Rubix

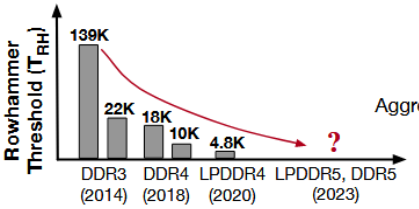
1.1 为什么要使用Rubix

目前学术界存在着多种RowHammer缓解措施, 这些方案大多需要**计数器**来追踪行激活数：

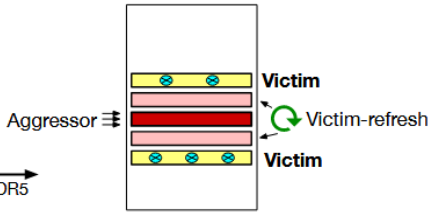
- **Blockhammer**通过**延迟访问**来控制**频繁访问行的访问速率**，从而确保在 64 毫秒内，任何行**不会出现超过** T_{RH}**次激活**。由于攻击者无法在单行上执行大量激活（Half-Double攻击的条件），因此它可以防范较为复杂的攻击。
- **AQUA** 在被 $T_{RH}/2$**次** 激活（由于跟踪器重置导致阈值减半）时将**攻击者行**迁移到内存中的**隔离区域**。AQUA 通过破坏攻击者和受害者之间的空间相关性，使得攻击者难以成功。
- **安全行交换 (SRS)** 在被$T_{RH}/3$**次**激活后，将**攻击者行与内存中另一个随机选择的行进行交换**。与 AQUA 一样，SRS 会破坏攻击者与受害者之间的空间相关性。

这些安全缓解措施都会在计数器达到**某个阈值** T_{RH} **之后** 实施一些行为来防止出现RowHammer, 但是这些操作会**造成一定的开销**, 在阈值设定较大时不容易触发缓解措施, 因此带来的开销尚在可接受范围内. 一旦**阈值降低(比如降低到128)**, 就很容易触发缓解措施, 造成严重的性能下降.

下图展示了阈值从 1 K 降低到 128 时 AQUA、SRS 和 Blockhammer的平均性能下降情况。在阈值为 128 时，AQUA 有 **15% 的性能损失，SRS有 60% ，Blockhammer 则有 600%**，



(a)



(b)

Average Slowdown

T_{RH}	AQUA	Secure Row-Swap	BlockHammer
1K	<1%	3.4%	10%
512	2.4%	10%	37%
256	6.4%	25%	140%
128	15%	60%	600%

(c)

1.2 Rubix怎么解决这个问题

在一个**刷新窗口内(64ms)**, 程序会频繁访问一小部分内存row, 造成这些部分的内存非常容易超过阈值而触发缓解措施.

Rubix将一个row拆分成 多个部分, 并将这几个部分随机散布在整个内存空间里, 这样就可以将原本一个row的 100次访问变成对10个row的各10次访问.

为了体现 Rubix的效果, 下述测试都把 **RowHammer缓解措施的计数器阈值设置为128.**

1.3 Hot Rows

Hot Rows指的是在一个**刷新窗口内(64ms),** 激活次数很多的row.文中描述了两种 Hot Rows, 分别是激活次数为 64 次以上（ACT-64+）和激活次数为 512 次以上（ACT-512+）的“热行”.

下面这张表测试了常见的几个workloads运行的指标, 作者跳过忽略其中 250 亿条指令，然后模拟 2.5 亿条指令.

- MPKI: 每千条指令 cache miss数
- Unique Rows: 访问过的Row的数量, 重复访问的只计算一次

Table 2. Workloads Characteristics: MPKI, Unique Rows Touched (within 64ms), and Hot-Rows (within 64ms).

Workload	MPKI (LLC)	Unique Rows Activated	Total number of "Hot-Rows"	
			ACT-64+	ACT-512+
blender	12.78	8.8K	347K	2.9K
lbm	20.87	29.4K	70.3K	0
gcc	6.12	10.4K	21.8K	384
cactuBSSN	2.57	5.2K	12.2K	0
mcf	5.81	4.9K	10.5K	425
roms	3.33	27.9K	6.6K	9
perlbench	0.71	11.4K	1.7K	0
xz	0.40	10.8K	496	0
nab	0.53	4.4K	189	0
namd	0.37	3.4K	105	0
imagick	0.13	1.1K	89	0
bwaves	0.21	1.7K	20	0
wrf	0.02	702	20	0
exchange2	0.01	122	14	0
deepsjeng	0.25	68.1K	12	0
povray	0.01	390	8	0
parest	0.10	2.4K	3	0
leela	0.02	879	0	0
Average	3.01	10.7K	9528	206

1.3.1 line-to-row映射对 Hot Rows的影响

现在给定一个系统， 该系统的**总内存为 4kb**, 包含**100w个row**, 每个row的大小为**4kb**且由**64个line**组成, 且一个 4kb的page放置在同一个row当中. 现在要使用三种不同访存模式(每次访问以***Line***为基本单位)访问 **100w** 次, 且将访存行为限制在**4mb**里(空间局部性).

1.3.1.1 顺序访问

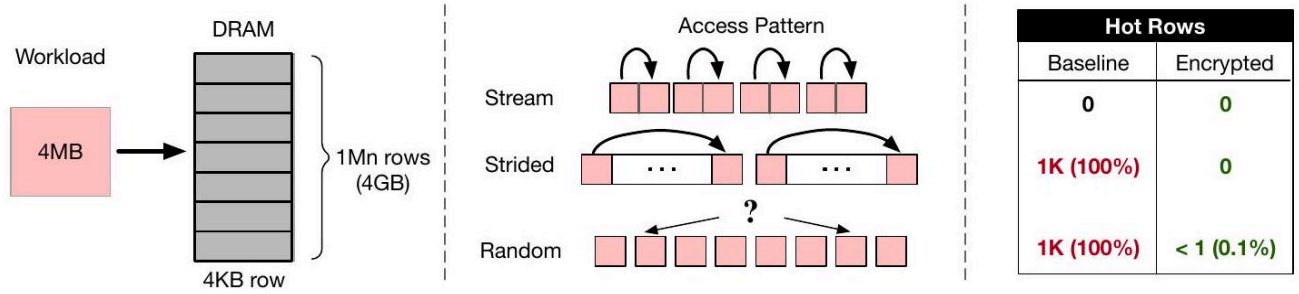
对于顺序访问，**第一次访问会导致 row 激活**，随后的 63 次访问都会命中rowbuffer。因此，一百万次内存访问总共只会导致 15.6 K 次激活，这些激活分布在 1 K 行上，每行约 16 次激活，**没有产生热行**.

1.3.1.2 步幅64间隔访存

stride-64 方式的步幅为 64 行，每次访问都会转到不同的页。由于每次内存访问都会导致激活，因此此方式会产生 1 百万次激活，均匀分布在 1 K 页面上，每行都会被激活 1 K 次。因此，**所有 1 K 行都是热行**。

1.3.1.3 随机访存

随机访问时几乎不会命中缓冲区，因此 1 百万次访问会导致 1 百万次激活，分布在 1 K 行上。每行的平均激活次数为 1000（标准差为 32），其中超过 99% 的行激活次数超过 900。因此，我们认为**所有 1K 行都是热行**。



将连续line放置在相同row中的传统映射会导致**间隔访问模式和随机访问模式**都出现热行。

再考虑采用加密行地址来访问内存系统的映射方式，这种方式可以把4MB 的 64K 行分散到内存中的 100 万行里。
通过二项分布估算 大概有61.5 K row中**不存在来自同一row的Line**，，1.9 K row有 2 个来自同一row的line，40 row有 3 行（没有行有 4 行或更多行）。

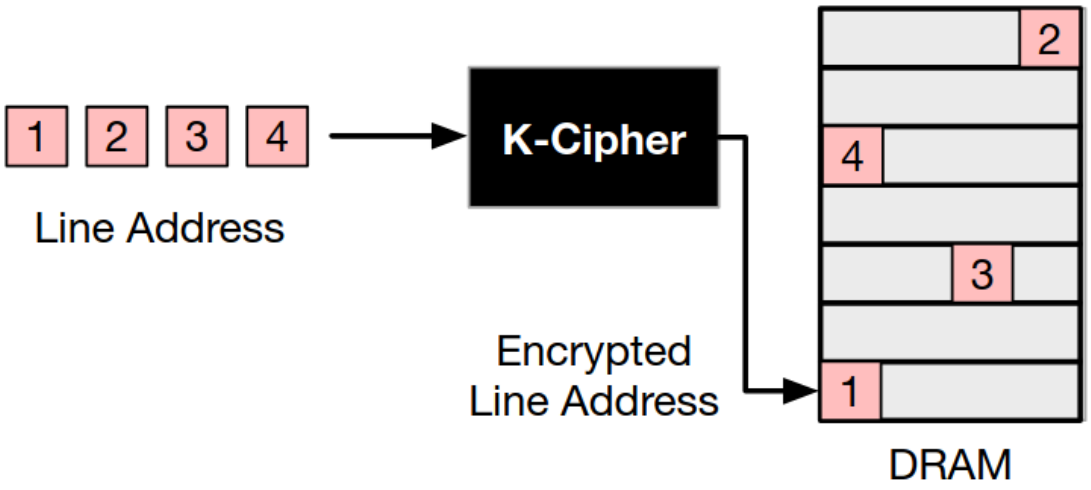
对于顺序和间隔访问，每行被访问 16 次。因此，我们有 61.5 K 行有 16 次激活，1.9 K 行有 32 次激活，40 行有 48 次激活。**因此，没有行是热行**。对于随机，我们估计预期的热行数量为 0.4，因此**不会有热行**。

1.4 Rubix-S

1.4.1 最简单的Rubix-S

Rubix-S 使用 K-Cipher 进行地址空间随机化，K-Cipher 是一种低延迟的可编程位宽密码。K-Cipher 位于 **内存控制器(CPU)** 中，时延为 3 个周期（采用 10nm 工艺技术 ）。在内存访问时，它加密用于访问内存的**line**地址。由于我们有 16GB 内存，因此使用 28 位密码。加密随机化了**line到row**的映射。

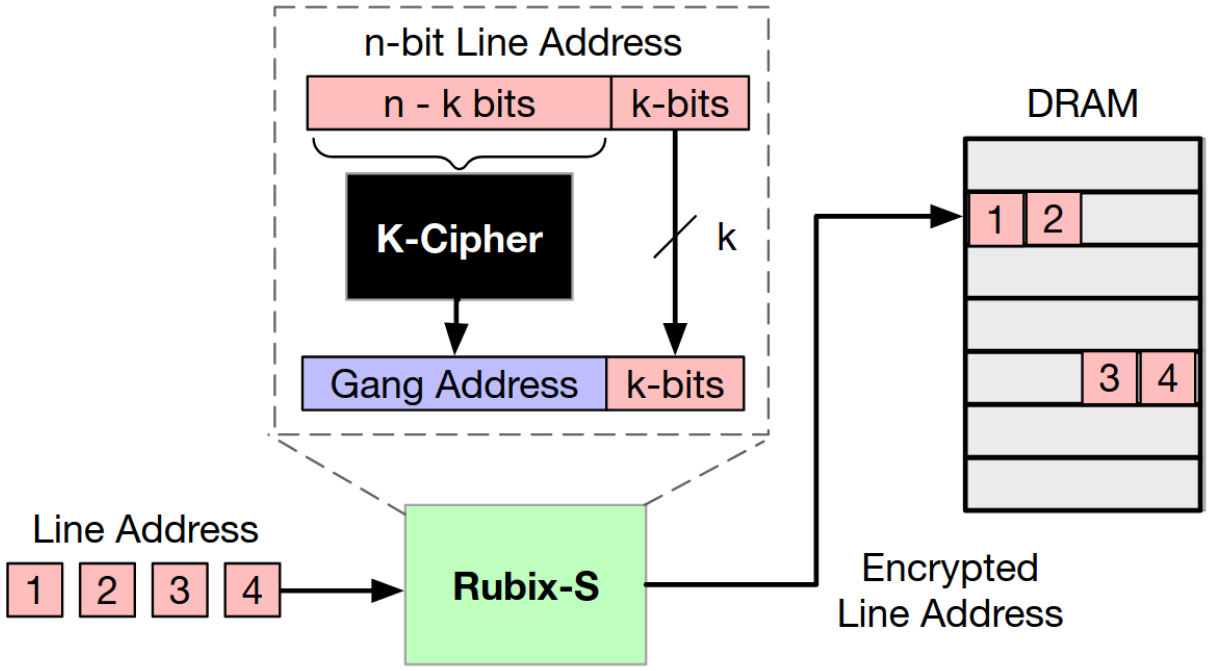
具体的**line到row**的映射取决于 K-Cipher 的 **96 位密钥。密钥在启动时设置为随机值**（基于 PRNG）。由于每个系统都有不同的密钥，因此每个系统的内存映射也会不同。



1.4.2 利用好 RowBuffer

虽然行地址加密几乎消除了热行，但它将行缓冲命中率降至近乎零。Rubix通过加密 2-4 个连续行的组来最大程度地减少热行，同时保留一些行缓冲命中。

在这种方案下 K-cipher**只加密高 n-k 位地址** ，下面以**Rubix-S(GSX)代表一个组内有X个line**的方案.



1.4.3 结果

1.4.3.1 热行数量

平均而言, Rubix(GS4)将热行的数量**下降了 220x, 为33**

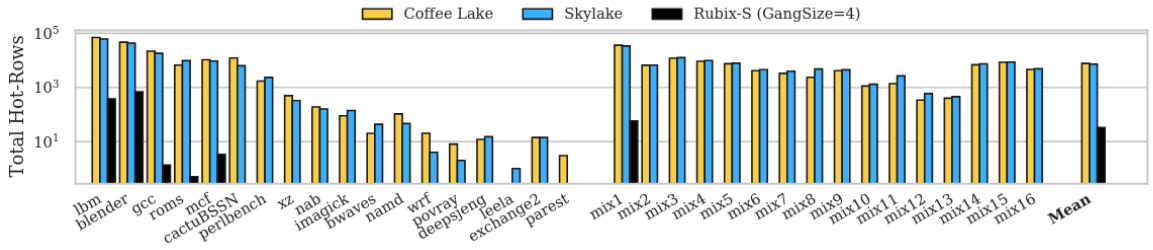


Figure 7. Number of hot-rows (activations of 64 or more) with Intel mappings and Rubix-S with Gang-Size of 4 (GS4). Mean implies arithmetic mean. While baselines have more than 7K hot rows on average, Rubix-S (GS4) reduces it by 220x to 33.

1.4.3.2 性能损失

以**无防护coffee lake**下的性能为baseline, 使用AQUA作为防护方案造成了**平均15%** 的性能下降，而通过Rubix-S(GS4)加强后将其降低到1%.

不同的测试用例，极端的情况

SRS和BlockHammer分别导致60%和600%的性能下降。通过Rubix-S加强后使得SRS和BlockHammer性能损失分别为 **3.1%（SRS使用 GS 4）**和 **2.8%（BlockHammer 使用 GS 1, 为什么要使用GS 1）**，而且在应用层性能上保持了最坏情况下也只有42%的性能损失（SRS使用 lbm）和仅11%的性能损失（对于BlockHammer）.

总体而言，Rubix-S使得在超低阈值（128）下实现安全缓解成为可能，仅带来2-3%的开销。***尽管我们没有改变访问调度和DRAM页面策略，但微调它们可能会进一步减少开销。***

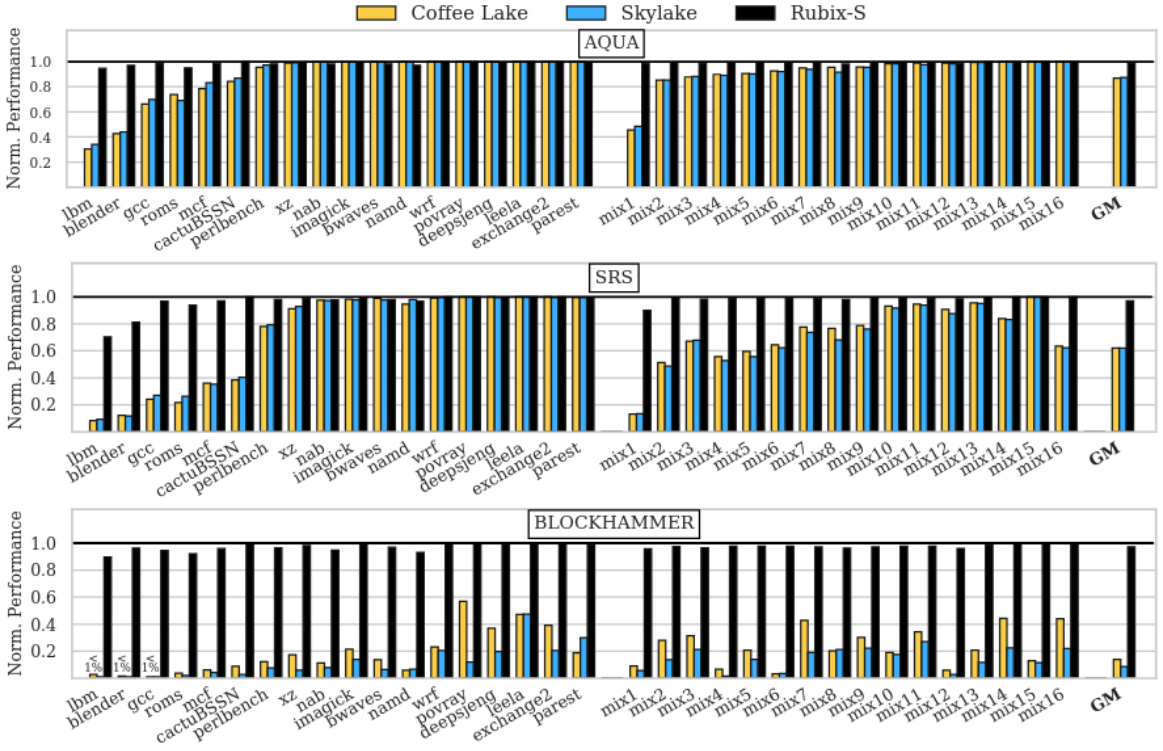


Figure 8. Performance of secure mitigations at T_{RH} of 128 for Intel mappings and Rubix-S, normalized to an unprotected Coffee Lake baseline. Rubix-S uses GS4 for AQUA and SRS, and GS1 for Blockhammer, and reduces the average slowdown to 1.1%, 3.1%, and 2.9%, respectively (down from 15%, 60%, and 600%), making them viable at ultra-low thresholds.

1.4.3.3 gang-size的选择

GS的大小选择本质上是在 热行数量和 rowbuffer 命中率之间的权衡, 以下是不同 GS下各防护的性能损失。

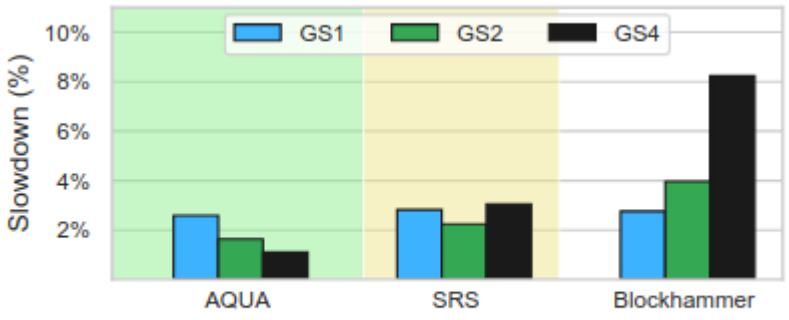


Figure 9. Performance of Rubix-S with Gang-Size of 1-4.

由于高缓解开销，**Blockhammer 最适合使用 GS**1，它消除了热点行。
AQUA 具有较低的缓解开销，因此 最适合GS4，它保留了一定的行缓冲区命中。
对于 SRS，GS2 在行缓冲区命中和最小化热点行之间提供了最佳平衡。

1.4.3.4 对RowBuffer命中的影响

Coffee Lake和Skylake策略下RowBuffer的平均命中率分别是55%和63%。
使用Rubix-S后RowBuffer的**平均命中率为0(GS1)、19%(GS2)和31%(GS4)**

1.4.3.5 对存储和功耗的影响

Rubix在k-cipher和地址映射逻辑上的功率很低， 主要额外开销来自于RowBuffer命中率降低. 这里使用[镁光的功率计算器.](Micron Technology Inc. [n. d.]. System Power Calculators. ([n. d.]). https://www.micron.com/support/tools-and-utilities/power-calc.)

Rubix-S在GS为4时将DRAM功耗**提高了120mW（增加4.3%）**，在GS为1时**提高了300mW（增加10.6%）**，这是由于行缓冲区命中率低于baseline，导致额外的激活。
Rubix-S在启用安全措施后的功耗仍保持在baseline的10%以内，因为几乎没有触发缓解措施.

1.4.4 安全性证明

1.5 Rubix-D

1.5.1 整体结构

Rubix-D通过与随机生成的密钥进行异或操作来实现随机化，并且这种映射会逐渐从旧的密钥变化到新的密钥。

首先介绍一下 Rubix-D的整体结构，在该方案下内存空间在垂直方向也进行一次分组, 每一个row都将自己**划分为v个gang**(图中示例为4, 按照不同颜色区分), 那么第 i 组就由所有row的第 i 个gang组成.
每一个组都有一个独立的密钥(实际是两个密钥， currKey和nextKey, 后续再做解释),一共有 v 个密钥. 当发起一次访存行为时, 地址 L被划分为如图所示的 (L1, L2, L3)三个部分, 随后 L1会被转化成 $L1' = L1 \oplus K_{L2}$, 随后按照 (L1', L2, L3)去DRAM中找对应位置.

使用数据结构将它算法化

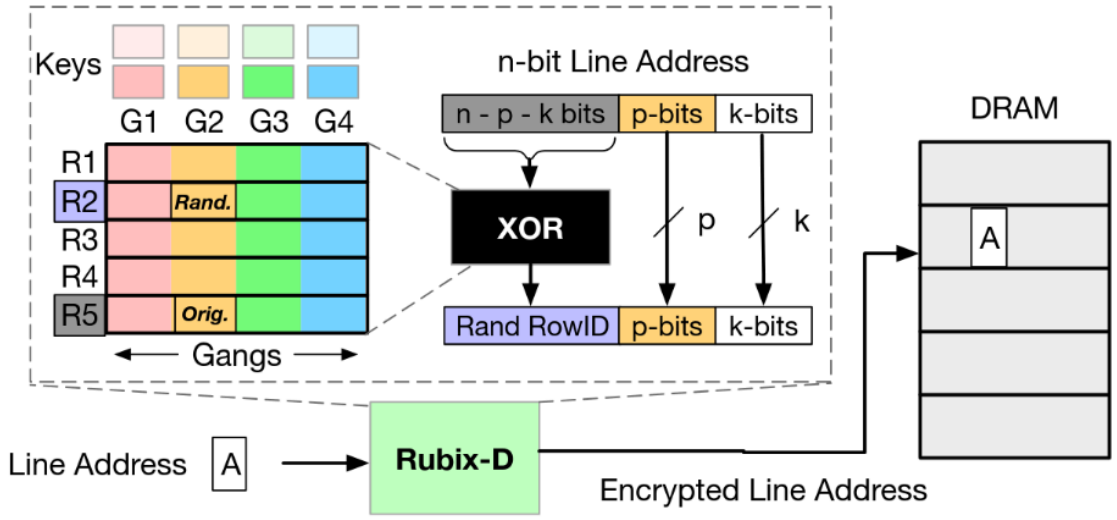


Figure 11. Overview of Rubix-D: gangs within a vertical group (G1, G2, etc.) are routed to random rows in memory.

1.5.2 密钥轮转

在实际运行 Rubix-D时， 由于需要动态的进行密钥轮转(**该值由基于硬件的 PRNG 生成**), 因此需要两个变量来记录当前密钥和下一个密钥. 在切换密钥之后需要一步一步将内存映射由currKey映射转化成按nextKey映射. 具体步骤如下图所示, 为了方便描述这里假设一共只有8个row:

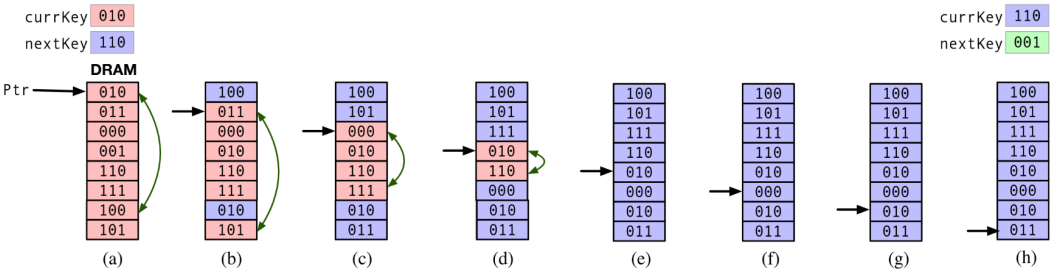
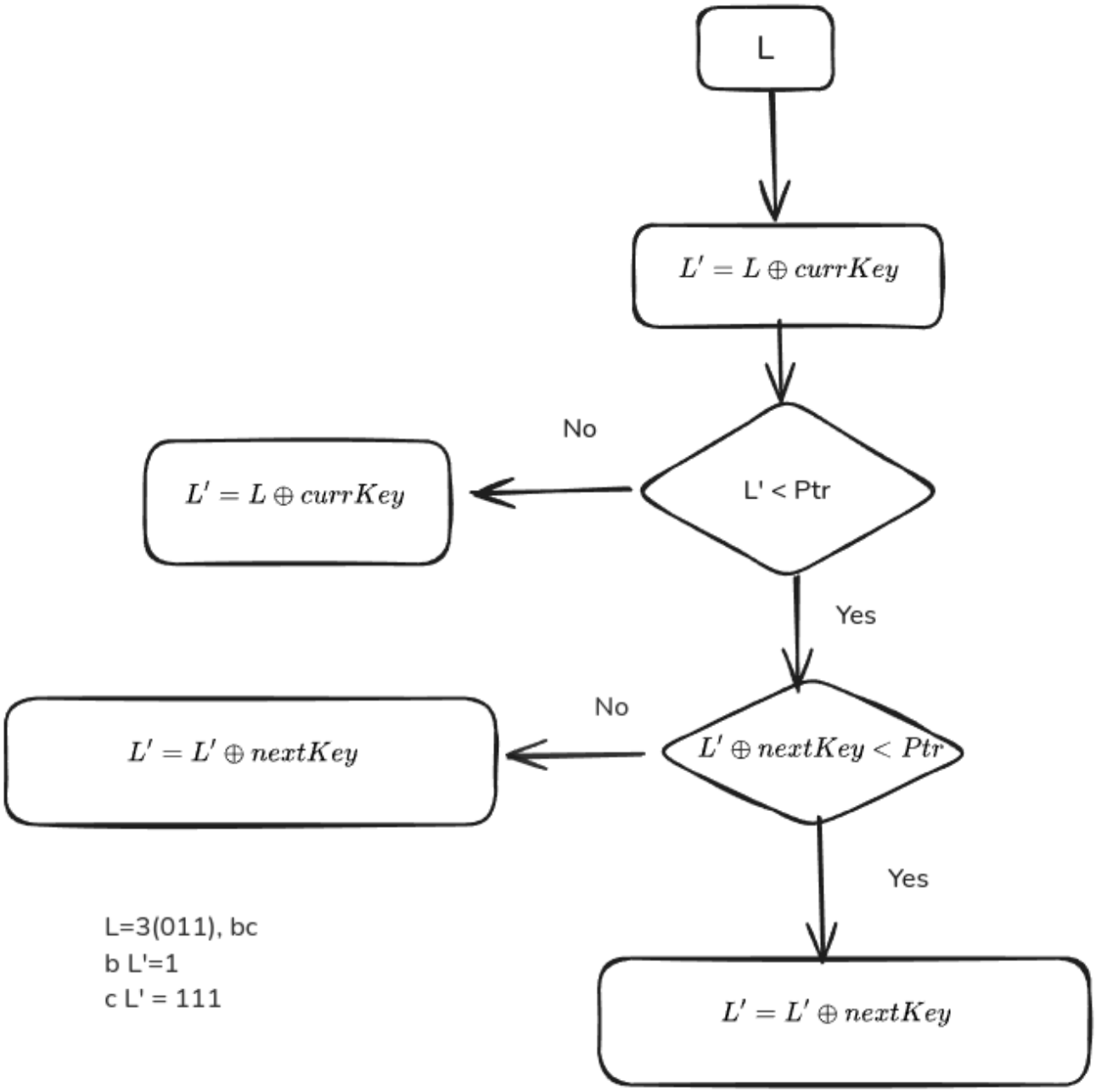


Figure 10. An example of dynamically changing xor-based mapping. The effective address is the line-addressed xor-ed with a key. The dynamic remapping algorithm gradually remaps all the lines from a currKey (010) to the nextKey (110).

系统维护一个指针 ptr 表示重映射进行到了第几个row, 在最开始的阶段 ptr 指向第 0 行, 并将其与第 $ptr \oplus currKey \oplus nextKey$ 行的进行交换, 随后以此依次往下, 直到交换了一半(因为后面的都已经交换过了, 因此跳过).在全部处理完了之后ptr被重置为 0, 等待下一次密钥轮转.

自己设计一个算法来解决列太多的问题， 选择部分

在这个系统之下需要重新设计 上一小结的 L1转化方案， 按照以下步骤进行:



以上的 currKey, NextKey, **ptr均存储在SRAM的额外区域内**, 完成以上检查和地址转换工作只需要1 cycle.

以上分组做了一定的简化，在实际操作中一般有 $2^5 = 32$个组, 一个组里有2^{21} 个gang, 一个gang内有$2^2 = 4$个line. 因此大概需要 512 bytes的额外空间.

1.5.3 Re-mapping的时机

Re-mapping是以组为单位的， 每次访问一个组时有 1%(Remapping-Rate, RR) 的机率会触发密钥轮转，也就是Re-mapping.

对于GS4，内存控制器**从源行和目标行中分别读取4line并进行交换**（open-row-X, read-DataX, open-row-Y, read-DataY, write-DataX-to-Y, open-row-X, writeDataY-to-X）。交换操作涉及**3次激活（ACT）、8次列读取（CAS读取）和8次列写入（CAS写入）**，消耗带宽和功耗.

在 RR = 1%时会增加大概 1.5%的激活次数.

但是每次 remap-period 都会进行大概 200 million次的激活, 我们可以通过划分v-group，使得每第N行**形成一个v-segment**，从而减少重映射周期。每个v-segment都有自己的一组键和指针。当N=32时，v-segment的重映射周期为625 million 次激活；然而，这需要16 KB的SRAM开销用于元数据。

这个办法, 又增加了一次分组， 感觉不太好

1.5.4 额外空间和功耗

对于分段的Rubix-D，存储开销与段的数量成正比（例如，32个段需要16KB的SRAM）。使用美光的功耗计算器计算的DRAM功耗在GS4时增加了130mW（比baseline多4.2%），在GS2时增加了180mW（增加5.8%），在**GS1时增加了320mW（增加10.9%）**。

1.5.4.1 结果

GS1和GS2基本消灭了热行(没有讲是否分段)

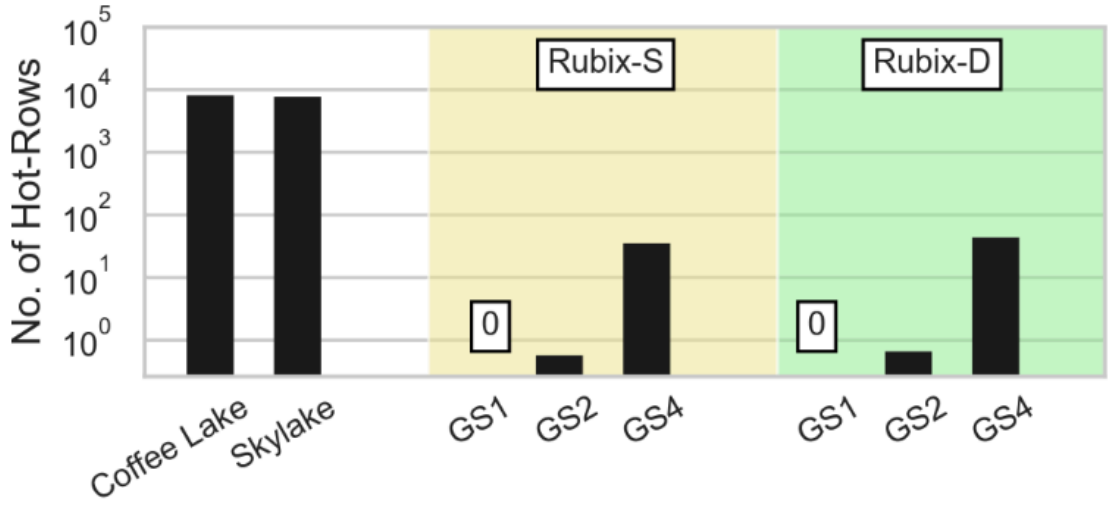


Figure 12. Hot-rows in baseline and Rubix (atleast 100x less).

在 **RR为 1%** 并且没有分段时平均仅产生 1-3% 的低开销。
在 **GS4 时受益于行缓存局部性，AQUA 几乎不会触发缓解措施**。
SRS 在 $T_{RH}/3$ 的较低阈值下运行，触发了更多缓解措施，**在 GS2 时表现最佳**，且热行几乎可以忽略不计。
BlockHammer 具有较高的缓解开销，在 **GS1** 时热行最少的情况下表现最佳。
Rubix-D 在**最坏情况**下的**性能损失仅为 10%**，**而baseline（BlockHammer）则超过 100 倍**。

这里的图中没有体现出 GSX

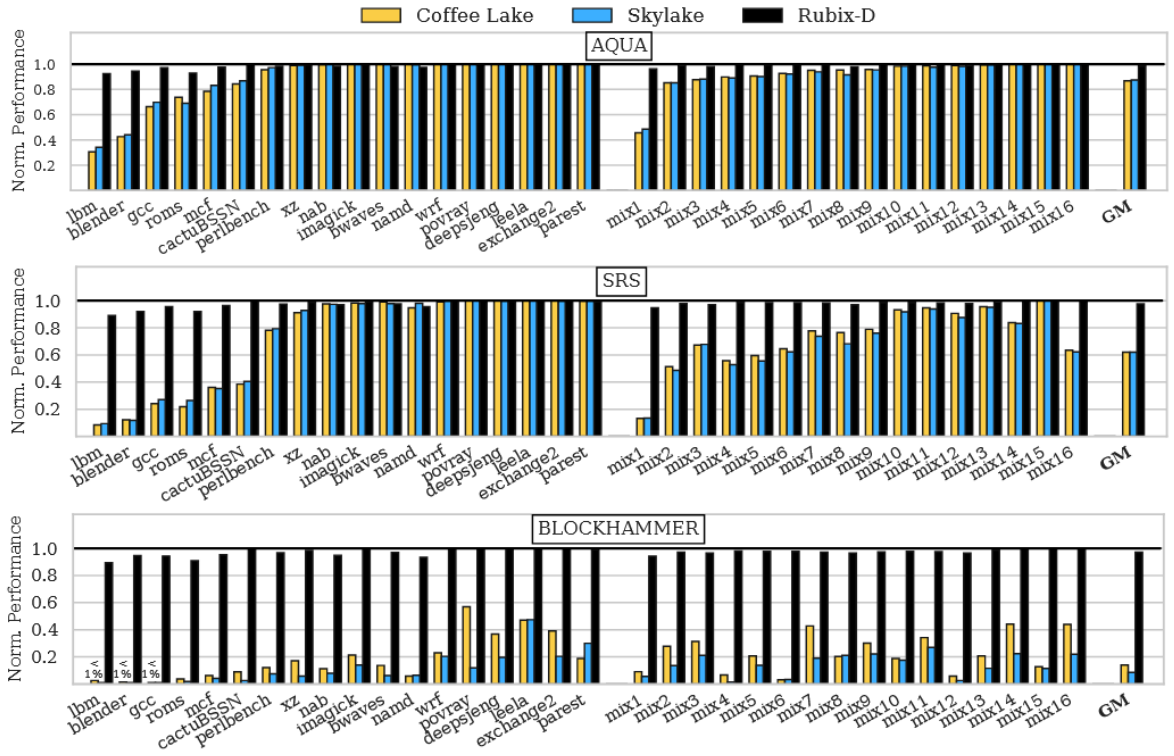


Figure 13. Performance of secure mitigations at T_{RH} of 128 with Intel mappings and Rubix-D, normalized to unprotected Coffee Lake baseline. With GS4 for AQUA, GS2 for SRS, and GS1 for BlockHammer, Rubix-D incurs a low average slowdown of 1.5%, 2.3%, and 2.8%, respectively (down from 15%, 60%, and 600%).

1.5.4.2 Mapping带来的影响

下表展示了Rubix映射在**没有使用任何缓解措施**的情况下的性能损失。由于**行缓冲区命中率低于baseline映射**，Rubix的开销较低，**为1%-3%。**

Rubix-D的开销略高于Rubix-S，因为动**态重映射需要额外的激活。**随机化在最小化热行数量的同时，只带来了较小的性能开销。

	Rubix-S	Rubix-D
GS4	1%	1.3%
GS2	1.6%	1.9%
GS1	2.6%	2.7%

1.5.4.3 多通道带来的影响

我们评估了Intel Coffee Lake和Rubix映射在8核模拟的子集工作负载上的表现，**使用了2通道和4通道配置（32GB DDR4内存和16MB LLC）。**
如下图所示，**Intel的映射引入了很大的开销，分别为15%、45%和380%**，尽管它将一个gang的四个Line分配到4个通道中，但由于连续的line最终以间隔模式落在同一row中。**Rubix打破**

了line与row之间的空间相关性，导致开销仅为1-3%（2通道SRS的Rubix-S为4%）。

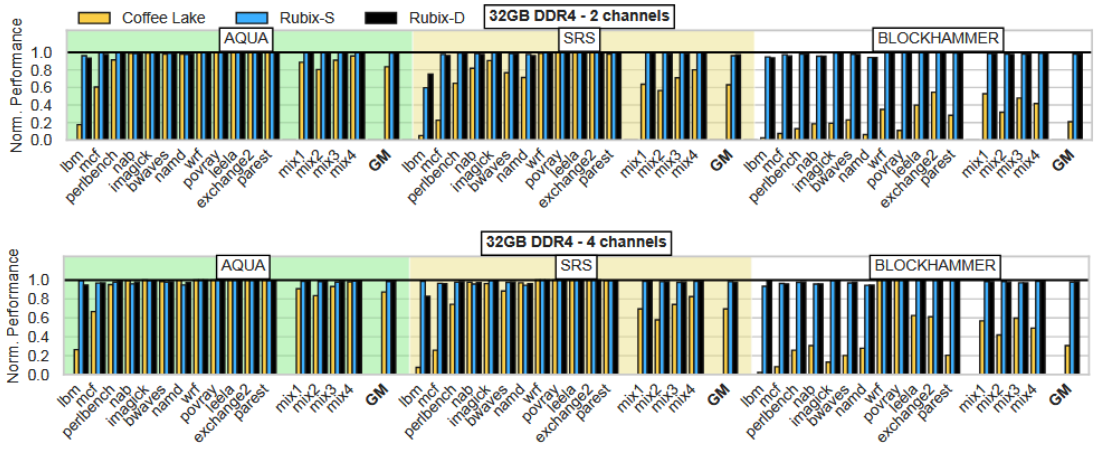


Figure 15. Normalized performance of secure mitigations with Intel and Rubix mappings for an 8-core multi-channel system. While Intel mappings incur impractical average overheads of 15%-380% (AQUA-BlockHammer), Rubix reduces it to 1%-4%.

1.5.4.4 内存密集型任务

我们使用内存密集型顺序访存工作负载评估了Rubix和baseline映射，使用了1 GiB数组（LLC MPKI超过50）。

下图展示了Rubix相对于未保护的Coffeelake和Skylake映射的性能。**Rubix消除了所有顺序访存工作负载中的热点行。**平均而言，**Rubix相对于Coffeelake映射（相对于Skylake映射为5%到8%的性能损失）引入了2%到5%的性能损失，这是由于行缓冲命中率较低**（RubixD由于动态重映射而引入了更多的性能损失）。总体而言，即使对于内存作为性能瓶颈的工作负载，Rubix的成本也很低。

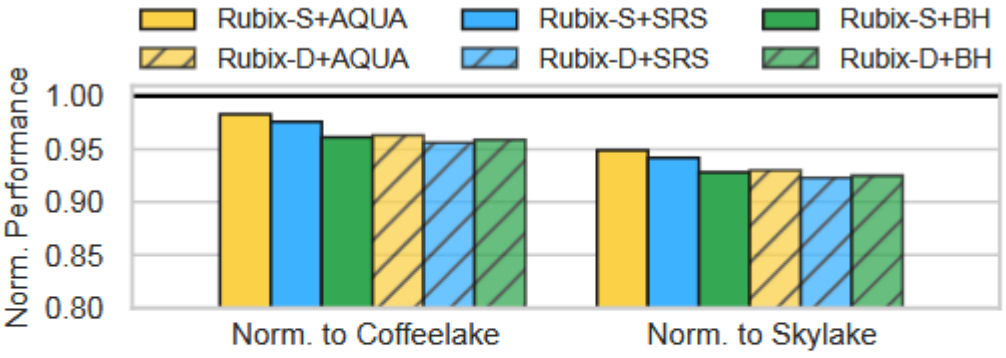


Figure 16. Rubix with secure mitigations incurs 2% to 8% average slowdown (geomean) for memory-intensive workloads compared to unprotected baseline memory mappings.

1.6 想法

- Rubix与别的防护方案相结合
- 更好的 Rubix-D
- 针对Rubix的Dos攻击能影响到什么程度

- 由于 Rubix-D的密钥轮转是概率触发，所以有可能会出现上一次变化密钥时未彻底完成内存交换,就再次变化密钥
- 降低的功耗主要来自与 更少达到阈值导致更少触发缓解措施, 如果单从自身来讲可能增加了很多功耗