4.1 节）根据秘密值执行规则的存储器访问，例如，像在示例中一样（参见第 3 章)。

在临时指令序列访问可访问地址之后，即：这是隐蔽通道的发送者；该地址被缓存以用于随后的访问。然后，接收器可以通过测量地址的访问时间来监视地址是否已经加载到缓存中。因此，发送者可以通过访问被加载到所监视的 cache 中的地址来发送“1”位，并且通过不访问这样的地址来发送“0”位。

使用多个不同的 cache 线，就像我们在第 3 节中的示例一样，允许一次传输多个比特。对于 256 个不同字节值中的每一个，发送者都访问不同的缓存行。通过对所有 256 个可能的缓存行执行 Flush + Reload 攻击，接收者可以恢复一个完整的字节而不是一个比特。然而，由于 Flush + Reload 攻击比临时指令序列花费更长的时间 (通常为几百个周期)，所以一次只发送一个比特就更有效率。攻击者可以简单地通过相应地转移和屏蔽秘密值来做到这一点。

请注意，隐蔽通道不限于依赖缓存的微体系结构状态。任何可以被指令（序列）影响并且可以通过侧信道观察到的微体系结构状态可被用来构建隐蔽通道的发送端。例如，发送者可以发出占用某个执行端口（如 ALU）的指令（序列）来发送“1”位。接收器在同一执行端口上执行指令（序列）时测量延迟。高延迟意味着发送者发送“1”位，而低延迟意味着发送者发送“0”位。Flush + Reload 缓存隐蔽通道的优点是抗噪声和高传输速率[10]。此外，泄漏可以从任何 CPU 核心[35]观察到，即重新调度事件不会显著影响隐蔽通道。

5 Meltdown

在本节中，目前的 Meltdown 是一个强大的攻击，允许从一个无特权的用户程序中读取任意的物理内存，这个程序由第四节提供的构建块组成。首先，我们讨论攻击设置以强调这种攻击的广泛适用性。其次，我们提供了一个攻击概述，展示了如何将 Meltdown 安装在个人电脑的 Windows 和 Linux 上以及云中。最后，我们讨论 Meltdown 的具体实现，允许以高达每秒 503 KB 的速度转储内核内存。

攻击设置。在我们的攻击中，我们考虑了云中的个人电脑和虚拟机。在攻击场景中，攻击者在被攻击的系统上执行任意非特权

的代码，即攻击者可以以普通用户的权限运行任何代码。 但是，攻击者没有对机器进行物理访问。此外，我们假设系统受到诸如 ASLR 和 KASLR 等先进软件防御以及 SMAP、SMEP、NX 和 PXN 等 CPU 特性的充分保护。最重要的是，我们假设一个完全没有 bug 的操作系统，因此，不存在可以利用的软件漏洞来获得内核特权或泄漏信息。攻击者针对的是秘密用户数据，例如密码和私钥，或其他任何有价值的信息。

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

清单 2：Meltdown 的核心指令序列。一个无法访问的内核地址被移动到一个寄存器，引发一个异常。随后的指令在引发异常之前已经被乱序执行，通过间接内存访问泄漏内核地址的内容。

5.1 攻击描述

Meltdown 结合了第 4 节讨论的两个构建块。首先，攻击者使 CPU 执行一个临时指令序列，该序列使用存储在物理存储器某处的不可访问的秘密值（参见 4.1 节）。临时指令序列充当隐蔽信道的发送器（参见 4.2 节），最终将秘密值泄露给攻击者。

Meltdown 包括 3 个步骤：

第 1 步：攻击者所选的内存位置的内容被加载到寄存器中，这是攻击者无法访问的。

第 2 步：临时指令根据寄存器的秘密内容访问缓存行。

第 3 步：攻击者使用 Flush+Reload 来确定被访问的 cache 线路，因此秘密存储在选定的内存位置。

通过对不同的内存位置重复这些步骤，攻击者可以转储内核内存，包括整个物理内存。

清单 2 显示了使用 x86 汇编指令的临时指令序列的基本实现和隐蔽通道的发送部分。请注意，攻击的这一部分也可以完全用 C 语言等高级语言实现。下面我们将讨论 Meltdown 的每一步骤以及清单 2 中的相应代码行。

第 1 步：读取秘密。要将数据从主存储器加载到寄存器中，主存储器中的数据将使用虚拟地址进行引用。在将虚拟地址转换为物理地址的同时，CPU 还检查虚拟地址的许可位，即该虚拟地址是用户可访问的还是只能由内核访问。正如在第 2.2 节中讨论的那样，通过一个许可位进行基于硬件的隔离被认为是安全的，并且被硬件厂商推荐。因此，现代操作系统总是将整个内核映射到每个用户进程的虚拟地址空间。

因此，所有内核地址在翻译时都会生成一个有效的物理地址，并且 CPU 可以访问这些地址的内容。访问用户空间地址的唯一区别是 CPU 引发异常，因为当前的权限级别不允许访问这样的地址。因此，用户空间不能简单地读取这样的地址内容。然而，Meltdown 利用了现代 CPU 的乱序执行，它仍然在非法内存访问和引发异常之间的小时间窗口中执行指令。

在清单 2 的第 4 行中，我们将存储在 RCX 寄存器中的目标内核地址的字节值加

载到由 AL 表示的 RAX 寄存器最低有效字节中。2.1 节中含有更详细的解释，MOV 指令由内核提取，解码成微指令，分配并发送到重排缓冲器。架构寄存器（例如清单 2 中的 RAX 和 RCX）被映射到底层物理寄存器，从而实现乱序执行，试图尽可能多地利用流水线。随后的指令（第 5-7 行）已经被解码并分配为微指令。当操作等待对应的执行单元执行时，微指令会被发送到保存微指令的保留站。如果执行单元已经被使用到相应的容量，或操作数的值还未被计算出来，微指令的执行就会被延迟。

当内核地址被加载到第 4 行时，CPU 很可能已发出后续指令，使其作为乱序执行的一部分，相应的微指令在保留站中等待内核地址到达。只要在公共数据总线上观察到数据，微指令就可以开始执行。

当微指令完成执行时，它们按顺序退出，在退出期间，也将继续处理执行指令期间发生的任何中断和异常。因此，如果加载内核地址的 MOV 指令退出，则可进行注册异常并刷新流水线，以消除后续乱序执行所产生的不良后果。然而，这一点与我们下文所描述的攻击步骤 2 之间存在竞争。

正如 Gruss 等人所报道的那样[9]，预取内核地址有时会成功，有时会失败。我们发现预取内核地址可以提高攻击性能。

步骤 2：传输秘密值

步骤 1 中不按顺序执行的指令序列必须以其成为临时指令序列的方式进行选择。如果在 MOV 指令退出之前执行这个临时指令序列（即引发异常），且临时指令序列根据秘密值执行计算，则会将其秘密值传输给攻击者。

如前所述，我们利用缓存攻击，使用 CPU 的缓存构建低噪隐蔽信道。因此，临时指令序列必须将秘密值编码为微架构缓存状态，类似于第 3 节中的“玩具”示例。

我们在内存中分配一个探测数组。为了传输秘密值，临时指令序列包含基于秘密（不可访问）值计算的地址间接存储访问。在清单 2 的第 5 行中，来自步骤 1 的选择值乘以页面大小，即 4KB。秘密值的乘法确保了对阵列的相互作用仍具有很大的空间距离。这可以防止硬件预取器将相邻的内存位置加载到缓存中。在这里，我们一次读取一个字节，假设有 4 KB 的页面，我们的探测阵列是 256×4096 字节。

请注意，在乱序执行中，我们对寄存器值“0”有一个噪声偏差。我们在 5.2 节讨论这个原因。但是，由于这个原因，我们在临时指令序列中引入了一个重试逻辑。如果我们读取“0”，则需要尝试再次读取密码（步骤 1）。在第 7 行中，多重秘密值被添加到阵列基地址，形成隐蔽信道的目标地址。读取该地址来缓存相应的缓存行。临时指令序列会根据在步骤 1 中读取的秘密值影响缓存状态。

由于步骤 2 中的临时指令序列对引发

异常造成阻碍，因此减少步骤 2 的运行时间可以显着提高攻击的性能。例如，阵列的地址转换被缓存在 TLB 中会增加对某些系统的攻击性能。

第 3 步：接收秘密值

在步骤 3 中，攻击者通过利用将 cache 状态（步骤 2）转移回侧信道攻击（即微体系结构隐蔽信道的接收端）来恢复秘密值（步骤 1）。正如在第 4.2 节中讨论的那样，Meltdown 依靠 Flush + Reload 将缓存状态转换为体系结构状态。

当步骤 2 的临时指令序列被执行时，探测器阵列中被缓存的缓存行的位置仅取决于在步骤 1 中读取的秘密值。因此，攻击者遍历探测器阵列的所有 256 个页面，并测量每个 cache 的访问时间（即偏移量）。缓存行的页面编号直接对应于秘密值。

通过重复 Meltdown 的所有 3 个步骤，攻击者可以遍历所有不同的地址来转储整个内存。但是，由于对内核地址的内存访问引发程序终止，我们使用 4.1 节中描述的方法之一来处理或控制异常情况。

由于所有主要操作系统通常也将整个物理内存映射到每个用户进程的内核地址空间（参见 2.2 节），所以 Meltdown 不仅限于读取内核内存，还能够读取整个物理内存。

5.2 优化和限制

0 的情况

如果从难以访问的内核地址读取时触发异常，那么数据应该被存储的寄存器清零。因为如果异常未处理，则用户空间应用程序被终止，并且来自不可访问的内核地址的值可以在存储在崩溃进程的核心转储中的寄存器中。解决这个问题的直接方法是将相应的寄存器清零。如果寄存器清零快于后续指令的执行（清单 2 中的第 5 行），则攻击者可能会在第三步中读取一个错误的值。为了防止临时指令序列继续有一个错误的值，即'0'，Meltdown 重试读取地址，直到它遇到一个不同于'0'（第 6 行）的值。由于临时指令序列在引发异常之后终止，所以如果秘密值为 0，则不存在 cache 访问。因此，如果完全没有 cache 命中，则 Meltdown 假定秘密值确实为“0”。

循环由不为 0 的读取值或由非法内存访问引发的异常终止。请注意，这个循环不会显着减慢攻击速度，因为在任何情况下，处理器都会在非法内存访问之前运行，无论是循环还是线性控制流。在任何一种情况下，从异常处理或异常控制返回的控制流时间在没有无循环情况下都保持不变。因此，事先读取“0”并尽早恢复，极大地提高了速度。

单比特传输详见 5.1 节，攻击者立即通过隐蔽信道传输 8 比特，并执行 28 = 256 次刷新+重新加载测量来进行恢复。但是，在执行更多的临时指令序列和执行更多的刷新+重新加载测量之间有一个明显的折衷，攻击者可以通过隐藏信道在一次传输中传

输任意数量的比特，通过使用 MOV 指令读取更多的比特，从而获得更大的数据值。此外，攻击者可以使用临时指令序列中的附加指令来进行屏蔽。我们发现，临时指令序列中的附加指令数量对攻击的性能仅产生微不足道的影响。

上面描述的通用攻击的性能瓶颈是指用于刷新+重新加载测量上的时间。实际上，几乎整个时间都将用于 Flush + Reload 测量上。通过仅发送一个比特，我们可以省略除 Flush + Reload 测量之外的所有测量，即在缓存行 1 上的测量。如果发送的比特是“1”，则观察缓存行 1 上的 cache 命中。否则，我们观察不到缓存行 1 上的 cache 命中。

一次只发送一个位也有其固有的缺点。如果我们一次读取和传输多个数据，则对于实际用户数据来说，所有比特为“0”的可能性可能相当小。单个位为“0”的可能性通常接近 50%。因此，一次读取和传送的位数是隐式错误减少与隐蔽信道整体传输速率之间的折衷。

但是，由于这两种情况下的误码率都很小，所以我们的评估（参见第 6 章）是基于单比特传输机制。

使用 Intel TSX 的异常控制。在第 4.1 节中，我们讨论了防止由于无效的内存访问而引起异常的选项。使用英特尔 TSX（一种硬件事务存储实现），我们可以完全避免这种情况[17]。使用 Intel TSX，可以将多条指令分组到一个事务处理，即全部执行或者完全不执行指令。如果事务中的一条指令失败，已经执行的指令将被恢复，但不会产生异常。

如果我们用清单 2 中的代码封装这样的 TSX 指令，任何异常都会被检测到。然而，微体系结构的效果仍然是可见的，即 cache 状态在硬件中被持久地操纵[7]。这会导致更高的信道能力，因为控制异常的速度明显快于陷入内核以处理异常，并在之后继续执行的速度。

KASLR 处理

2013 年，内核地址空间布局随机化（KASLR）引入了 Linux 内核（从版本 3.14 [4]开始），允许在引导时随机化内核代码的位置。但是，直到 2017 年 5 月，KASLR 才在 4.12 版本中默认启用[27]。使用 KASLR 时，直接的物理映射也是随机的，因此不会固定在某个地址，这样攻击者在安装 Meltdown 攻击之前需要获得随机偏移。但是需要注意的是，随机化被限制在 40 位。

因此，如果我们假设目标机器的设置 8 GB 的 RAM，以 8 GB 的测试地址空间就足够了。这允许在最坏的情况下，仅用 128 次测试覆盖 40 位的搜索空间。如果攻击者能够从测试地址成功获取值，则攻击者可以从该位置继续转储整个内存。这样即使在几秒钟之内受到 KASLR 的保护，也可以在系统上安装 Meltdown。

6 评估

在本节中，我们将评估 Meltdown，从而实现一些性能。第 6.1 节讨论 Meltdown 可能泄漏信息，第 6.2 节评估 Meltdown 的性能，包括对策。最后，我们在 6.4 节讨论 AMD 和 ARM 的局限性。

表 1 显示了我们成功复制 Meltdown 的配置列表。对于 Meltdown 的评估，我们使用了笔记本电脑以及带有 Intel Core CPU 的台式电脑。对于云安装，我们测试了运行在 Amazon Elastic Compute Cloud 和 DigitalOcean 上的 Intel Xeon CPU 上运行的虚拟机中的 Meltdown。请注意，出于道德方面的原因，我们并没有在指向其他租户的物理内存的地址上使用 Meltdown。

6.1 信息泄漏和环境

我们在 Linux（比较 6.1.1 节）和 Windows 10（比较 6.1.3 节）上评估了 Meltdown。在这两个操作系统上，Meltdown 都可以成功泄漏内核内存。此外，我们还评估了 KAISER 补丁对 Linux 上的 Meltdown 的影响，表明 KAISER 可以防止内核内存泄漏（参见 6.1.2 节）。最后，我们讨论在 Docker 等容器中运行时的信息泄漏（参见 6.1.4 节）。

6.1.1 Linux

我们成功地评估了 Linux 内核多个版本（从 2.6.32 到 4.13.0）的 Meltdown。在所有这些版本的 Linux 内核中，内核地址空间也映射到用户地址空间。因此，所有内核地址也映射到用户空间应用程序的地址空间，但是由于这些地址的权限设置，任何访问都被阻止。当 Meltdown 绕过这些权限设置时，如果内核基地的虚拟地址是已知的，则攻击者可以泄漏完整的内核内存。由于所有主要的操作系统也将整个物理内存映射到内核地址空间（参见 2.2 节），所有的物理内存也可以被读取。

在内核 4.12 之前，内核地址空间布局随机化（KASLR）在默认情况下是静止的[30]。如果 KASLR 处于活动状态，则通过搜索地址空间，仍然可以使用 Meltdown 来查找内核（参见 5.2 节）。攻击者也可以通过遍历虚拟地址空间来简单地去除随机的直接物理地图。没有 KASLR，直接物理映射从地址 0xffff 8800 0000 0000 开始，并线性映射整个物理内存。在这样的系统中，攻击者可以使用 Meltdown 来转储整个物理内存，只需读取从 0xffff 8800 0000 0000 开始的虚拟地址即可。

表 1：实验设置

环境	CPU 模式	内核
实验室	Celeron G540	2
实验室	Core i5-3230M	2
实验室	Core i5-3320M	2
实验室	Core i7-4790	4
实验室	Core i5-6200U	2
云	Xeon E5-2676 v3	12
云	Xeon E5-2650 v4	12

在较新的系统中，默认情况下 KASLR

处于活动状态，直接物理映射的随机化被限制为 40 位。由于映射的线性，它甚至更受限制。假设目标系统至少有 8 GB 的物理内存，攻击者可以以 8 GB 的步长测试地址，最多可以测试 128 个内存位置。攻击者可以再次转储整个物理内存。

因此，出于评估目的，我们可以假设随机化被禁用，或者在预计算步骤中已检索到偏移量。

6.1.2 Linux 与 KAISER 补丁

Gruss等人的KAISER补丁[8]实现了内核和用户空间之间更强的隔离。KAISER 不会映射用户空间中的任何内核内存，除了 x86 架构所需的某些部分（例如中断处理程序）外。因此，在用户空间中没有有效的映射到内核内存或物理内存（通过直接物理映射），这些地址就不能被解析。除了必须映射到用户空间的少数内存位置外，Meltdown 不会泄漏任何内核或物理内存。

我们证实 KAISER 确实可以防止崩溃，并且没有造成任何内核或物理内存的泄漏。

如果 KASLR 处于活动状态，并且剩余的少量内存位置是随机的，则由于它们的几千字节的小尺寸，找到这些内存位置则变得毫无意义。第 7.2 节从安全性角度讨论了这些映射内存位置的含义。

6.1.3 微软 Windows

我们成功评估了最新的 Microsoft Windows 10 操作系统上的 Meltdown。根据 Linux 上的结果（参见 6.1.1 节），Meltdown 也可以在 Windows 上泄漏任意内核内存。这并不奇怪，因为 Meltdown 不会利用任何软件问题，它利用的是硬件问题。

与 Linux 相比，Windows 不具有身份映射的概念，该映射将物理内存线性映射到虚拟地址空间。相反，很大一部分物理内存映射到页面缓冲池、非分页缓冲池和系统 cache 中。此外，Windows 也将内核映射到每个应用程序的地址空间。因此，Meltdown 可以读取内核地址空间中映射的内核存储器，即没有被交换的内核的任何部分，以及分页和非分页池中映射的任何页面以及系统 cache。

请注意，一个物理页面映射到一个进程中，但却不在另一个进程的（内核）地址空间中，即不能使用 Meltdown 攻击的物理页面。但是，大部分的物理内存仍然可以通过 Melt down 来访问。

我们成功地使用 Meltdown 读取了 Windows 内核的二进制文件。为了验证泄漏的数据是实际的内核内存，我们首先使用 Windows 内核调试器来获取包含实际数据的内核地址。泄漏数据后，我们再次使用 Windows 内核调试器将泄漏数据与实际内存内容进行比较，确认 Meltdown 可以成功泄漏内核内存。

6.1.4 容器

我们评估了共享内核的容器（包括

Docker， LXC 和 OpenVZ）中运行的 Meltdown，并发现其可以毫无限制地安装攻击。在容器中运行 Meltdown 不仅泄漏底层内核的信息，还可以泄漏运行在同一主机上的其他信息。

大多数解决方案的共同之处在于每个容器使用相同的内核，即内核在所有容器之间共享。因此，通过共享内核的直接物理映射，每个容器都具有整个物理内存的有效映射。特别是对于 Intel TSX，只有没有特权的指令才会被执行。

因此，共享内核的容器隔离在使用 Meltdown 的情况下会被完全损毁。对于廉价的托管服务提供商而言，这一点尤为重要，用户不能通过完全虚拟化的机器进行隔离，而只能通过容器进行隔离。攻击是在此设置下进行的：在我们的控制之下，泄漏来自不同用户容器的内存内容。

6.2 Meltdown

为了评估 Meltdown 的性能，我们从内核内存泄露了已知的值。这使我们不仅可以确定攻击者泄漏内存的速度有多快，而且还可以确定错误率，即预期有多少字节的错误。我们的平均阅读率高达 503 KB/s，使用异常阻止时错误率低至 0.02%。在性能评测方面，我们专注于 Intel Core i7-6700K，因为它支持 Intel TSX，以便在异常处理和异常阻止之间进行公平的性能比较。

如第 5 节所述，我们的所有测试都使用了 Flush+Reload 作为隐藏通道来泄漏内存。我们评估了异常处理和异常阻止（参见 4.1 节）的性能。对于异常处理，我们使用了信号处理程序，如果 CPU 支持它，我们还使用 Intel TSX 异常阻止。Kocher 等人对使用条件分支的异常阻止进行了广泛评估[19]。为避免文章冗长，我们在本文中省略了对这一评估的描述。

6.2.1 异常处理

异常处理是更为通用的实现，因为它不依赖于任何 CPU 扩展，因此可以不受任何限制地使用。 对异常处理的唯一要求是操作系统支持捕获分段错误并在此之后继续操作。所有现代操作系统都是如此，尽管操作系统的具体实现不同。在 Linux 上，我们使用了信号，而在 Windows 上，我们依赖于结构化异常处理程序。

通过异常处理，当泄漏 12MB 内核内存时，我们的平均读取速度达到了 123KB/s。在 12MB 内核数据中，只有 0.03%被错误地读取。因此，错误率为 0.03%，信道容量为 122KB/s。

6.2.2 异常阻止

可以使用条件分支或使用 Intel TSX 来实现异常阻止。Kocher 等人对条件分支进行了详细的描述[19]，因此我们在此只 Intel TSX 进行异常阻止评估。与异常处理相比，Intel TSX 不需要操作系统支持，因为它是指令集扩展。但是，由于 Intel TSX 是一个非常新的扩展，只能在最新的 Intel CPU 上

被使用，也就说自 Broadwell 微体系结构以来的 CPU 上。

我们再次泄露了 12MB 的内核内存来测量性能。通过异常阻止，我们的平均读取速度达到了 503KB/s。此外，异常阻止的错误率为 0.02%，甚至低于异常处理的错误率。因此，我们通过异常阻止获得的信道容量是 502 KB / s。

6.3 攻击中的 Meltdown

列表 3 中显示了在运行的 Ubuntu 16.10 和 Linux 内核 4.8.0 的 Intel Core i7-6700K 上利用 Meltdown 进行内存转储。在该示例中，我们可以识别对运行在机器上的 Web 服务器的请求的 HTTP 报头。“XX”代表在侧信道未产生任何结果的字节，即无 Flush+Reload 攻击。攻击的额外重复可能仍然能够读取这些字节。

```
79cbb30: 616f 61 4e 6b 32 38 46 31  34 67 65 68 61 7a 34 |aoaNk28F14gehaz4|
79cbb40: 5a74 4d 79 78 68 76 41 57  69 69 63 77 59 62 61 |ZtMyxhvAWiicwYba|
79cbb50: 356a 4c 76 4d 70 4b 56 56  32 4b 6a 37 4b 5a 4e |5jLvMpKVV2Kj7KZN|
79cbb60: 6655 6c 6e 72 38 64 74 35  54 62 43 63 7a 6f 44 |fUlnr8dt5TbCczoD|
79cbb70: 494e 46 71 58 6d 4a 69 34  58 50 39 62 43 53 47 |INFqXmJi4XP9bCSG|
79cbb80: 6c4c 48 32 5a 78 66 56 44  73 4b 57 39 34 68 6d |lLH2ZxfVDsKW94hm|
79cbb90: 3364 2f 41 4d 41 45 44 41  41 41 41 51 51 45 42 |3d/AMAEDAAAAQEB|
79cbba0: 4141 41 41 41 3d 3d XX  XX XX XX XX XX XX XX XX |AAAAA==.........|
79cbbb0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbbc0: XXXX XX 65 2d 68 65 61 64  XX XX XX XX XX XX XX |...e-head.......|
79cbbd0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbbe0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbbf0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbc00: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbc10: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbc20: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbc30: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbc40: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
79cbc50: XXXX XX XX 0d 0a XX 6f  72 69 67 69 6e 61 6c 2d |.......original-|
79cbc60: 7265 73 70 6f 6e 73 65  2d 68 65 61 64 65 72 73 |response-headers|
79cbc70: XX44 61 74 65 3a 20 53  61 74 2c 20 30 39 20 44 |.Date: Sat, 09 D|
79cbc80: 6563 20 32 30 31 37 20  32 3a 32 39 3a 32 35 |ec 2017 22:29:25|
79cbc90: 2047 4d 54 0d 0a 43 6f  6e 74 65 6e 74 2d 4c 65 | GMT..Content-Le|
79cbca0: 6e67 74 68 3a 20 31 0d  0a 43 6f 6e 74 65 6e 74 |ngth: 1..Content|
79cbcb0: 2d54 79 70 65 3a 20 74  65 78 74 2f 68 74 6d 6c |-Type: text/html|
79cbcc0: 3b20 63 68 61 72 73 65  74 3d 75 74 66 2d 38 0d |; charset=utf-8.|
79cbcd0: 0a53 65 72 76 65 72 3a  20 54 77 69 73 74 65 64 |.Server: Twisted|
79cbce0: 5765 62 2f 31 36 2e 33  2e 30 0d 0a XX 75 6e 63 |Web/16.3.0...unc|
79cbcf0: 6f6d 70 72 65 73 73 65  64 2d 6c 65 6e XX XX XX |ompressed-len...|
```

列表 3：在 Intel Core i7-6700K 的 Ubuntu 16.10 上显示 HTTP 报头的内存转储。

```
f94b7690: e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |................|
f94b76a0: e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |................|
f94b76b0: 70 52 b8 6b 96 7f XX XX  XX XX XX XX XX XX XX XX |pR.k............|
f94b76c0: 09 XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b76d0: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b76e0: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX 81 |................|
f94b76f0: 12 XX e0 81 19 XX e0 81  44 6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX  XX XX XX XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX  XX XX XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |...........inst|
f94b7760: 61 5f 30 32 30 33 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX  XX XX XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX  XX XX XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |...........secr|
f94b77c0: 65 74 70 77 64 30 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX  XX XX XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f  61 64 64 6f 6e 73 2e 63 |https://addons.c|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c  6c 61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69  61 2f 61 64 64 6f 6e 5f |ser-media/addon_|
f94b7840: 69 63 6f 6e 73 2f 33 35  34 2f 33 35 34 33 39 39 |icons/354/354399|
f94b7850: 2d 36 34 2e 70 6e 67 3f  6d 6f 64 69 66 69 65 64 |-64.png?modified|
f94b7860: 3d 31 34 35 32 32 32 34  34 38 31 35 XX XX XX XX |=1452244815.....|
```

列表 4：在公开保存密码的 Intel Core i7-6700K 的 Ubuntu 16.10 上的 Firefox 56 内存转储（参见图 6）。

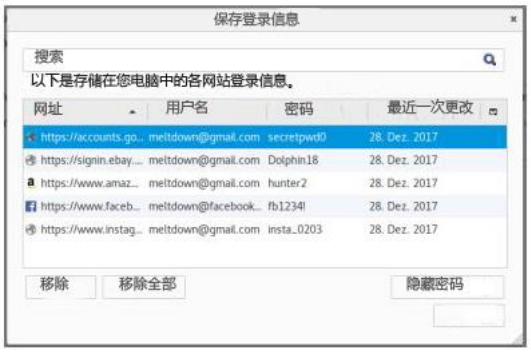

图 6：Firefox 56 密码管理器显示了利用列表 4 中的 Meltdown 泄露的存储密码。

列表 4 显示了在同一台机器上利用 Meltdown 的 Firefox 56 内存转储。我们可以清楚地看到存储在图 6 所示的内部密码管理器中的一些密码，即 Dolphin18、insta 0203 和 secretpwd0。该攻击还收集了一个似乎与 Firefox 地址相关的 URL。

6.4 ARM 和 AMD 的限制

我们也尝试在几个 ARM 和 AMD CPU 上重现 Meltdown 错误。但是，我们没有成功地将内核内存泄漏到第 5 节中描述的攻击，无论是 ARM 还是 AMD。出现这种情况的原因是多方面的。首先，我们的实现可能太慢，更好的版本可能会成功。例如，一个更为简单的无序执行流水线可能会导致竞争状态向数据泄漏转移。类似地，如果处理器缺乏某些特征，例如没有重新排序缓冲器，那么我们当前的实现可能不能泄漏数据。然而，对 ARM 和 AMD 来说，第 3 节

所描述的玩具例子都是可靠的，说明通常会出现乱序执行，并且非法内存访问的指令也会执行。

7 应对策略

在本节中，我们将讨论针对 Meltdown 攻击的对策。起初，由于问题根源于硬件本身，所以我们想讨论可能的微码更新和硬件设计中的一般变化。

其次，我们要讨论用于减轻 KASLR 侧信道攻击的 KAISER 对策，该对策歪打正着地阻止了 Meltdown 攻击。

7.1 硬件

Meltdown 绕过硬件强制隔离的安全域。在 Meltdown 中没有软件漏洞。因此，任何软件补丁（例如 KAISER [8]）都会留下少量的被暴露的内存（参见 7.2 节）。目前还没有明文说明这种修复是否需要开发全新的硬件，或者可以使用微码更新来修复。

当 Meltdown 采用无序执行的时候，一个很重要的对策就是完全禁止乱序执行。然而，性能影响将是毁灭性的，因为现代 CPU 的并行性不能再被利用。因此，这不是一个可行的解决方案。

Meltdown 是内存地址的获取与相应的地址检查之间某种形式的竞争条件。串行化权限检查和寄存器提取可以防止 Meltdown，因为如果权限检查失败，则永远不会获取内存地址。但是，这对于每次内存提取都会带来很大的开销，因为在完成权限检查之前，内存提取必须停止。

更现实的解决方案是引入用户空间和内核空间的硬分割。这可以通过现代内核使用 CPU 控制寄存器（例如 CR4）中的新的硬分离位来实现。如果硬分割位被设置，内核必须驻留在地址空间的上半部分，用户空间必须驻留在地址空间的下半部分。通过这种硬分割，内存提取可以立即识别目标的这种提取是否会违反安全边界，因为权限级别可以直接从虚拟地址派生而无需进一步查找。我们认为这种解决方案的性能影响是最小的。此外，还确保了向后兼容性，因为硬拆分的比特不是默认设置的，而且内核只在支持硬分割特性时才设置它。

需注意的是，这些对策只能防止 Meltdown，而不是 Kocher 等人描述的 Specter 攻击类别[19]。同样，Kocher 等人提出了一些对策[19]对 Meltdown 也不起作用。需要强调的是，针对这两种攻击采取应对策略是非常重要的。

7.2 KAISER

由于硬件不易修补，因此需要软件解决方法，直到可以部署新的硬件为止。Gruss 等人[8]提出了 KAISER，一种内核修改，无需内核映射到用户空间。这种修改是为了防止侧信道攻击破坏 KASLR[13,9,17]。但是，它也可以防止 Meltdown，因为它可以确保不会有效映射到用户空间中可用的内核空间或物理内存。KAISER 将在 Linux 内核的

最新版本中以内核页面表隔离（KPTI）[25]的名称被提供。该补丁也将被移植到较旧的 Linux 内核版本。Microsoft Windows 10 Build 17035 中也引入了一个类似的补丁程序建立[15]。而且，Mac OS X 和 iOS 也有类似的功能[22]。

尽管 KAISER 提供了针对 Meltdown 的基本保护，但它仍然有一些限制。由于 x86 架构的设计，需要在用户空间中映射多个权限内存位置[8]。这会留下 Meltdown 的残留攻击面，即，这些存储位置仍然可以从用户空间读取。即使这些内存位置不包含任何证书（如证书），它们仍可能包含指针。泄漏一个指针可能足以再次破坏 KASLR，因为可以从指针值计算随机数。

不过，KAISER 是目前最好的短时间解决方案，因此应立即部署在所有系统中。即使在 Meltdown 的情况下，KAISER 也可以避免在用户空间映射的内存位置上有任何内核指针，这些指针会泄漏关于随机偏移量的信息。这将需要每个内核指针的 trampoline 位置，即中断处理程序不会直接调用内核代码，而是通过 trampoline 函数。trampoline 函数只能映射到内核中。它必须以不同于其余内核的偏移来随机化。因此，攻击者只能泄漏指向 trampoline 代码的指针，而不能泄露剩余内核的随机偏移量。每个内核内存都需要使用这种 trampoline 代码，这些内存仍然需要映射到用户空间并包含内核地址。这种方法是性能和安全性之间的权衡，在未来的工作中必须进行评估。

8 讨论

Meltdown 从根本上改变了我们对操纵微体系结构元素状态的硬件优化安全性的观点。早在 20 多年前，人们便已知晓，硬件优化可以改变微体系结构元件的状态，从而危害软件安全的事实[20]。到目前为止，工业界和科学界都认为这是高效计算所必需的。今天，当密码算法不能抵御由硬件优化引入的微体系结构泄漏时，才认识到这是一个漏洞。Meltdown 彻底改变了整个局面。Meltdown 将粒度从相对较低的空间和时间粒度（例如，每隔几百个缓冲区攻击周期为 64 个字节）转换为任意的粒度，从而允许攻击者读取每个比特。面对这种情况，任何（密码）算法都无法保护自己。KAISER 是一个短期的软件修复程序，但我们发现的问题更为重要。

我们期望在现代 CPU 中进行更多的性能优化，以某种方式影响微体系结构状态，甚至不一定通过缓存。因此，为提供某些安全保证而设计的硬件（例如，运行不可信代码的 CPU）需要重新设计以避免类似 Meltdown 和 Spectre 的攻击。如果没有考虑到底层硬件，那么即使是无误差的软件，若明确地写成阻止侧信道攻击，也是不安全的。

通过将 KAISER 集成到所有主要操作系统中，迈出阻止 Meltdown 的重要一步。KAISER 也是操作系统范式改变的第一步。

不需总将所有内容映射到地址空间，而是只需要映射最低限度的内存位置作为减少攻击面的第一步。但是，这些也是不够的，可能需要更强的隔离。在这种情况下，我们可以通过为每个操作系统强制一定的虚拟内存布局来交换性能和安全的灵活性。由于大多数现代操作系统已经使用基本相同的内存布局，这可能是一个很有希望的方法。

Meltdown 也严重影响了云服务供应商，特别是如果客户没有完全虚拟化。出于性能原因，许多托管或云服务供应商没有虚拟内存的抽象层。在这种通常使用容器（如 Docker 或 OpenVZ）的环境中，所有客户都共享内核。因此，客户之间的隔离可以简单地通过 Meltdown 来避开，充分暴露同一主机上所有其他客户的数据。对于这些供应商而言，将其基础架构改为全面虚拟化或使用软件解决方案（如 KAISER）将极大增加成本。

即使 Meltdown 得到修复，Spectre [19] 仍然是一个问题。Spectre [19] 和 Meltdown 需要不同的防御。只解决一个其实并未消除整个系统的安全风险。我们预计，Meltdown 和 Spectre 会开启一个新的研究领域，即研究性能优化在多大程度上改变了微体系结构状态，该状态又是如何转化为体系结构状态，以及如何防止此类攻击。

9. 结论

在本文中，我们介绍了 Meltdown，这是一种基于软件的新型侧信道攻击，利用现代处理器的乱序执行从非权限用户空间程序中读取任意内核和物理内存位置。它不需要借助任何软件的脆弱性并且独立于操作系统，Melt-down 使攻击者能够以高达 503KB/s 的速度读取云中其他进程或虚拟机的敏感数据，从而影响数百万台设备。我们发现，KAISER [8]（即最初提出的防止对 KASLR 进行侧信道攻击的对策）能够在一定程度上阻止 Meltdown。我们在此强调，KAISER 需要在每个操作系统上得到部署，这可以作为一个短期的解决方案，直到 Meltdown 在硬件中被修复，从而防止攻击者大规模利用 Meltdown。

致谢

我们要感谢 Anders Fogh 在 2016 年的 BlackHat USA 和 2016 年的 BlackHat 欧洲峰会上所取得的丰硕的成果，使得我们能够发现 Meltdown。Fogh[5]已经怀疑，可能存在为了在用户模式下阅读内核存储器而滥用投机性执行，但他的实验并未取得成功。我们还要感谢 Jann Horn 对早期草案的评论。Jann 在六月份曾向英特尔披露了这一问题。随后进行的 KAISER 补丁活动反映了我们调查这一问题的原因。此外，我们希望英特尔、ARM、高通和微软就早期草案提供反馈意见。

我们还要感谢英特尔公司提供了负责任的披露流程的奖惩措施，并提供了清晰的时间表，对所有相关研究人员的信息进行了专业处理。此外，我们也要感谢 ARM 在披

露此问题时所做出的快速反应。

欧洲科学研究委员会（ERC）根据欧盟地平线 2020 研究和创新计划（第 681402 号资助协议）对这项工作进行了一定的支持。

参考文献

[1] BE N G E R , N., VA N D E PO L , J., SM A RT, N. P., A N D YA RO M , Y. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In CHES' 14 (2014).

[2] CH E N G , C.-C. The schemes and performances of dynamic branch predictors. Berkeley Wireless Research Center, Tech. Rep (2000).

[3] DE V I E S , A. M. AMD Takes Computing to a New Horizon with RyzenTM Processors, 2016.

[4] ED G E , J. Kernel address space layout randomization, 2013.

[5] FO G H , A. Negative Result: Reading Kernel Memory From User Mode, 2017.

[6] GR A S , B., RA Z AV I , K., BO S M A N , E., BO S , H., A N D GI U F - F R I DA , C. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS (2017).

[7] GRU S S , D., LE T T N E R , J., SC H U S T E R , F., OH R I M E N KO , O., HA L L E R , I., A N D CO S TA , M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In USENIX Security Symposium (2017).

[8] GRU S S , D., LI P P, M., SC H WA R Z , M., FE L L N E R , R., MAU - R I C E , C., A N D MA N G A R D , S. KASLR is Dead: Long Live KASLR. In International Symposium on Engineering Secure Software and Systems (2017), Springer, pp. 161–176.

[9] GRU S S , D., MAU R I C E , C., FO G H , A., LI P P, M., A N D MA N - G A R D , S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In CCS (2016).

[10] GRU S S , D., MAU R I C E , C., WAG N E R , K., A N D MA N G A R D , S. Flush+Flush: A Fast and Stealthy Cache Attack. In DIMVA (2016).

[11] GRU S S , D., SP R E I T Z E R , R., A N D MA N G A R D , S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In USENIX Security Symposium (2015).

[12] HE N N E S S Y, J. L., A N D PAT T E R S O N , D. A. Computer architec- ture: a quantitative approach. Elsevier, 2011.

[13] HU N D , R., WI L L E M S , C., A N D HO L Z , T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In S&P (2013).

[14] IN T E L. Intel R 64 and IA-32 Architectures Optimization Reference Manual, 2014.

[15] IO N E S C U , A. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER)., 2017.

[16] IR A Z O QU I , G., IN C I , M. S., EI S E N BA RT H , T., A N D SU NA R , B. Wait a minute! A fast, Cross-VM attack on AES. In RAID' 14 (2014).

[17] JA N G , Y., LE E , S., A N D KI M , T. Breaking Kernel Address Space Layout Randomization with Intel TSX. In CCS (2016).

[18] JI M E´ N E Z , D. A., A N D LI N , C. Dynamic branch prediction with perceptrons. In High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on (2001), IEEE, pp. 197–206.

[19] KO C H E R , P., GE N K I N , D., GRU S S , D., HA A S , W., HA M BU R G , M., LI P P, M., MA N G A R D , S., PR E S C H E R , T., SC H WA R Z , M., A N D YA RO M , Y. Spectre Attacks: Exploiting Speculative Exe- cution.

[20] KO C H E R , P. C. Timing Attacks on Implementations of Diffe- Hellman, RSA, DSS, and Other Systems. In CRYPTO (1996).

[21] LE E , B., MA L I S H E V S K Y, A., BE C K , D., SC H M I D , A., A N D LA N D RY, E. Dynamic branch prediction. Oregon State University.

[22] LE V I N , J. Mac OS X and IOS Internals: To the Apple' s Core. John Wiley & Sons, 2012.

[23] LI P P , M., GRU S S , D., SP R E I T Z E R , R., MAU R I C E , C., A N D MA N G A R D , S. ARMageddon: Cache Attacks on Mobile De- vices. In USENIX Security Symposium (2016).

[24] LI U , F., YA RO M , Y., GE , Q., HE I S E R , G., A N D LE E , R. B. Last-Level Cache Side-Channel Attacks are Practical. In IEEE Symposium on Security and Privacy – SP (2015), IEEE Computer Society, pp. 605–622.

[25] LWN. The current state of kernel page-table isolation, Dec. 2017. [26] MAU R I C E , C., WE B E R , M., SC H WA R Z , M., GI N E R , L.,GRU S S , D., AL B E RTO BOA N O , C., MA N G A R D , S., A N D RO¨ M E R , K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In NDSS (2017).

[27] MO L NA R , I. x86: Enable KASLR by default, 2017.

[28] OS V I K , D. A., SH A M I R , A., A N D TRO M E R , E. Cache Attacks and Countermeasures: the Case of AES. In CT-RSA (2006).

[29] PE R C I VA L , C. Cache missing for fun and profit. In Proceedings of BSDCan (2005).

[30] PH O RO N I X. Linux 4.12 To Enable KASLR By Default, 2017. [31] SC H WA R Z , M., LI P P, M., GRU S S , D., WE I S E R , S., MAU R I C E , C., SP R E I T Z E R , R., A N D MA N G A R D , S. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In NDSS' 18 (2018).

[32] TE R A N , E., WA N G , Z., A N D JI M E´ N E Z , D. A. Perceptron learning for reuse prediction. In Microarchitecture (MICRO), 2016 49th Annual

IEEE/ACM International Symposium on (2016), IEEE, pp. 1–12.

[33] TO M A S U L O , R. M. An efficient algorithm for exploiting multi- ple arithmetic units. IBM Journal of research and Development 11, 1 (1967), 25–33.

[34] VI N TA N , L. N., A N D IR I D O N , M. Towards a high performance neural branch predictor. In Neural Networks, 1999. IJCNN′ 99. International Joint Conference on (1999), vol. 2, IEEE,pp. 868–873.

[35] YA RO M , Y., A N D FA L K N E R , K. Flush+Reload: a High Reso- lution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium (2014).

[36] YE H , T.-Y., A N D PAT T, Y. N. Two-level adaptive training branch prediction. In Proceedings of the 24th annual interna- tional symposium on Microarchitecture (1991), ACM, pp. 51–61.

[37] ZH A N G , Y., JU E L S , A., RE I T E R , M. K., A N D RI S T E N PA RT, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In CCS′ 14 (2014).