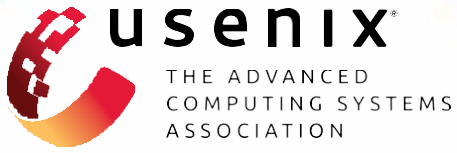


半双人床：从下一排开始锤打

格拉茨技术大学的安德烈亚斯·科格勒；格拉茨技术大学和拉玛安全研究学院的乔纳斯·朱芬格；谷歌的萨尔曼·卡齐和金永圭；亚马逊网络服务公司的莫里茨·利普；尼古拉斯·博查特，谷歌；邵世昌，**Rivos**；马蒂亚斯·尼斯勒，谷歌；格拉茨技术大学的丹尼尔·格拉斯

https：//www.usenix.org/conference/usenixsecurity22/presentation/kogler-shalf double

本文被收录在第**31**届**USENIX**安全研讨会论文集中。

2022 年 **8** 月 **10** 日至 **12** 日 • 美国马萨诸塞州波士顿

978-1-939133-31-1

第**31**届**USENIX**安全研讨会论文集的开放访问由**USENIX**赞助。

安德烈亚斯 · 科格勒[1] 乔纳斯 · 尤芬格 [1,2] 萨尔曼Qazi[3] 尤恩古Kim[3] 莫里茨Lipp[4,] 尼古拉斯 · 博伊查特 [3] 埃里克Shiu[5] 马蒂亚斯 · 尼斯拉尔 [3] 丹尼尔 · 格鲁斯 [1]

[1] 格拉茨技术大学 [2] 拉玛安全研究 [3]*Google* [4] 亚马逊网络服务 [5]*Rivos*

摘要

Rowhammer 是现代 DRAM 中的一个漏洞，对一行（侵略者）的重复访问会产生电干扰，其累积效应会翻转相邻行（受害者）中的位。因此，Rowhammer 防御的前提是攻击者与受害者之间的邻接关系，包括 LPDDR4 和 DDR4 中的攻击者与受害者，最明显的是 TRR。

在本文中，我们提出了 Half-Double[1]，这是 Rowhammer 的升级，可以攻击除直接相邻行之外的行。使用半双，我们通过将许多对距离-2 行的访问与对距离-1 行的少量访问相结合，在受害者中引入错误。我们的实验表明，这些因素的累积效应会导致受害者行产生足够的电干扰，从而引发位翻转。我们在一个完全最新的系统上进行了概念验证攻击，证明了半双的实际相关性。我们在端到端的半双倍攻击中使用侧通道、一种名为 *BlindHammering* 的新技术、一种新的喷涂技术和 Spectre 攻击。在最近配备 ECC 和 TRR 保护的 LPDDR4x 内存的 Chromebook 上，平均攻击时间不到 45 分钟。

1 简介

Rowhammer 是一个普遍存在的 DRAM 问题，由其组成行之间的意外耦合引起 [31]。通过反复访问一行（即*攻击者*），攻击者可以通过加速电荷泄漏来破坏相邻行（即*受害者*）中的数据。作为绕过硬件和软件内存保护的一种强大手段，Rowhammer 已被用作许多不同攻击的基础（第2.3节）。

以前，Rowhammer 被认为是在一行距离内进行操作的：攻击者只会翻转其两个紧邻的位，每侧一个。这从直观上讲是合理的：作为一种耦合现象 [54]，Rowhammer 效应在距离最近的情况下应该是最强的。事实上，这一假设支撑了许多针对 Rowhammer 提出的对策（第2.3节），特别是那些依赖于检测攻击者并在其预期受害者中刷新电荷的对策（例如 [31,35,40]）。事实上，*目标行刷新*（*TRR*）是一种广泛部署的量产对策，作为 LPDDR4/DDR4 芯片的一部分，属于这种*检测和刷新*类别 [15]。

在本文中，我们提出了 *Rowhammer 的新升级版 Half-Double*，我们展示了它的效果，即它不仅限于直接相邻的节点。使用半双，我们可以通过将许多对*远侵略者*（距离为 2）的访问与对*近侵略者*（距离为 1）的少数访问相结合，来翻转受害者中的位。两个攻击者都是必要的：只访问前者不会连续翻转两个比特，而只访问后者会变成一个很容易缓解的经典攻击。根据我们的实验，近侵略者似乎充当了桥梁，将远侵略者的 Rowhammer 效应传递给了受害者。令人担忧的是，TRR 通过其缓解刷新促进了 Half-Double，将它们的接收者行变成了近侵略者，与远侵略者合谋，而远侵略者是首先需要刷新的。实际上，治疗变成了疾病。

虽然 Half-Double 的发现和评估是这项工作的主要贡献，但我们也证明了它在概念验证开发中的实际意义。然而，目前的系统限制了攻击者的控制，引入了 4 个挑战：第一（**C1**），对手需要在 DRAM 库中分配连续的内存。然而，在没有物理地址 [48] 和巨型页面 [13，19] 的情况下，我们必须引入一种将*伙伴分配器*信息与 DRAM 定时侧通道相结合的新方法，以可靠地检测连续内存。第二种（**C2**），ECC 保护的内存可以使位翻转不可观察，这取决于攻击者无法控制的受害者数据，对手无法像之前的 Rowhammer 攻击那样对内存进行模板化，因为锤击需要知道单元数据。由于最先进的 [8,13,15,19,45,51] 没有解决这个问题，我们引入了一种称为*盲锤击*的新技术，以诱导位翻转，尽管 LPDDR4x 具有 ECC 机制。第三（**C3**），最近基于 ARM 的系统中地址空间大小的减小打破了以前攻击中的页表喷射机制 [19，44，48，51]。因此，我们开发了一种新的喷涂技术，这种技术仍然没有得到缓解。最后（**C4**），在没有模板的情况下，我们需要一个预言机来告诉我们 Rowhammer 是否引发了可利用的位翻转，而不会导致漏洞利用崩溃。为此，我们引入了一种使用基于幽灵的预言机来利用可利用位翻转的新方法。我们将这些技术结合成一个端到端的验证概念，即半双倍攻击[2]，它将无特权的攻击者升级为任意系统内存读写访问，*即内核特权*。在配备 TRR 保护 LPDDR4x 内存的完全更新的 Chromebook 上，半双倍攻击可在 45 分钟内完成。

总结而言，我们做出了以下贡献：

1. 我们发现了一种新的 Rowhammer 效应：半双，并评估了一组设备和模块的易感性。
2. 我们进行了彻底的根源分析，以实证证明 TRR 是造成半双效应的原因。
3. 我们分析了当今系统中存在的权宜之计缓解措施，并表明通过使用 Half-Double 的新漏洞，我们可以绕过它们并构建端到端的攻击。
4. 我们的端到端半双攻击在最新的 Chromebook 上运行，并将半双攻击效果与漏洞利用技术、侧信道和幽灵攻击相结合。

大纲。我们在第2节中提供了背景知识，并在第3节中介绍了一种新的 Rowhammer 模式符号。我们在第4节概述了半双效应，并在第5节实证验证了这是一种新的效应。我们在第6节中开发了端到端攻击。我们在第7节讨论了相关工作及其意义，并在第8节进行了总结。

负责任的披露。作者的一部分与内存供应商之间预先存在的合同义务意味着我们无法提供有关修复缺陷的努

[1]以一种比单针高但比双针短的钩针编织针法命名。
USENIX 协会

[2]我们的开源概念验证实现可以在以下网址找到：https://github.com/iaik/halfdouble USENIX 协会

力的有效性或实质的详细信息。我们认为，尽管如此，这项工作仍应发表，因为披露的影响不太可能对消费者安全产生重大影响，因为在我们的论文发表之前，Rowhammer 类型的漏洞的存在是 DRAM 设计中的一个已知限制[15,31]。因此，我们认为公开披露我们的新变体将有助于而不是阻碍系统的安全部署。

我们负责任地披露了半双倍，通知了受影响的内存供应商，引发了惯例禁运。在禁运到期后，该漏洞通过博客文章公开 [43]。

2 背景

在本节中，我们提供了 DRAM、Rowhammer 效应和广泛部署的 TRR 缓解措施的背景。

2.1 DRAM 组织

主存储器系统由多个通道组成，这些通道是存储控制器和 DRAM 芯片之间的独立链路。由于 DRAM 芯片具有较窄的数据总线，因此将其中几个芯片分组到一个列中，该列的聚合数据总线宽度与通道的宽度相匹配。多个等级可以分时共享一个通道。一个等级中的芯片在组织上步调一致，就像一个更大的芯片。因此，我们可互换地使用术语“等级”和“芯片”。每个等级由基于电容器的 *DRAM* 单元行组成。要访问一行，必须提高其字线的电压，将它的单元连接到它们各自的位线。该过程被称为激活，然后涉及位于位线另一端的行缓冲器，以检测电压扰动并将其放大到 0ŕ 或‘1’。这让我们回到了原点，因为细胞恢复到原来的状态：完全放电或完全充电。只要同一行保持激活状态，后续访问就会从行缓冲区提供服务。这种行击中比行冲突更快，行冲突必须激活不同的行。为了提高行命中的概率，将列组内的行分区到具有专用行缓冲区的存储体中。电容器以及 DRAM 单元会随着时间的推移而失去电荷。因此，所有行必须定期刷新，通常为 32-64 毫秒 [27]。刷新操作会随着时间的推移均匀分布，即每次刷新命令都会刷新一小部分行。我们强调，刷新行与激活行完全相同[38]（见第3节）。

2.2 DRAM 地址逆向工程

各种攻击都需要在 DRAM 中特定放置数据，这促使人们对 DRAM 寻址功能进行逆向工程。Pessl 等人 [42] 以及后来的 Barenghi 等人 [6] 使用了行缓冲器定时侧通道。Jung 等人 [29] 甚至通过基于热量的硬件故障攻击逆向设计了物理芯片位置。Helm 等人 [21] 使用性能计数器来测量行命中率和未命中率。所有这些方法都将地址分组为地址集，这些地址集相互之间存在行冲突或行命中，即它们位于同一银行。然后，他们计算哪些比特组合表示该集合，这通常是比特的线性 XOR 组合。Helm 等人 [21] 表明，寻址功能可以在不同的地址范围或通道之间变化。虽然较早的作品发现行索引只是地址位的一个子集 [19，42，48，56]，但最近的作品也发现 XOR 组合也用于索引位[49]。作为对 Rowhammer 的回应，今天的物理地址对用户程序是隐藏的 [32]，使得依赖于它们的攻击方法不适用于对最新系统的攻击。

2.3 电锤

在更高的密度下，芯片更有可能受到芯片内串扰引起的干扰误差的影响 [39]。2014 年，

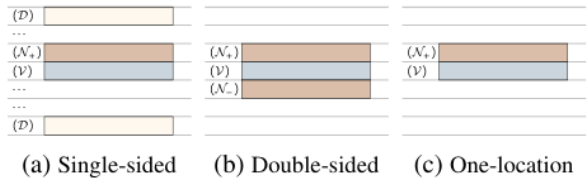


图 1：Rowhammer 访问模式：红色矩形 () 表示被敲击的行，即附近的攻击者 N，而蓝色矩形 () 表示最有可能发生位翻转的行，即受害者行 V。单面敲击访问一组无关的行，我们称之为诱饵$\mathcal{D}$ (□)。

Kim 等人 [31] 证明了 DRAM 芯片中来自内存访问的行间干扰错误，并将其称为行锤 [23]。最近，Walker 等人 [54] 对 Rowhammer 的底层物理进行了全面分析。

现有的 Rowhammer 访问模式（图1）因受害者和攻击者的相对位置而异。首先，在单侧模式 [48] 中，攻击者交替访问两行：攻击者和我们所说的诱饵。需要访问诱饵（同一组中的任意行）来翻转行缓冲区，以确保攻击者确实被激活。还有一种“放大”变体 [19]，其中两个攻击者被放置在彼此旁边。顾名思义，它们有时会相互加强，在各自的受害者身上产生比其他情况更多的比特翻转。其次，在双面模式 [48] 中，受害者被夹在两个攻击者之间。众所周知，这是最糟糕的访问模式，会导致最多的位翻转。还有一种“多面”扩展 [15]，涉及大量不同程度的侵略者和受害者。第三，单点模式 [18] 类似于单侧模式，只是它避开了诱饵。相反，它等待存储器控制器清除行缓冲器，然后再访问侵略者，以确保它被激活。

Rowhammer 漏洞已在沙盒环境 [48]、本地环境 [7、18、48、49]、虚拟机 [26、45、56]、JavaScript[8、13、19]、移动设备 [14、51] 和网络 [36、50] 中得到了证实。Rowhammer 漏洞利用通常借用传统的漏洞利用技术，如内存喷射 [19，48，56]、梳理 [51] 和页面去重 [8，45]，将目标数据结构放置在正确的内存位置。有许多改进锤击的建议，包括特殊指令 [44]、负载危害 [25]、页表访问 [59]、板载 FPGA[55] 和服务质量技术的内存压力 [1]。

已经提出了许多防御措施 [18]，主要集中在检测[10,20,22,24,35,40,41,53,58]、中和 [8,9,19,28,45,51] 或消除 [4,9,16,30,31] 软件或硬件中的 Rowhammer。目标行刷新（TRR）是一种已经集成到某些 DDR4 模块和 LPDDR4 标准中的防御措施，我们在第2.4节中讨论了它。

R_B

装饰

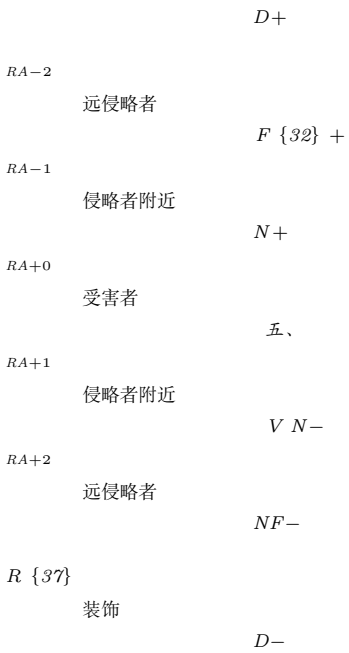


图 2：围绕受害者行的单个银行内的行的行注释。

2.4 缓解刷新（又称“TRR”）

从（LP）DDR4 代开始，供应商在其芯片内部实施了不透明和专有的防御措施。Frigo 等人 [15] 发现，此类措施似乎涉及两个主要组成部分：(i) 识别潜在攻击者的采样器，以及 (ii) 对其潜在受害者进行缓解刷新的抑制剂。此外，采样器的跟踪能力有限，当攻击者将激活交错到多行时，可能会被欺骗。

相比之下，我们的半双攻击利用了抑制器的缺点，抑制器是硬连接的，只对直接相邻的节点执行缓解刷新，而不考虑 Rowhammer 的远程影响。事实上，我们展示了缓解刷新如何通过将接收者行转变为共谋者来促进半双攻击——更具体地说，它变成了需要缓解刷新的远方的近侵略者。

在本文中，我们交替使用术语“缓解刷新”和“TRR”（目标行刷新）。尽管在以前的工作中使用了它，但后者是一个轻微的用词不当，因为它指的是以前提出（但从未采用）的 DRAM 命令，该命令允许 CPU 的内存控制器在刷新命令 [5] 的同时发送行地址。[3]

3 A 系统化的 Rowhammer 模式符号

在本节中，我们介绍了一种新的 Rowhammer 模式的系统符号，使我们能够对现有的攻击进行分类，并在第4节中描述了半双攻击的效果。

我们的符号描述了 Rowhammer 模式及其与银行内实际行位置的关联性。我们假设行索引表示银行内的物理行位置。对于符号，我们假设具有连续行索引的行在物理上是相邻的。图2显示了银行内部的行，如下所示。受害者行（V）是 Rowhammer 攻击的目标，该行内的位翻转用于测量实验中模式的效率。受害者行的直接邻居是所谓的近侵略者行($\mathcal{N}_+, \mathcal{N}_-$)，紧随其后的是远侵略者行($\mathcal{F}_+, \mathcal{F}_-$)。这三种类型的行位于银行内部的一个连续范围内。我们将距离该范围较远的行表示为诱饵行（D）。上远侵略者行($\mathcal{F}_+$)的绝对行位置用$\mathcal{R}_{A-2}$表示，这允许我们用一个索引来寻址这些行。为了描述 Rowhammer 模式，我们使用特殊的符号，例如$(\mathcal{A}_i \to (\mathcal{B} \to \mathcal{C})^\beta)^\infty$，其中 i 是所选模式的当前重复次数。因此，第一次存储器访问是到 A_0。然后，该模式访问行 B 和 C，并重复这两次访问 β 次。在访问了 B 和 C 之后，我们继续进行下一次迭代，即接下来访问第 A_1 行，依此类推。

通过这种表示法，我们根据行局部性对已知的 Rowhammer 模式进行了比较。双面 Rowhammer [48] 使用两个近侵扰者来攻击受害者行，即我们可以将模式表示为$(\mathcal{N}_+ \to \mathcal{N}_-)^\infty$。单边 Rowhammer [48] 有效地利用了一个附近的攻击者来攻击受害者行和 7 个诱饵访问，即我们可以将模式表示为$(\mathcal{N}_+ \to (\mathcal{D}_i)^7)^\infty$。然而，这些诱饵行的目的通常是在使用开放行策略的 DIMM 上触发行冲突，否则，访问将从行缓冲区提供（参见第2.1节）。最近的 Rowhammer 类型攻击，如 TRRespass [15] 和 Smash [13]，也使用了近侵略者和诱饵行。然而，在这两种情况下，多个受害者行都是交错定位的，以利用有限的 TRR 采样器大小并耗尽受保护行的数量。这使得攻击可以引发受害者行中的翻转，因为 TRR 缓解措施不再保护它们，所以受害者行受到的攻击频率较低。

总之，所有现有的 Rowhammer 模式都使用附近的攻击者行进行攻击，即它们是距离为 *1* 的模式，直接围绕单个或多个受害者行。TRR 缓解措施旨在缓解这些距离-*1* 型攻击。TRR 通过采样器检测到这些对附近攻击者的重复访问，然后抑制器在位翻转发生之前刷新受害者行（参见第2.4节）。TRR 刷新的详细实现是特定于供应商的，没有公开记录。我们假设与 Liu 等人 [38] 类似，TRR 刷新是通过关闭当前打开的行，然后打开受害行将行加载到行缓冲区中，从而刷新受害者的内容来实现的。然而，这提出了是否存在实际可利用的 *2* 距离模式的问题。

4 半双效应和利用

本节概述了半双攻击、其新的锤击模式以及攻击的挑战。

4.1 半双效应

通过半双攻击，我们提出了两种新的 Rowhammer 模式，即四元组模式和加权模式（或更详细地说，加权单加诱饵(WS+D)）。

四模式（模式1）将双面 Rowhammer 模式向外移动一行：

$$(\mathcal{F}_+ \to \mathcal{F}_-)^\infty. \quad (1)$$

[3] “伪 TRR”通过发送一对激活和预充电命令来手动刷新所需行，从而模拟该行为。

然而，由于这只会从实际受害者行中消耗少量电荷，因此四重模式结合了 TRR 刷新的效果来攻击受害者行。该模式使用远侵略者($\mathcal{F}_+, \mathcal{F}_-$)进行锤击。TRR 采样器检测到远侵略者受到攻击，在足够数量的行激活后，TRR 抑制剂向近侵略者发出刷新命令（试图减轻位翻转）($\mathcal{N}_+, \mathcal{N}_-$)。TRR 刷新机制关闭打开的行，并依次激活附近的侵略者行以刷新它们。TRR 刷新机制的这些额外激活通过从受害者行（V）中进一步消耗电荷来帮助我们打击受害者行。

加权模式（模式2）将一半的锤子分配给上方的远侵略者($\mathcal{F}_+$)，而将另一半分配给受害者行下方的行。因此，我们将其表示为

$$(\mathcal{R}_{A+4+3\cdot i} \rightarrow \mathcal{F}_+ \rightarrow \mathcal{R}_{A+6+3\cdot i} \rightarrow \mathcal{F}_+)^{\infty}. \qquad (2)$$

这种模式的直觉是在受害者下方的行上转移一个双面 Rowhammer 模式，同时将一半的锤子分配给远处的攻击者($\mathcal{F}_+$)。由于银行内部的最大行数有限，如果$\mathcal{R}_{A+6+3\cdot i}$在物理行范围之外，$i$ 将循环到零，在受害者下面重新开始该模式。加权模式的头两次重复产生以下序列：$\mathcal{R}_{A+4}$、F_+、$\mathcal{R}_{A+6}$、F_+、$\mathcal{R}_{A+7}$、F_+、$\mathcal{R}_{A+9}$、F_+。与四重模式类似，加权模式会攻击远处的诱饵，但会访问受害者下方的诱饵。该模式访问远侵略者，触发近侵略者上的缓解刷新机制（TRR），通过从受害者（V）中进一步排出电荷来协助锤击。

我们用以下假设 H 描述了半双模式的影响，在该假设下，半双模式会导致受害者行发生翻转。

假设 H： 远侵略者（65）的锤击会触发近侵略者（$N+$, $\mathcal{N}_-$)）的缓解刷新（TRR），通过消耗受害者行（V）的电荷来暗中协助对受害者行的锤击。然而，如果不激活远侵略者（*69*），近侵略者（68）的刷新就无法从受害者行中提取足够的电荷。

与多边 Rowhammer [13,15] 相比，半双模式不依赖于 TRR 资源的耗尽。相反，这些模式结合了 TRR 刷新机制，从而有助于 Rowhammer 攻击。因此，这种模式也适用于 TRR 机制完美应对距离-*1* Rowhammer 攻击的情况。我们在第5.1.2节和第7节中评估并讨论了半双和最先进的 Rowhammer 攻击之间的差异。

Rowhammer 漏洞通常涉及解决比特翻转之外的几个挑战。对于最先进的系统上的半双精度，我们确定了 4 个挑战：**C1** 连续内存的分配（没有物理地址信息或巨大的页面），**C2** 在没有模板的情况下找到位翻转（以绕过对模板的防御），**C3** 使用受限的喷射资源进行内存喷射，以及 **C4** 位翻转验证（由于盲敲造成的不确定性）。对于挑战 1 和挑战 3，我们可以扩展现有的技术。然而，对于挑战 2，ECC 存储器阻碍了位翻转模板，因为 ECC 代码依赖于攻击者未知的相应单元中的数据。我们称之为 *BlindHammering* 的新方法通过不针对位翻转进行模板化来规避这个问题。然而，这引入了不确定性，从而产生了挑战 4，我们通过将盲锤攻击与幽灵攻击相结合来解决这个问题。因此，我们更加关注挑战 2 和挑战 4，因为它们需要新颖的方法。

挑战 1：连续内存的分配。第一个挑战是获得对银行中相邻行的访问权限。由于之前的 Rowhammer 攻击，物理地址信息目前不可用 [32]。并非所有系统都支持大页面或页面融合机制，因此这两种方法都不适用于我们的攻击。因此，我们首先设计了一种新颖的连续存储器检测，结合基于异或的 DRAM 寻址功能的一般结构知识，以获取底层物理地址的信息，即使设备的 DRAM 寻址功能未知。结合对伙伴分配器行为的理解，我们获得了有关底层物理地址的信息。其次，由于半双模式对行位置的精确要求，我们需要使用时序侧信道对银行的行索引功能进行逆向工程。最后，我们将虚拟地址通过连续内存映射到存储体和行。

挑战 2：内存模板的替代方案。在控制了存储体内的连续行之后，下一个挑战是在内存中找到可翻转的位置。由于 DRAM 单元的差异，一些单元比其他单元更容易翻转 [54]。目前的技术水平是预先为易受 Rowhammer 翻转影响的存储器位置进行模板化。然而，一些 ECC 存储器阻碍了这一步骤，因为如果对精确数据或对 ECC 代码表现相同的数据执行模板化，位翻转只能在攻击中再现。攻击者通常没有这些信息，阻碍了内存模板方法。因此，我们提出了一种新的无记忆模板的方法，即盲锤法。

挑战 3：记忆按摩。这个挑战的重点是用盲锤攻击可利用的目标填充内存。我们利用的目标是页表条目。我们的目标是在这些页表条目中的物理页码。我们使用一种方法，在父进程的多个子进程之间映射共享内存，用额外的页表填充内存，而不用其他不可利用的数据页填充主内存。

挑战 4：位翻转验证。这一挑战的重点是确定诱发位翻转的位置。由于盲目锤击，我们无法直接检查锤击是否成功，因为访问潜在的损坏页表条目（PTE）会被操作系统检测到，从而终止漏洞利用。我们使用一种新的 Spectre 预言机 [33] 来解决这个问题，该预言机可以确定地址是否可以安全访问。我们还开发了一种架构替代预言机，并评估了这两种方法的优缺点。

我们在第6节中解决了上述挑战，并完全控制了系统主内存，证明了半双效应可以在现实系统中得到利用。

5 半双经验评估

为了证明假设 H 成立并解释半双倍效应，我们进行了以下观察：

1. 我们证明了半双效应的存在，即在当前的 TRR 保护系统中诱发比特翻转（第5.1节）。

2. 我们表明，如果没有对附近攻击者的（TRR 诱导）刷新，就不会发生半双倍攻击，并且 TRR 刷新与观察到的位翻转次数之间存在关系，即（与直觉相反）更多的 TRR 刷新会导致受害者行中出现更多的位翻转（第5.2节）。

为了获得无噪声的观测结果，我们使用了一个 FPGA 板，可以完全控制所有的刷新和内存访问，我们对数据的稳定性没有要求（第5.3节）。由于本节的重点是展示上述要点，因此假设 H 成立，我们不会在本节中限制特定的威胁模型。

5.1 半双倍速率（Half-Double）的 TRR 保护 LPDDR4x

在本小节中，我们使用 TRR 保护系统上的*四元组模式*演示半双。[4]我们证明这种模式可以产生位翻转，并记录观察到的位翻转次数来衡量性能。

5.1.1 测试系统和 DRAM 寻址功能

我们使用了 10 个商品系统（完整列表见表9）。我们使用 Pessl 等人 [42] 的方法对 DRAM 寻址功能进行逆向工程（参见第2.2节）。由于他们的方法只将物理地址映射到给定的存储体，但没有恢复我们需要的*四元组模式的精确行索引*，我们在存储体内的行命中与行冲突之间使用额外的时序侧通道 [49]（参见第2.2节）来获取行索引信息。我们发现，我们两台相同的基于 ARM 的联想 Chromebook 具有 Tatar 等人 [49] 描述的相同的行加密功能，其中第 3 位与行索引中的第 2 位和第 1 位进行 XOR 运算。我们在图3中展示了 Chromebook 的完整 DRAM 寻址和索引功能。通过设备特定的功能，我们将物理地址映射到存储体和行索引，从而测试第4.1节中的半双模式。

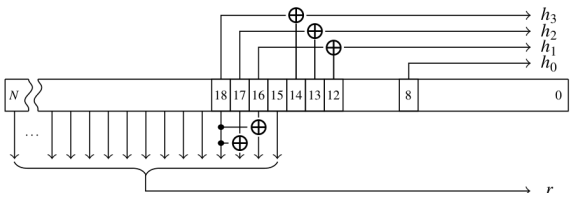


图 3：Chromebook 中逆向工程 DRAM 寻址功能。

5.1.2 四边形模式的评估

我们使用两种策略来测试*四重模式*，以达到 DRAM：不可缓存的内存 [52] 和内存刷新 [31]。对于不可缓存的内存，我们将*远侵略者*、*近侵略者*和*受害者*标记为不可缓存，允许在一个刷新间隔内进行更多的锤击尝试。内存刷新方法要慢得多，依赖于架构的刷新指令来将*远端攻击者*从缓存中清除。

在我们的评估中，我们分配了大量的内存，并使用 Linux 的 pagemap 接口 [32] 来提取物理地址。我们分析块的物理地址，并将与它们的存储体相对应的虚拟地址分组。之后，我们从同一银行搜索映射到表示*四元模式*的连续行范围的地址，即我们找到行$\mathcal{R}_{A-2}$到$\mathcal{R}_{A+2}$，参见图2。

现代内存控制器通过将掩码与行数据异或来对数据进行加扰。Cojocar 等人 [11] 表明，所有行的数据掩码都是相同的。我们通过经验观察数据扰乱，并相应地将*远侵略者*和*近侵略者*行的所有字节设置为 0x55，并用 0xaa 填充*受害者*行的字节。我们发现，这最大限度地增加了我们在测试设备上看到的*四元组模式*攻击中的比特翻转次数。

[4]我们在第5.2节和第5.3节中分析了*加权模式*。

锤击以一个紧密的循环运行，接近*远处的侵略者*。我们运行了 *2000 万次 Quad 模式*，并检查了*受害者*行中的位翻转。表1显示了两种方法的结果。Chromebook$_2$ 的翻转次数是相同 Chromebook$_1$ 的 36 倍。OnePlus 5T 显示出与 Chromebook 类似的翻转趋势。使用不可缓存的内存，我们可以在 Chromebook 上引发 10 到 20 倍的位翻转。然而，在使用不可缓存的内存时，OnePlus 5T 并没有显示出巨大的增长。我们还观察到从 1 到 0 的更多翻转，与 Kim 等人 [31] 的研究结果相似。然而，我们得出结论，在任何一种情况下，攻击都是可能的，尽管如果不可缓存的内存可用，攻击速度会更快。

表 1：在受影响的 LPDDR4x 系统上使用不可缓存的内存和刷新指令的 *Quad 模式*的性能。

System	$N_{Hammers}$	$UC_{0\to1}$	$UC_{1\to0}$	$Flush_{0\to1}$	$Flush_{1\to0}$
Chromebook$_1$	23 274	27	40	2	5
Chromebook$_2$	23 586	235	2379	12	101
OnePlus 5T	25 687	2	30	1	24
Pixel 3	32 921	11	5	0	0
HTC U11	21 840	-	-	3	17

关键见解： Half-Double 能够在 TRR 保护的内存上产生位翻转。

为了比较半双效应与当前最先进的多边 Rowhammer 模式，我们在最易受攻击的商业系统（即 Chromebook$_2$）上使用 TRRespass[15] 进行了三个实验，最多有 20 个攻击者。首先，我们将公开可用的 TRRespass 工具移植到 ARM 上，包括行扰乱和不可缓存的内存支持，以搜索位翻转。其次，我们在锤击工具中实现了模式以进行交叉验证。我们在相同条件下，使用 12 个侵略者多边模式之一的*四边模式*评估了锤击不可访问的内存。我们没有观察到多边模式的任何位翻转，而*四边模式*在相同的时间范围内诱导了 956 次翻转。这个实验得出的结论是，有些商品设备受到半双模式的影响，但不受（或受影响较小）其他最先进的模式的影响。第7节进一步讨论了半双和多边 Rowhammer。

5.2 确定 TRR 的作用

通过在商品系统上的实验，我们无法排除观察到的翻转是由 TRR（或其他行刷新）单独引起的*距离-1 翻转*，或者它们实际上是由*远侵略者*锤击引起的*距离-2 位翻转*。根据之前的工作 [54]，我们也观察到少量的*距离-2 位翻转*，其频率不足以解释半双（参见第5.3节）。此外，Helm 等人 [21] 发现了复杂的寻址功能，这些功能会根据实际物理位置而变化。为了排除这种可能的错误来源，我们使用具有 LPDDR4x 存储器的商用 SoC 平台来测量刷新（例如来自 TRR 的刷新）的影响。

我们从供应商那里获得了关于银行内部实际物理行位置与此 SoC 平台上的物理地址之间关系的精确但机密的

信息。对于这个系统，我们可以关闭和打开内存刷新。但是，这个开关不仅禁用了 TRR 刷新，还禁用了内存控制器发出的刷新，以符合刷新间隔（例如 64 毫秒）或 pTRR 发出的刷新。完全禁用刷新会使系统无法使用，因为 DRAM 单元在短时间内会失去电荷并损坏数据。为了仍然能够运行实际的软件，我们构建了一个在启用和禁用刷新之间交替的工作周期机制，我们将其表示为*舞蹈*（即在刷新开启和关闭之间跳舞）。在这些舞蹈实验中，我们启用 25% 的时间进行刷新，即启用 64 毫秒，禁用 192 毫秒。

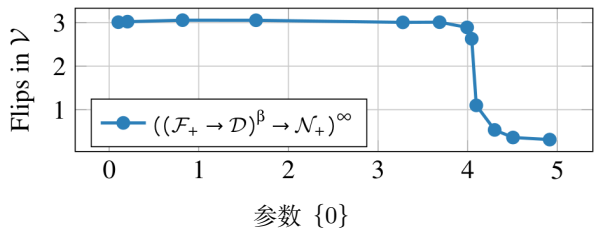


图 4：单面情况下，*受影响区域*中观察到的位翻转数量与稀释参数的关系。

*舞蹈*实验允许暂时限制刷新的次数，从而观察刷新与位翻转次数之间的相关性。虽然刷新被禁用，但它们不会无意中帮助我们的半双模式，即没有干扰的 TRR 刷新。虽然禁用刷新的窗口比标准刷新周期（例如 64 毫秒）长，但它足够短，可以避免不稳定。如果我们的假设 H 成立，即 TRR 刷新有助于半双效应，我们预期随着刷新次数的减少，位翻转的数量也会减少。

为了证明 TRR 刷新有助于观察到的第5.1节中的位翻转，我们进行了以下实验：我们设计了三**个**模式类别来演示半双效应：第一个类别表明该效应不是由*距离-2* 锤击引起的，第二个类别表明模拟的 TRR 刷新也会触发该效应，第三个类别表明只有模拟的 TRR 刷新单独不会触发该效应。用于验证假设的模式是围绕*受害者行*构建的。这些观察表明，只有 TRR *对近侵略者*的刷新（或其他访问）与我们*远侵略者*的访问相结合，才能触发半双效应，从而证实了我们的假设 H。在我们的实验中，我们使用了一个*容易受到 Rowhammer 攻击的受害者行*，即我们通常能够通过加权模式引发三次位翻转。

第一类验证了观察到的比特翻转并非仅由*距离为 2* 的锤击引起。单面模式S1访问*远侵略者*($\mathcal{F}_+$)和*诱饵* (D)。

$$\{77\}\ (\mathrm{S1})$$

双面图案D1用对较低*远侵略者*($\mathcal{F}_-$)的访问取代了这一*诱饵*。

$$(\mathcal{F}_+ \to \mathcal{F}_-)^{\infty} \qquad (\mathrm{D1})$$

在 TRR 实验中，我们观察到大量的位翻转，但在我们的实验中，禁用 TRR 刷新后，这两种模式都没有显示出任何大量的位翻转。观察到的位翻转次数太少，无法将其可视化，也无法通过*距离-2* Rowhammer 位翻转来解释半双效应（我们在第5.3节中进一步详细介绍）。因此，这表明 TRR 刷新有助于半双效应，支持我们的假设 H。

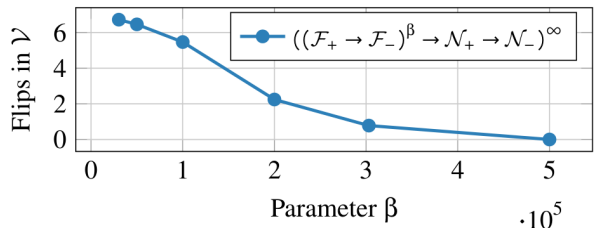


图 5：在双面情况下，在稀释参数下观察到的*受害者*位翻转数量。

第二类模式将半双模式扩展到禁用 TRR 时*接近攻击者*的位置，并模拟 TRR 刷新。在执行对*附近攻击者*的模拟 TRR 访问之前，模式S1和D1重复 β 次。单面模式S2通过访问*近侵扰者*($\mathcal{N}_+$)来模拟 TRR。

$$((\mathcal{F}_+ \to \mathcal{D})^{\beta} \to \mathcal{N}_+)^{\infty} \qquad (\mathrm{S2})$$

双面模式D2通过访问两个*近侵扰者*($\mathcal{N}_+, \mathcal{N}_-$)来模拟 TRR。

$$((\mathcal{F}_+ \to \mathcal{F}_-)^{\beta} \to \mathcal{N}_+ \to \mathcal{N}_-)^{\infty} \qquad (\mathrm{D2})$$

参数 {82} 是一个*稀释参数*，允许我们改变执行多少次模拟的 TRR 刷新，即在对*远侵略者*进行一定数量的访问后，我们对*近侵略者*进行访问。

对于低稀释参数（*即对近攻击者的访问次数非常高*），这些模式的行为类似于传统的单侧或双侧 Rowhammer 攻击，没有 TRR 保护，即导致许多位翻转。我们通过经验证实了这一点，如图4所示的单面图案S2和图5所示的双面图案D2。我们看到稀释参数 β 与位翻转次数之间存在相关性，即模拟刷新次数越少，位翻转次数越少。附录B中还提供了另一种表示方法。从这些观察中，我们得出结论，*接近侵略者的访问*导致了观察到的位翻转，支持了我们的假设 H。然而，我们还需要测试零假设，就像我们在下面做的那样。

为此，**第三类**模式实施安慰剂模式，以验证第二类模式中的位翻转并非仅由对近侵略者的访问引起（即零假设）。在给定的稀释度下，模式访问*诱饵*以保持与模式S2和D2相同的接近攻击者的整体访问速率。因此，单侧模式S3访问一个*附近的攻击者*($\mathcal{N}_+$)。

$$((\mathcal{D}_1 \to \mathcal{D}_2)^{\beta} \to \mathcal{N}_+)^{\infty} \qquad (\mathrm{S3})$$

双面图案D3访问($\mathcal{N}_+, \mathcal{N}_-$)。

$$((\mathcal{D}_1 \to \mathcal{D}_2)^{\beta} \to \mathcal{N}_+ \to \mathcal{N}_-)^{\infty} \qquad (\mathrm{D3})$$

表 2：在我们的 FPGA 设置中观察到的位翻转数量（括号中包含位翻转的行）。锤击持续时间决定了访问次数（锤击计数）。与稀释因子相比，锤击持续时间对比特翻转次数和受影响行的影响更大。即使在一个 64ms 的刷新间隔内，稀释因子为 3712，也适合 950272 次访问，其中 256 次访问模拟 TRR 的近侵略者（参见第3节），仍然会在所有 32 行中引起位翻转。因此，仅对近侵略者的 256 次访问以及对远侵略者的 950016 次访问就足以攻击任何行。

	Accesses	296 960	356 352	415 744	475 136	534 528	593 920	653 312	712 704	772 096	831 488	890 880	950 272
	Duration	20 ms	24 ms	28 ms	32 ms	36 ms	40 ms	44 ms	48 ms	52 ms	56 ms	60 ms	64 ms
Dilution Factor	58	1 (1)	3 (3)	5 (5)	6 (6)	15 (12)	26 (19)	35 (20)	44 (23)	57 (28)	83 (30)	115 (32)	173 (32)
	116	1 (1)	3 (3)	4 (4)	6 (6)	14 (11)	24 (19)	32 (20)	40 (22)	51 (27)	73 (30)	117 (32)	152 (32)
	232	1 (1)	3 (3)	4 (4)	5 (5)	12 (10)	24 (19)	31 (20)	39 (21)	51 (27)	68 (30)	112 (32)	149 (32)
	464	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	24 (18)	32 (20)	39 (21)	49 (26)	70 (30)	109 (32)	148 (32)
	928	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	25 (18)	32 (20)	39 (21)	49 (25)	70 (29)	108 (32)	146 (32)
	1856	0 (0)	2 (2)	3 (3)	5 (5)	11 (8)	22 (17)	32 (20)	37 (21)	49 (25)	66 (29)	110 (32)	140 (32)
	3712	0 (0)	2 (2)	3 (3)	5 (5)	10 (7)	22 (16)	30 (20)	37 (21)	49 (25)	64 (27)	99 (31)	139 (32)
	7424	0 (0)	2 (2)	3 (3)	5 (5)	8 (6)	18 (15)	29 (19)	36 (20)	48 (25)	66 (27)	92 (31)	128 (31)
	14 848	0 (0)	0 (0)	2 (2)	4 (4)	7 (6)	15 (12)	22 (15)	32 (19)	40 (22)	58 (27)	80 (30)	109 (30)
	29 696	0 (0)	0 (0)	2 (2)	2 (2)	3 (3)	8 (7)	11 (9)	19 (14)	28 (18)	41 (25)	57 (27)	82 (29)

表 3：FPGA 分析中使用的模块。M_1 不受半双刷新影响，M_3 在默认刷新窗口（64 毫秒）内受影响，M_2 在较长窗口内受影响。

Module	Freq.	Size	Ranks	Banks	Pins	Half-Double
M_1	2666	4 GB	1	8	x16	✗
M_2	3200	4 GB	1	8	x16	✓(>64 ms)
M_3	3200	8 GB	1	8	x16	✓

当改变稀释参数时，我们观察到第二类模式仍然在受害者中产生位翻转的点，而第三类模式不再产生位翻转。更具体地说，模式S3显示在模式S2之前位翻转减少。我们观察到，在双面情况下，模式D3的位翻转次数在较低的稀释参数下比模式D2的位翻转次数下降得更快。由于我们只访问了与受害者行 V 无关的诱饵 D_i，因此这种下降可以用对远侵略者 F 的缺失访问来解释，这支持了我们的假设 H。我们得出结论，TRR 抑制剂的额外访问有助于对受害者进行半双效应的锤击。

关键见解： TRR 刷新辅助半双，但不是根本原因。它们单独不会引起受害者的位翻转。

5.3 无噪声 FPGA 实验

为了确认我们的结果没有噪声，我们使用 *ZCU104* FPGA 平台[5]，在该平台上我们可以完全控制所有刷新和内存访问，并且对数据稳定性没有保留要求。与第5.2节相反，我们可以在基于 FPGA 的平台上禁用所有刷新，因为平台本身不会在 DIMM 中存储任何数据，即 FPGA 不将 DIMM 用作系统内存。

我们分析了表3中列出的三个现成的 DDR4 DIMM，发现 M_1 不易受半双效应的影响。虽然其他两个受到影响，但我们只能在 M_3 上以默认的 64 毫秒刷新间隔演示位翻转，而 M_2 需要加倍的刷新间隔（128 毫秒）。因此，在我们的分析中，我们重点关注了默认情况下容易受到 Half-Double 影响的 DIMM M_3。在我们的实验中，我们使用与第5.2节相同的模式，并在这种高度可控且无噪声的设置中确认了我们的结果。

$$((\mathcal{F}_+ \to \mathcal{F}_-)^{\beta} \to \mathcal{N}_+ \to \mathcal{N}_-)^{\infty} \tag{D2}$$

在模式D2中，我们分别锤击 32 行，并在本次实验中改变稀释参数 β 和总锤击持续时间（即锤击次数、访问次数）。此外，我们将锤击持续时间从 20 毫秒变化到 64 毫秒，步长为 4 毫秒。与稀释参数相反，我们引入了稀释因子 d_f。这个因素稍微改变了 β 的表示。模式D2的稀释因子与稀释参数之间的关系为 $d_f = \beta + 1$ 58 的稀释因子是指每 58 个锤子中有 1 个距离-1 锤子。因此，我们可以通过将总锤击数除以稀释因子来直接计算对附近攻击者的访问。稀释系数在每一步加倍时从 58 变化到 29696。

表2显示了该实验的结果，我们可以观察到两个预期的效果：首先，位翻转的数量随着锤子持续时间的增加而增加。无论测试的稀释系数如何，我们都可以在一个默认刷新间隔（64 毫秒）内诱导所有 32 行中的位翻转。其次，比特翻转的数量随着稀释度的提高而减少。然而，比特翻转的减少比锤子持续时间的减少要平坦得多。即使使用最高测试稀释度，我们也会在一个默认刷新间隔内将 32 行中的 29 行翻转。

为了强调半双效应与距离-1 和距离-2 Rowhammer 效应是不同的现象，我们在 FPGA 系统上测试了另外两种模式，并将其与之前实验中获得的结果进行了比较。我们使用距离-1 的双面 Rowhammer $(\mathcal{N}_+ \to \mathcal{N}_-)^{\infty}$和距离-2 的变体 $(\mathcal{F}_+ \to \mathcal{F}_-)^{\infty}$。我们再次锤击 32 行，并测量观察到的细胞和行翻转。

表 4：距离-1 双面锤击 $(\mathcal{N}_+ \to \mathcal{N}_-)^{\infty}$以及观察到的每个单元和行的位翻转。

[5] https://github.com/antmicro/litex-rowhammer-tester

Hammers	Time (ms)	Cells	Rows
18 000	1.212	2	1
24 000	1.616	23	18
30 000	2.020	136	31
36 000	2.425	495	32
42 000	2.829	1395	32
48 000	3.233	2870	32
54 000	3.637	5099	32
60 000	4.041	7749	32

Hammers	Time (ms)	Cells	Rows
4 000 000	270	1	1
5 000 000	336	1	1
6 000 000	404	2	2
7 000 000	472	2	2
8 000 000	538	3	3
9 000 000	606	2	2
10 000 000	674	3	3

表4显示了*距离-1*双面锤击的结果，其中将翻转引入所有 32 行所需的锤子数量仅为 36000。这比半双工攻击小 25 倍，表明半双工攻击不仅仅是*距离-1*的 Rowhammer。然而，36000 次访问也远远高于 TRR 实现可以在标准 64ms 刷新间隔内执行的操作。即使在低稀释系数（如 58）下，在一个刷新间隔内也需要大约 2 088 000 次访问，即大约是标准刷新间隔内访问次数的两倍。

表5显示了*距离-2*双面锤击，我们观察到需要 400 万次锤击访问才能获得一个*距离-2*位翻转，即访问次数是 64 毫秒刷新间隔内的四倍。因此，半双也可以不用*距离-2*位翻转来解释。

根据第5.2节，这再次表明 H 成立。根据表2的结果，我们可以模拟半双效应何时发生。稀释因子为 3712，锤子数量为 950272，对*附近攻击者*的访问总数为 256。因此，只有 256 次对*近侵略者*的访问，加上 950016 次对*远侵略者*的访问，就足以在 32 行中的每一行中引起翻转。然而，如果我们将这些数字与*距离-1*和*距离-2*实验中的等效访问进行比较，我们远远低于在这两种情况下看到哪怕一个位翻转所需的访问次数。

关键见解：如果*距离-1* Rowhammer 和*距离-2* Rowhammer 效应会引发半双效应，那么它们需要的访问次数将超过标准 64 毫秒刷新间隔内的访问次数。因此，我们得出结论，H 是 Half-Double 最合理的解释。

6 半双攻击漏洞

在本节中，我们将演示半双攻击的实际攻击能力。攻击分为多个阶段，每个阶段解决一个挑战（见第4.2节），最终从不可信的可执行文件中完全控制系统。我们的攻击旨在引起 PTE 的物理页帧号发生位翻转。如果损坏的页帧号指向攻击者控制的数据而不是原始页表，则对手可以伪造额外的页表条目。这使得对手可以任意读写整个系统内存。

表 5：*距离-2* 双面锤击 $(\mathcal{F}_+ \to \mathcal{F}_-)^\infty$以及观察到的每个单元和行的位翻转。

威胁模型。我们假设受害者在基于 ARM 或 x86 的系统上运行不受信任的可执行文件或 Android APP，以进行我们的攻击。我们的攻击不会利用操作系统或其他运行程序中的任何软件漏洞，而只会使用侧信道信息和操作系统提供的接口。此外，我们假设系统上使用的 LPDDR4x DRAM 同时受到 ECC 和 TRR 保护。然而，我们的攻击并不依赖于像以前针对 TRR 的 Rowhammer 攻击那样耗尽 TRR 资源 [15]。我们评估了 Chromebook、OnePlus 5T 和联想 T490s 的挑战，以展示其在多种架构和操作系统上的适用性（见附录A）。

从虚拟内存访问到半双模式。在我们的利用的第一步中，我们将虚拟地址映射到 DRAM 组内的实际物理行位置，这是使用半双模式进行锤击的构建块。虽然我们可以使用*四元模式*和*加权模式*，但我们更关注*四元模式*，因为它可以更快地引起位翻转。对于*四元组模式*，我们需要控制至少五行相邻行，其中中间行未映射，由受害者进程使用。通过 DRAM 寻址功能（参见第5.1.1节和图3），我们可以确定存储体和行内的物理位置。但是，所需的物理地址信息对于无特权可执行文件是不可用的。我们在第6.1节中解决了这一挑战（**C1**）。

诱导位翻转。在第二步中，我们需要将潜在的位翻转目标放置在正确的内存位置。模板位翻转在实际攻击中很可能不可复制，因为受害者行中的数据在模板化和攻击阶段是不同的，DRAM 的集成 ECC 机制取决于受害者行中存储的实际数据。因此，在攻击期间试图破坏目标数据时，模板化过程中的位翻转可能不会发生。因此，在第6.2节中，我们提出了两种解决 **C2** 的新方法。第一种技术使用一种可替代的模板化过程，该过程具有 ECC 感知能力。第二种技术称为*盲锤*，是模板制作的一种通用替代方法。然而，正如我们下面概述的那样，验证是否发生了可利用的位翻转，这本身就是一个挑战。

表 6：*Chromebook* 上的*页面距离模式*。每个图案都有一个独特的*页面距离*，以黄色突出显示。

P	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}	d_{16}	…
P_0	8	9	8	9	8	9	8	9	8	9	8	9	8	9	8	**1**	…
P_1	8	7	8	11	8	7	8	11	8	7	8	11	8	7	8	**3**	…
P_2	8	9	8	5	8	9	8	**13**	8	9	8	5	8	9	8	5	…
P_3	8	7	8	7	8	7	8	**15**	8	7	8	7	8	7	8	7	…

放置可利用的数据。在第三步中，我们使用 PTE 填充系统的内存。然而，由于现代基于 ARM 的设备出于性能原因

而受到地址空间限制，我们无法使用与之前的 Rowhammer 攻击相同的喷射技术。在第6.3节中，我们通过生成子进程来增加内存中页表的数量，从而解决了挑战 **C3**。

位翻转验证。我们不能直接进入被殴打的受害者行列。尝试访问损坏的映射也是致命的，因为 Linux 内核会在发生故障时检测到损坏的 PTE，并终止相应的用户空间进程。在第6.4节中，我们解决了 **C4** 问题，并使用幽灵攻击来防止我们的攻击应用程序发生不可恢复的崩溃。

结合所有步骤，我们获得了完整的**端到端利用**，可以对整个系统的内存进行读写访问。

6.1 C1：内存分配

要使用半双模式，对手需要访问同一银行内至少 5 行相邻的行。我们提出了三种不同的方法来解决这一挑战。如果已知 DRAM 寻址功能，则通过使用独特的银行访问模式，或者通过使用未知的基于 XOR 的 DRAM 寻址功能的结构，来访问大型页面

通过巨大的页面。Chromebook 上运行的 Chrome 操作系统以及 T490s 上运行的 Ubuntu 都激活了透明的大页面。对于 2 MB 的巨型页面，虚拟地址和物理地址的最低 21 位是相同的。这涵盖了我们在测试系统中找到相邻行所需的所有位，有效地解决了这一挑战（参见图3）。

通过 DRAM 寻址功能。禁用大页面可以缓解上述方法。不幸的是，HalfDouble 需要特定的行索引信息，这些信息超出了先前工作中的连续性信息 [14，34，47]。然而，我们可以结合 DRAM 寻址功能（参见图3）和 Linux 和 Chrome OS 中使用的伙伴分配器 [17] 的信息来检测连续的内存块，并重建我们需要的额外的物理地址位。

在处理连续的物理内存范围时，由于 DRAM 寻址功能，内存范围的页面分布在多个存储体上。我们现在只选择给定银行的页面，并迭代所有分配的页面以分析页面距离。页距是同一组中两页之间的距离。如果我们修复银行并分析分配的内存，我们会观察到 *Chromebook* 上的页面距离遵循四种模式之一。表6显示了这四种页距模式。这些图案的周期为 16，每个图案都有一个唯一的页距，用黄色突出显示。我们只发现了四种模式，因为我们可以跳过 DRAM 寻址功能的第 8 位，因为我们总是用虚拟地址来控制它，只留下八个剩余的存储体。然而，我们只观察到四种模式，因为两家银行共享相同的模式。

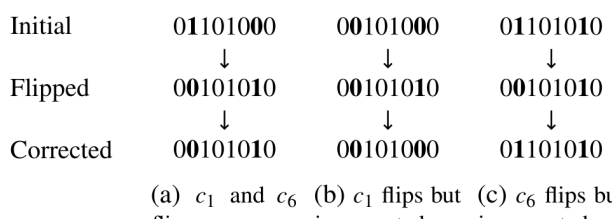
表 7：通过唯一的页距离重建物理地址位。

Page	From							To						
Distance	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}
1	B	1	1	1	1	1	1	$\bar{B}$	0	0	0	0	0	0
3	B	1	1	1	1	1	0	$\bar{B}$	0	0	0	0	0	1
13	B	1	1	1	0	0	1	$\bar{B}$	0	0	0	1	1	0
15	B	1	1	1	0	0	0	$\bar{B}$	0	0	0	1	1	1

所有四种页距模式在每个周期都有一个唯一的距离（1、3、13 或 15）。这种独特的值使我们能够重建物理地址位 12 至 17，因为 DRAM 寻址功能从位 12 开始（参见图3）。如果我们提前物理内存中唯一的页距离（4 kB），则物理地址的基础地址位 12 到 17 会发生变化。然而，由于使用定时侧通道 [34,47] 进行页距离分析，我们知道由唯一页距离提前的页属于同一存储体。因此，改变地址位不影响 DRAM 寻址功能的结果。每家银行只能出现一次这种情况。由于独特的页距离，我们知道我们落入两个银行中的一个，因此，这个分析只留下位 18 未知。表7显示了根据位 18，每个唯一页距离值的重建物理地址位。这 50% 的概率相当于攻击的相应减速。

通过 DRAM 寻址结构。我们通过在 Z3 定理证明器中制定基于 XOR 的 DRAM 寻址函数的一般结构来推广以前的方法 [12]。Schwarz 等人 [47] 使用类似的求解器从已知的 DRAM 寻址函数中恢复物理地址部分，Kwong 等人 [34] 使用求解器从连续存储器中恢复存储体信息。相比之下，我们通过存储体访问模式检索连续的内存信息，而无需知道 DRAM 寻址功能。

在漏洞利用中，我们记录了迭代虚拟地址范围的每一页时的银行访问模式，并通过行冲突时间侧通道确定相应的银行亲和度 [42]。求解器使用基于 XOR 的 DRAM 寻址功能的模式和底层结构来实现以下问题：当遍历连续的物理内存时，可以通过基于 *XOR* 的寻址功能生成这种存储体访问模式吗？如果约束条件无法满足，则底层内存范围不连续，或者寻址函数不是基于异或的。然而，如果约束是可满足的，求解器会找到寻址函数和物理起始偏移，从而生成这种访问模式。约束条件详见附录C。我们将 XOR 掩码的位数限制为仅覆盖物理地址位 12 至 20，因为页面偏移控制位低于 12，而高于 20 的位不需要用于查找相邻行。



(a) c_1 and c_6 (b) c_1 flips but (c) c_6 flips bu

图 6：使用 ECC 对 8 个存储单元进行错误纠正（来自 c_7 to c_0）。我们实际上从未看到过单个位翻转。

评价。使用 2 MB 页面进行内存扫描需要不到 10 秒，使用页面距离（即 Chromebook 上的 {97}，Chromebook 上的 {99}，OnePlus 5T 上的 {101} 和 T490s 上的 {102}）需要不到 3 分钟。

最后，我们通过生成由最多 128 页的均匀生成的连续内存块组成的 512 页的物理地址范围来评估求解器的正确性。这个内存范围通过 DRAM 寻址功能转换为存储体访问模式，其中我们额外对存储体索引进行了加密。我们改变求解器接收的输入模式长度，在整个内存范围内滑动求解器，并计算 F-score 度量。求解器在银行访问模式长度为 64 个样本，平均扫描速度为1.079 MB s^{-1}的情况下，实现了 0.97 的平均 F 分数。附录C提供了有关功能性能和正确性的更多详细信息。

6.2 C2：内存模板的替代方案

由于半导体生产差异，一些单元比其他单元更容易受到 Rowhammer 的影响 [54]。因此，大多数 Rowhammer 位翻转都是可重复的，并且它们的方向($0 \to 1$或$1 \to 0$)也是固定的 [31]。受影响的系统（见表1）使用具有 ECC 的 LPDDR4x DRAM，具有典型的单纠错码[6]。我们通过观察经验性地验证了这一点，我们没有看到单一的，而是只有双位的翻转。原因是 ECC 存储器需要至少两次位翻转才能在代码字内显示效果。否则，位翻转被纠正并且不可利用。图6用 8 个数据位（奇偶校验位未显示）直观地展示了这种效果。因此，我们提出了两种技术来应对半双攻击，以解决这种数据依赖性问题，一种是 ECC 存储器的改进模板技术，另一种是 *BlindHammering*，它不需要任何位翻转模板。

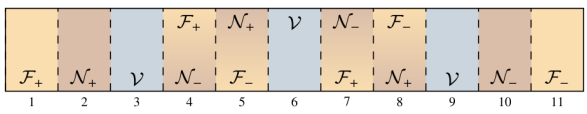


图 7：用于盲锤的斑马图案。请注意，近侵略者和远侵略者如何根据受害者行而变化。

ECC 感知模板。经典内存模板用特定的字节值填充攻击者行，并用该值的相反值填充受害者行。在我们的实验中，当用 0x55 填充攻击者行并用 0xaa 填充受害者行时，我们发现了许多弱单元。然而，当进入下一个利用阶段，受害者行中填满了 PTE 时，我们最初不再观察到任何翻转。原因如前所述，如图6所示：ECC 存储器上的位翻转取决于存储在单元中的数据 [34]。因此，我们需要调整模板化阶段，以包含我们目标数据的假定结构，即当目标为页帧编号时，使用伪 PTE。因此，在模板化过程中，我们用伪造的 PTE 填充受害者行，其中页帧编号与常规模板化方法中的填充方式相同，例如，我们用 0x55555555555555 填充攻击者行，用 0x68000AAAAAAFD3 填充受害者行。在评估过程中，我们跟踪在页帧编号字段偏移处产生翻转的所有地址。之后，我们使用这些地址在受害者行中的目标页面中诱导位翻转。

盲目锤击。ECC 模板的缺点是它需要对目标数据有精确的了解。*BlindHammering* 进一步概括了我们的攻击，并且对受害者行中的目标数据不做任何假设。相反，它通过不依赖于改变受害者数据时位翻转的可重复性来规避 ECC 数据依赖性问题。*Blind-Hammering* 跳过模板阶段，在潜在的受害者行中用真实的页表尽可能多地锤击行。*Blind-Hammering* 通过映射连续内存（参见第6.1节）然后取消映射部分分配的内存，为蓝色显示的受害者行腾出空间，从而创建斑马图案（参见图7）。本质上，它直接在不可访问的受害者行上执行模板，并使用受害者自己的数据来进行攻击。因此，盲目锤击的权衡与模板化阶段相似。虽然盲锤攻击能够定位 ECC 内存，但它有一个明显的缺点：攻击者无法简单地读取数据来检查位是否翻转。我们在第6.3节中进一步阐述了这个问题，激发了我们在第6.4节中解决的挑战 **C4**，即需要验证一个比特是否以一种可利用的方式翻转，而不会使攻击者进程崩溃。

评价。我们在 Chromebook 上评估了盲锤攻击 2，并观察到 30 个可利用的位翻转（13 次翻转 $0 \to 1$，11.6 小时内 17 次翻转$1 \to 0$）。这使我们平均每小时可利用 2.59 次翻转，或平均 23.2 分钟产生一次可利用的翻转。我们使用 Chromebook_2 的总位翻转与可利用位翻转的比率来估计其他设备上的利用时间。在 OnePlus 5T 上大约需要 6.4 小时，在 HTC U11 4.0 小时和 Chromebook_1 4.2 小时上，在 PTE 中翻转一个可利用的位。

6.3 C3：存储器准备（喷涂）

在本节中，我们使用我们的攻击目标（PTE）填充目标系统的内存。基于 ARM 的现代平台，即移动平台，可以将页表的级别从默认的 4 个页表级别减少到只有 3 个页表级别，以优化页表遍历（在 TLB 未命中时）的性能。然而，这也使每个进程可用的虚拟地址空间减少了 512 倍。由于受影响的设备（见表1）使用这种方法，我们只有 512 GB 的虚拟地址空间可用。虽然这仍然远远超过设备所拥有的物理内存量，但它严重限制了使用文件映射进行页表喷射的实用性。之前的工作已经使用文件映射和其他内存映射来用页表填充内存 [19，44，48，51]。然而，由于只有 512 GB 的虚拟地址空间可用，我们只能创建需要少于 262 144 个页表的映射，即占用 1 GB 的内存。因此，我们只能占用 Chromebook 上 4 GB 可用空间的 25%。这将攻击持续时间增加了 8 倍以上（因为并非所有页面都可以映射）。

为了绕过这种加剧的影响，我们提出了一种名为“儿童喷雾”的新技术。*Child Spray* 不是只通过映射来分配页表来喷射我们自己的虚拟内存，而是生成与父进程共享内存的子进程。共享内存仅存在于物理内存中一次，但每个进程都有自己的页表层次结构，有效地将页表散布在物理内存中。*Child Spray* 的唯一缺点是，一个被攻击的 PTE 可以指向一个子进程的页表，从而导致额外的工程步骤才能成功利用。

通过这种喷涂方法和盲目锤击，位翻转现在可以在任何时间发生在任何页表中，因为我们不知道哪些单元是脆弱的，以及页表在哪里。我们可以通过检查共享内存映射是否仍然具有预期的内容来定期检查页表是否发生了变化。然而，与模板位翻转相反，页表中的盲目位翻转是不可预测的，因此 PTE 经常变得无效。因此，我们需要一种方法来测试页表中的位翻转是否发生，而不会导致攻击者进程崩溃。我们通过以下方式解决这一挑战。**评价**。*Child Spray* 在 Chromebook 上以 79.39 MBs^{-1}(n=10, $\sigma_{\bar{x}}$=0.24) 的速度运行两个子进程，在 Chromebook 上以 54.09 $\text{MB}\,\text{s}^{-1}$ 54.09 $\text{MB}\,\text{s}^{-1}$ (n=10, $\sigma_{\bar{x}}$=1.437) 的速度运行，

[6]我们没有看到任何由于错误检测而导致的冻结，这表明它只是单错误纠正，不支持双错误检测。

清单 1：我们的推测 *Oracle* 的示例代码。攻击者可以了解指针 [0] 到指针 [4] 是否是可访问的内存位置，或者是否会引发 CPU 故障。如果检测到故障，攻击者会单独探测每个地址。

```
1 if (misprediction)
2     access(probe + (pointer[0] & 1) + ... + (pointer[4] & 1));
3 if (flush_reload(probe) == CACHE_HIT)
4     // Report valid address
```

{117}，联想 T490s 为 99.88 MBs {118}（$n = 10$，{119}。因此，平均不到 1 分钟内存中就充满了页表。T490s 不需要 *Child Spray*，因为系统支持 4 个页表级别。

6.4 C4：稳健的位翻转验证

通过盲锤，我们无法验证位翻转是否成功。读取损坏的数据（如模板化）是不可能的，因为它不在我们的流程中。由于操作系统在访问错误时检测到损坏，访问可能重新映射的内存通常会因损坏的 PTE（例如，映射到无效的物理内存区域，或设置保留位）而导致攻击者的进程崩溃。因此，我们开发了一种新技术，将半双行哈默攻击与幽灵 [33] 侧通道机制相结合，使我们能够安全地确定是否可以访问某个地址，或者访问该地址是否会破坏攻击者的进程。借助推测性 *Oracle*，我们可以检查映射是否损坏，而不会触发操作系统自身的检测。

6.4.1 推测性预言机

如果物理页帧号指向一个非法的内存位置，那么对其进行读取或写入操作将引发 CPU 故障，例如数据异常终止 [2]。由于 CPU 故障在不易受熔断影响的系统上进行推测执行时无法成功 [37]，我们可以使用推测执行来确定位翻转是以有缺陷的方式还是以可利用的方式破坏了条目。这使我们能够避免访问，这些访问会使操作系统终止我们的攻击过程。

我们的推测性预言机在基于 ARM 的移动电话上使用类似于 Lipp 等人 [37] 在 Meltdown 攻击中使用的 Spectre 进行异常抑制。我们的 Chromebook 使用带有 ARM 内核的 Mediatek MT8183 SoC，不易受到 Meltdown 的影响 [3]。因此，取决于故障负载的负载不是用户可见的，而是在此 ARM 微架构上执行的。

我们的推测性预言机通过训练分支预测器来使用异常抑制，以暂时执行探测代码 [37]，参见清单1。我们根据指针（要测试的地址）暂时加载带有偏移量的探测器。对于推测性预言有两种可能的情况：指针是**有效的**，因此，它的值被转发到探测负载。因此，探测器被加载到缓存中。指针无效，因此，探测加载未被执行，因此，未被加载到缓存中。使用 Flush+Reload[57]，我们确定探针是否被缓存，从而确定指针是否有效。

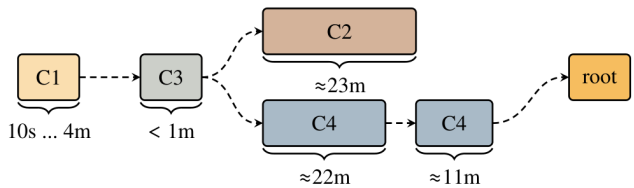


图 8：在 Chromebook 上执行的端到端漏洞利用的持续时间。对于挑战 **C1** 和 **C4**，可能有更快的替代方案（参见第6.1节和第6.4节）。总运行时间受挑战 **C2** 的限制，即诱发可利用的位翻转所需的时间。之后，平均需要一半的 **C4** 来找到位翻转。

由于我们的探测工具在瞬态域中运行，因此可以在加载之前纠正错误预测的分支，或者分支可能根本不会被错误预测。因此，即使指针有效，探测器也可能不会被缓存。因此，我们多次重复幽灵攻击，如果指针有效，则更有可能看到缓存命中。然而，如果指针无效，探测器永远不会被加载到缓存中，因此，如果我们不能在重复一定次数后观察到缓存命中，我们可以推断指针以很高的概率无效。此外，我们可以通过将多个地址链接到探测地址作为额外的依赖关系来一次探测多个地址，如清单1所示。这显著提高了运行时间，因为它允许在单独探测候选地址之前一次粗略扫描多个候选地址。

评价。我们评估了我们的推测预言机的成功率和运行时间。由于只有在测试地址有效的情况下才能观察到探测器上的缓存命中，因此我们的方法具有零假阴性率（将无效地址分类为有效）。因此，如果我们在触发探测工具的重复试验中没有观察到缓存命中，我们只能将有效地址错误分类为无效。

我们评估了目标地址分类的成功率和不同数量的 Spectre 攻击的运行时间。我们一次探测 5 个地址，其中所有地址都有效，或者其中一个随机地址无效。我们重复实验 10000 次，其中 5000 次针对有效地址，5000 次针对无效地址。通过一次探测尝试，我们在 Chromebook 上已经达到了 99.01% 的成功率，平均运行时间为 0.008 毫秒（n=10000，σ_{fix}=0.002）。Chromebook {122} 在两次尝试中成功率达到 99.68 %，平均运行时间为 0.025 毫秒（n = 10 000，{123}）。在 OnePlus 5T 上，我们通过三次尝试，成功率达到 99.24 %，运行时间为 0.034 毫秒 {124}。我们还评估了基于 x86 系统的技术，出于性能原因，我们在该系统中使用了 RSB 预测失误。联想 T490s 在 20 次尝试中成功率达到 99.94 %，运行时间为 0.018 毫秒（n = 10 000，{125}）。

表 8：挑战概述、替代方案和跨多个平台的可用性。

	Alternative	Requirement	Available on	Prior Work
C1	Physical Address Access	OS-enabled	Linux-based Systems	[48]
	Huge Pages	OS-enabled	Linux-based Systems	[13,19]
	Bank Differences	Known Functions	DDR-based Memory	[34,47]
	Solver	XOR-based Functions	DDR-based Memory	[34,47]
C2	Templating	no ECC	Systems without ECC	[8,13,15,19,44,48,51]
	Blind-Hammering	Half-Double affected	see Tables 1 and 3	-
	ECC-Aware Templating	Half-Double affected	see Tables 1 and 3	[11,34]
C3	Spray Children	`fork`	ARM64, x86	-
	Spray Page Tables	4 Page Table Levels	ARM64, x86	[19,44,48,51]
C4	`vfork`	OS-enabled	Linux-based Systems	
	Speculative Oracle	Hardware	ARM64, x86	-

因此，由于几乎所有地址都保持有效且没有位翻转，因此使用我们的推测预言机进行验证需要 19.0 分钟的 CPU 时间来扫描 2GB 的 PTE 以查找位翻转。但是，在盲目锤击期间，该扫描可以在后台的第二个内核上运行（见图8）。

建筑替代方案。作为推测式预言机的替代方案，我们提出了一种架构方法，即使用 vforksystem 调用。vfork 与 fork 类似，创建调用进程的精确副本，唯一的区别是不复制页表。其主要目的是为子进程提供一个更快的 fork 版本，通过 exec 立即执行另一个进程。我们的 vfork oracle 创建一个子进程，扫描 2GB 的 PTE 以查找位翻转。如果页面转换被破坏，子进程将被内核杀死，否则会干净利落地返回。通过检查子进程的死亡方式，我们可以知道访问地址范围是否安全。如果子进程被杀死，我们使用共享内存将最后访问的地址传递给父进程。这将 vfork 调用的数量减少到每个损坏的 PTE 一个。

{126} 在 Chromebook 上 {127}，{128} {129} 在 T490s 上。带宽数量接近系统的最大内存带宽。因此，通过这种方法，在 Chromebook 上，验证仅消耗 56 秒的 CPU 时间，在 T490s 上仅消耗 4.3 秒，即可扫描 2GB 的 PTE 以查找位翻转。这种方法的缺点是，通过禁用我们在内核中对 vforkin 的特定使用来减轻这种影响是微不足道的，例如，在 OnePlus 5T 上，vfork 指令被别名化为 fork。尽管如此，推测性的预言仍然可用。

6.5 端到端攻击评估

图8显示了所有攻击步骤的组合运行时间，其中连续性（**C1**）和喷射（**C3**）的时间不到 5 分钟。在我们的 *Chromebook* 上，**盲锤（C2）**平均需要 23.2 分钟才能找到可利用的位翻转。第四，投机 *Oracle*（**C4**）与 *BlindHammering* 并行运行，耗时 22 分钟。在位翻转之后，漏洞清理、扫描物理内存、设置页表以方便任意读写，这些操作在不到 3 分钟的时间内完成。因此，总运行时间通常保持在 45 分钟以下，但会随着可利用位翻转发生的时间而变化。在我们的其他设备上，总利用时间主要由翻转可利用位所需的时间决定，因为相比之下其他利用步骤可以忽略不计（参见第6.2节）。表8总结了挑战和适用解决方案的要求。

7 讨论

最近，*TRRespass*[15] 对锤击模式进行了模糊处理，并表明各种受 TRR 保护的 DDR4 DIMM 仍然容易受到 Rowhammer 的影响。在测试的 42 个模块中，生成的多面（3 至 19 面）图案在 13 个模块上有效。他们利用的根本效果是 TRR 实现的优化，其中 DRAM 模块只计算对有限数量的行的访问，攻击者可以耗尽这些行。然后，TRR 无法跟踪对邻近攻击者（距离-*1*）行的访问次数。与 TRRespass [15] 和 Smash [13] 的多边 Rowhammer 模式相比，我们的半双模式不依赖于 TRR 资源的耗尽。相反，这些模式直接将 TRR 刷新机制整合到攻击中。因此，我们的模式甚至适用于 TRR 机制完美检测和缓解距离-*1* Rowhammer 的情况。

适用于其他系统（包括 x86）。我们还评估了其他系统（尤其是 x86）上端到端攻击的所有构建块。半双倍攻击也存在于 x86 系统上受 TRR 保护的 DDR4 内存中，一些 x86 处理器（如 Xeon）使用 pTRR 对附近的攻击者进行类似的访问，因此可用于半双倍攻击。连续内存检测也适用于其他基于 Arm 和 x86 的系统。*Blind-Hammering* 在 x86 和 Arm 上都能工作。在 x86 系统上喷涂更容易，因为不需要子进程，并且之前工作的技术仍然适用。我们表明，推测预言机可以针对特定系统进行调整（例如，使用 Spectre-RSB 而不是 Spectre-PHT，或者调整阈值），也可以被不依赖于微体系结构行为的 vfork 所取代。

缓解措施。为了缓解半双倍，我们讨论了一种短期防御措施来保护下一代 DRAM 芯片，以及一种更通用的 Rowhammer 防御措施。首先，我们提出了(p)TRR±2，将现有的面向受害者的基于刷新的缓解措施扩展到包括距离为 *2* 的攻击者。这种缓解措施将最大限度地减少硬件更改，因为只有基于 TRR 的设计的抑制剂需要适应刷新额外的行。此外，如表2的结果所示，更复杂的设计将以比距离 *1* 行更低的频率刷新距离 *2* 行。其次，随着 DRAM 单元密度的进一步提高，我们假设半双效应的影响将会增加，从而增加了对更通用的 Rowhammer 保护的需求。Saileshwar 等人 [46] 提出用面向攻击者的防御来取代面向受害者的防御，如 TRR。他们建议使用置换层在达到某个激活计数后，将攻击者行与同一组内的另一行进行交换。这种机制在统计上打破了攻击者和受害者行之间的局部性，因此极不可能连续攻击同一个受害者。

为了使受影响的系统对我们的端到端漏洞利用更加坚固，我们建议解决连续内存分配和位翻转验证问题（参见第6.1节和第6.4节）。如果底层系统分配器确保分配永远不会返回连续的页面，那么攻击者必须采取蛮力方法来找到正确的远侵略者，以诱导半双位翻转。此外，`vfork`系统调用可以别名化为 fork，从而将此小工具从系统中删除。

8 结论

我们提出了一种新的、完全的 Rowhammer 效应，Half-Double。半双通过将大量访问“远”攻击者（距离为 2）与少量（数十个）访问“近”攻击者（距离为 1）相结合，在受害者中引发错误。这在具有缓解刷新的 DRAM 上是有

问题的，因为 Rowhammer 保护(e.g., "TRR")，保护措施会隐式访问附近的攻击者，因此，它不是阻止 Rowhammer 协助 Half-Double 引发位翻转。我们对 Half-Double 进行了全面评估，并展示了它在端到端 Rowhammer 攻击中的实际意义。为了克服对最近现成的设备进行端到端攻击的挑战，我们使用了侧信道攻击、一种称为盲锤的新技术、一种新的页表喷射技术和一种基于幽灵的防崩溃位翻转验证。我们的端到端概念验证攻击，即半双倍攻击，使攻击者可以在完全最新的系统上对整个内存进行任意读写访问，正如我们在具有 ECC 和 TRR 保护的 LPDDR4x 内存的 Chromebook 上展示的那样，平均运行时间仅为 45 分钟。

致谢

我们感谢匿名审稿人，尤其是我们的指导者 Shaanan Cohney，感谢他们的指导、评论和建议。亚马逊的慷慨捐赠为该项目提供了部分资金。本文中表达的任何意见、发现、结论或建议都是作者的观点，不一定反映资助方的观点。

工具书类

[1] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting

anti-DoS measures to accelerate Rowhammer attacks. In *HOST*, 2017.

[2] ARM. *ARM Architecture Reference Manual ARMv8*. ARM, 2013.

[3] ARM. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism, 2018. URL: https://developer. arm.com/support/arm-security-updates/ speculative-processor-vulnerability.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. *ACM SIGPLAN Notices*, 2016.

[5] K.S. Bains, J.B. Halbert, C.P. Mozak, T.Z. Schoenborn, and Z. Greenfield. Row hammer refresh command, January 2014. US Patent App. 13/539,415. URL: https://google.com/patents/US20140006703.

[6] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical dram mappings for rowhammer attacks. In *International Verification and Security Workshop (IVSW)*, 2018.

[7] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.

[8] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*, 2016.

[9] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn' t touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, 2017.

[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[11] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *S&P*, 2019.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337-340. Springer, 2008.

[13] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *USENIX Security Symposium*, 2021.

[14] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, 2018.

[15] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, 2020.

[16] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. ARMOR: A Run-time Memory Hot-Row Detector, 2015. URL: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer.

[17] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, 2004.

[18] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O' Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[21] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable reverse engineering of intel dram addressing using performance counters. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020.

[22] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters - CPU Hard ware Performance Counters for Security. In *Black Hat Briefings*, 2015.

[23] Rei-Fu Huang, Hao-Yu Yang, Mango C.-T. Chao, and Shih-Chin Lin. Alternate hammering test for application specific DRAMs and an industrial case study. In *Annual Design Automation Conference (DAC)*, 2012.

[24] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping microarchitectural attacks before execution. ePrint 2016/1196, 2017.

[25] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*, 2019.

[26] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[27] Jedec Solid State Technology Association. Low Power Double Data Rate 3, 2013. URL: http://www.jedec.org/standards-documents/docs/jesd209-4a.

[28] JEDEC Solid State Technology Association. Low Power Double Data Rate 4, 2017. URL: http://www.jedec. org/standards-documents/docs/jesd209-4b.

[29] Matthias Jung, Carl C Rheinländer, Christian Weis, and Norbert Wehn. Reverse engineering of drams: Row hammer with crosshair. In *International Symposium on Memory Systems*, 2016.

[30] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters*, 14, 2015.

[31] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[32] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace, 2015. URL: https://git.kernel.org/cgit/linux/ kernel/git/torvalds/linux.git/commit/?id= ab676b7d6fbf4b294bf198fb27

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[34] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P*, 2020.

[35] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In *ISCA*, 2019.

[36] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. *arXiv:1711.08002*, 2017.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[38] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. *ACM SIGARCH Computer Architecture News*, 40(3):1-12, 2012.

[39] Onur Mutlu. The RowHammer problem and other issues we may face as memory becomes denser. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[40] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*, 2020.

[41] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[42] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[43] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Introducing Half-Double: New hammering technique for DRAM Rowhammer bug, 2021. URL: https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html.

[44] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In *International Symposium on Hardware Oriented Security and Trust*, 2016.

[45] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, 2016.

[46] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *ASPLOS*, pages 10561069, 2022.

[47] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[48] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings*, 2015.

[49] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *RAID*, 2018.

[50] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC*, 2018.

[51] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[52] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *DIMVA*, 2018.

[53] Saru Vig, Siew-Kei Lam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Rapid detection of rowhammer attacks using dynamic skewed hash tree. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2018.

[54] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On dram rowhammer and the physics of insecurity. *IEEE Transactions on Electron Devices*, 2021.

[55] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *arXiv:1912.11523*, 2019.

[56] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, 2016.

[57] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[58] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID*, 2016.

[59] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. TeleHammer: Cross-PrivilegeBoundary Rowhammer through Implicit Accesses. *arXiv:1912.03076*, 2019.

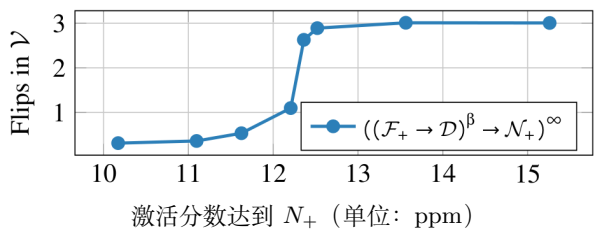


图 9：在单侧情况下，在 100 万次访问中，受害者对近侵略者的访问部分中观察到的位翻转次数。

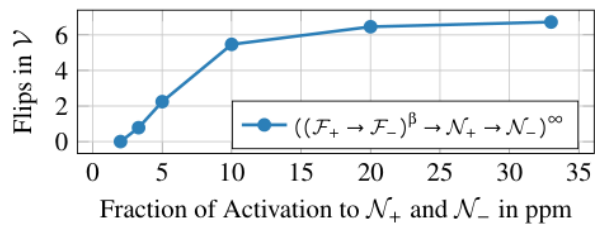


图 10：在双面情况下，在 100 万次访问中，受害者对邻近攻击者$(\mathcal{N}_+, \mathcal{N}_-)$的访问部分中观察到的位翻转次数。

评估记忆部分总结

表9提供了我们在本文中获得和评估的所有 DDR4、LPDDR4 和 LPDDR4x 部件的完整列表。首先，我们通过 ZCU104 FPGA 平台将部件划分为第5.3节中分析的 DIMM。对于这些 DIMM，我们可以完全控制 DRAM 寻址和刷新间隔，以评估距离 *1*、距离 *2* 和基于半双锤击的

性能（参见表2、4和5）。其次，我们在第5.1节评估了移动设备和 PC 部件，其中我们首先对 DRAM 寻址功能 [42] 进行了逆向工程，并使用四元组模式进行锤击。表1显示了受影响设备的最终位翻转。我们观察到受 HalfDouble 影响的总体 7 个部分。对于未受影响的移动和 PC 部件，我们只能推测底层内存是否具有半双抗性，或者未知的行扰是否阻止了安装四元模式。因此，我们不能得出结论，底层内存确实不受 Half-Double 的影响。

B 模拟 TRR 下比特翻转的替代表示

在本节中，我们给出了图4和图5的不同表示。我们没有使用 β 参数，而是绘制了在 *100 万次访问*中，*受害者*中观察到的比特翻转次数与*接近攻击者*的分数之比。图9显示了单面情况下的数据（即与图4相同的数据）。图10显示了双面情况下的数据（即与图5相同的数据），

表 9：所有被评估的内存部件，包括它们的生产日期、底层内存结构以及我们评估它们的测试系统或操作系统的信息。我们指出了明显受到 Half-Double 影响的部件。

	Name	Year-Week	CPU / SoC	RAM	Size	Manufacturer	Test System / Operating System	Half-Double
DIMMs	M_1	2019-48	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✗
	M_2	2020-32	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✓
	M_3	2020-42	-	DDR4	8 GB	Confidential	ZCU104 FPGA Platform	✓
Mobile Devices	Chromebook$_1$	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Chromebook$_2$	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Pixel 3	2018-40	SDM845	LPDDR4x	4 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.9	✓
	HTC U11	2017-18	MSM8998	LPDDR4x	4 GB	Unknown	Android 9 with Kernel Version 4.4	✓
	OnePlus 5T	2017-47	SDM835	LPDDR4x	6 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.4	✓
	Samsung S9 (SM-G960F/DS)	2018-10	Exynos 9810	LPDDR4x	4 GB	Unknown	Android 10 with Kernel Version 4.9	✗
	Samsung S7 (SM-G935F)	2016-10	Exynos 8890	LPDDR4	4 GB	Unknown	Android 8 with Kernel Version 3.18	✗
PC	Lenovo T490s	2019-13	Intel i5-8265U	DDR4	16 GB	Samsung	Ubuntu 20.04.3 LTS with Kernel Version 5.11	✗
	Minisforum TL50 MiniPC	2021-43	Intel i5-1135G7	LPDDR4	16 GB	SK Hynix	Ubuntu 20.04.3 LTS with Kernel Version 5.13	✗
	Minisforum X35G MiniPC	2020-43	Intel i3-1005G1	LPDDR4	16 GB	Micron	Ubuntu 20.04.1 LTS with Kernel Version 5.4	✗

C 连续内存求解器

本节详细介绍了求解器的实现，以及基于真实设备的重建 DRAM 寻址功能的额外性能和正确性分析。**求解器实现**。求解器使用 Z3 定理证明器 [12] 实现。为了检测连续的内存区域，我们首先将基于 XOR 的 DRAM 寻址函数的结构作为约束来实现。求解器求解 N 个异或掩码，我们将其表示为 M_i f 或$0 \le i < N$，以及连续物理范围 B 的基址。总体思路是，对于每个给定的输入样本（即页面），增加基址 B，就像该范围是连续的一样，否则将导致无法满足的约束。每个输入样本 x_i 都来自一个集合 X_i，其中 x_i 是当前样本索引。首先，我们定义函数$F_i(x)$，该函数计算物理内存范围内第 x 页的第 i 个设置位：

$$F_i(x) = \bigoplus (M_i \wedge (B + x \cdot 0x1000)).$$

我们将$\bigoplus(x)$表示为对 x 的所有位进行异或运算，将 $\wedge$ 表示为按位与运算。其次，我们将集合索引定义为每个集合位的二进制连接：

$$S(x) = F_0(x) \parallel \cdots \parallel F_{N-1}(x).$$

对于包含在一个集合中的每个页面，我们强制该集合索引与该集合的第一个成员的索引相同，即集合x_i^0: 的第一页

$$\text{assert}(S(x_i) = S(x_i^0)) \forall x_i \in \mathbb{X}_i.$$

最后，我们限制所有其他未包含在一个集合中的页面偏移量必须具有不同的集合索引：

$$\text{assert}(S(y) \neq S(x_i^0)) \forall y \notin \mathbb{X}_i.$$

如果满足这些约束，则可以通过基于 XOR 的 DRAM 寻址函数生成基础范围。如果不满足，则存储器区域不是连续的，或者寻址功能不是基于 XOR 的。

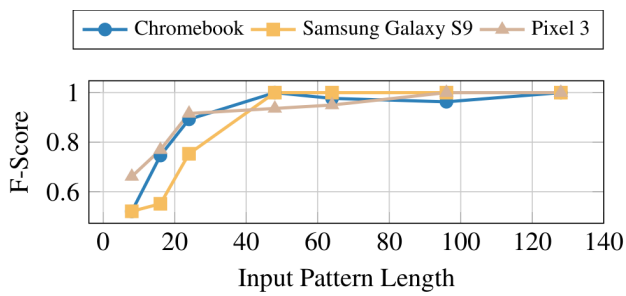


图 11：基于求解器的邻接检测在不同模式长度下的 F-分数。

评价。我们通过随机连接最多 128 页的连续范围，并将这些物理页面转换为具有真实逆向工程 DRAM 寻址功能的存储体访问模式，来验证求解器的正确性。在评估中，我们将求解器滑动到这个生成的图案上，并改变求解器接收的输入样本的数量。图11显示了不同 DRAM 寻址函数和各种模式长度的结果 F 分数度量。我们观察到 F-score 随着模式长度的增加而增加。正如预期的那样，因为求解器在内部有更多的约束条件可以依赖。我们看到，在模式长度为 128 页的情况下，求解器对每个模式的 F-score 为 > 0.99。