基于虚拟化内存隔离的 Rowhammer 攻击防护机制

石培涛, 刘宇涛, 陈海波

上海交通大学并行与分布式系统研究所, 上海 中国 200240

摘要 随着虚拟化技术的发展与云计算的流行, 虚拟化环境下的安全防护问题一直受到广泛的关注。最近的 Rowhammer 攻击打破了人们对于硬件的信赖, 同时基于 Rowhammer 攻击的各种攻击方式已经威胁到了虚拟化环境下的虚拟机监视器以及其他虚拟机的安全。目前业界已有的对 Rowhammer 攻击的防御机制或者局限于修改物理硬件, 或者无法很好的部署在虚拟化环境下。本文提出一种方案, 该方案实现了一套在虚拟机监视器层面的 Rowhammer 感知的内存分配机制, 能够在虚拟机监视器层面以虚拟机的粒度进行 Rowhammer 攻击的隔离防护。测试表明, 该方案能够在不修改硬件, 以及引入较小的性能开销(小于 6%的运行时开销和小于 0.1%的内存开销)的前提下, 成功阻止从虚拟机到虚拟机监视器以及跨虚拟机的 Rowhammer 攻击。

关键词 虚拟化安全; 内存分配; Rowhammer 攻击; Xen
中图分类号: TP309.2 DOI 号 10.19363/j.cnki.cn10-1380/tn.2017.10.001

Defense against Rowhammer Attack with Memory Isolation in Virtualized Environments

SHI Peitao, LIU Yutao, CHEN Haibo

Institution of Parallel and Distributed Systems, Shanghai Jiaotong University, Shanghai 200240, China

Abstract The virtualization security has increasingly gained widespread attention with the spreading of cloud computation in recent years. And some common hardware-software contracts which were supposed to be the base of security system have been violated by some attacks like “rowhammer”. Adversaries have used rowhammer attack to break the isolation between virtual machines and hypervisor as well as to threaten the security in the virtualization environment. To date, all the known defenses against rowhammer either require the modification on hardware or are hard to be deployed in the virtualization environment. We present a novel method, which can prevent the spreading of rowhammer attacks by isolating the memory of different security domains (e.g., the kernel of hypervisor and the virtual machines). We extent the physical memory allocator of Xen to be aware of rowhammer. Our solution does not require any modification to the hardware, and it is transparent to the guest VMs. The evaluation shows its effectiveness in preventing against rowhammer attacks, as well as the efficiency in introducing negligible overhead (the runtime performance overhead is lower than 6%, and the memory cost is lower than 0.1%).

Key words virtualization security; rowhammer attack; memory allocator; Xen

1 引言

随着虚拟化技术的不断发展, 云计算平台也得到了技术支持, 越来越多的企业将自身的服务部署在拥有完善的云生态的云平台上[1]。因此虚拟化环境下的安全防护得到了广泛关注[2]。在过去的安全防护中我们通常将处理器、内存等硬件组成认为是可信的, 例如认为用户进程在没有操作系统对应权限时是无法直接修改内存上内核的数据[3]。但是最近的一些研究[4]发现这种假设被破坏了。动态随机存取内存(DRAM)在设计中遗留的缺陷有可能被攻击者利用软件来触发, 从而攻击操作系统或者重要软件。

Rowhammer 攻击[3]就是其中一种针对 DRAM 的攻击。它利用内存设计上被称作为“Disturbance Error”的缺陷: 当内存中的基本单元被短时间内频繁地充电和放电时, 有一定概率泄漏电量给附近的行地址线路(DRAM Row)的基本单元的电容, 最终导致其附近行地址线路的基本单元内存储的数据发生突变, 这使得内存上的数据从物理层面被修改。攻击者可以利用该缺陷修改内存上原本不具有读写权限

通讯作者: 陈海波, 博士, 教授, haibochen@sjtu.edu.cn。
本课题得到国家重点研发计划 No. 2016YFB1000104 资助。
收稿日期: 2017-07-18; 修改日期: 2017-08-10; 定稿日期: 2017-08-23

的数据, 从而扩大缺陷影响, 最后实现破坏系统隔离性等攻击。

在 2014 年 Kim.Y [3]提出 Rowhammer 攻击后, 已经发现在目前市场上主流的 DDR3 以及部分 DDR4 内存中都存在这种缺陷。2015 年 Google 提出了利用 Rowhammer 攻破 Chrome 浏览器内部的沙盒机制[5]; RowhammerJS[6]则实现了利用高级语言直接远程攻击网页服务器; 2016 年的 Drammer [7]攻击在 ARM 平台下利用 Rowhammer 攻击让安卓环境下恶意程序获取 root 权限。同时还有文献[8, 9]实现了针对虚拟化环境下的跨虚拟机攻击等多种攻击方式。这些攻击方式不仅跨越了 X86 和 ARM 等处理器架构, 还跨越了普通操作系统和虚拟化环境。

目前已有不少针对 Rowhammer 攻击的防御机制。但是大多数是对于 DRAM 内存的物理架构上的优化[10,11], 这些物理硬件上的改进并不能很好地部署在已有的系统上。而利用纯软件方法来防御 Rowhammer 攻击的机制已有 ANVIL[12]与 G-CATT[13]等。前者针对在普通操作系统环境下的 Rowhammer 攻击进行模式匹配检测并且阻止攻击。但目前已经有攻击方式通过内存直接访问(Direct Memory Access)的方式来绕过 ANVIL 的检查[6]。后者则是首次提出了利用物理内存隔离的思想来阻止 Rowhammer 攻击的蔓延, 但是它只能阻止普通操作系统环境下恶意用户态进程攻击操作系统内核这一类型的 Rowhammer 攻击。以上的纯软件的防御方法均不能很好地复用在虚拟化环境下的 Rowhammer 攻击的防御上。

针对上述问题, 本文针对虚拟化环境下的 Rowhammer 攻击, 提出了一套纯软件的防御机制——RDXA (Rowhammer Defense on Xen Allocator)系统。该系统在虚拟机监视器层通过实现一套 Rowhammer 感知的内存分配机制, 保证虚拟机监视器与虚拟机之间、不同的虚拟机之间所对应的物理内存不会被物理存放在同一个 bank 的相邻 row 上面, 从而防止不同区域的内存通过 Rowhammer 攻击的方式相互影响。

本文的主要贡献如下:

1) 提出了一个不需要对物理硬件进行修改, 同时对于上层的客户虚拟机透明的 Rowhammer 解决方案 RDXA 系统。

2) RDXA 系统实现了在 DRAM 物理内存层面为虚拟机以及虚拟机监视器提供内存隔离机制。相较于以往的物理地址隔离, RDXA 系统通过更加底层的内存隔离来阻止跨虚拟机到虚拟机、虚拟机到虚拟机监视器之间的 Rowhammer 攻击。

3) 本文实现了一个利用时间旁路分析来逆向物理地址到内存地址的映射关系的工具。并用其分析出了本文所使用处理器的物理地址到内存地址的映射关系。

4) 基于逆向分析的结果,本文将 RDXA 系统应用到现有的虚拟机监视器 Xen 系统上, 并且测试了该系统的安全性和性能。经过测试证明了 RDXA 具有不足 6%的运行性能损失和小于 1%的内存损失, 同时也能够很好地防御现实世界中的 Rowhammer 攻击。

本文的结构如下: 在第二章节介绍背景和相关的 Rowhammer 攻击与防御技术; 第三章节和第四章节分别介绍了 RDXA 系统的框架设计及其具体的组成部分的实现; 第五章节对 RDXA 的安全性与性能进行了测试与评估; 第六章节探讨了当今的 Rowhammer 防御机制的优缺点以及本方法的局限性; 第七章节总结了 RDXA 系统并对未来工作进行展望。

2 背景

2.1 DRAM 内存架构与 Disturbance Error

内存是由 DRAM(动态随机存储器)芯片组成的。图 1 所示的是典型的 DRAM 内存空间布局[14], 在图 1 的例子中, 一块 2GB 大小的 DDR3 Dual Inline Memory Module(DIMM)中有两个 rank, 而每个 rank 则分别有 8 个 chip, 每个 chip 内都有 8 个 bank, 这 8 个 bank 在内存读取时是并行读取的[15]。在每个 bank 中有许多的基本单元(cell), 每一个基本单元由一个电容和一个晶体管构成, 电容可储存 1 比特数据量, 电容的电势高低分别对应二进制数据 0 和 1。内存中的基本单元按矩阵形排列, 每一行和每一列都会有一个对应的行地址线路(row)和列地址线路(column)。

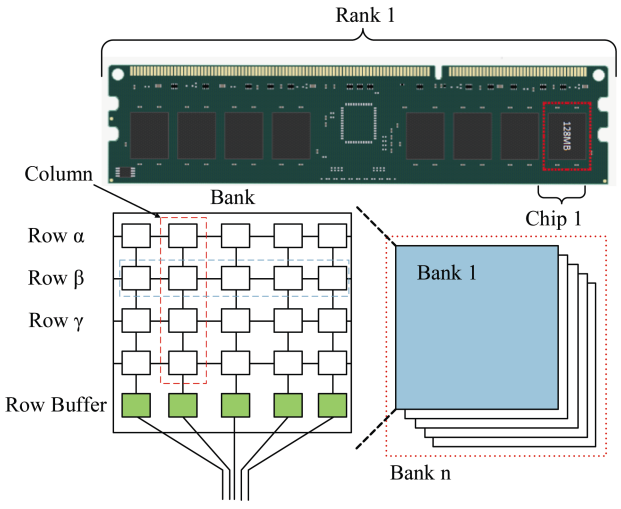


图 1 典型的 DRAM 内存空间布局

内存寻址时先通过行地址找到对应的行地址线路, 再通过列地址找到对应的基本单元。而每个 bank 内都有一行 row buffer 作为行缓冲来加速对于 row 的访问。

Disturbance Error[16]则是目前的 DRAM 内存, 特别是市场上主流的 DDR3 以及部分 DDR4 类型内存在设计时由于内存基本单元排列过于紧密而产生的一种硬件性缺陷。因为内存基本单元之间排列过于紧密, 当内存上某一个 row 的基本单元被频繁访问从而被不断的充电和放电时, 有一定概率泄露部分电量给其相邻 row 的基本单元的电容。当短时间(两次 DRAM 内存刷新时间间隔)内泄露的电量积累到足够时则会导致相邻 row 的某个基本单元内存储的数据发生 0~1 或者 1~0 的位翻转突变。特别是当某一个易发生 Disturbance Error 的 row(称为 victim row)的左右两个 row 都被频繁访问时, 能加大 victim row 上基本单元的突变概率。如图 1 所示, 当攻击者频繁访问 row β相邻的 row α与 row γ时, 作为 victim row 的 row β 就有很大的概率发生位突变。Disturbance Error 为攻击者提供了一种通过简单的内存访问就能修改内存上原本不具有读写权限的内存数据的方法。

2.2 Rowhammer 攻击技术

Rowhammer 攻击是利用软件方式来触发 Disturbance Error 这一硬件错误的攻击方式。目前的 Rowhammer 攻击通常是通过某种方式将关键数据(例如内核中的页表项等)放在易受攻击的 victim row 上, 然后通过短时间内频繁访问其相邻的 row 上的数据来触发电容的不断刷新, 最后实现 Rowhammer 攻击, 让 victim row 上的关键数据发生某个比特上的位翻转突变。例如攻击者通常利用 Rowhammer 攻击操作系统的页表的步骤如下: 先发现易发生 Rowhammer 的物理地址, 然后通过大量分配内存, 概率性地将该物理地址所在的 row 分配为页表项, 通过 Rowhammer 攻击翻转页表项所在页的权限位, 让攻击者拥有对应页表页的读写权限, 从而能够操纵内核的内存来获取更高级的权限。

尽管 Rowhammer 攻击是一种基于概率的攻击方式, 但是在各个架构和平台上已经发展出了多种基于 Rowhammer 的攻击方式。

如图 2 攻击方式一所示, 在普通环境下, 恶意进程直接攻击操作系统的内核部分, 从而获得更高级的权限。例如 Google 利用 Rowhammer 攻破了 Chrome 浏览器沙盒机制, 以及实现了在 X86 平台下 Linux 系统下普通进程的提权攻击[4]。RowhammerJS[5]实现了利用高级语言 JavaScript 远程的实现 Rowhammer 攻击。Drammer[6]则在 Android 环境下让一个普通进程获取 root 权限。

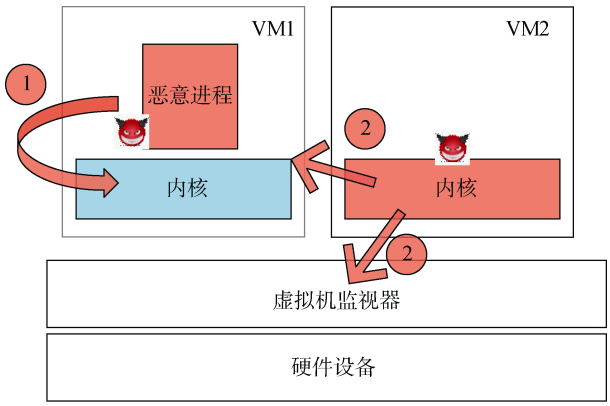


图 2 普通与虚拟化环境下的 Rowhammer 攻击

除了在普通的操作系统环境下的攻击, 在虚拟化环境下, Rowhammer 攻击也有着许多的攻击方式。如图 2 中攻击方式二所示, 其主要是虚拟化环境下的跨域攻击, 通过攻击虚拟机监视器和另外一个虚拟机, 从而操纵更多的数据。文献[7]提出了一种跨虚拟机的攻击方式, 该攻击可以在 XEN PV 的环境下, 让某个恶意虚拟机能够访问其他虚拟机与虚拟机监视器所在内存并修改。而 Flip Feng Shui[8]则是利用不同虚拟机之间内存去重的功能, 实现了跨虚拟机攻击。

2.3 相关工作

针对 Rowhammer 攻击的防御工作主要分为两个方向, 一个是基于硬件的改进措施, 通过重新设计 DRAM 内存架构来阻止 Rowhammer 攻击的产生; 另一个方向是通过纯软件的方式来防御 Rowhammer 攻击。硬件防御机制可以更有效的阻止 Rowhammer 攻击, 而软件防御机制无需修改硬件设备, 可以很好的部署在已有的商业系统中。

截至目前, DRAM 内存上 Disturbance Error 发生的根本机制尚未被研究明确[5], 硬件防御机制无法完全阻止 Disturbance Error 的出现。因此现在针对 Rowhammer 攻击的硬件防御机制都是通过修改 DRAM 内存的架构来增加攻击者触发 Disturbance Error 的难度。

例如在惠普和联想等计算机厂商都采用的防御措施[17]: 提升 DRAM 内存的刷新速率来减少内存刷新的时间间隔。但是这种方法会带来更大的处理器耗能以及会影响内存的吞吐率。同时已经有文献[11]证明存在 Rowhammer 攻击能够在 15ms 左右的内存刷新时间间隔内完成攻击, 所以仅仅两倍的加速刷新是不够的, 但是四倍加速对于处理器和内存都会

带来很大的性能影响。文献[18]提出了增加随机的 DRAM 内存行地址刷新机制来减少 Rowhammer 攻击成功的几率。Project Armor[19]则是提出额外的 row buffer 硬件来增加内存访问的缓存, 从而加强实现 Rowhammer 攻击的复杂性。ECC 内存[20]则是在内存中加入奇偶校验码来保证当只有一个 bit 发生位错误时能够自动纠错。这些对于 Rowhammer 攻击的防御工作都需要修改物理硬件, 针对当下大部分商业云环境下使用的内存都是型号较旧的 DDR3 内存的情况, 这些防御很难部署在已有的系统上。

基于纯软件的方式的 Rowhammer 防御机制只有以下几种:

1. Google 提出禁止用户态程序使用 CLFLUSH 指令来阻止用户态的进程发起 Rowhammer 攻击[4]。在此之前的 Rowhammer 攻击都依赖于 CLFLUSH 指令来排除 Cache 对于频繁访问内存的影响。但是 RowhammerJS 提出了使用 Cache Evict Set 的方式来绕过 CLFLUSH 同样实现攻击, 其通过巧妙安排 Cache 上的数据在不调用 CLFLUSH 的情况下将访问的数据从 Cache 中移除[5]。这样就可以绕过 CLFLUSH 指令完成 Rowhammer 攻击。

2. ANVIL 则是通过在 X86 平台下的 Precise Event Based Sampling 硬件特性和对于 L3 Cache Miss Rate 的性能计数器来进行取样并分析[11]。将 L3 缓存命中率低, 同时内存行访问的局部性高的 Rowhammer 攻击与一般的程序访存行为区分开来。但是 ANVIL 具有较大的误判率而且它过于依赖 X86 平台上的硬件特性。在 Drammer 中提出了通过内存直接访问技术可以直接绕过缓存访问内存[6]。这种情况下, 将无法利用 ANVIL 中的判断 L3 缓存命中率的方式来鉴别 Rowhammer 攻击。

3. G-CATT[13]与上述的 Rowhammer 防御机制采取了不同的防御思路, 它并不阻止 Rowhammer 攻击的产生, 而是通过在更底层提供内存隔离的方法来防御 Rowhammer 攻击。G-CATT 将普通操作系统下的内存分为用户态与内核态两个安全实体。通过修改操作系统的内存分配器使得用户态内存都被分配到每个 bank 的较低的 row 处, 而内核态的内存则都被分配到每个 bank 的较高的 row 的位置, 每个 bank 中间的一个 row 将作为不被使用的隔离 row。通过内存分配上实现 row 隔离保证了内核态的内存不受到来自用户态进程的恶意攻击。但是 G-CATT 只提供了用户态与内核态这两个安全实体间的内存隔离。该防御机制只能够阻止普通操作系统环境下恶意用户态进程攻击操作系统内核这单一类型的 Rowhammer 攻击。

由于 Rowhammer 攻击产生于物理硬件的缺陷, 所以截止目前针对 Rowhammer 攻击的防御机制或者需要修改物理硬件、或者需要依赖于某些处理器上的硬件特性、或者只能针对某些特殊场景下的攻击行为, 均无法很好的部署在已有的商业系统中。

而且针对图 2 中攻击方式二所示的针对虚拟化环境下的 Rowhammer 攻击, 目前并没有对应的防御机制。上述的一些防御机制[4,11]虽然可以部署在虚拟化环境下, 但是它们都已经被文献证明可以被通过某种方式绕过。

本文针对虚拟化环境下的 Rowhammer 攻击, 实现了 RDXA 这一纯软件的防御机制, 能够在虚拟机监视器的内存分配阶段保证不同的安全实体间所对应的物理内存不会被物理存放在同一个内存芯片上同一个 bank 的相邻 row 上面, 从而阻止了跨不同安全实体的 Rowhammer 攻击。RDXA 系统在实现 Rowhammer 防御时不依赖于任何的硬件特性以及硬件修改, 其既能保证虚拟机监视器的高效性, 又能保证对上层客户虚拟机的透明性。

2.4 威胁模型

本文假设运行在云平台的虚拟化环境下的客户虚拟机可能是恶意的, 或者容易受到恶意攻击从而被控制。恶意虚拟机为了扩大其 Rowhammer 攻击的影响力, 会通过攻击虚拟机监视器或者攻击其他虚拟机。上述攻击方式被称作虚拟化环境下的跨域 Rowhammer 攻击, 即从虚拟机到虚拟机, 虚拟机到虚拟机监视器的 Rowhammer 攻击。针对上述攻击方式, 本文不试图通过阻止 Rowhammer 攻击的产生来进行防御, 而是利用更加底层的内存隔离机制来阻止跨域 Rowhammer 攻击的蔓延。

本文假设虚拟化环境下虚拟机监视器是可信的, 攻击者通过其他方式直接攻击虚拟机监视器的情况不在本文讨论范围中。

3 RDXA 框架设计

3.1 概述

RDXA 系统是针对虚拟化环境下 Rowhammer 攻击的一套防御机制。其主要通过修改虚拟机监视器的内存分配器, 在内存分配时保证不同安全实体间所对应的物理内存不会被物理存放在同一个内存芯片上同一个 bank 的相邻 row 上面。RDXA 系统运行在虚拟机监视器所在的虚拟化层, 包含两个部分: 1. 内存布局旁路逆向分析工具; 2.Rowhammer 感知的内存分配器。

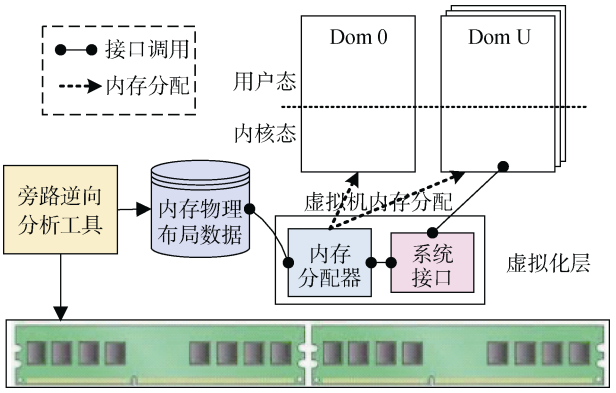


图 3　RDXA 系统架构图

如图 3, 整个 RDXA 的防御分为 2 个阶段:

1.线下的内存物理布局数据逆向阶段。在该阶段, 我们利用旁路逆向分析工具通过基于访问时间的旁路(side channel)分析方法, 逆向出不同物理地址和实际内存在芯片上布局的关系, 并将相关信息存储在数据库中;

2.运行时的虚拟机监视器的物理内存分配阶段。在该阶段, 虚拟机监视器中的 Rowhammer 感知的内存分配器会根据数据库中的内存物理布局数据, 将物理内存分为 DRAM 内存上相互不相邻的若干块区域(zone), 然后通过内存分配算法将这些区域分配给不同的安全实体(例如虚拟机监视器的内核部分与不同的虚拟机)。

3.2　内存物理布局数据逆向阶段

在实现 Rowhammer 感知的物理内存分配机制之前, 我们需要了解物理地址是如何被分配到内存中的, 比如物理地址中的哪些位对应实际内存地址所在的 channel、bank、row 等信息。然而这些物理地址到内存地址的映射关系并没有被 Intel 等芯片厂商透露[21]。因此本文基于访问时间的旁路(side channel)分析方法[22], 逆向出不同物理地址和实际内存在芯片上布局的映射关系。

如图 1 所示, 在 DRAM 内存芯片中的每个 bank 都有一个行缓冲器(row buffer)来缓存最近使用的行。因此, 如果程序交替访问同一个 bank 中两个不同的行, 由于行缓冲器的冲突, 我们会观察到一个高延迟。这种访问延迟的增加形成了一个时延通道(timing channel)[22]。我们可以通过这个通道来确定两个物理地址是否被分配到了同一个 bank 的不同 row。通过交替访问两个只有特定 bit 不同的物理地址, 利用时延通道观察它们是否在同一个 bank 的不同 row 中。并结合内存控制器的一些特点, 就能够推断出物理地址中这些特定 bit 的功能, 例如是否决定了该物理地址所在的 bank, row, column 的信息。

3.3　虚拟机监视器的物理内存分配阶段

利用上一阶段所获取的物理地址和内存布局的映射关系, 可以在本阶段将虚拟机监视器的物理内存在内存地址层面进行隔离。

RDXA 系统修改了虚拟机监视器底层的内存分配器, 在内存分配时能够主动的将不同安全实体的内存分配到 RDAM 内存上物理隔离的不同区域。

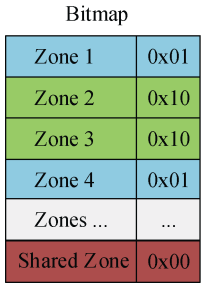


图 4　使用 Bitmap 维护所有的 zones 的拥有者

首先我们将所有的物理地址翻译成内存地址后, 根据其 row 的大小将整个 DRAM 内存分为若干个不同的 zone。每个 zone 的第一个 row 将会被保留为不可使用, 通过这种方式保证任意两个相邻的 zone 之间都隔离至少一个 row 的距离。

然后如图 4 所示, RDXA 使用一个全局的 bitmap 来维护这些 zones 所对应的安全实体。每当一个安全实体(可以是虚拟机监视器的内核部分或者是单独的虚拟机)需要内存时, 利用该安全实体的唯一标志(例如虚拟机的 domain ID 等)从 bitmap 中找到对应的 zone, 从这个 zone 内的空闲链表中找到内存块分配给该实体。而当某个安全实体是第一次分配内存或者之前的 zone 的内存空间不够用时, 则会在 bitmap 中找到一个新的 zone 分配给该安全实体。这样每个安全实体都有自己对应的若干 zone 空间。每个安全实体间都隔离至少一个 row 的距离。在图 4 中 zone 1 到 zone 4 的内存就被分配给了 ID 为 0x01 与 0x10 的两个安全实体, 这两个安全实体分别代表虚拟机监视器与 Dom 0 虚拟机。

而对于不同安全实体间共享内存, 本文则认为这部分共享内存只能是不被保护的普通内存。所以会通过 mmap 将这部分内存重新分配到一个单独的、所有安全实体共享的 shared zone 上。

4　RDXA 系统实现

4.1　概述

本文在以 Xen [23]作为虚拟机监视器的基础上, 开发了 RDXA 系统。其修改了 Xen 的底层内存分配

器, 保证了在内存分配时不同安全实体所在的内存区域始终保证至少一个 row 的距离。XEN 底层内存分配器使用 buddy system[24]作为分配策略。该内存分配策略作为一种高效, 简单的内存分配方式, 被通用于 Xen, Linux 等多种操作系统中。

在 RDXA 系统的实现过程中, 我们主要实现了一个旁路逆向分析工具(\$4.2)以及 Rowhammer 感知的内存分配器(\$4.4), 同时为了让 RDXA 可以适应于不同的安全策略, 我们设置了不同的安全实体为内存隔离的粒度, 而安全实体的划分也是 RDXA 实现时一个重要参数(\$4.3)。下面的小节我们将分别介绍相关内容的具体实现。

4.2 旁路逆向分析工具

为了逆向出物理地址到内存地址的映射关系, 我们实现了一个用户态的, 基于时延通道的旁路分析工具。该工具主要利用的是当程序交替访问同一个 bank 的不同 row 上的数据时, 由于 row buffer 的存在, 会导致一个高延迟的特性。

算法 1 测试访问地址的时间延迟的伪代码

```
1  int get_access_time(int* p1, int* p2, int iteration)
2  {
3      int t1 = get_current_time();
4      access_memory(p1, p2, iteration);
5      int t2 = get_current_time();
6      return t2 – t1;
7  }
8
9 void access_memory(int* p1, int* p2, int iteration)
10 {
11     while (iteration--> 0){
12         *p1;
13         *p2;
14         clflush(p1);
15         clflush(p2);
16     }
17 }
```

我们首先在内存中利用 mmap 分配一大块物理地址连续的内存空间。然后利用 Linux 的/proc/self/pagemap 的接口找到两个物理地址 P1 与 P2, P1 与 P2 只有某个特定第 X 位的 bit 不同。然后通过算法 1 所示的伪代码交替访问这两个内存地址。如表 1 所示(其中 S 表示 Same, D 表示 Different), 当 P1 与 P2 当访问时间是一个高延迟时, 可以判定 P1 和 P2 处于同一个 bank 的不同 row 上, 从而判断出 X 位是否是只决定 row 的 row bit。

在 Intel 处理器中为了提高内存访问的吞吐率, 会将某些决定 bank、rank 或者 channel 的 bit 和一些高位的 row bit 进行 XOR 操作, 并用这个 XOR 的结果决定 bank、rank 或 channel 的信息。

为了寻找这种 XOR 的关系, 我们需要利用上述方法构造四个物理地址 P1-P4, 其中 P1 与 P2 的第 X 位不同, P1 与 P3 的第 Y 位不同。而 P1 与 P4, P2 与 P3 则对应的 X, Y 两个 bit 均不同。通过图 4 所示的伪代码获取其中某些组合是否是属于高延迟, 则可以判断出是否 X⊕Y 决定 bank、rank 或者 channel。如表 2 所示(S 表示 Same, D 表示 Different, U 表示 Unknown), 当存在(P1, P4), (P2, P3)组合是高延迟但是(P1, P2), (P1, P3)组合是低延迟时, 则可以判断 X⊕Y 组合决定 bank、rank 或者 channel 的信息, 而高位的 X 位 bit 同时也决定了该地址所在的 row。

表 1 P1 与 P2 只在第 X 位不同时的访问延迟

测试	X	延迟	bank	row
P1&P2	0 1	高	S	D

表 2 在第 X 位, 第 Y 位不同时的访问延迟

测试	X	Y	X⊕Y	延迟	bank	row
P1&P2	0 1	0 0	0 1	低	D	U
P1&P3	0 0	0 1	0 1	低	D	U
P1&P4	0 1	0 1	0 0	高	S	D
P2&P3	1 0	0 1	1 1	高	S	D

最后我们结合处理器上不同芯片的各自特性(例如在 Intel Xeon 处理器上总是高位是 row bits, 低位是 column bits)便可以逆向出内存物理地址中的哪些位的 bit 决定了内存的 channel、bank、row 和 column 等信息。以本文实验测试的机器为例, 所使用的处理器上的内存控制器是 Xeon E3-1200 v2/IVY Bridge DRAM Controller, 系统使用的 DRAM 内存大小为 8GB。所有的内存总共有两个 channels, 每个 channel 有 1 个 DIMMs; 每个 DIMMs 有两个 ranks, 每个 rank 有 8 个 banks, 每个 bank 有 2^{15} 个 rows, 每个 row 的大小为 2^{13} Byte。而经过旁路逆向分析工具获取的最后物理地址到内存地址的映射关系如表 3 所示。可以看到 b_{13} 与 b_{16} 分别是 channel 和 rank bits, 但它们都各自 XOR 许多的 row bits 来增加物理地址在访问 DRAM 时的并行性。而 b_{14}、b_{15} 与 b_{16} 则是分别与其他的 row bits 异或后再共同决定该物理地址所在的 bank 信息。

表 3 物理地址和内存布局的映射关系实例

Channel bits	Rank bits	Bank bits	Row bits	Column bits
$b_7 \oplus b_8 \oplus b_9 \oplus b_{12} \oplus \mathbf{b_{13}} \oplus b_{18} \oplus b_{19}$	$\mathbf{b_{16}} \oplus b_{20}$	$\mathbf{b_{14}} \oplus b_{18}, \mathbf{b_{15}} \oplus b_{19}, \mathbf{b_{17}} \oplus b_{21},$	b_{18}~b_{32}	b_0~b_{12}

4.3 安全实体的划分

在本文中对于Rowhammer攻击的防御的粒度是在“安全实体”这一级别的, 而安全实体的划分则是非常重要的。对于不同的安全策略, 安全实体的划分可以有多种方式。

较为粗粒度的安全划分可以将整个内存分为两个安全实体: 虚拟机监视器与客户虚拟机, 这种划分可以很好地防御已有的从虚拟机到虚拟机监视器的攻击, 同时也产生较低的性能损失。但是缺点是无法防御跨越虚拟机的 Rowhammer 攻击。

更细粒度的划分也可以将不同的虚拟机分为不同的安全实体。这样的划分则让 Rowhammer 攻击只能发生在虚拟机内部, 恶意的虚拟机无法利用 Rowhammer 攻击扩大影响范围, 从而阻止跨虚拟机的 Rowhammer 攻击。

更进一步的可以将一个虚拟机内部也划分出多个安全实体, 结合虚拟机内部的安全策略可以在虚拟机内部实现进程级别的隔离粒度。但是这样会使得该方案无法实现对虚拟机的透明性。

更精确的安全实体的划分方式可以带来更加细粒度的安全隔离, 但是也会相应的带来较大的性能损失。对于安全性与性能的取舍则取决于虚拟机监视器采取的安全策略。

4.4 Rowhammer 感知的内存分配器

在 Xen 的内核中存在若干个内存分配器, 其中最底层的内存分配器是 heap allocator[25]。该内存分配器使用的是 buddy system 的分配策略, 每次分配的内存大小都是 2 的若干次幂个页。而 Xen 的内核以及不同的虚拟机则向该内存分配器不断地提出内存分配请求来获取内存。

在上一阶段获取物理地址到内存地址的映射关系后, 便可以获取每个物理地址对应的在 DRAM 内存的位置。由表 1 可知在物理地址中总是高位 bits 表示 row 的地址, 据此我们可以根据该物理内存中表示 row 的 bits 位数将整个机器的物理内存分为若干个 zones, 每个 zone 的大小是 2^K 个 rows(K 小于 row bits 的数目), 这里的 K 是一个可调整的参数。当 K 较大时, 每个 zone 内的内存较大, 可以减少被保留不分配的那个 row 在 zone 中所占比例从而提高内存分配效率, 但是对应的 zone 的数目较少, 则会限制 Xen 上可以运行的虚拟机的数目。反之亦然。

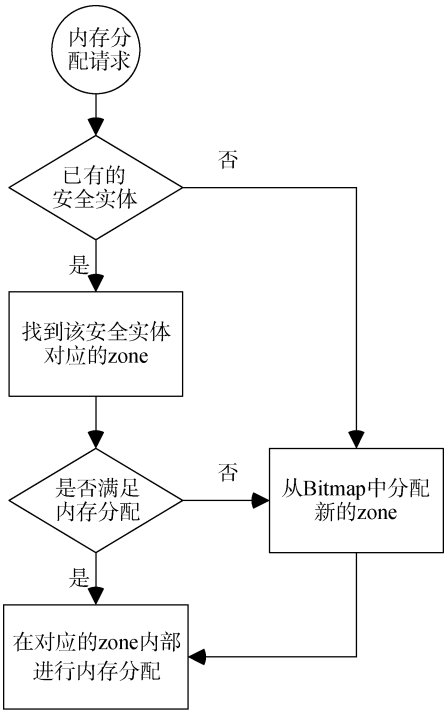


图 5 RDXA 内存分配流程图

RDXA 系统建立了一个全局的 bitmap 来表示这些 zones 的使用情况及其拥有者。在每次 Xen 的内核或者虚拟机这些安全实体向内存分配器发送请求时。RDXA 根据图 5 所示的内存分配流程图来找到最后进行内存分配的 zone。

首先 RDXA 根据请求参数中的 domain ID 来判断请求来自哪个安全实体。对于第一次出现的 domain ID, 将从 bitmap 中分配一个 zone 给该安全实体, 之后该安全实体分配的内存将总是来自这个 zone。直到该 zone 内的内存已经不够分配时, 则会在 bitmap 上继续分配一个 zone 给该安全实体。在每个 zone 内部的内存分配则继续采用 heap allocator 使用的伙伴系统原则。在 zone 内部将内存按照大小分成若干个空闲链表, 每次内存分配时都从对应需求内存大小的空闲链表中找到一个空闲块然后分配。

同理, 对于每次内存释放的请求, 则会根据该地址对应在内存中的 row 信息将其释放回对应 zone 里的空闲链表中。当某个虚拟机被销毁时, 对应的 zone 内的内存会被全部释放, 而这个 zone 对应的拥有者也会清空, 等待重新分配给其他的虚拟机。

通过将物理地址划分成内存上 row 隔离的各个 zones, 然后再通过内存分配器保证不同的安全实体都只使用自己所拥有的 zones 的内存, RDXA 可以实现让所有的安全实体都有着内存上的 row 隔离。

最后, 为了阻止攻击者利用虚拟机之间的内存共享机制进行跨虚拟机的 Rowhammer 攻击, RDXA 系统在分配共享内存时, 会将该块内通过 mmap 操作重新存放到一个单独的 shared zone 中, 而这块 zone 是作为所有安全实体所共享的区域。我们默认对于该块区域的内容不做 Rowhammer 防护, 也禁止虚拟机将自己的内核部分进行共享。

5 测试与评估

5.1 测试环境

本文实验环境中 CPU 为 Intel Xeon E3-1200, 内存为 8GB, 虚拟机监视器为 Xen 4.8.0 版本, 使用的 Domain 0 虚拟机的操作系统为 Ubuntu 16.04, Linux 内核版本为 4.4.24。使用的客户虚拟机的操作系统为 Debian 8, Linux 内核版本为 4.10.2。

由于本文实验所使用的机器硬件限制, 本机总共内存大小为 8GB, 在该硬件条件下运行较多虚拟机时虚拟机性能下降严重。因此在本文的性能测试中, 均采用启动一个虚拟机 Dom0 与一个虚拟机 DomU 的情景进行测试。性能测试时每个虚拟机设置的所需最大内存为 4GB。

5.2 性能参数

在 RDXA 中存在若干参数可能会影响 RDXA 防御机制的安全级别。第一个参数是安全实体的划分, 越细粒度的划分会提供更高的隔离级别, 而粗粒度的划分则会减少系统的性能损失。本文为了防御从虚拟机到虚拟机监视器和跨虚拟机的 Rowhammer 攻击, 采用的是以虚拟机为粒度的安全实体划分方式。第二个参数则是每个 zone 所占的内存大小 2^K 个 row 中的参数 K。本文首先测试了不同的 K 值对于 RDXA 系统的性能影响, 由于在测试中仅启动两个虚拟机, 我们发现 K 值的变化对于虚拟机创建与运行时的性能影响较不明显。但是实验表明, 若设置的 K 值太小, 在虚拟机创建的过程中某些较大的内存分配请求会一直分配失败, 从而导致虚拟机创建失败。为了保证虚拟机分配内存时不同大小的请求总能成功, 在本机的测试环境中 K 值的合理范围是 6 至 14。

在本文的后续测试中我们设置 K 值为 9, 将整个内存分为 2^6 个 zones, 每个 zone 大小为 2^9 个 row, 即 2^{15} 个页。这样本系统理论上可以支持最大为 63 个虚拟机。

5.3 安全测试

目前在虚拟化环境下实现的 Rowhammer 攻击已经有文献[7]与 Flip Feng Shui[9]这两种攻击方式。但是由于这两种攻击方式并不开源, 本文无法直接测试 RDXA 在该种攻击下能否阻止 Rowhammer 攻击的发生。为了证明 RDXA 系统的安全性, 我们基于 Google 提供的开源提权攻击[5]实现了一个虚拟化环境下的攻击。该攻击和已有的 Rowhammer 攻击类似, 利用 Rowhammer 从系统低权限的部分攻击系统页表, 从而获取更高级的权限。首先我们手动的找到了一个测试内存上的“Disturbance Error”错误, 然后利用内存溅射攻击, 将大量的内存分配给攻击虚拟机, 概率性地将虚拟机监视器的页表内存分配到错误内存所在的 row 上。最后通过 Rowhammer 攻击让虚拟机监视器的页表内存发生突变, 从而让一个虚拟机获取对于虚拟机监视器内核任意部分内存的读写权限。

为了测试 RDXA 系统能否防御现实中的基于 Rowhammer 的从虚拟机到虚拟机监视器的页表攻击, 保证其划分的安全实体间不会有相邻的 row 接触, 本文对比了该攻击在被 RDXA 保护下的 Xen 与普通的 Xen 虚拟机监视器中成功攻击所需次数。由于我们主动暴露的“Disturbance Error”所在的 row 有一定的概率不在虚拟机监视器的页表上, 所以在普通 Xen 环境下的 Rowhammer 攻击平均需要 34 次才能成功。而对应的在 RDXA 保护下的 Rowhammer 攻击在尝试了 2000 余次后仍未成功, 我们有理由相信这是由于 RDXA 的保护, 所有的虚拟机监视器的页表内存距离恶意虚拟机所在的内存至少有一个 row 的距离, 所以该攻击无法利用恶意虚拟机的内存将 Rowhammer 攻击扩散到虚拟机监视器。

5.4 性能评估

为了测试在虚拟机运行时 RDXA 带来的性能影响, 本文测试了在一个新建的虚拟机内部运行 SPEC CPU 2006[26]和 PARSEC 3.0[27]等基准测试程序产生的性能损失。其中 SPEC CPU 2006 是针对单核处理器的性能基准测试集, 而 PARSEC 3.0 是多线程应用程序组成的测试程序集。

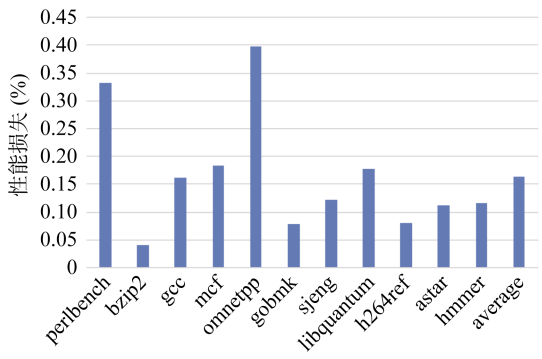


图 6 性能测试: SPEC CPU2006

本文首先对 SPEC CPU 2006 这个基准程序测试集中的 ref 数据集, 即真实数据组成的数据集进行了测试。如图 6 所示, RDXA 系统相对于 Xen 造成的性能损失最大不超过 0.4%, 平均的性能损失只有 0.16%, 在误差范围内可以几乎忽略其性能损失。

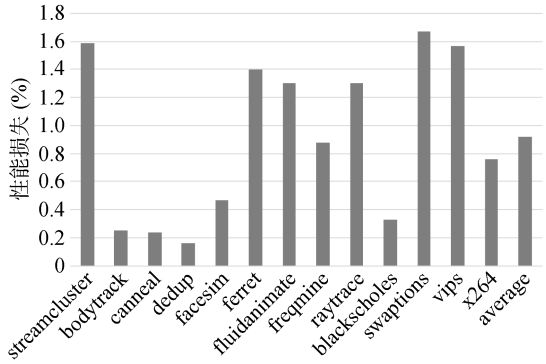


图 7　性能测试: PARSEC 3.0

如图 7 所示, 当对于基准程序测试集 PARSEC 3.0 中的 simlarge 大小的数据集进行测试时, RDXA 相对于 Xen 造成的性能损失最大不超过 1.8%, 平均的性能损失只有 0.92%。

由于 RDXA 系统所在的 heap allocator 是 Xen 系统中最底层的内存分配器, 而在它上层还有一个分配粒度更小的 TLFS 内存分配器[28], 所以在上层虚拟机运行时, 大部分的内存分配请求只会发送到 TLFS 内存分配器, 很少的内存分配请求会到达底层的 heap allocator。因此如图 6 和图 7 所示, 即使在虚拟机内运行大量分配内存的基准程序测试集, 对于虚拟机监视器底层的内存分配器的内存分配请求也并不多, 因此 RDXA 系统造成的性能损失非常小。

所以本文还测试了实际使用中, 常见的大量内存被分配和释放的场景——虚拟机的启动过程。考虑到虚拟机 Dom 0 和其他的客户虚拟机 Dom U 的不同, 本文分别测试了在启动这两种虚拟机时 Xen 和 RDXA 系统所使用的时间。如表 4 所示, 可以看到 RDXA 在启动虚拟机这种大量内存分配和使用的场景下的性能损失仍然不超过 6%。

表 4　启动虚拟机场景下的性能测试

场景\系统(时间)	Xen	RDXA	性能损失
启动 Dom0	12.08s	12.76s	5.63%
启动 DomU	51.24s	53.42s	4.25%

同时本文也测试了 RDXA 系统内 allocation 函数在单独的一次内存分配时造成的性能损失。如图 8 所示, 在分配大小为一个页的内存时, 内存分配所需时间最小, 因为最小粒度的内存总是存在于空闲链表上。因此 RDXA 的性能损失主要来自于: 找到该内存分配请求所来源的安全实体, 利用该安全实体 ID 找到其拥有的 zone。

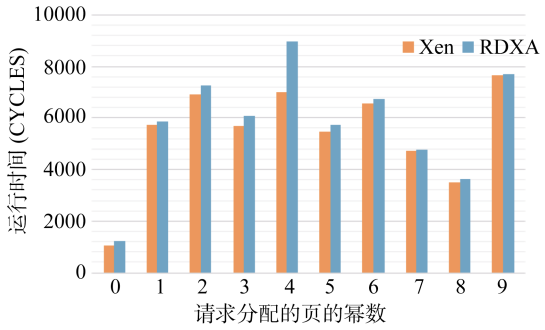


图 8　一次内存分配请求的运行时间

对比 RDXA 与 Xen 的内存分配时间, 在请求的页的大小为 2^4 个页时 RDXA 的性能损失最大, 约为 27%。这是由于在 RDXA 系统中每次内存分配都只在其拥有的 zones 中寻找内存碎片, 而这显然会增加对应大小内存碎片不足的概率。当该粒度大小的内存碎片不足时需要对更大的内存碎片切割后才能成功分配内存。这个过程则令 RDXA 的性能损失增大。

同时我们可以发现图 8 中 XEN 与 RDXA 在运行时最大粒度的内存请求为 2^9 个页, 这个内存请求小于我们划分的一个 zone 的大小(2^9 个 row, 即 2^{15} 个页)。所以我们划分的 zone 的大小可以满足系统运行时所有的内存分配请求。

由于 RDXA 系统中分配的全局 Bitmap 所占的内存空间远小于隔离 row 所占空间。因此 RDXA 系统中内存开销要来自于每个 zone 中的隔离 row。每个 zone 的大小为 2^K 个 row, 其中一个 row 作为隔离 row 无法被使用, 所以理论上内存空间损失率为 $1/2^K$(K 的范围是 6 至 14)。我们估算了在本测试所设置参数下 RDXA 系统所造成的内存空间损失。其中全局的 bitmap 所占用的内存大约 512Byte; 而每个 zone 中为了实现隔离牺牲了一个 row 的内存不能够被分配。对于每个大小为 2^9 个 row 空间的 zone 总会造成一个 row 的空间的损失。所以最终所以损失的内存大小为 2^{11} 个页, 相对于总共的内存大小, 产生的内存损失率约为 0.2%。

6　思考与讨论

针对 Rowhammer 攻击的防御机制已经有许多, 如章节$2.3 所示, 按照防御方法的主要实现方式划分, 可以将这些防御机制分为硬件防御机制与软件

防御机制。由于截止现在 DRAM 内存中 Disturbance Error 的发生原因尚不明确, 而且在最新的 DDR4 类型的内存设计中还无法完全杜绝 Disturbance Error 的发生[29]。所以上述防御机制都是试图在 Rowhammer 攻击发生过程的某个步骤进行阻止。按照 Rowhammer 攻击的发生过程来划分, 可以将已有的针对 Rowhammer 攻击防御机制分为以下三种思路:

1. 阻止攻击者触发 Disturbance Error 的发生;
2. 在 Disturbance Error 发生后更正内存错误;
3. 阻止攻击者利用 Rowhammer 攻击更高权限的安全实体内存。

本小节将对这三种思路的防御机制进行系统性的总结。对比各种思路下的防御机制的优缺点。

当下大多数的防御机制都是采用的第一种思路来阻止 Rowhammer 攻击。例如章节$2.3 中所提出的加速内存刷新速率是为了减少触发 Disturbance Error 的时间周期; 文献[18]提出的加入内存随机行地址刷新机制可以减少内存错误的发生概率; Project Armor 提出在内存中加入额外的 row buffer 则是通过 row buffer 的缓存效果增加攻击者触发 Rowhammer 现象的复杂度。禁止 CLFLUSH 指令则是试图令用户态进程无法直接访问内存, 从而无法实现 Rowhammer 攻击; ANVIL 则是试图在攻击者触发 Disturbance Error 的过程中通过处理器上的性能计数器获取攻击者的内存访存行为, 然后通过模式匹配识别出恶意的 Rowhammer 攻击, 通过刷新对应内存的行地址来阻止 Rowhammer 攻击的发生。

上述的这些防御机制都试图阻止攻击者触发 Disturbance Error 错误, 在 Rowhammer 攻击发生的源头就将其阻止。由于硬件设计的缺陷导致现在的内存中无法杜绝 Disturbance Error, 这些在源头处阻止错误发生的防御机制确实是最有效的 Rowhammer 攻击的防御方法。但是如 ANVIL、加速内存刷新速率以及禁止用户 CLFLUSH 指令等软件防御机制都已经被证明可以被某种攻击方式绕过。而且由于 Disturbance Error 是由软件触发的硬件错误, 难以通过简单的方式直接阻止, 所以上述的攻击方式或者依赖于对硬件设备的修改、或者依赖于处理器上的物理特性。这些依赖使得上述防御机制难以适用于目前的商业云环境。

采取第二种防御思路, 在 Disturbance Error 发生后更正内存错误的防御机制都是 ECC 内存以及类似的内存加密等技术。这些技术虽然可以在内存发生一个 bit 的错误后通过自动纠错来阻止 Rowhammer 攻击的发生, 但是性能损失较大。由于奇偶校验的简单性导致其只能自动纠错一个 bit 的位错误, 对于 Rowhammer 攻击中可能出现的多位错误则是无法纠正。而且当攻击者意识到 ECC 技术的存在时, 可以通过多次触发 Disturbance Error 来对 ECC 内存进行 DDOS(服务拒绝访问)攻击。

第三种防御思路并不试图阻止 Rowhammer 攻击的产生, 而是在 Rowhammer 攻击的蔓延阶段, 即攻击者利用 Rowhammer 现象攻击具有更高级权限安全实体时进行隔离阻止。现有的 G-CATT 防御机制是采取了类似的思路进行安全防御, 在物理内存这一层面上进行安全实体的隔离从而阻止 Rowhammer 攻击的蔓延。但是 G-CATT 只针对普通操作系统环境下 Rowhammer 攻击。因为普通操作系统下用户态进程的创建与销毁过于频繁, 而且每个进程所需要的内存是在运行时动态分配, 这种频繁的内存分配与回收使得难以实现细粒度的内存隔离机制。所以 G-CATT 在普通操作系统层面只实现了用户态内存与内核内存这两个粗粒度的安全实体划分。G-CATT 针对的使用场景是恶意的用户态进程攻击作系统内核这一类型的 Rowhammer 攻击。

本文实现的 RDXA 系统则是对第三种防御机制在虚拟化环境的应用与优化。由于虚拟化场景下虚拟机的内存分配往往是预先配置且静态分配的, 所以 RDXA 系统将虚拟机监视器的内存分为了若干个相互物理隔离的 zone, 然后在虚拟机的启动与销毁的过程中动态的将不同的 zone 分配给虚拟机。不同的安全实体通过不同的 zone 之间物理内存层面的隔离保障了它们所在的内存区域彼此之间相距至少一个 row 的距离。相对于 G-CATT 只能将操作系统划分为两个粗粒度的安全实体, 而且只能防御单一的攻击场景。RDXA 系统实现了更加细粒度的安全隔离, 实现了在一个系统内多个(大于两个)安全实体之间的相互隔离, 可以防御虚拟机到虚拟机以及虚拟机到虚拟机监视器之间的 Rowhammer 攻击。

采用第三种思路的防御机制都具有共同的局限性: 只能够在安全实体的粒度上进行防御, 对于更细粒度的 Rowhammer 攻击则无法阻止。例如在 G-CATT 中只能防御用户态进程到内核的攻击。而 RDXA 系统也不例外, 本文在实现时以不同虚拟机的粒度来进行安全实体的划分, 这保证了在虚拟化环境下无法实现跨虚拟机以及虚拟机到虚拟机监视器的 Rowhammer 攻击, 但是无法保证发生在虚拟机内部的 Rowhammer 攻击。

不过从章节$2.2 中可知, 截止现在所有的攻击方式都是通过 Rowhammer 来扩大攻击影响范围, 让

攻击者获取更高的权限。所以已有的 Rowhammer 攻击总是会从低权限的安全实体攻击更高权限的安全实体。RDXA 系统能够有效的防御跨安全实体的攻击, 即能阻止目前已有的绝大多数的 Rowhammer 攻击。而且由于本文实现的 RDXA 系统是基于虚拟化层面的安全隔离, 与 G-CATT 在操作系统内部的隔离并不冲突, 所以 RDXA 与 G-CATT 工作可以正交耦合, 通过结合两种防御机制达到更细粒度的安全隔离。

对于其他的 Rowhammer 攻击防御机制, 当攻击者在知道该防御机制的系统设计时, 可以通过多种方式绕过该防御机制。例如针对禁止 CLFLUSH 指令的防御方式, 攻击者可以通过设置 Cache Evit Set 的方式达到同样将内存从 cache 中移除的目的。而针对于 ANVIL 这种模式识别的防御方式, 攻击者可以使用 DMA 的方式直接绕过缓存, 从而让 ANVIL 无法识别缓存 Miss rate, 从而防止 Rowhammer 攻击被识别和阻止。而针对于本文实现的 RDXA 系统, 当攻击者知道我们的系统设计时, 仍然无法通过任何方式绕过我们的内存隔离机制。攻击者只能在我们划分的安全实体内部进行 Rowhammer 攻击, 但是无法将 Rowhammer 攻击蔓延到其他安全实体, 更无法攻击具有更高权限的安全实体。

7 总结

本文设计并实现了一个在虚拟化环境下针对 Rowhammer 攻击的防御系统 RDXA。该系统在虚拟机监视器层通过实现一套 Rowhammer 感知的内存分配机制, 在对上层虚拟机透明的情况下保证不同安全实体间所对应的物理内存不会被物理存放在同一个芯片上同一个 bank 的相邻 row 上面。RDXA 系统能够在性能损失较小的情况下有效的阻止虚拟化环境下虚拟机到虚拟机监视器、跨虚拟机的 Rowhammer 攻击, 从而阻止 Rowhammer 攻击的蔓延。

为了保证对于上层虚拟机的透明性, 本文在划分安全实体的粒度时设置最小粒度为不同的虚拟机实体。这样的划分对于虚拟机内部的 Rowhammer 攻击无法防御。因此在未来的工作中, 我们需要结合虚拟机内部的内存分配机制。将安全实体划分为用户进程等更小的粒度, 通过调用虚拟化层的接口让虚拟机内部的内存分配也能同时保持安全实体粒度的隔离性, 从而实现真正的细粒度的限制 Rowhammer 攻击的蔓延。

参考文献

[1] Amazon Inc. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, 2011.

[2] Kandukuri B R, Rakshit A. Cloud security issues[C]//Services Computing, 2009. SCC'09. IEEE International Conference on. IEEE, 2009: 517-520.

[3] Elkaduwe D, Derrin P, Elphinstone K. Kernel design for isolation and assurance of physical memory[C]//Proceedings of the 1st workshop on Isolation and integration in embedded systems. ACM, 2008: 35-40.

[4] Kim Y, Daly R, Kim J, et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors[C]//ACM SIGARCH Computer Architecture News. IEEE Press, 2014, 42(3): 361-372.

[5] Seaborn M, Dullien T. Exploiting the DRAM rowhammer bug to gain kernel privileges[J]. Black Hat, 2015

[6] Gruss D, Maurice C, Mangard S. Rowhammer. js: A remote software-induced fault attack in javascript[M]//Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, 2016: 300-321.

[7] Van der Veen V, Fratantonio Y, Lindorfer M, et al. Drammer: Deterministic rowhammer attacks on mobile platforms[C]// Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016: 1675-1689.

[8] Xiao Y, Zhang X, Zhang Y, et al. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation[C]// Proceedings of the 25th USENIX Security Symposium. 2016.

[9] Razavi K, Gras B, Bosman E, et al. Flip feng shui: Hammering a needle in the software stack[C]//Proceedings of the 25th USENIX Security Symposium. 2016.

[10] JEDEC Solid State Technology Association. Low Power Double Data Rate 4 (LPDDR4), 2015.

[11] CISCO Inc. Mitigations Available for the DRAM Row Hammer Vulnerability. http://blogs.cisco.com/security/mitigtions-available-for-the-dram-row-hammervulnerability.

[12] Aweke Z B, Yitbarek S F, Qiao R, et al. ANVIL: Software-based protection against next-generation rowhammer attacks[J]. ACM SIGPLAN Notices, 2016, 51(4): 743-755.

[13] Brasser F, Davi L, Gens D, et al. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory[J]. 2017.

[14] D. R. A. M. (2017, June 10). In Wikipedia, The Free Encyclopedia. Retrieved 14:26, June 12, 2017, from https://en.wikipedia.org/w/index.php?title=D.R.A.M.&oldid=784953922

[15] JEDEC Solid State Technology Association. DDR3 SDRAM Specification, 2010

[16] Kim Y, Daly R, Kim J, et al. RowHammer: Reliability Analysis and Security Implications[J]. arXiv preprint arXiv:1603.00747, 2016.

[17] HP Inc. HP Moonshot Component Pack Version 2015.05.0. http://

h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/component-pack/index.aspx. Accessed: 2015-08-11.

[18] Dae-Hyun Kim, P.J. Nair, and M.K. Qureshi. Architectural support for mitigating row hammering in dram memories. Computer Architecture Letters, 14(1): 9-12, Jan 2015.

[19] Mohsen Ghasempour, Mikel Lujan and Jim Garside. Armor: A Run-Time Memory Hot-Row Detector. http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/index.html. Accessed: 2015-08-11.

[20] Qin F, Lu S, Zhou Y. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs[C]//High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on. IEEE, 2005: 291-302.

[21] Pessl P, Gruss D, Maurice C, et al. DRAMA: Exploiting DRAM addressing for cross-cpu attacks[C]//Proceedings of the 25th USENIX Security Symposium. 2016.

[22] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In Security and Privacy (SP), 2013 IEEE Symposium on, pages 191–205, May 2013.

[23] Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization[C]//ACM SIGOPS operating systems review. ACM, 2003, 37(5): 164-177.

[24] Li K, Cheng K H. A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system[J]. Journal of Parallel and Distributed Computing, 1991, 12(1): 79-83.

[25] Cherkasova L, Gupta D, Vahdat A. When virtual is harder than real: Resource allocation challenges in virtual machine based it environments[J]. Hewlett Packard Laboratories, Tech. Rep. HPL-2007-25, 2007: 15.

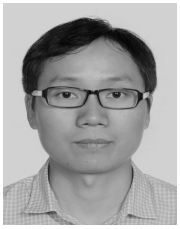
[26] Henning J L. SPEC CPU2006 benchmark descriptions[J]. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1-17.

[27] Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: Characterization and architectural implications[C]//Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008: 72-81.

[28] Masmano M, Ripoll I, Crespo A, et al. TLSF: A new dynamic memory allocator for real-time systems[C]//Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on. IEEE, 2004: 79-88.

[29] Lanteigne M. How rowhammer could be used to exploit weaknesses in computer hardware[J]. 2016.

石培涛 于 2015 年在上海交通大学大学软件工程专业获得工学学士学位。现在上海交通大学软件工程专业攻读工学硕士学位。研究领域为虚拟化安全。Email：beidao@sjtu. edu.cn



陈海波 于 2009 年在复旦大学计算机系统结构专业专业获得工学博士学位。现任上海交通大学教授、博士生导师。主要研究方向为系统软件、系统结构与系统安全。Email：haibochen @sjtu.edu.cn

刘宇涛 于 2017 年在上海交通大学大学软件工程专业获得工学博士学位。现任华为公司高级工程师。研究领域为操作系统、虚拟化安全和手机安全。