SpecHammer：结合Spectre和Rowhammer进行新的投机攻击

密歇根大学Youssef Tobahytobah@umich.edu

Andrew Kwong密歇根大学 ankwong@umich.edu

Ingab Kangigkang@umich.edu

丹尼尔·根金佐治亚理工学院 genkin@gatech.edu

康G.辛密歇根大学 kgshin@umich.edu

密歇根大学

2022年IEEE安全与隐私研讨会（SP）|978-1-6654-1316-9/22/31.00美元©2022年IEEE | DOI:10.109/SP46214.2022.9833802

***摘要*——最近的*Spectre攻击*揭示了分支预测的性能提升是如何以削弱安全性为代价的。Spectre Variant 1（v1）展示了攻击者控制的变量如何传递给推测性执行的代码行，从而向攻击者泄露秘密信息。此后，人们提出了许多防御措施来防止Spectre攻击，每种防御措施都试图阻止所有或部分Spectre变体。特别是，使用污染跟踪的防御被认为是抵御所有形式的Spectre v1的唯一方法。然而，我们表明，通过将Spectre与众所周知的Rowhammer漏洞相结合，可以绕过迄今为止提出的防御措施。通过使用Rowhammer修改受害者值，我们放宽了攻击者需要与受害者共享变量的要求。因此，依赖于这一要求的防御，如污染跟踪，不再有效。此外，如果没有这一关键要求，可能用于发动Spectre攻击的代码片段数量将急剧增加；Linux内核版本5.6中存在的那些通过Rowhammer位翻转从大约100个增加到大约20000个。攻击者可以使用这些代码片段窃取敏感信息，如堆栈Cookie或金丝雀，或使用新的*三重*代码片段读取内存中的任何地址。我们在用户和内核空间中演示了对示例受害者的组合攻击的两个版本，展示了攻击泄露敏感数据的能力。**

1 介绍

计算机体系结构开发长期以来一直强调在常见情况下优化性能，通常以牺牲安全性为代价。投机性执行是顺应这一趋势的一个特征，因为它以有害的安全成本提供了显著的性能提升。此功能试图在确定正确的路径之前预测程序的执行流，从而节省正确预测的时间，并在预测错误的情况下简单地回滚任何执行的代码。然而，这样的预测可能会错误地推测恶意代码或值是安全的，使攻击者能够暂时绕过保护措施，并在误用窗口内运行恶意代码。

Spectre[25]和Meltdown[31]首次证明了这种投机性和无序利用的潜力，揭示了一类源于瞬态执行的新漏洞。这些攻击动摇了计算机体系结构和安全领域，导致大量工作涉及瞬态执行攻击[4]、[5]、[24]、[33]、[42]和防御[5]、[38]、[39]、[46]、[51]。

远离信息泄露，Rowhammer[23]是一个互补的漏洞，它破坏了存储在机器主存储器中的数据和代码的完整性。更多具体来说，DRAM DIMM中晶体管的紧密封装允许攻击者通过快速访问物理上相邻的内存行，在不可访问的内存地址中引发位翻转。与Spectre类似，Rowhammer也产生了许多漏洞[3]、[11]、[13]、[17]、[18]、[27]、[32]、[34]、[37]、[40]、[43]、[45]、[47]，包括最近绕过专用防御，如目标行刷新（TRR）[50]和纠错码（ECC-RAM）[9]。

虽然Spectre和Rowhammer都单独进行了广泛的研究，但对这两个漏洞的组合知之甚少。事实上，只有一项先前的工作，GhostKnight[55]，考虑了将这两种技术结合起来的新利用潜力。从高层次上讲，GhostKnight证明，尽管推测性内存访问具有瞬态性质，但它会导致Rowhammer无法单独到达的地址发生位翻转，从而导致这些内存位置发生位翻转。然而，GhostKnight只展示了如何使用Spectre来增强Rowhammer，而忽略了如何使用Rowhammer来增强Spectre的补充问题。注意到大多数现代机器都容易受到Spectre和Rowhammer的攻击，在本文中，我们提出了以下问题：

Rowhammer漏洞可以用来加强Spectre攻击吗？特别是，攻击者能否以某种方式利用Rowhammer来缓解Spectre的主要局限性，即受害者代码中有一个由攻击者控制输入的代码片段？最后，联合攻击对现有的Spectre缓解措施有什么影响？

A.我们的贡献

我们证明，事实上，Rowhammer和Spectre可以结合使用，以规避所提出的防御措施，并增加广泛使用的代码中可利用的代码片段的数量。在接下来的内容中，我们将对这种被称为SpecHammer的组合攻击进行高级概述，并讨论我们在内核中发现的新可利用的代码片段。

攻击方法。SpecHammer的核心思想是通过使用Rowhammer位翻转将恶意值插入受害者代码片段来触发Spectre v1攻击。我们提出了两种形式的SpecHammer：第一种放宽了对普通Spectre代码片段（以下称为*双*代码片段）的限制，第二种使用新的三代码片段，只需一次位翻转即可提供任意读取。

双重代码片段漏洞。通常，Spectre v1允许攻击者向Spectre代码片段发送任何恶意值，并在受害者的地址空间内任意读取内存。Spectre v1的主要弱点是，它需要在受害者的代码中使用一个代码片段，该代码片段使用攻击者控制的偏移变量，从而限制了Spectre v1.的攻击面。然而，SpecHammer第一个版本的目标是满足Spectre代码片段所有要求的代码部分，但*不向攻击者提供任何直接控制受害者偏移量的方法*。通过使用Rowhammer，可以修改偏移量并触发对此类受害者的Spectre攻击，以泄露敏感数据。这种攻击消除了Spectre v1的主要弱点，允许对更广泛的代码进行攻击。

不幸的是，Rowhammer最多只能用于翻转给定内存字的几个位，这限制了攻击者对受害者偏移量的控制。尽管如此，我们证明了攻击者即使在有限的控制下，仍然能够泄露敏感数据。例如，可以翻转偏移量中的位，使其指向刚好超过数组的边界。这允许泄漏秘密堆栈数据，例如旨在防止缓冲区溢出攻击的堆栈金丝雀[10]。也就是说，我们展示了如何利用双代码片段漏洞来泄露此类秘密，绕过堆栈保护机制。

三重代码片段漏洞。虽然第一个漏洞对防止缓冲区溢出攻击的共同防御构成了威胁，但它的范围比最初的Spectre攻击更有限，Spectre攻击在受害者的地址空间中泄漏了任意内存。然而，第二种类型的SpecHammer攻击可用于转储内存中任何地址的数据。此方法依赖于*三重*代码片段，其行为与Spectre v1代码片段相似，除了它具有三重嵌套访问功能。使用此功能，攻击者可以修改偏移量以指向攻击者控制的数据。此数据可以设置为指向秘密数据，这会导致在嵌套数组访问中使用秘密数据，就像Spectre v1中所做的那样。攻击者控制的数据可以被修改，以指向攻击者地址空间内的任何秘密，包括在利用驻留在内核中的三重代码片段时的内核内存。因此，单比特翻转允许任意内存读取，而双代码片段在泄漏地址方面受到更多限制。**挑战。**实施这些SpecHammer攻击带来了几个关键挑战：

1 我们必须找到包含有用位翻转的地址，这些位翻转可以迫使受害者在误操作下访问机密数据。

2 我们需要按摩内存，迫使受害者将数组偏移变量分配到包含这些有用翻转的地址。对于驻留在内核中的目标，这意味着对内核堆栈内存进行按摩。

3 我们必须证明，在Spectre v1代码片段中翻转数组偏移值可能会在误用情况下泄漏数据。

4 最后，我们需要在敏感的现实世界代码中找到代码片段，以了解放宽代码片段要求的影响。**挑战1：制作足够的链球翻转。**SpecHammer需要在特定的页面偏移处进行位翻转，以泄露秘密数据。为此，我们使用了之前工作[16]、[44]、[48]、[50]中附带的代码库，以便测试DRAM DIMM对Rowhammer攻击的敏感性。不幸的是，这些存储库产生的翻转数量表明，很难找到具有足够位翻转的DIMM来实际执行SpecHammer。

然而，正如我们在第四节中所示，我们观察到所有这些存储库都对缓存数据进行了关键的疏忽：它们首先初始化受害行，然后*在DRAM*（不是缓存）*中*引发位翻转，但在检查翻转之前忽略了刷新受害缓存行。这导致他们在检查翻转时观察*缓存数据*，从而使DRAM阵列中的许多翻转无法被观察到。通过纠正这些疏忽，我们能够在最坏的情况下将DDR3的位翻转次数增加248倍，在最好的情况下增加525倍，在DDR4的最好情况下增加16倍，这表明位翻转比以前的工作建议的要常见得多。这不仅使我们能够运行SpecHammer，而且使Rowhammer攻击比以前想象的更实用。

挑战2：堆叠按摩。对于SpecHammer攻击，Rowhammer位翻转的目标是一个用作数组索引的变量。这种偏移通常被分配为局部变量，这意味着它们位于堆栈上。Rowhammer攻击依赖于将目标按摩到易受比特翻转影响的物理地址上。然而，据我们所知，只有一项先前的工作[40]证明了锤击堆栈变量，依赖于内存重复数据删除来根据需要按摩堆栈数据。由于默认情况下已禁用重复数据删除，因此SpecHammer需要一种新的方法来将受害者堆栈按摩到位。此外，这种攻击最有吸引力的目标是驻留在内核中的代码片段，因为它们可用于泄漏内核数据，因此非常需要*内核堆栈按摩*原语。

然而，之前的内核按摩示例侧重于PTE，而不是堆栈[43]，或者是在移动设备上执行的，利用了Android独有的功能[47]。因此，我们开发了用于按摩用户和内核堆栈的新原语，以便在不使用重复数据删除的情况下进行堆栈锤击（第五节）。

挑战3：概念验证（PoC）演示。作为概念证明，我们在第六节中演示了在用户和内核空间中对示例人工受害者的攻击的变化。我们演示了用户空间中的双代码片段攻击和内核空间中的三代码片段攻击，因为每种攻击都适用于其各自的空间。这些PoC攻击是对广泛使用的代码中已经发现的代码片段进行最终攻击的基础。我们证明DDR3的泄漏率高达24位/秒，DDR4的泄漏率为19位/分钟。

挑战4：内核代码片段。为了更好地了解放宽代码片段要求的影响，我们发现了Linux内核中存在的代码片段数量，与SpecHammer代码片段的数量相比，原始的Spectre v1限制。如第七节所示，我们发现，在原始要求下，大约有100个普通的双代码片段，只有2个三代码片段。修改该功能以搜索易受SpecHammer攻击的代码片段，导致它报告了大约20000个双代码片段，以及大约

170个三重代码片段。因此，我们发现内核中潜在的代码片段数量比以前理解的要多。**捐款摘要。**本文做出了以下贡献：

- 结合Rowhammer和Spectre，放宽了攻击者控制Spectre代码片段偏移量的关键要求，在Linux内核中发现了20000多个额外的代码片段（第三节和第七节）。
- 开发在用户和内核空间中篡改受害者堆栈的新方法，使攻击者能够利用Linux内核中存在的众多代码片段（第五节）。
- 纠正先前Rowhammer技术的疏忽，在最佳情况下将比特翻转率提高525倍（第四节）。
- 演示如何使用SpecHammer代码片段来获取缓冲区溢出攻击的堆栈金丝雀，以及如何使用三重代码片段分别从示例用户和内核空间受害者的任何内存地址提供任意读取（第六节）。

2 背景

我们提供了了解新的组合攻击SpecHammer所需的Spectre和Rowhammer的必要背景信息。由于Spectre依赖于之前的缓存侧通道，因此也解释了相关的缓存攻击。

1 缓存侧通道攻击

缓存最初是为了弥合处理器速度和内存延迟之间的差距而设计的，但无意中导致了一个强大的侧通道被用于多次攻击[25]、[35]、[36]、[52]、[53]。通过定时内存访问，攻击者可以判断数据是从缓存（快速访问）还是DRAM（慢速访问）中提取的，从而可以观察到受害者的内存访问模式。

与SpecHammer最相关的是FLUSH+RELOAD技术[53]。目标是使用缓存来观察受害者对受害者和攻击者共享的内存的访问模式。例如，如果受害者访问依赖于秘密值的特定地址，了解受害者访问的地址可能会泄露有价值的秘密信息。

该技术首先通过刷新受害者可能使用clflush指令访问的任何缓存行来准备缓存。然后，受害者被允许运行，并且*只*会访问依赖于秘密数据的特定地址，*只*将相应的块加载到缓存中。接下来，攻击者访问受害者*可能*访问过的所有内存块，同时对每次访问进行计时。如果访问速度较慢，则意味着数据需要从DRAM移动到缓存，这意味着受害者没有访问块内的任何地址。但是，如果访问速度很快，则会从缓存中提取数据，这意味着受害者必须访问了与同一缓存行对应的地址。因此，通过利用缓存命中与缓存未命中之间延迟的巨大时间差异，攻击者可以准确地辨别受害者与哪些地址交互，从而辨别用于控制访问哪些地址的任何秘密数据。

2 幽灵

投机和无序执行。为了提高性能，现代处理器利用乱序执行来避免在后续指令准备运行时必须等待指令完成。在线性执行流的情况下，处理器利用*无序*（OoO）执行，按程序顺序运行指令，并且只有在所有前面的指令都被提交后才提交指令。当程序具有取决于某些指令结果的分支执行路径时，处理器会使用*推测执行*，预测分支将采用哪条路径。如果预测不正确，在推测窗口中运行的任何代码都将被撤消，从而导致可忽略的性能开销。

瞬态执行攻击。由于OoO或推测执行，在先前指令提交之前运行指令会产生一段短暂的执行期。长期以来，这种瞬态执行窗口一直被认为是良性的，因为任何不应该运行的代码都会被回滚，只有正确的代码才会被提交。然而，通过Meltdown[31]和Spectre[25]攻击，最近的研究表明，OoO和推测执行如何被攻击者用来强迫程序使用恶意值运行，不受安全防护的限制，只有*在*瞬态执行完成*后*才能生效。当代码回滚时，恶意值会留下架构副作用（例如将数据放置在缓存中），即使在瞬态执行中也会被用来泄漏数据。SpecHammer专注于Spectre和投机执行领域。

```
1if (x<array_size) {
2y=array1[x]
3z=array2[y*4096];
4 .
```

清单1:Spectre v1代码片段

幽灵攻击。Spectre[25]提出了攻击者利用推测执行的多种方式。我们关注Spectre v1，下面的例子说明了这一点。假设受害者包含清单1中所示的代码行，并且x是攻击者控制的变量。该攻击需要首先训练分支预测器来预测*if*语句将被输入。然后，攻击者可以更改x，使读取array1[x]访问array1末尾以外的秘密值。即使x可能超出界限，由于推测执行，秘密值仍将被访问，因为分支预测器已经相应地进行了训练。虽然从array2读取的数据永远不会提交给z，但推测执行仍然会导致array2使用秘密值y作为索引，并将（“secret”*4096）+array2基址处的数据加载到缓存中。

然后，攻击者使用FLUSH+RELOAD[53]检查提取了什么缓存行，以揭示array2索引，暴露秘密值。这种攻击的一个关键假设是，攻击者控制着x，因为她需要将x更改为用于通过array1访问机密数据的恶意值。

代码片段的流行。由于Spectre攻击依赖于受害者代码中是否存在代码片段，因此敏感代码中代码片段的普遍性成为一个关键问题。研究人员已经开发了工具[19]、[29]、[51]，以自动化在目标代码中查找代码片段的过程。例如，内核调试工具smatch[29]被扩展为能够报告Linux内核中的Spectre v1代码片段。在内核版本5.6上，smatch报告了大约100个代码片段。

后续攻击。Spectre发现后，出现了许多论文，详细介绍了如何将替代变体用于新的攻击载体[4]、[7]、[20]、[24]、[26]、[33]、[41]、[42]。这些包括执行推测性写入[24]，在网络上运行Spectre攻击[42]，以及将Spectre与其他侧通道组合以利用需要在条件语句中进行单个数组访问的“半代码片段”[41]。

3 落锤

Rowhammer漏洞[23]提供了一种修改攻击者无法直接访问的值的方法。该漏洞利用了DRAM阵列使用电容器存储数据位的事实，其中充满电的电容器表示1，放电的电容器指示0。随着晶体管变得越来越小，DRAM变得越来越密集，将电容器更紧密地封装在一起。[23]发现，通过快速访问DRAM中的值，使其快速放电并恢复到原始值，干扰效应会增加相邻行电容器的泄漏率。因此，通过快速访问（或“锤击”）攻击者行，攻击者可以释放相邻存储器位置中翻转1到0（或0到1）的相邻电容器。

DRAM组织和双面落锤。DRAM阵列由多个通道组成，每个通道对应一组列，其中每个列包含多个存储体。每个存储体由一系列行组成，行由包含单个数据位的电容器组成。虽然可以通过快速访问单个DRAM行来引起翻转[17]，但使用双面Rowhammer（即交替锤击围绕单个受害者行的两个攻击者行）要高效得多。通过增加相邻访问的数量，电容器的泄漏率增加，大大提高了诱导翻转的效率。双面Rowhammer需要锤击同一存储体内的相邻DRAM行。然而，攻击者无法直接看到与之交互的值的DRAM地址。相反，他们只能看到虚拟地址。这些被映射到物理地址，物理地址被映射到DRAM地址。

剥削。与Spectre一样，Rowhammer利用修改不可访问内存的能力激发了许多漏洞。这始于Seaborn和Dullien[43]，他们展示了如何使用翻转来执行沙盒转义，以及覆盖页表条目。随后出现了许多漏洞[1]、[3]、[17]、[27]、[32]、[34]、[37]、[40]、[45]、[47]，-展示了Rowhammer如何用于移动设备上的特权升级[47]，使用JavaScript在网络浏览器中翻转比特[16]，以及通过网络远程攻击受害者[32]、[45]。Gruss等人[17]还展示了可以击败多少Rowhammer防御。

幽灵骑士。据我们所知，只有一项先前的工作《幽灵骑士》[55]展示了如何将Spectre和Rowhammer结合起来进行更强大的攻击。由于Spectre允许访问给定地址空间内的任意内存，GhostKnight观察到，快速访问一对攻击者地址可能会导致推测域的翻转。这有效地增加了Rowham-mer的攻击面，因为它允许在只有推测执行才能到达的地址上进行位翻转。

3 SpecHammer

我们的SpecHammer组合攻击展示了如何使用Rowhammer来增强Spectre，绕过拟议的防御措施，并放宽对Spectre v1代码片段的要求。我们提出了两个版本：双代码片段攻击和三代码片段攻击，每个版本都在攻击的能力和对受害者代码中代码片段可用性的假设之间进行了不同的权衡。

1 双代码片段攻击：取消攻击者控制

如第二节所述，Spectre v1的一个关键限制是攻击者必须控制用作受害者数组索引的变量。我们通过使用Rowhammer在不直接访问的情况下修改索引变量来放宽这一限制。

```
1if (x<array_size) {
2victim_data=array1[x]
3z=array2[受害者数据*512];
4   .
```

清单2：伪代码双代码片段

攻击概述。从高层次上讲，双代码片段漏洞利用的目标是发动Spectre v1攻击，即使攻击者无法直接控制阵列偏移量。我们使用Rowhammer修改此偏移值，使数组访问秘密数据并通过缓存侧通道泄漏。

清单2显示了我们攻击的第一个版本利用的一个代码片段，它使用了与Spectre v1相同的代码片段。除了假设受害者的代码中存在此类代码代码片段外，我们还假设受害者的地址空间包含一些秘密数据。最后，与Spectre v1攻击不同，我们不假设对x的值有任何对抗性控制。攻击者不是直接控制x，而是利用Rowhammer触发x值的位翻转，使array1[x]访问秘密数据。

第一步：内存模板。任何基于Rowhammer的攻击的第一步都是对内存进行模板化，以找到包含有用位翻转的受害者物理地址，即导致x指向所需数据的翻转。如第二节所述，模板主要包括锤击许多物理地址，直到找到一对与受害者行相对应的攻击者，并进行有用的翻转。在找到一个具有合适翻转的物理地址后，我们使用内存按摩技术（见第五节）来确保x的值驻留在这个物理地址中，使其容易受到Rowhammer引起的位翻转的影响。

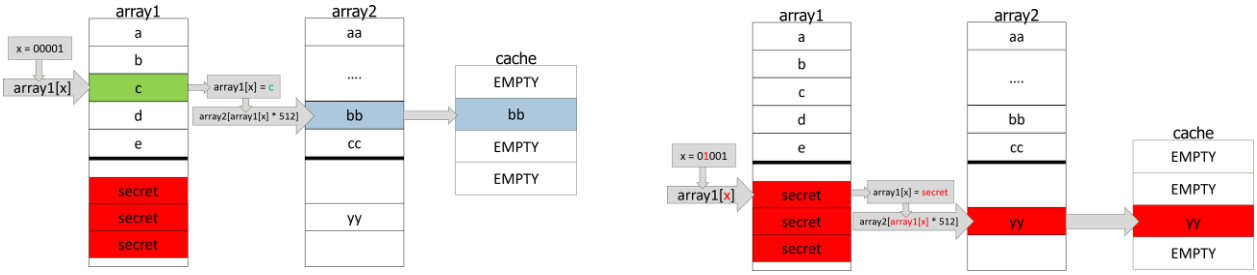

图1：攻击场景示例。（左）具有法律价值的训练阶段。（右）具有恶意值的攻击阶段。

第二步：分支预测器训练。在将受害者tim的代码放置在Rowhammer易受攻击的位置后，攻击者通过正常执行受害者代码来训练受害者的分支预测器。当我们使用合法值x执行受害者的代码时，会出现x<array1 _size的情况，这会导致CPU的分支预测器被训练来预测清单2第一行中的if被采用。见图1（左）。

第三步：锤击和误算。接下来，攻击者锤击x，导致图1（右）中的状态，其中位翻转（标记为红色）增加了x的值，使其指向array1末尾之后的秘密数据。攻击者还需要事先从缓存中删除x的值，以确保下次读取时使用DRAM中的翻转值，而不是之前缓存的值。在驱逐array1_size后，攻击者会触发受害者的代码。由于array1_size未被缓存，CPU使用分支预测器，并假设采用清单2第1行中的if进行正向推测。接下来，由于影响x的位翻转，对array1的访问使用了恶意偏移，导致secret被用作array2的索引，从而导致依赖于secret的存储块被加载到缓存中。最后，CPU最终会检测并尝试撤销错误推测的结果，根据程序顺序将受害者恢复到正确的执行状态。然而，正如Spectre[25]所发现的那样，CPU缓存的状态不会恢复，导致array2的一个秘密依赖元素被缓存。见图1（右）。

步骤4：冲洗+重新加载。为了从推测域中恢复泄露的数据，攻击者使用FLUSH+RELOAD侧信道[53]来检索秘密。更具体地说，攻击者在对每次内存访问的持续时间进行计时的同时访问array2的每个值。由于array2的所有值之前都已从缓存中清除，如果在驱逐和攻击的这个阶段之间没有发生访问，攻击者的定时访问应该很慢。但是，如果定时访问很快，则该内存块必须是最近访问过的。在这种情况下，由于在推测过程中访问了array2[secret*512]，攻击者在测量偏移secret*512时应该观察到快速访问，从而学习secret的值。

2 三重代码片段攻击：允许任意内存读取

第III-B节中介绍的攻击假设攻击者可以使用Rowhammer翻转受害者物理内存中的任意位。然而，在实践中，Rowhammer引起的位翻转不足以翻转泄漏任意地址所需的位数。攻击者最多可以翻转数组偏移量的几个位，从而限制她可以到达的地址。为了在Rowhammer提供的有限控制下提供任意读取，我们开发了另一种利用“三重代码片段”的变体。只需一个位翻转，攻击者就可以使用三重代码片段将数组偏移指向攻击者控制的数据。然后，可以将此数据设置为指向内存中的任何值，从而允许攻击者通过一次翻转泄漏任意数据，如下所述。

```
1if (x<array_size) {
  2attacker_offset=array0[x]
  3victim_data=array1[攻击者抵消]
  4y=array2[受害者数据*512];
5 .
```

清单3：伪代码三元组代码片段

攻击概述。对于三重代码片段攻击，我们使用了一种新型的代码代码片段；示例请参见清单3。在较高层次上，虽然原始Spectre v1假设受害者使用攻击者控制的变量x对两个数组进行嵌套访问（例如array2[array1[x]]），但在这里我们假设受害者使用x执行三重嵌套访问，即array2[aarray1[array0[x]]]。

通过使用这些代码片段，攻击者可以修改最内部的数组偏移量（x），使array0[x]指向攻击者控制的数据。这反过来又允许她向array2[array1[]发送任意偏移量，从而能够从受害者的地址空间中恢复任意信息。更具体地说，我们的攻击过程如下。

步骤1+2：记忆分析和分支预测器训练。与第III-A节一样，攻击者首先分析机器的物理内存，旨在找到包含有用位翻转的物理地址。然后，攻击者正常执行受害者的代码，从而训练分支预测器观察到清单3第1行中的if通常被使用。**第三步：锤击和误算。**接下来，攻击者锤击x，导致图2中的状态，其中位翻转（标记为红色）增加了x的值，使其指向array0的末尾，进入攻击者控制的数据。与第III-A节的情况一样，攻击者在驱逐array1_size后触发受害者的代码，这会导致CPU退回到分支预测器上，推测性地执行清单3第1行中的分支，就像它被占用一样。袭击者

控制地址array0+x中的值，这将导致攻击者控制的值作为第2行array0[x]的输出加载。CPU继续进行错误的推测，执行array1[array0[x]]（第2行和第3行），导致攻击者控制（通过array0[x]）从内存中寻址受害者负载。然后，在第4行访问array2之后，array1[array0[x]]的值通过缓存侧通道泄漏。

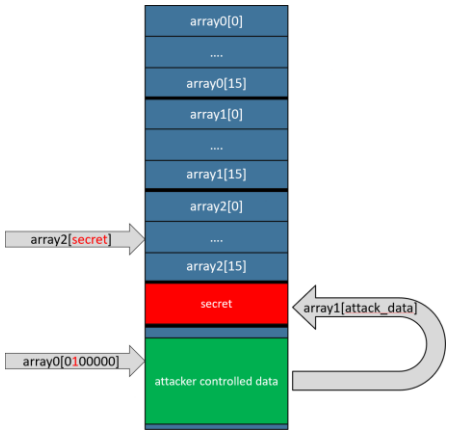

图2：三重代码片段示例

步骤4：冲洗+重新加载。最后，与第III-A节的情况一样，攻击者使用FLUSH+RELOAD侧通道来泄露在推测过程中访问的值。

与Double Gadgets进行比较。虽然三重代码片段需要在受害者的代码中进行三重嵌套数组访问，但它们也提供了一个优势，即读取受害者的数据不再需要多次精确的位翻转。特别是，由于只使用一个位翻转将array0[x]指向攻击者控制的数据，因此可以使用相同的位翻转值读取多个值。通过改变array0[x]的值并重复发起攻击，攻击者可以使用一次精心控制的位翻转来转储整个受害者地址空间。

内核攻击。当对驻留在内核中的代码片段执行此攻击时，这种攻击尤其危险，因为单个位翻转可用于读取整个内核空间。乍一看，防止*内核*对*用户*访问的Supervisor Mode Access Prevention（SMAP）似乎可以通过禁止内核访问清单3第2行上的用户控制数据来防止攻击。然而，在第VI-B节中，我们展示了如何绕过这种缓解措施，演示了攻击者如何使用系统调用将数据注入内核，然后使用单个位翻转从代码片段指向此受控内核数据。由于SMAP不会阻止*内核*到*内核*的读取，因此即使启用了SMAP，这种技术也允许执行三重代码片段攻击。

4 内存模板

第三节中提供的高级描述假设了两个关键的先决条件。首先，*内存模板*步骤用于查找有用的易翻转地址。接下来，使用*内存按摩*步骤强制目标受害者变量使用此地址。在本节中，我们将描述内存模板化过程，将堆栈按摩推迟到第五节。

模板的目标是获得“有用”的位翻转，这意味着它们可用于翻转数组偏移变量并触发SpecHammer攻击。易受位翻转的影响取决于单个DIMM的性质，需要锤击多个地址来了解哪些地址包含有用的翻转。用于模板的技术主要借鉴了现有的工作，因此我们保持了高层次的描述，让读者参考适当的先前工作[27]、[37]，并在附录a中给出了更详细的描述。

1 从虚拟地址获取DRAM行索引

如第二节所述，当连续击打挤压受害者排的两排侵略者排时，Rowhammer的效果要高得多，这种技术称为双面锤击[23]。通过双面Rowhammer找到翻转需要控制三个连续的DRAM行。然而，作为无特权的攻击者，我们没有直接的方法来确定我们的虚拟页面如何映射到DRAM行，从而阻止我们执行双面锤击。因此，在开始锤击之前，我们必须对这种映射进行反向工程。由于虚拟地址映射到物理地址，而物理地址又映射到DRAM行将，因此我们必须同时获得*虚拟到物理*和*物理到DRAM的*映射。

对于后者，我们使用Pessl的DRAMA技术[37]。对于前者，我们只需要用于确定相应秩、存储体和信道的物理地址位。对于使用DDR3的Haswell处理器，这些是最低的21位。因此，我们可以使用RAM Bleed[27]中提出的技术来获得一个连续的2MiB页面，为我们提供较低的21个物理地址位。由于此技术依赖于最近限制的页面类型信息文件，我们使用了一种新技术，该技术依赖于世界可读的buddyinfo文件（见附录a）。此步骤所需的时间不受使用新buddyinnfo技术的影响。

对于使用DDR4内存的较新架构，我们遵循TRRespass[14]的方法，使用在Linux内核版本5.14（撰写本文时的最新版本）中默认启用的透明巨型页面。请注意，对于单DIMM配置，最多只需要位21。对于两个DIMM配置，可以使用内存按摩技术来获得4MB的连续内存。

2 颤抖的记忆

将所有获得的记忆按行排序后，我们用反映我们所需翻转的值来初始化攻击者和受害者。在我们的例子中，我们试图增加数组偏移值以指向秘密数据，这意味着我们想将特定的受害者位从0翻转到1。因此，我们将潜在的受害者行初始化为包含所有0。由于双面锤击在受害者比特夹在相反值的两个比特之间时最有效[23]，[27]，我们将攻击者行设置为全1，给出1-0-1的攻击者-受害者-攻击者条纹配置。

诱导翻转。与之前的工作和现有的Rowhammer模板代码[16]、[44]、[50]、[54]一样，我们反复阅读

并从缓存中清除攻击者行，以确保每次读取直接访问DRAM并对相邻行造成干扰。在执行固定次数的读取后，我们读取受害者行以检查是否有任何位翻转，在这种情况下，这意味着受害者行值中的任何位置都将位设置为1。我们保存包含有用翻转的地址（即，会导致数组偏移指向秘密的位翻转），并进入内存按摩阶段。请注意，上述步骤*忽略了刷新受害者地址缓存*行。因此，当我们试图读取受害者以检查我们是否诱导了翻转时，我们可能会读取缓存的初始数据。

需要有用的翻转。在众多DDR3 DIMM上运行现有的Rowhammer代码[16]时，我们遇到了大约每小时2到5次翻转的较低翻转率。然而，对于我们的SpecHammer攻击，我们需要特定的位翻转（4KiB页面中的单个位位置），才能从数组指向秘密，这意味着在一般情况下，找到所需的位需要很长的时间。克服这一点的一种选择是测试许多DIMM，直到找到一个特别容易受到Rowhammer攻击的DIMM，将攻击限制在这种易受攻击的DIMM上。然而，我们观察到现有Rowhammer存储库中存在一个与缓存受害者数据问题有关的疏忽，这导致易受影响的DIMM在翻转时*看起来*很坚固，而事实上，绝大多数翻转只是被缓存数据*掩盖*了。通过修改这些现有的存储库，我们发现相同的DIMM每小时容易发生数千次翻转，这使我们能够对以前认为安全的DIMM进行攻击。

先前工作中报告的翻转率偏低。在检查了许多公共Rowhammer存储库[16]、[44]、[50]后，[54]旨在测试DIMM对Rowhammer的漏洞，我们观察到它们都造成了前一段中提到的受害者行缓存疏忽。通过执行上述步骤，读取受害者行以检查位翻转可能会导致读取缓存的初始化数据，从而导致对任何测试的DIMS上可获得的翻转实际数量的严重低估。报告的任何翻转都可能是由于其他内存访问替换了这些缓存行，导致受害者数据无意中从缓存中删除。在附录C中，我们描述了我们进行的实验，以证明缓存效应确实是掩盖位翻转的原因。**链球技术的比较。**为了充分了解这种疏忽对查找位翻转的影响，我们将之前的工作与受害者缓存刷新修改进行了比较。结果如表I所示。我们使用1-0-1条纹配置在两小时内使用双面锤击运行每个程序，然后使用0-1-0进行2小时的测试。两次运行的总翻转次数如表所示。

请注意，Rowhammer.js[16]的存储库包含一个错误，在确定哪些地址位于同一存储体上时使用*虚拟地址*而不是*物理地址*，因此分为2个条目：一个用于未修改的Rowhammer.js，另一个用于已删除错误的同一代码，不包括缓存刷新疏忽。最后，我们使用了TRResspass[14]，这是最新的Rowhammer模板库，专门用于DDR4，因为它使用了旨在绕过DDR4专用防御的技术。我们对这些存储库所做的更改详见附录B。

我们在Haswell i7-4770 CPU上使用Ubuntu 18.04和Linux内核版本4.17.3进行DDR3实验。对于DDR4实验，我们使用带有Ubuntu 20.04和Linux内核版本5.8.0的Coffee Lake i7-8700K CPU。DDR4 DIMM的型号均为M378A1K43BB2-CC。**结果。**对于DDR3，与去除了寻址错误的Rowhammer.js相比，我们的代码在最坏的情况下将翻转率提高了248倍，在最好的情况下提高了525倍。至于TRResspass，我们发现修改代码以包含受害者缓存刷新会导致DDR4 DIMM上发生6到8次翻转。虽然之前的Rowhammer调查发现翻转次数更多[8]，[22]，但他们使用了通用机器上无法使用的技术。在[22]的情况下，目标是在*电路级*别了解DIMM对Rowhammer的脆弱性，因此通过FPGA对DIMM进行了测试，以消除可能减少翻转次数的更高级别的干扰源。同样，[8]试图在服务器上实现翻转，他们的技术只能在多插槽系统上工作。相比之下，我们使用的代码是为测试自己的机器是否存在Rowhammer错误而设计的，并展示了冲洗受害者行如何大大增加翻转次数。

为了验证这些额外的翻转是缓存刷新的结果，我们进行了额外的实验，以验证数据实际上是从内存中提取的，而不是每次翻转的缓存中提取的。这些实验详见附录C。

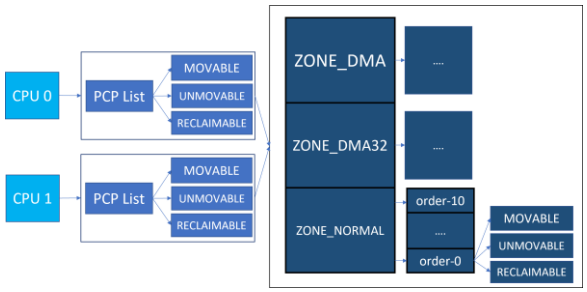

图3:Linux内存组织

5 内存（堆栈）按摩

拥有一个有用的、易翻转的地址后，下一步是将受害者变量强制放入该地址。目标受害者是一个用作数组偏移量的变量。这些变量通常被分配为局部变量，因此位于受害者的堆栈上。因此，为了翻转这些变量并触发攻击，我们需要将受害者的堆栈放置在从模板步骤获得的易翻转页面上。之前只有一项工作演示了堆栈按摩[40]，并使用（现已禁用的）页面重复数据删除来实现。请注意，位翻转对应于特定的DRAM地址，这些地址固定在特定的*物理地址*上。然而，物理地址可以通过页表映射映射到各种不同的虚拟地址。因此，目标是强迫受害者使用特定的*物理页面*。

模型	三星（DDR3）	Axiom（DDR3）	海力士DDR3	三星（DDR4）	三星（DDR4）	三星（DDR4）
落锤试验[44]	1.	0	0	-	-	-
罗哈默吉斯[16]	4.	9	2.	-	-	-
rowhammerjs（更正地址）	15	38	32	-	-	-
rohammerjs与受害者脸红	7,883	11,005	7,943	-	-	-
TR重新发布[50]	-	-	-	947	2,976	2,134
TR受害者出现潮红症状	-	-	-	7,916	17,958	15,611

表一：之前的Rowhammer技术和我们新的缓存刷新技术之间的比较。由于前4行中列出的技术是为DDR3设计的，因此我们没有在DDR4 DIMM上运行它们。同样，TRResspass是为DDR4设计的，并没有在DDR3上运行。请注意，rowhammerjs引用了其“本机”目录中的代码。

此外，如果受害者驻留在内核代码中，攻击者需要按摩*内核堆栈*，与按摩用户空间堆栈相比，这增加了额外的复杂性，因为无特权的攻击者无法直接操纵内核页面。虽然之前的工作已经通过强制PTE使用某些页面来演示内核按摩，但它们使用的方法对于内核堆栈按摩来说太不精确了[43]。这种现有技术只是取消映射易翻转的页面，并用PTE填充物理内存，直到使用最近未映射的页面。对于内核堆栈按摩，需要生成新的线程来分配内核堆栈。由于生成新线程是资源密集型的，我们不能用堆栈线程喷洒大部分内存，必须将内存操纵到一种状态，以最大限度地提高使用目标页面进行*有限*喷洒的可能性。其他先前的工作已经证明了更具确定性的技术，但这些技术是Android特有的[47]。在本节中，我们开发了一种通过利用Linux的物理页面分配器“伙伴分配器”（见附录a）及其每CPU（PCP）列表系统来按摩*内核*内存的新技术。在描述我们的技术之前，我们提供了我们为实现结果而操纵的内存结构的背景。概述如图3所示。**内存区域。**在伙伴分配器中，内存页面被组织在伙伴分配器内，除了按顺序排序外，空闲页面也按其*区域*排序。区域表示物理地址的范围。每个区域都有一个特定的免费页面*水印级别*。如果该区域的总可用内存降至水印级别以下，则请求将由下一个最优选的区域处理。例如，一个进程可以从ZONE NORMAL请求页面，但是，如果空闲NORMAL页面的数量太少，分配器将尝试为来自ZONE DMA32[15]的请求提供服务。

页面顺序。在每个区域内，页面按*大小*（也称为*顺序*）排列成块，其中order-x块包含2^x个连续页面。分配器总是试图从最小的顺序满足请求，但如果没有小顺序块可用，则较大的块将被分成两半，一半用于满足请求[15]。

迁移类型。页面按*migratetype*进一步组织。迁移类型决定了在使用页面时是否可以更改虚拟到物理地址的映射。例如，如果一个进程控制着映射到具有迁移类型MOVABLE的物理页面的虚拟页面，则可以通过将相同的虚拟地址映射到不同的物理地址来替换物理页面[28]。

PCP列表。最后，PCP列表（也称为页面*帧缓存*）[6]本质上是最近存储的缓存释放的订单为0页。每个CPU对应一组按区域和迁移类型组织的先进先出列表。每当发出订单-0请求时，分配器将首先尝试从相应的PCP列表中提取页面。如果列表为空，则从好友分配器的order-0自由列表中提取页面。当页面被释放时，它们总是被放置在相应的PCP列表中。即使释放了连续的高阶块，每个单独的页面也会被放置在PCP列表上，并且只有当它们从PCP列表返回到好友分配器自由列表时才会合并。因此，该系统用于快速获取最近在同一CPU上释放的页面，而不需要直接访问好友分配器。

1 用户空间堆栈按摩

基于现有的用户空间按摩技术[6]、[27]，主要目标是释放攻击者当前拥有的易翻转页面，然后强制使用最近释放的页面进行受害者分配。在堆栈按摩的情况下，这意味着强制进行新的堆栈分配。这里介绍的技术遵循与先前工作[6]、[27]中类似的步骤。虽然之前的作品使用此过程来按摩通过mmap分配的页面，但我们按摩受害者堆栈。**堆栈分配。**用户空间堆栈在生成新进程或线程时分配，并使用ZONE NORMAL、migratetype MOVABLE内存。此外，即使它们通常使用多个页面，也会将请求作为多个订单0请求处理，这意味着页面是从PCP列表中提取的。从用户空间中的mmap调用获得的页面也使用NORMAL、MOVABLE内存，这意味着堆栈页面和受控翻转易受攻击的页面属于同一类型。因此，通过取消映射释放易翻转的页面将使该页面位于用于堆栈分配的同一PCP列表中。**按摩步骤。**现在了解Linux堆栈分配，使用以下步骤执行堆栈按摩：**步骤1：饲料**分配。首先，在分配堆栈之前，我们进行“饲料”分配，以说明受害者所做的任何分配。目标变量可能不在受害者堆栈的第一页上。因此，在受害者分配包含目标的堆栈页面之前，我们必须首先计算受害者将使用多少页面，并分配这样数量的素材页面。

步骤2：取消页面映射。然后我们释放易受攻击的翻盖页面，将其放置在PCP列表中，然后释放饲料页面，将它们放置在易翻转页面上方的同一列表中。**第三步：受害者分配。**最后，我们生成受害者进程，迫使它执行预测的分配和目标堆栈分配。在

目标分配将从PCP列表中删除饲料页面，迫使堆栈使用目标页面。

结果。这种技术的准确率约为63%，这是可以接受的，因为只需要进行一次攻击。如果这一步失败，我们可以再次尝试按摩，并期望在两次尝试内成功。我们可以通过运行攻击的后续步骤（即调用包含代码片段的受害者并攻击攻击者）并检查缓存侧通道上的数据来检查按摩失败。如果没有观察到数据，我们会重新尝试按摩。

2 内核空间堆栈按摩

在内核中定位代码片段同样需要强制堆栈变量使用特定的易翻转页面。与用户空间堆栈分配一样，在创建新线程或进程时分配内核堆栈，该堆栈用于该线程或进程进行的所有系统调用。然而，与用户空间堆栈不同，内核堆栈使用UNMOVABLE内存，这意味着它们从PCP列表中提取的页面与用户空间mmap和unmap调用所使用的页面不同。因此，攻击者需要一种方法来强制内核使用“用户页面”（MOVAVABLE页面）而不是“内核页面”（UNMOVABLE页面）。我们从Seaborn[43]中观察到，当内存处于压力下时，内核确实会使用用户页面，并建立在Seaborn的技术之上，以允许更精确的内存按摩技术，从而可以按摩内核堆栈。

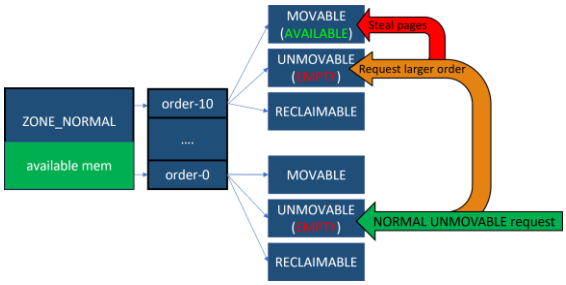

图4：**物理页面窃取**

分配器受压。如上所述，当该区域的空闲页面总数低于水印时，将使用下一个最受欢迎的区域。然而，由于区域包含多种迁移类型，所请求的迁移类型的自由列表可能为空，但总区域内存可能高于水印。在这种情况下，分配器调用一个*窃取函数*，从给定的“回退”迁移类型中窃取页面，并将其转换为最初请求的类型。如图4所示，此函数试图从回退类型中窃取最大的可用块。对于不可移动内存，第一个回退是可重复内存，第二个回退是可动内存。

内核按摩步骤。内核堆栈按摩所需的步骤与用户空间堆栈按摩相似。关键的区别在于，攻击者必须首先施加内存压力，迫使内核使用用户页面。

第一步：清空内核页面。作为非特权攻击者，我们不能直接分配UNMOVABLE页面。然而，每次通过mmap进行分配时，都需要一个页表条目（PTE）来映射虚拟和物理页。由于PTE使用内核内存，因此每次mmap调用都使用用户内存和内核内存。然而，一个页面内可以容纳多个PTE，PTE的地址取决于其相应的虚拟地址。我们需要有效地使分配足够大，以便每个PTE都需要一个新页面，但要足够小，以便进程不会因分配太多内存而终止。在2MB对齐地址处映射页面提供了最小的分配大小，以便每个PTE分配一个新页面。这样的分配会一直进行到没有可移动页面为止，使用pagetypeinfo文件来监视剩余页面的数量。后续映射将使用PTE的RECLAIMABLE页面。一旦必要的页面被耗尽，下一个内核分配将使用最大的可用MOVABLE块。

在无法访问页面类型信息的机器上，我们改用buddyinfo（对所有内核版本都是世界可读的），并同时监视MOVABLE和UNMOVABLE块的耗尽情况（同时执行步骤1和步骤2），只耗尽顺序为4或更高的UNMOVTABLE块。（与页面类型信息相比，buddyinfo的更详细解释见附录A。

第二步：清空用户页面。内存现在处于一种状态，将迫使内核使用最大的可用MOVABLE块。然而，我们需要内核使用特定的单个页面（包含位翻转的页面）。因此，我们需要确保目标页面位于此块中。使最大的可用块尽可能小是有利的，以提高内核使用目标页面进行堆栈分配的机会。因此，下一步是尽可能多地排出高阶自由块，而不会将自由页的总数降至水印以下。在我们的机器中，我们能够排出所有4阶或更高阶的块。

步骤3：释放目标页面。目标是释放目标页面，使其位于最大的可用块中。但是，释放此页面会将其发送到PCP，而不是好友分配器自由列表。即使它没有PCP，如果它没有任何免费伙伴，它也将保持在0级免费列表中。释放的目标页面需要合并成一个顺序为4的块，这样剩下的最大空闲块就包含易翻转的目标页面。幸运的是，正如第四节所解释的那样，我们已经保证目标页面是order-4（或更大）块的一部分。因此，我们可以释放目标页面及其所有伙伴，以确保它合并成最大的可用块。

最后一个障碍是PCP列表，因为即使取消映射连续的高阶块，所有页面也会被放置在相应的PCP列表中。但是，zoneinfo文件显示了在任何给定时间每个PCP列表中驻留的页面数量以及每个列表的最大长度。因此，可以取消映射其他页面，直到PCP列表中的页面数量达到最大长度（根据zoneinfo，我们的机器上有186页）。这迫使页面从PCP列表中删除并发送到好友分配器自由列表，将目标页面放置在MOVABLE内存的最大自由块中。

实验配置	SMAP	页面类型信息	THP	泄漏
i7-4770，DDR3，Linux 4.17.3	关闭	可读	不适用	20b/s
i7-7700、DDR4、Linux 5.4.1	ON	受限制的	麦迪逊	6b/m
i9-9900K，DDR4，Linux 5.4.1	ON	受限制的	麦迪逊	6b/m
i7-1700K，DDR4，Linux 5.4.0	ON	受限制的	麦迪逊	6b/m

表二：用于我们实验的配置列表。所有缓解措施都处于默认配置中。

步骤4：分配内核堆栈。释放目标页面后，知道下一个内核堆栈分配将使用用户内存，我们现在可以强制进行内核堆栈分配。然而，释放页面以强制目标页面退出PCP将略微缓解内存压力，这意味着一些不可移动的页面将是空闲的。内核堆栈分配将消耗这些页面，随后的分配将把包含目标页面的块转换为不可移动块。此外，由于内核的伙伴系统，块将被一分为二，其中一半用于内核堆栈，另一半移动到较低阶的UNMOVABLE自由列表。目标页面可能位于其中一半，必须继续进行分配，以确保目标页面用于内核堆栈。

因此，我们使用*内核堆栈喷雾*，分配许多内核堆栈，直到UNMOVABLE页面再次耗尽。我们通过生成许多线程来执行内核堆栈喷射。每个线程都可以在空循环中旋转，直到喷涂完成，然后通过让线程对受害者进行系统调用并敲打目标变量，直到我们观察到泄漏，来逐一进行测试。一旦找到目标页面的线程，其他线程就会被释放。我们现在可以翻转驻留在内核中的堆栈变量。

结果。这种技术与页面类型信息技术的准确率约为66%（与buddyinfo的准确率为60%）。我们希望它在两次尝试中取得成功。

6代码片段开发

此时，我们已经强制用户空间和内核空间中的受害者堆栈使用易翻转的地址。现在，我们可以翻转数组偏移值、强制错算和泄漏目标值。作为概念验证，我们分别在用户和内核空间中演示了对示例受害者的端到端双重和三重代码片段攻击。这些示例用于验证攻击泄露数据的能力。

设置。对于双代码片段攻击，我们使用带有Ubuntu 18.04和Linux内核版本4.17.3的Haswell i7-4770 CPU，这是我们机器上附带的默认版本。使用的DRAM由一对三星DDR3 4GiB DIMM组成。对于三重代码片段攻击，除了配备Kaby Lake i7-7700、Coffee Lake Refresh i9-9900K和Comet Lake i7-1700K处理器的机器外，我们还使用同一台机器。后三台机器分别使用DDR4 8GB DIMM，并分别运行Linux内核版本5.4.1、5.4.1和5.4.0。这些配置如表二所示。请注意，这两款较新的处理器具有Haswell不支持的额外防御功能。即使有这样的防御，我们也会展示我们的攻击。KASLR已在所有机器上启用。此外，透明巨型页面（THP）被设置为默认设置，即用户可以通过madvise系统调用进行分配。

A. 双代码片段——堆叠金丝雀泄漏

在本节中，我们将演示如何使用驻留在用户空间代码中的双代码片段窃取堆栈金丝雀。

堆叠金丝雀。堆栈金丝雀是放置在堆栈上的值，与返回指针相邻，作为防御缓冲区溢出攻击的机制。试图使缓冲区溢出并写入返回指针的攻击者将覆盖金丝雀，从而导致程序停止。由于其低成本和防止缓冲区溢出攻击的有效性，金丝雀长期以来一直被广泛部署为有效、轻量级的堆栈溢出防御机制[10]。

即使它们是随机生成的，属于父进程的子进程的堆栈金丝雀也将始终具有相同的堆栈金丝桃。因此，如果子进程的金丝雀被泄露，假设代码存在内存崩溃漏洞，就有可能对属于同一父进程的任何子进程执行缓冲区溢出攻击。例如，OpenSSH通过单个守护进程生成的子进程处理加密。泄露这些子进程中的任何一个的金丝雀，都可以绕过任何其他子进程的这种防御，从而泄露密钥。

```
    luint16_t数组1、数组2;
    2if（x<array_size）{
3       受害者数据=array1[x]
        4z=array2[受害者数据*512];
5   .
```

清单4：双代码片段

受害者示例。此示例攻击的受害者位于攻击者生成的线程中，受害者由一个双代码片段组成，如清单4所示，其中每个数组的类型都是uint16_t（第1行）。阵列位于受害者和攻击者共享的内存中，但通过使用PRIME+PROBE侧通道，可以进行没有此要求的攻击[35]。编译代码时，堆栈中包含秘密金丝雀，如果金丝雀被修改，则停止执行。让受害者驻留在攻击者生成的线程中允许用户空间堆栈按摩，但扩展到任何可以强制生成的进程，如OpenSSH[27]。

偷金丝雀。由于它们位于受害者堆栈的末尾，刚好经过目标数组的末尾，堆栈金丝雀成为双代码片段攻击的主要目标。读取金丝雀需要翻转数组偏移量的低位，以便相应的数组访问点刚好经过数组末尾到达堆栈金丝雀。

堆栈金丝雀通常为32到64位长，存储在返回指针正下方的地址。Spectre v1攻击每恶意偏移值窃取一个“单词”数据，其中一个单词对应于最内层数组的数据类型。在我们的受害者中，array1是一个uint16_t数组。每个恶意值x都指向并窃取一个16位值，这意味着该代码片段必须使用四次，每次使用不同的恶意值。

目标翻转。Rowhammer钻头翻转需要将偏移量推过受害者阵列的末端，并指向堆栈金丝雀。由于堆栈金丝雀被分成多个单词，我们可以找到一个具有多个位翻转的受害者行，或者允许受害者自然地循环值，并在必要的时间将偏移量推到金丝雀的不同单词上。我们使用后一种方法，因为我们在机器上观察到很少有包含多个翻转的行。

记忆模板和按摩。我们按照第四节中的描述执行内存模板，以找到有用的位翻转。受害者偏移量位于堆栈中的特定页面偏移量处，这意味着所需的翻转必须发生在相同的偏移量处。记忆被模板化约2.5小时，以找到这个特定的翻转。

包含此翻转的页面将取消映射，并生成受害线程，迫使受害线程内的偏移变量使用易翻转的页面。

触发幽灵。受害者只能使用用于其抵消的合法值来运行，这会训练分支预测器。我们等待受害者将偏移量设置为与金丝雀的给定目标词对应的适当值。对于这个例子，受害者和攻击代码是同步运行的，但FLUSH+RELOAD可用于准确监控受害者代码的执行，以提供攻击者同步[52]。然后，我们从缓存中删除偏移量，迫使代码片段在误用状态下使用翻转的值。金丝雀的一个单词被访问并用作将数据加载到缓存中的偏移量，使我们能够使用FLUSH+RELOAD来检索目标。受害者值保持不变，重复锤击以检索金丝雀的其余部分。

泄漏率。如前所述，阵列一次访问16位，这意味着每次翻转和FLUSH+RELOAD实例都会泄漏16位。我们观察到泄漏率约为8b/s，这意味着整个金丝雀在大约8秒内以100%的准确率泄漏。

B.三重代码片段-任意内核读取

第二个示例演示了如何使用内核系统调用中的三元组代码片段来实现对内核内存的任意读取。这尤其危险，因为内核内存在所有进程之间共享，这意味着有权访问内核内存的攻击者可以观察到内核为同一台机器上运行的任何进程处理的值。

```
1if (x<array_size) {
  2attacker_offset=array0[x]
  3victim_data=array1[攻击者抵消]
  4y=array2[受害者数据*512];
5.
```

清单5:Triple代码片段

受害者示例。此攻击的示例受害者是一个系统调用，我们在其中插入了一个三元组代码片段，如清单5所示。由于系统调用是以内核权限执行的，因此内核中的任何数据都可能被泄露。对于这个例子，我们在系统调用的代码中针对一个10个字符的字符串，该字符串超出了目标数组的范围。此外，攻击者和受害者共享三重代码片段中使用的阵列。

内存模板。正如在双代码片段攻击中所做的那样，我们首先找到一个有用的位翻转。这里翻转的目的是迫使受害者数组（在内核中）指向攻击者控制的数据。因此，需要一个特定的高阶位翻转来从受害者指向我们控制的数据区域。为了减少找到位翻转所需的时间，我们配置受害者，使其可以在堆栈中的任何位置使用数组偏移，方法是在每个偏移位置都包含受害者变量。因此，不需要在特定偏移处找到翻转；我们只需要更改页面内任何对齐的64位字的特定位。

攻击者控制的数据。控制受害者地址空间中数据的一种方法是简单地在用户空间堆上分配一个大的内存块，并用所需的值填充这个块。然后，位翻转会导致受害者从内核内存指向用户内存中的数据。然而，这需要打破内核地址空间布局随机化（KASLR），以便精确地知道目标内核地址和受控用户空间地址之间的差异。此外，Supervisor Mode Access Prevention（SMAP）会阻止内核读取用户内存，并且在最近几代Intel处理器上默认启用[2]。因此，我们将数据注入内核的地址集，这些地址集与目标翻转地址相差一个比特。

SMAP旁路。我们借鉴了内核堆喷雾攻击[12]、[21]，这些攻击演示了用攻击者控制的数据填充内核堆的方法。这些技术利用了sendmsg或msgsend等系统调用，它们使用kmalloc分配内核堆内存，然后将用户数据移动到这些内核地址。为了防止这些系统调用在返回之前释放数据，攻击者使用userfaultfd系统调用来暂停内核。此系统调用允许用户定义自己的线程，以处理指定页面上的任何页面错误。当攻击者调用数据插入系统调用（如sendmsg）时，他们传递具有N个页面数据的参数，但只分配N-1个物理页面。当sendmsg试图将数据从用户空间复制到内核空间时，它将在最后一个页面上遇到页面错误。由userfaultfd分配的线程故障处理程序被配置为在无休止循环中旋转，在将N-1页用户数据复制到内核内存后，使sendmsg陷入困境。

堆栈数据插入。虽然上述方法可用于将攻击者控制的数据插入内核的*堆*中，但堆插入对SpecHammer没有用，因为内核堆地址与内核堆栈地址永远不会只有一个比特的差异。然而，许多系统调用，包括sendmsg，都采用放置在内核堆栈上的用户定义的*消息头*。为了确保插入的值将落在距离翻转目标一位翻转的地址上，我们生成了许多线程，这些线程都使用sendmsg插入内核堆栈数据，从而提供了高概率（87%）的地址匹配。**控制页面偏移。**唯一剩下的问题是页面内的偏移。内核系统调用的堆栈偏移量始终是固定的，我们需要将数据插入到一个页面偏移量与翻转目标相匹配的地址中。幸运的是

攻击者，有许多系统调用（例如 sendmsg、recvmsg、setxatr、getxatr、msgsn）允许写入内核堆栈的256个字节，并提供了一系列偏移选项。此外，这些系统调用也可以从其他系统调用中调用（例如socket、send、sendto、recv、sendmmsg、recvmmsg），在调用前面列出的系统调用之前，每个系统调用都会使用不同数量的堆栈空间，这基本上允许攻击者在堆栈中“滑动”插入数据的位置。例如，我们发现第VII-B节中给出的示例代码片段的目标变量的页面偏移量为0xd20（在生成新线程期间调用时），sendmmsg可用于控制内核堆栈上从0xcf0到0xd70的数据。因此，三重代码片段攻击可以通过从受害者内核地址指向攻击者控制的内核地址来工作，从而允许攻击在SMAP存在的情况下工作。由于KASLR只随机化内核的基址，因此这些地址之间的差异保持不变，从而抵消了KASLR。

内核堆栈按摩。 接下来，我们运行第V-B节中的内核堆栈按摩技术，强制系统调用使用易翻转的页面作为其数组偏移量。我们分配了许多线程作为堆栈喷射的一部分，并且有可能没有一个内核堆栈包含易翻转的页面。因此，我们检查每个线程的目标页面，如果找不到该页面，我们重复模板和按摩步骤，直到目标页面进入内核堆栈。

触发幽灵。最后，包含目标页面的线程会发出包含受害者代码片段的系统调用，该调用会重复运行一个合法偏移值的循环，以训练分支预测器。偏移值偶尔会被敲打并从缓存中删除，导致最内部的数组指向处于误占用状态的用户数据。FLUSH+RELOAD侧信道用于确认目标秘密（在本例中为受害者字符串的值）已正确泄露。然后，我们修改攻击者控制的数据，以指向攻击者地址空间内的任何秘密值，并重复锤击以泄露下一个目标值。**离线阶段性能** 在禁用SMAP且页面类型信息不受限制的Haswell计算机上运行时，找到具有有用翻转的页面并将其放入内核所需的时间为34分钟。虽然我们新的buddyinfo和SMAP旁路技术的准确性略有降低，但它们相反地*减少*了找到翻转和找到有用页面所需的时间。buddyinfo技术放宽了对清空用户页面的要求（只清空4个或更大的区块，而不是清空所有区块），这意味着每次按摩尝试所需的时间更少。

此外，SMAP技术允许一*系列*位是有用的，因为我们需要从（受害者）内核堆栈指向（我们受控的）内核堆栈的任何翻转。这两个内存区域在内核堆栈受害者和受控用户空间区域的情况下更接近，这意味着我们可以在许多低阶位（位5到28）中进行选择，而不是被迫翻转从内核空间指向用户空间的唯一高位（位45）。因此，虽然这种技术引入了另一个概率元素（准确率为87%），但找到一个有用的翻转来执行攻击所需的时间减少了。因此，攻击平均需要9分钟才能找到一个有用的翻转并将其放入所有机器的内核中。

泄漏率。array1的类型为uint8_t，这意味着每次误检都会泄漏8位数据。在执行了必备的模板和按摩步骤后，DDR3上的泄漏率为16至24b/s。我们以100%的准确率泄露了目标字符串。在DDR4上运行时，需要进行多面锤击，这需要更多的锤击时间，从而将泄漏率降低到约4至19m/min（平均6b/min），在表二中列出的三台DDR4机器上也能达到100%的精度。

7 LINUX内核中的代码片段

1 代码片段搜索

史玛奇。Smatch[29]最初是为查找Linux内核中的错误而设计的。然而，在Spectre被发现后，添加了一个检查幽灵功能，可以搜索代码片段。它搜索在条件语句后发生嵌套数组访问的代码段，并且数组中的偏移量由无特权用户控制。它还检查嵌套访问是否发生在最大可能推测窗口内，以及访问是否使用数组索引nospec宏，该宏通过将数组偏移限制为指定大小来净化数组偏移。

工具修改。我们修改了该工具，删除了攻击者控制偏移的条件，并仅搜索了攻击者*不*控制偏移的代码片段。此外，我们还添加了一个函数来搜索三重代码片段，该函数检查嵌套数组访问的值是否用作第三次数组访问的偏移量。

结果。在Linux内核5.6上运行未经修改的checkspectre函数时，我们发现大约有100个双代码片段，只有2个三代码片段。修改该功能以搜索SpecHammer代码片段会导致它报告大约20000个双代码片段和约170个三代码片段。

绕过泰特追踪。如此多的潜在代码片段暴露了更多漏洞，让Spectre攻击敏感的真实代码。此外，oo7[51]是唯一可以有效缓解所有形式的Spectre[4]的防御措施，但对SpecHammer代码片段无效。这种防御识别使用*不*受信任*的*数组偏移值（即来自无特权用户的值）的嵌套数组访问。任何使用这种偏移的代码片段都被认为是“受污染的”，并且被阻止执行越界内存访问。然而，由于新发现的代码片段使用的变量不能被攻击者直接修改，因此它们被认为是值得信赖的，并且不会被oo7缓解。

附加代码片段。即使在对smatch进行修改以包含没有攻击者控制的偏移的代码片段之后，我们观察到smatch仍然无法检测到所有潜在的SpecHammer代码片段，这表明现有的代码片段检测工具不足以找到所有可利用的代码。

2 内核代码片段漏洞

为了了解smatch未发现的代码片段的性质，我们选择手工探索内核源代码，以识别可能利用Rowhammer提供的灵活性新开发的潜在代码片段。例如，除了操纵数组偏移量外，Rowhammer位翻转还允许间接修改指针。修改单个结构指针可能会导致指针解引用链以依赖秘密的缓存访问结束。这指向了一种与Spectre[25]中提出的代码片段相比的新型代码片段，因为它依赖于指针差异而不是嵌套数组访问。一个特别的例子是内核的page_alloc.c文件。

图5：alloc上下文**结构指针**

pagealloc.c此文件包含用于所有物理页面分配的代码。get_page_from_freelist函数特别包含SpecHammer代码片段；清单6展示了一个仅包含相关代码行的简化版本。请注意，该代码片段不包含共割数组访问，而是解引用连续的结构指针，并将结果用于数组访问。如图5所示，allocation_context（ac）结构指针尤为重要，因为函数中使用的许多变量都是从该指针获得的。

```
1   get_page_from_freelist{
2     构造alloc_context*ac;
3     结构体zoneref*z=ac->preferred_zoneref;
4     结构区*区;
5.
6     用于（区域=z->zone；区域；z=find_next_zone（z，ac-
    >zone_highidx）；
7     区=z->区）{
8       .
        9preferred_zone=ac->preferred_6zoneref;
        10idx=preferred_zone->classzone_idx;
11      ....
        12z->低内存服务[idx];
13    }
14 }
```

通过操纵ac的值指向攻击者控制的代码区域，可以控制从ac解引用中获得的所有变量，并控制受害者的执行流。更具体地说，攻击者正常运行该函数，教导预测器将进入清单6第6行的for循环。然后，可以通过锤击修改ac，使第3行（z=ac->preferred_zoneref）和第6行（zone=z->zone）的解引用将zone设置为NULL。这会触发错误猜测，因为for循环应该立即终止，但由于之前的训练，它实际上会开始第一次迭代。此外，ac已被设置为，在这种错误配置期间，清单6第9行和第10行的解引用链导致idx等于秘密数据，从而导致第12行的秘密相关访问（lowmem_reserve[idx]），可由缓存侧通道恢复。

结果。为了实证验证这种行为，我们根据需要对page-aloc.c文件进行翻转，并发现有可能操纵函数的控制流，导致泄漏内核数据的误用。我们通过插入使用-FLUSH+RELOAD通道的代码，恢复了插入到内核代码中的一个8位字符，该字符通常不在操纵数组的范围内。这可以用PRIME+PROBE替换，以在不修改page_alloc的情况下检索机密。

8 缓解措施

幽灵。开发一个专注于幽灵方面的防御可能是更困难的选择。虽然Spectre的其他变体得到了有效和高效的缓解[4]、[30]、[46]，但Spectre v1更多地被视为由分支预测引起的固有安全漏洞，没有简单的解决方案。

Taint跟踪是以前已知的唯一一种防御所有形式的Spectre v1[4]，[51]的防御方法，但由于它依赖于联合攻击中不存在的Spectre限制，因此受到了新的联合攻击的阻碍。其他旨在防御Spectre v1的防御措施[5]、[38]、[39]提供了不完整的保护，仅在特定情况下有效，并且通常会带来过高的性能成本[4]。

罗哈默。另一方面，对于Rowhammer来说，从PARA开始，已经开发了许多硬件和软件防御来防止或检测比特翻转[23]。PARA随机刷新行，为具有重复访问的行赋予更多权重。然而，这并不能保证保护即将翻转的行，而只能提供很高的刷新概率。对于需要单比特翻转的三重代码片段攻击，PARA不能保证保护。

与PARA类似的防御，目标行刷新（TRR）*确实保证*了每当两个攻击者行超过某个激活阈值时都会进行刷新。然而，TRResspass[14]最近表明，尽管有TRR，但通过执行分散的攻击者行访问，可以获得位翻转。此外，通过应用这种技术，发现DDR4比DDR3更容易发生位翻转[22]。

另一种常见的针对位翻转的硬件防御是纠错码（ECC）。最初设计用于捕捉由自然错误引起的位翻转，这些函数能够纠正单次翻转，并在给定行内检测最多两次翻转。然而，ECCploit[9]展示了一个由单翻转校正产生的定时侧通道，该通道允许攻击者找到包含多个翻转的行。通过模拟翻转多个比特，Rowhammer攻击可以被ECC检测不到，使ECC成为一种无效的防御。

9 致谢

本文报告的工作部分得到了美国陆军研究办公室的资助，资助号为W911NF-21-1-0057；美国国家科学基金资助号CNS-164613 和 CNS-1954712；空军科学研究办公室（AFOSR），奖项编号FA9550-20-1-0425；以及AMD和英特尔的礼物。

工具书类

1 Z.B.Aweke、S.F.Yitbarek、R.Qiao、R.Das、M.Hicks、Y.Oren和T.Austin，“Anvil：基于软件的下一代行锤攻击防护”，*ACM SIGPLAN通告*，第51卷，第4期，第743-755页，2016年。

2 A.Baumann，“硬件是新的软件”，*第16届操作系统热点研讨会*。ACM，2017，第132-137页。

3 E.Bosman、K.Razavi、H.Bos和C.Giuffrida在*2016年IEEE安全与隐私（SP）研讨会*上发表的“机器去重：作为高级利用载体的内存去重”。IEEE，2016，第987-1004页。

4 C.卡内拉、J.范·伯克、M.施瓦茨、M.利普、B.冯·伯格、P.奥尔特纳，
F.Piessens、D.SetPyushkin和D.Gruss，“瞬态执行攻击和防御的系统评估”，载于2019年*第28届{USENIX}安全研讨会（{USENIX}Security 19）*，第249-266页。

5 C.卡拉斯。（2018）Rfc:推测性负载共享（幽灵变体#1缓解措施。

6 A.Chakraborty、S.Bhattacharya、S.Saha和D.Mukhopadhyay，“解释框架：利用页面帧缓存进行分组密码的故障分析”，载于*2020年欧洲设计、自动化与测试会议与展览（DATE）*。IEEE，2020，第1303-1306页。

7 G.Chen、S.Chen、Y.Xiao、Y.Zhang、Z.Lin和T.H.Lai，“Sgxpectre：通过投机执行从sgx飞地窃取情报机密”，载于*2019年IEEE欧洲安全与隐私研讨会（EuroS&P）*。IEEE，2019，第142-157页。

8 L.Cojocar、J.Kim、M.Patel、L.Tsai、S.Saroiu、A.Wolman和O.Mutlu，“我们是否容易受到rowhammer的影响？云提供商的端到端方法论”，*2020年IEEE安全与隐私研讨会（SP）*。IEEE，2020，第712-728页。

9 L.Cojocar、K.Razavi、C.Giuffrida和H.Bos，“利用纠错码：关于ecc内存对行锤攻击的有效性”，发表在*2019年IEEE安全与隐私研讨会（SP）*上。IEEE，2019，
第55-71页。

10 C.Cowan、F.Wagle、C.Pu、S.Beattie和J.Walpole，“缓冲区溢出：十年漏洞的攻击和防御”，*载于DARPA信息生存性会议和博览会*论文集。*DISCEX'00*，第2卷。IEEE，2000，第119-129页。

11 F.de Ridder、P.Frigo、E.Vannacci、H.Bos、C.Giuffrida和K.Razavi，“SMASH：来自JavaScript的同步多面Rowhammer攻击”，发表于2021年8月的*USENIX Sec*。在线的可用：纸张=https://download.vusec.net/papers/smashsec21.pdfWeb=https://www.vusec.net/projects/smashCode=https://github.com/vusec/smash

12 L.Dixon，“使用用户错误fd”，2016年。在线的可用：https://blog.lizzie.io/using-userfailfd.html

13 P.Frigo、C.Giuffrida、H.Bos和K.Razavi，“大密码单元：用gpu加速微架构攻击”，*2018年IEEE安全与隐私研讨会（SP）*。IEEE，2018，第195-210页。

14 P.Frigo、E.Vannacc、H.Hassan、V.Van Der Veen、O.Mutlu、C.-Giuffrida、H.Bos和K.Razavi，“Trrespass：利用目标行刷新的多方面”，*2020年IEEE安全与隐私研讨会（SP）*。IEEE，2020，第747-762页。

15 M.Gorman，“理解linux虚拟内存管理器”，《*IEEE软件工程学报*》，2004年。

16 D.Gruss，“使用驱逐测试dram‘rowhammer’问题的程序”，2017年5月。在线的可用：https://github.com/IAIK/rowhammerjs

17 D.格鲁斯、M.利普、M.施瓦茨、D.根金、J.朱芬格、S.奥康奈尔，W.Schoechl和Y.Yarom，“落锤防御墙的另一个翻转”，*2018年IEEE安全与隐私研讨会（SP）*。IEEE，2018，第245-261页。

18 D.Gruss、C.Maurice和S.Mangard，“Rowhammer.js:javascript中的远程软件诱导故障攻击”，发表在*入侵和恶意软件检测以及漏洞评估国际会议*上。施普林格，2016，第300-321页。

19 M.Guarnieri、B.Ko¨pf、J.F.Morales、J.Reineke和A.Sa´nchez，“Spectrector：推测信息流的原理检测”，*2020年IEEE安全与隐私研讨会（SP）*。IEEE，2020，第1-19页。

20 J.霍恩。（2018）投机执行，变体4：投机商店绕过。在线的可用：https://bugs.chromium.org/p/project-zero/问题/细节?id=1528

21 invitus，“Linux内核堆喷涂/uaf”，2017年。在线的可用：https://invictus-security.blog/2017/06/15/linux内核堆喷涂uaf/

22 J.S.Kim、M.Patel、A.G.Yağlıkcı、H.Hassan、R.Azizi、l.Orosa和O.Mutlu，“重温rowhammer：现代dram设备和缓解技术的实验分析”，*2020年ACM/IEEE第47届计算机体系结构国际研讨会（ISCA）*。IEEE，2020，第638-651页。

23 Y.Kim，R.Daly，J.Kim，C.Fallin，J.H.Lee，D.Lee，C.Wilkerson，K.Lai和O.Mutlu，“在内存中翻转位而不访问它们：dram干扰错误的实验研究”，*ACM SIGARCH计算机体系结构新闻*，第42卷，第3期，第361-372页，2014年。

24 V.Kiriansky和C.Waldspurger，“推测缓冲区溢出：攻击和防御”，*arXiv预印本arXiv:1807.037572018*。

25 P.Kocher，J.Horn，A.Fogh，D.Genkin，D.Gruss，W.Haas，M.Hamburg，M.Lipp，S.Mangard，T.Prescher*等人*，“幽灵攻击：利用推测执行”，载于*2019年IEEE安全与隐私研讨会（SP）*。IEEE，2019，第1-19页。

26 E.M.Koruyeh、K.N.Khasawneh、C.Song和N.Abu Ghazaleh，“幽灵归来！使用返回堆栈缓冲区的投机攻击”，载于2018年*第12届进攻性技术研讨会（{WOOT}18）*。

27 A.Kwong、D.Genkin、D.Gruss和Y.Yarom，“Rambleed：读取内存中的位而不访问它们”，*2020年IEEE安全与隐私研讨会（SP）*。IEEE，2020，第695-711页。

28 C.Lameter和M.Kim，“页面迁移”，2016年。在线的可用：https://www.kernel.org/doc/Documentation/vm/page迁移

29 J.LCorbet，“用smatch发现幽灵漏洞”，2018年。在线的可用：https://lwn.net/Articles/752408/

30 M.Linton和P.Parseghian，“关于cpu推测执行问题缓解措施的更多细节”，2018年。在线的可用：https://security.googleblog.com/2018/01/有关cpu 4.html缓解措施的更多详细信息

31 M.Lipp、M.Schwarz、D.Gruss、T.Prescher、W.Haas、A.Fogh、J.Horn，
S.Mangard、P.Kocher、D.Genkin*等人*，“崩溃：从用户空间读取内核内存”，载于2018年*第27届{USENIX}安全研讨会（{USENIX}安全18）*，第973-990页。

32 M.Lipp、M.Schwarz、L.Raab、L.Lamster、M.T.Aga、C.Maurice和D.Gruss，“Nethammer：通过网络请求引发rowhammer故障”，*2020年IEEE欧洲安全与隐私研讨会（EuroS&PW）*。IEEE，2020，第710-719页。

33 G.Maisuradze和C.Rossow，“ret2spec：使用返回堆栈缓冲区的推测执行”，载于*2018年ACM SIGSAC计算机和通信安全会议论文集*，2018年，第2109-2122页。

34 O.Mutlu和J.S.Kim，“Rowhammer：回顾”，《*IEEE集成电路和系统计算机辅助设计*汇刊》，第39卷，第8期，第1555-15712019页。

35 D.A.Osvik、A.Shamir和E.Tromer，“缓存攻击和对策：aes的情况”，*在RSA会议*上的*密码学者轨道*上。施普林格，2006，第1-20页。

36 C.珀西瓦尔，“为了娱乐和利润而丢失缓存”，2005年。

37 P.Pessl、D.Gruss、C.Maurice、M.Schwarz和S.Mangard，“{DRAMA}：利用{DRAM}寻址进行跨cpu攻击”，第*25届{USENIX}安全研讨会（{USENIX}security 16）*，2016年，
第565-581页。

38 F.Pizlo，“幽灵和崩溃对我们bkit意味着什么”，2018年。在线的可用：https://webkit.org/blog/8048/幽灵和崩溃对webkit意味着什么/

39 T.C.项目，“现场隔离”，2018年。在线的可用：https://www.chromium.org/Home/铬安全/现场隔离

40 K.Razavi、B.Gras、E.Bosman、B.Preneel、C.Giuffrida和H.Bos，“翻转风水：在软件堆栈中敲打一根针”，*第25期{USENIX}安全研讨会（{USENIX}Security 16）*，2016年，第1页-18.

41 施瓦茨、利普、莫希米、范伯克、斯特克利纳，
T.Prescher和D.Gruss，“僵尸负载：跨权限边界数据采样”，载于*2019年ACM SIGSAC计算机和通信安全会议论文集*，2019年，第753-768页。

42 M.Schwarz、M.Schwarzl、M.Lipp、J.Masters和D.Gruss，“网络幽灵：通过网络读取任意内存”，载于*欧洲计算机安全研究研讨会*。施普林格，2019，第279-299页。

43 M.Seaborn和T.Dullen，“利用dram rowhammer漏洞获得内核权限”，《*黑帽*》，第15卷，第71页，2015年。

44 M.Seaborne，“dram‘rowhammer’问题测试程序”，2015年8月。在线的可用：https://github.com/google/落锤试验
45 A.Tatar、R.K.Konoth、E.Athanaspoulos、C.Giuffrida、H.Bos和K.Razavi，“链锤：网络和防御上的链锤攻击”，载于*2018年{USENIX}年度技术会议（{USENIX}{ATC}18）*，2018年，第213-226页。
46 P.Turner，“Retpoline：防止分支靶向注射的软件结构”，2018年。在线的可用：https://support.google.com/faqs/答案/7625886
47 V.van der Veen，Y.Fratantio，M.Lindorfer，D.Gruss，C.Maurice，G.Vigna、H.Bos、K.Razavi和C.Giuffrida，“Drammer：移动平台上的确定性行锤攻击”，载于*CCS*，2016年。
48 VandySec，“rowhammer-armv8”，2019年4月。在线的可用：https://github.com/VandySec/rowhammer-armv8
49 K.Viswanathan，“一些英特尔处理器上硬件预取器控制的披露”，2014年。在线的可用：https://software.intel.com/content/www/us/en/develop/articles/一些电话处理器上的硬件参考控制披露.html
50 vusec，“trresspass”，2020年3月。在线的可用：https://github.com/vusec/trespass
51 G.Wang、S.Chattopadhyay、I.Gotovchits、T.Mitra和A.Roychoudhury，“oo7:通过程序分析对幽灵攻击的低开销防御”，*IEEE软件工程学报*，2020年。
52 Y.雅罗姆。（2016）Mastik：一个微架构侧通道工具包。在线的可用：https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf
53 Y.Yarom和K.Falkner，“刷新+重新加载：一种高分辨率、低噪声的13缓存侧通道攻击”，第*23*届*USENIX安全研讨会*，2014年。
54 张、詹、巴拉苏布拉马尼安、库楚科斯和卡尔赛，“手臂上触发落锤硬件故障：回访”，载于*ASHES*，2018年。
55 张、程、张和苏亚，“幽灵骑士：通过投机执行破坏数据完整性”，载于*arXiv*，2020年。

附录

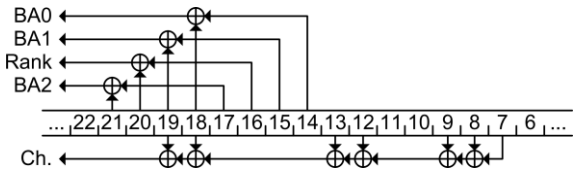

图6:Ivy Bridge/Haswell的物理到DRAM映射（摘自[37]）。

A.逆向工程虚拟到DRAM地址映射

以下部分解释了用于获得双面Rowhammer所需的虚拟到DRAM地址映射的技术。这些技术操纵Linux伙伴分配器，首先获得虚拟到物理地址映射[27]。然后使用定时侧通道来确定哪些物理地址对应于同一存储体中的行[37]，对物理到DRAM地址映射进行逆向工程。然而，这些技术依赖于pagetypeinfo文件进行内存操作，此后仅限于高权限用户。因此，我们使用世界可读的buddyinfo文件开发了一种新技术。

好友分配器。好友分配器是Linux用于处理物理页面分配的系统。它由按*顺序*和*迁移*类型组织的免费页面列表组成。顺序基本上是空闲内存块的大小。通常，来自用户空间的页面请求（例如，通过mmap）由订单0页面提供。即使用户请求了许多页面，她也可能会收到一个不连续的碎片页面块。如果没有请求大小的空闲块，则将最小的可用空闲块分成两半，称为*好友*，一个好友用于处理请求，而另一个好友则放置在下一个最大的空闲列表中。当页面返回自由列表时，如果其对应的好友也在自由列表中，则这两个页面将合并并移动到更高阶的自由列表中。Migratetypes本质上决定了页面是用于用户空间（MOVABLE页面）还是内核空间（UNMOVEABLE页面）[15]。

页面类型信息和buddyinfo文件。pagetypeinfo文件显示了每个订单和migratetype有多少可用块。虽然之前的技术[27]、[47]使用此文件来跟踪空闲内存的状态，但页面类型信息此后已被低权限用户无法读取。然而，一个名为buddyinfo的类似文件显示了每个订单有多少可用块，结合了内核和用户页面的数量。由于页面类型信息已被攻击者限制访问，我们提出了一种使用buddyinfo获取连续内存块的新技术。**获取连续的内存块。**为了控制连续DRAM行的集合，我们必须首先获得一大块连续的物理内存。对于第五节中描述的最终内存按摩步骤，位翻转需要驻留在至少16页长的连续内存块中。此外，正如我们将在下一段中看到的，2MiB块将有助于获取物理地址。但是，如果我们通过mmap请求2MiB块，分配器将通过分段内存而不是连续内存来处理此请求。因此，为了获得一个2MiB的连续块，我们首先分配足够的内存来耗尽所有较小大小（1MiB或更小）的用户块，迫使分配器为我们提供一个连续的2MiB块。

使用buddyinfo文件。然而，使用buddyinfo，我们只能看到剩余的用户*和*内核块的总和，但需要知道1MiB（及以下）*用户*块的数量何时低于2MiB内存。为了绕过这个问题，我们分配块，同时通过buddyinfo监控剩余的总量。通过将我们的分配放置在连续的虚拟地址上，我们确保我们的分配将主要使用用户块，因为很少需要用于新页表分配的内核块。因此，我们可以继续排空区块，观察1MiB区块*总数*的减少，直到达到*最小值*并再次增加。此行为表示没有剩余的用户块来满足请求，需要重新填充1MiB用户块自由列表。因此，观察到的*最小值*是可用的1MiB*内核页面*的数量，允许我们在任何给定时刻从总值中减去该值，以获得可用的1MiP用户页面的数量。

我们再次运行排水过程，减去内核页面的数量，直到剩余的1MiB用户页面等于0。我们可以使用相同的过程来耗尽较小的块，直到它们包含的内存小于2MiB。最后，我们通过mmap请求两个2MiB的内存块。由于分配器没有足够的小顺序块来满足分段页面的请求，因此它被强制执行以提供连续的2MiB块。我们的方法能够以与页面类型信息相同的100%准确率生成2MiB页面。由于在整个攻击过程中只需要执行一次计算内核块数量的额外步骤（而*不*是每次按摩尝试一次），因此使用*buddyinfo*技术的时间成本可以忽略不计。**物理地址。**为了获得虚拟到物理内存的映射，我们使用[27]中提出的技术。已经获得2MiB块后，我们可以通过找到块与对齐地址的偏移量来学习物理地址的最低21位。我们通过定时访问多个地址来学习同一存储体上地址之间的距离，从而获得这种偏移。通过确定块内每个页面的距离，我们可以检索偏移量。通过从虚拟到物理再到DRAM地址的映射，我们可以将虚拟地址分为与三个连续DRAM行对应的攻击者和受害者地址。

DRAM地址。接下来，我们需要物理到DRAM的地址映射。我们可以使用Pessl的定时侧信道来获得它[37]。这种技术利用了DRAM存储体的行缓冲区。在访问内存时，电荷从访问的行被拉入行缓冲区。从该缓冲区读取后续访问，减少访

问延迟。属于同一存储体的所有行共享一个行缓冲区。因此，对*同一存储体*中不同行的连续访问将增加延迟，因为每次访问都需要覆盖行缓冲区。通过访问成对的物理地址并将其分为快速和慢速访问，攻击者可以了解成对地址是否位于同一银行。攻击者可以比较同一存储体中足够多的地址的位，以检索从物理地址到DRAM的映射。

Pessl等人[37]提出了许多处理器的映射函数，如Haswell映射（如图6所示）。因此，对于针对Haswell的攻击，我们可以按原样使用此映射。对于较新的处理器，我们在几台机器上运行Pessl的攻击（如[50]中提供的），并获得Kaby Lake、Coffee Lake和Comet Lake处理器的映射。

DDR4上的连续块。我们之前解释了在Haswell机器上敲击时需要2MiB块，因为物理到DRAM的映射使用较低的21位。当机器使用两个通道，每个通道上有两个DIMM（4-DIMM配置）时，较新的处理器最多使用位24进行映射。最多22位用于两个DIMM配置，最多21位用于一个DIMM配置[11]。这些较新的处理器设计为使用DDR4。DDR4 Rowhammer技术，如TRRespass[14]，使用巨大的内存块来获得2MB的内存块，这足以用于一个DIMM配置。对于两个DIMM配置，可以使用内存按摩技术来获得4MB的连续块[11]。对于24位配置，精度会因未知位的数量而降低，这意味着在24位的最坏情况下，翻转次数会减少1/4。

B.对Rowhammer规范的修改

Rowhammer.js修改本节中的代码清单显示了我们对现有Rowhammer所做的更改

罗哈默.js	无缓存刷新	缓存刷新
击打	105,530,250	1.
失误	377,915	107,347,967
%翻找失误	100%	100%
翻转	12	2806

表三：刷新受害者地址对Rowhammer.js的影响

存储库，以防止缓存掩盖位翻转。清单7显示了对Rowhammer.js的本机代码所做的更改。从第530行开始的第一个更改修复了关于虚拟和物理地址的简单错误。原始代码将虚拟地址传递给get_dram_mapping函数，而此函数旨在使用物理地址。第二个修改发生在第561至576行。在这些额外的代码行中，我们在用测试值初始化任何受害行后立即刷新它们。这确保了当我们稍后读取这些行以检查翻转时，我们将直接从DRAM而不是缓存中读取。

TRRespass修改清单8显示了对TRRespas所做的修改。我们发现，需要将缓存刷新添加到多个代码区域，以尽量减少检查翻转时发生的点击次数。数据首先在init_stry函数中从第387行开始初始化。在TRRespass会话期间调用此函数一次，以初始化整个受害者数据区域。虽然由于初始化区域太大而无法一次性放入缓存，许多行自然会从缓存中删除，但许多初始化值仍保留在原始代码的缓存中。因此，我们在每次写入内存后都添加了刷新。由于TRRespass如何将地址组织成列（for循环中的col），后续列值不会导致后续地址访问。如果在给定的循环迭代中初始化的地址恰好位于已初始化的地址之前，则缓存的伙伴提取器可能会将已初始化和刷新的地址拉入缓存。因此，我们添加了一个额外的刷新（第400行），以从缓存中删除伙伴行。

然后，TRRespass使用从第571行开始的scan_stread函数检查翻转。当找到翻转时（如果res非零），翻转的数据将重新初始化为其初始值。但是，同一缓存行中可能仍有一些数据尚未检查。因此，我们刷新缓存以确保从DRAM而不是缓存中提取已检查的数据。在完成锤击会话后，TRRespass调用fill_strode（第284行），用初始数据填充受害者行。与init_stread函数类似，我们必须从缓存中清除此初始数据。最后，虽然hPatt_2_str函数（从第134行开始）不直接与受害者数据交互，但我们发现它的memset调用确实将受害者数据拉入缓存。这可能是由于处理器的伙伴缓存系统。因此，我们也会刷新此内存集数据。

C.验证缓存的效果。

为了确认读取实际上是在读取缓存数据，我们修改了现有的代码来测量

```
  .....
如果（偏移2>=0）
秒_行_页=页_每行[row_index+2].at（偏移2）；
如果(
    //*****已修复错误**********
      get_dram_mapping（（void*）（GetPageFrameNumber（页面映射，first_row_page）*0x1000））
    !=
      get_dram_mapping（（void*）（GetPageFrameNumber（页面映射，第二行_页面）*0x1000））
    //************************
 )
  {
  ....
    #ifdef查找爆炸物
（size_t尝试=0；尝试<2；++尝试）
    #endif
    {
/*****  *缓存刷新受害者**********
        int32_t偏移量=1；
（；偏移<2；偏移+=1）
      for（常量uint8_t*target_page8；
      每页页[行索引+偏移量]）
      {
常量uint64_t*target_page=（常量uint24_t*）
        目标_页8；
          for（uint32_t索引=0；索引<（512）；
          ++指数）{
              luint64_t*受害者_va=（uint64_t*）
              &目标页[索引]；
              asmvolatile（“clflush（%0）”：：“r”（victim_va）：%“内
存”）；
          }
          }
      //**********************************
      -hammer（第一页范围、第二页范围、读取次数）；
  ....
  }
```

清单7:Rowhammer.js修改

TRRespass	无缓存刷新	缓存刷新
击打	23,914,118	14,078
失误	2,081,626,490	2,105,526,350
%翻我失误	100%	100%
翻转	431	4795

表四：刷新受害者地址对TR的影响-
喘息

每次受害者地址检查发生的缓存命中和未命中次数。我们通过定时每次访问并将快速访问标记为缓存命中，将所有较慢的访问标记为高速缓存未命中来实现这一点。由于访问将整个*缓存行*拉入缓存，每行为64B，因此我们只测量每条缓存行的第一次访问，同一组中的所有其他访问都根据其第一个地址的时间进行标记。此外，我们测量了添加额外缓存刷新时观察到的命中和未命中次数，以确保我们从DRAM而不是缓存中读取受害者数据。最后，我们禁用了缓存预取器[49]，因为否则，访问单个集合会将其他集合拉入缓存，并使后续访问看起来像缓存命中，即使它们在锤击之前已被刷新。我们还验证了memset在我们的机器上不使用非时态（即非缓存）存储。对于DDR3测试，我们使用了运行Linux内核4.17.3的Haswell i7-4770处理器和三星DDR3 4GB DIMM。对于DDR4，我们使用了运行Linux内核5.4.0的Coffee Lake i7-8700K处理器和三星DDR4 8GB DIMM。每个实验进行2小时。数据以0-1-0条带模式初始化。

结果如表III（DDR3）和表IV（DDR4）所示。DDR3测试基于Rowhammer.js[16]和TRRespass上的DDR4[50]，因为它们是各自类型DIMM的最新Rowhammer存储库。对于这两个测试，在缓存未命中访问时都观察到了100%的翻转，这支持了我们的观察，即缓存掩盖了位翻转。在DDR3测试中，忽略使用受害者缓存刷新会导致绝大多数（99.64%）翻转检查读取缓存数据。缓存未命中时确实*会发生*不可忽视的377915次访问，这可能是原始代码能够观察到任何翻转的原因。然而，一旦添加了缓存刷新，几乎所有的访问都直接从DRAM读取，揭示了之前被缓存掩盖的大量翻转，导致翻转增加了233倍。

至于DDR4的结果，未修改的代码已经有大量的缺失。原因是，在被锤打之前，一次初始化了更大的数据区域，这导致由于缓存的大小有限，大部分数据被从缓存中驱逐出来。然而，额外的刷新能够将命中次数减少99.94%，大大减少了缓存掩盖的位翻转量。

```
  .....
字符*hPatt_2_str（HammerPatter*h_patt，int字段）
{
    静态字符patt_str[256]；
    字符*dAddr_str；

    memset（patt_str，0x00，256）；
    //*****新缓存刷新******
    clflush（patt_str）；
    clflush（patt_str+64）；
    clflush（patt_str+128）；
    clflush（patt_str+192）；
    clflush（patt_str+256）；
    //*******************************
  .....
无效填充成熟（DRAMAddr d）地址，uint8_t值，ADDRMapper*
映射器）
{
代表（size_t col=0；col<ROW_size；col+=（1<<6））{
        d_addr.col=col；
        DDRAM_pte d_pte=get_dram_pte（映射器，&d_addr）；
个      内存集（d_pte.v_addr、val、CL_SIZE）；
        //*****新缓存刷新******
        clflush（d_pte.v_addr）；
        clflush（（d_pte.v_addr）+CL_SIZE）；
        //*******************************
    }
  }
  .....
无效初始成熟（Hammer套房*套房，uint8_t val）{
  .....
        }
      }
    }
}
  .....
无效扫描_成熟（HammerSuite*套件，HammerPattern*h_patt，size_t adj_rows，uint8_t
  val）{
  .....
    if（res）{
      for（int off=0；off<CL_SIZE；off++）{
      ....
        内存集（pte.v_addr+off，t_vall，1）；
        //******新缓存刷新******
        clflush（pte.v_addr+关闭）；
        //*******************************
      }
      memset（（char*）（pte.v_addr）、t_val、CL_SIZE）；
      //******新缓存刷新******
      clflush（pte.v_addr）；
      clflush（（pte.v_addr）+CL_SIZE）；
      //*******************************
    }
```

清单8:TRRespass修改