题目

摘要

许多微体系结构攻击依赖于攻击者能够高效地找到小的驱逐集：映射到同一缓存组的虚拟地址组。这种能力已经成为缓存侧信道、行扫描和推测执行攻击的决定性基元。尽管它们非常重要，但在文献中对于找到小的驱逐集的算法并没有进行系统研究。在本文中，我们进行了这样一种系统性的研究。我们首先对问题进行了形式化，分析了一组随机虚拟地址是驱逐集的概率。然后，我们提出了基于阈值组测试思想的新算法，以线性时间将随机驱逐集缩减到它们的最小核心，改进了二次的现有技术水平。我们将对我们的算法的理论分析与严格的实证评估相结合，在评估中我们确定并隔离了影响其在实践中可靠性的因素，如自适应缓存替换策略和 TLB 抖动。我们的结果表明，我们的算法能够在比以前更短的时间内找到小的驱逐集，在以前被认为是不切实际的条件下也能实现这一点。

关键词：驱逐集；缓存侧信道攻击；阈值组测试

1 引言

对现代 CPU 微体系结构的攻击已经迅速从学术噱头演变为实际对手手中的强大工具。攻击的显著例子包括针对共享 CPU 缓存的侧信道攻击 [4]、针对 DRAM 的故障注入攻击 [3] 以及泄漏来自推测执行的信息的隐蔽通道攻击 [1]。

许多已记录的攻击的关键要求是攻击者能够将特定的缓存集合置于受控状态。例如，flush +reload [4] 攻击使用特殊指令来使目标缓存内容失效（例如在 x86 上的 clflush），为此它们需要特权执行和共享内存空间。另一类攻击称为 prime+probe，通过替换缓存内容并可以在用户空间或沙盒中无需权限执行。

用于替换缓存内容的基本单元称为驱逐集。技术上，驱逐集是一个（虚拟）地址集合，其中包含至少与缓存具有路数相同数量的元素。直觉上，当访问时，驱逐集将从缓存集中清除所有先前的内容。驱逐集使攻击者能够 (1) 将特定的缓存集合置于受控状态；(2) 通过测量对驱逐集的访问的延迟来探测受害者是否修改了这个状态。

访问足够大的虚拟地址集足以从缓存中逐出任何内容。然而，这样大的驱逐集会增加逐出和探测所需的时间，并因不必要的内存访问而引入噪声。因此，对于有针对性和隐蔽逐出缓存内容，人们寻求识别最小尺寸的驱逐集，这对于例如

- 对最后一级缓存进行定时攻击的内存使用进行细粒度监控；
- 强制内存访问以高频率命中 DRAM 而不是缓存，以在行扫描攻击 [中翻转位；
- 通过确保分支守卫未被缓存来增加被推测执行的指令数 [4]。

2 相关工作

2.1 相关研究

计算最小驱逐集被认为是一个具有挑战性的问题，相当于学习从虚拟地址到缓存集的映射 [2]。问题的难度由攻击者对物理地址位的控制程度决定。例如，在裸机上，攻击者完全控制到缓存集的映射；在大页上，它控制每个缓存切片内的映射到缓存集，但不控制到切片的映射；在常规的 4KB 页上，它仅部分控制到每个切片内集合的映射；在沙盒或强化环境中，可能根本无法控制映射。

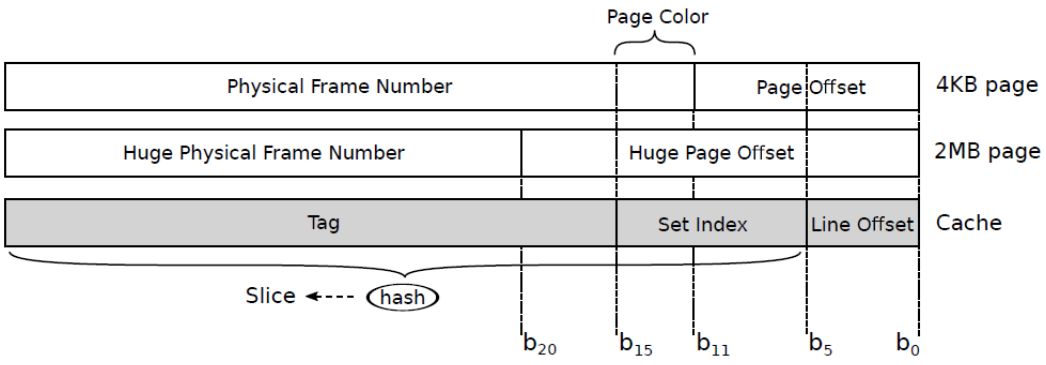
文献中有几种方法讨论了寻找最小驱逐集的算法，详见第 VII 节。这些算法依赖于一个两步法，首先收集足够大的地址集，使其成为一个驱逐集，然后逐步将该集合减小到其最小核心。不幸的是，这些算法通常只被视为实现其他目的的手段，比如设计新的攻击。因此，它们仍然缺乏在复杂性、实时性、正确性和范围方面的深入分析，这同时阻碍了对攻击和合理对策的研究的进展。

2.2 背景知识

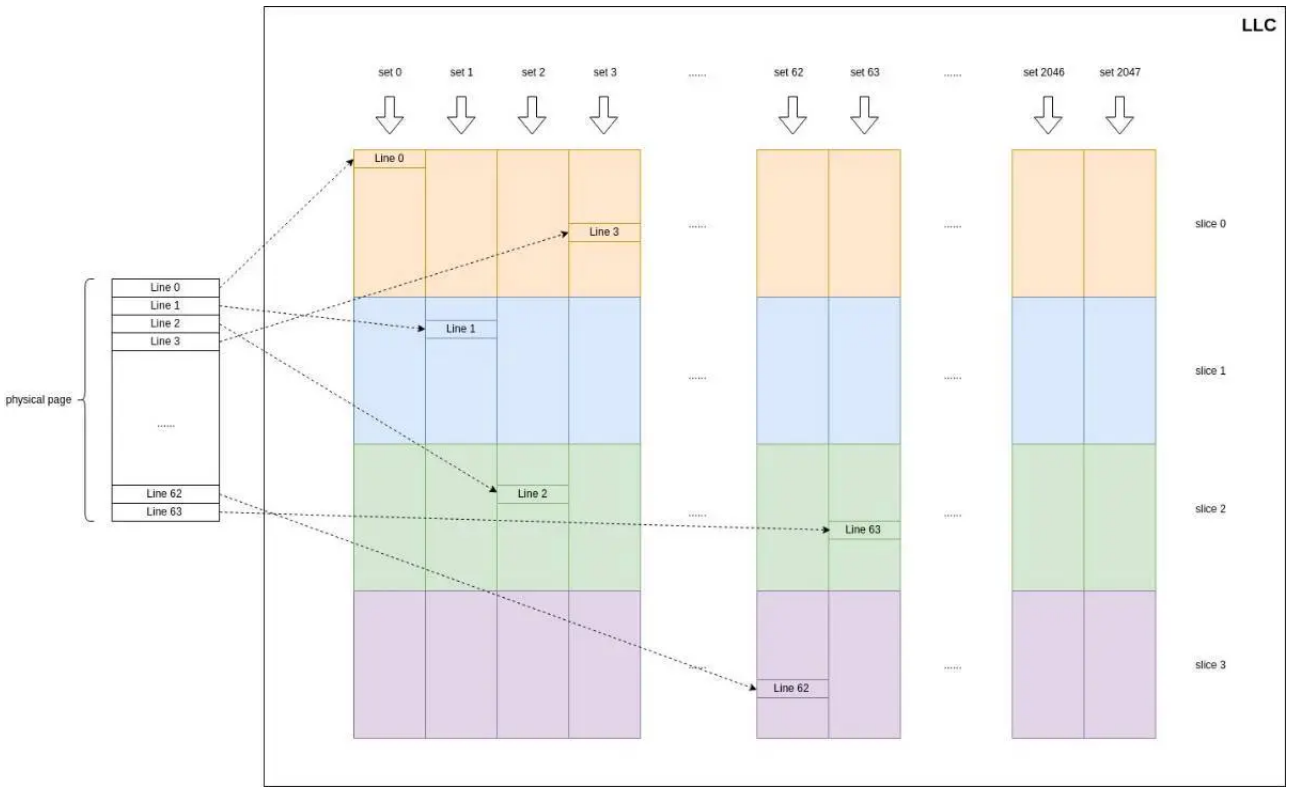
2.2.1 Cache

缓存是快速但容量有限的存储器，用于弥合 CPU 和主内存之间的延迟差距。为了充分利用空间局部性并减少管理开销，主内存在逻辑上被划分为一组块。每个块作为一个整体被缓存在相同大小的缓存行 (Cache line) 中。在访问一个块时，缓存逻辑必须确定该块是否存储在缓存中（缓存命中）或不在缓存中（缓存未命中）。为此，缓存被划分为大小相等的缓存组 (Cache set)。缓存组的大小或行数称为缓存的关联度 a(或 way)。

将主内存内容映射到 LLC 的缓存组是由内容的物理地址确定的。例如，一个具有 n 位物理地址、2^l 字节的缓存行和 2^c 个缓存组。物理地址 $y = (b_{n-1}, ..., b_0)$ 的最低 l 位形成行内偏移，确定在缓存行内的位置。地址 y 的 $(b_{c+l-1}, ..., b_l)$ 是 set 的索引，，用 $set(y)$ 表示。最高 $n - l - c$ 位形成 y 的标记 (tag)。请参见图 1，Intel Skylake 地址位。



现代 Intel CPU 将 LLC 分割成个切片，通常每个 CPU 核心一个或两个切片。切片由地址的最高 $n - l$ 位的未记录的 s 位哈希确定。通过切片，c 个 set 索引位只确定每个切片内的缓存组。总缓存大小 $|M| = 2^{s+c+l}a$，由切片数、每个切片的缓存组数、每行的大小和关联度确定。



Intel 中 LLC 简易架构

2.3 虚拟内存

虚拟内存是进程存储资源的一种抽象，提供了一个线性内存空间，与其他进程隔离并且比物理可用资源更大。操作系统通过 CPU 的内存管理单元（MMU）的帮助来负责将虚拟地址转换为物理地址。

物理内存被划分为大小为 2^p 的页面。常见的页面大小为 4KB（即 p = 12）或用于大页面的 2MB（即 p = 21）。将从虚拟地址到物理地址的转换建模为一个函数 pt。虚拟地址和物理地址在 $(x_{p-1}, ..., x_0)$ 上是相等的。pt 将最高的 n−p 位，称为虚拟页号（VPN），映射到物理帧号（PFN）。图 1 中包含了小页面和大页面的页面偏移和物理帧编号的可视化。

3 本文方法

3.1 驱逐集概念 Eviction sets

3.1.1 驱逐集的定义

两个虚拟地址 x 和 y 是同余的，用 $x \simeq y$ 表示，如果它们映射到相同的缓存组，当且仅当它们各自的物理地址 pt(x) 和 pt(y) 的 set 索引位 set(·) 和切片位 slice(·) 相同时成立。也就是说，$x \simeq y$ 当且仅当：

$$set(pt(x)) = set(pt(y)) \wedge slice(pt(x)) = slice(pt(y)) \tag{1}$$

同余是一个等价关系。关于同余的等价类 [x] 是将映射到与 x 相同的缓存组的虚拟地址的集合。

Definition 1. 一组虚拟地址 S 是

- 对于 x 的驱逐集，如果 $x \notin S$ 并且 S 中至少有 a 个地址映射到与 x 相同的缓存组：

$$|[x] \cap S| \geq a$$

- 对于（任意地址的）驱逐集，如果存在 $x \in S$ 使得 $S \setminus \{x\}$ 是对于 x 的驱逐集：

$$\exists x : |[x] \cap S| \geq a + 1$$

3.1.2 确定驱逐集

根据定义 1. 确定驱逐集需要验证是否满足式（1）。这需要访问物理地址的位，而用户程序无法执行此操作。本节提出了依赖于时序侧信道的测试，用于确定一组虚拟地址是否是驱逐集。

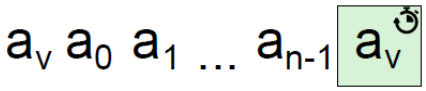


图 1. Test 1: 针对特定地址 a_v 的驱逐测试：(1) 访问 a_v(2) 访问 $S = a_0, ..., a_{n-1}$(3) 访问 a_v
如果步骤 3 的时间大于阈值，那么 S 是 a_v 的驱逐集

测试 1 允许用户程序检查 S 是否是特定虚拟地址 a_v 的驱逐集。该测试依赖于这样一个假设，即程序能够确定 a_v 是否被缓存。在实践中，只要程序可以访问一个时钟以区分缓存命中和未命中，这是可能的。测试 1 也可以作为测试集 S 是否对于任意地址都是驱逐集的基础，通过对所有 $a_i \in S$ 运行 $TEST(S \backslash a_i, a_i)$。然而，这需要的内存访问次数与 S 的大小成二次关系。

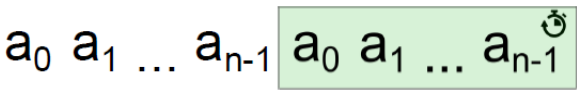


图 2. Test 2: 对于任意地址的驱逐测试：(1) 访问 $S = a_0, ..., a_{n-1}$(2) 再次访问 $S = a_0, ..., a_{n-1}$
如果步骤 2 的总时间大于阈值，那么 S 是驱逐集

测试 2 是对 S 的所有元素进行两次迭代，并测量第二次迭代的总时间。第一次迭代确保所有元素都被缓存。如果第二次迭代的时间超过某个阈值，说明其中一个元素已经从缓存中逐出，表明 S 是一个驱逐集。测试 2 的缺点在于它对噪声敏感，任何在第二次迭代期间引入的延迟都将导致正面的答案。

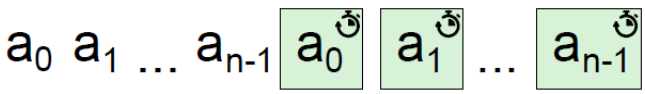


图 3. Test 3: 对于任意地址的稳健驱逐测试：(1) 访问 $S = a_0, ..., a_{n-1}$(2) 再次访问 S，并测量每个元素的访问时间。
如果（2）中超过 a 个元素的访问时间超过阈值，则 S 是一个驱逐集。

本文提出测试 3 作为测试 2 的一个变体。通过测量每次访问的个体时间而不是整体访问时间，可以（1）缩小杂散事件污染测量的时间窗口，以及（2）计数第二次迭代中的确切缓存未命中次数。虽然这提高了对噪声的稳健性，但也伴随着以执行的指令数量为代价。

3.1.3 概率函数

从用户空间控制的物理地址的 γ 位的索引位和不能控制的 $c-\gamma$ 位。γ 的值取决于是否考虑大页面还是小页面。本文中谈到“选择一组给定大小的随机虚拟地址”时，指的是选择在所有 γ 可控制的组索引位上相符的随机虚拟地址。我们现在确定这样一个集合成为驱逐集的概率。

虚拟地址碰撞的概率：首先计算两个在 γ 用户可控制的组索引位上相同的虚拟地址 x 和 y 实际上是一致的概率，并将这个事件称为碰撞，用 C 表示。由于 pt 在其余的 $c-\gamma$ 组索引位和 s 切片位上的作用是随机的，有：

$$P(C) = 2^{\gamma-c-s}$$

Example 1. *考虑图 1 的 Cache，有 8 个切片（s=3），每个切片有 1024 个缓存组 (c=10)*

- *对于大页面(即 $p = 21$), 用户控制所有组索引位, 即 $\gamma = c$, 因此碰撞的概率 $P(C) = 2^{-3}$*
- *对于 4KB 的页面（即 $p = 12$），可控位数为 $\gamma = p - l = 6$，因此碰撞的概率为 $P(C) = 2^{6-10-3} = 2^{-7}$*
- *极限情况（即 $p = l = 6$）对应于用户对虚拟地址到物理地址映射没有任何控制的情况。*

虚拟地址集 S 成为给定地址 x 的驱逐集的概率：本节分析虚拟地址集 S 成为给定地址 x 的驱逐集的概率。这个概率可以用二项分布随机变量 $X \sim B(N,p)$ 的术语来表示，其中参数 $N = |S|, p = P(C)$。对于这样的 X，找到 k 次碰撞的概率，即 $|[x] \cap S| = k$，由以下公式给出：

$$P(X = k) = \binom{N}{k} p^k (1-p)^{N-k}$$

根据定义 1，如果 S 包含至少 a 个与 x 同余的地址（见（1）），则 S 是一个驱逐集。这种情况发生的概率为：

$$\begin{aligned} P(|S \cap [x]| \geq a) &= 1 - P(X < a) \\ &= 1 - \sum_{k=0}^{a-1} \binom{N}{k} p^k (1-p)^{N-k} \end{aligned}$$

任意地址的驱逐集的概率：本文分析了集合 S 包含至少 $a+1$ 个映射到相同缓存组的地址的概率。为此，将问题视为一个单元占用问题。即，我们考虑 $B = 2^{s+c+\gamma}$ 个可能的缓存组（或箱），其中 $N = |S|$ 个地址（或球）均匀分布，询问至少填充一个组（或箱）超过 a 个地址（或球）的概率。

使用随机变量 $N_1, ..., N_B$ 来模拟这个概率，其中 N_i 表示映射到第 i 个缓存组的地址数，约束条件是 $N = N_1 + ... + N_B$。基于这个模型，至少有一个组有超过 a 个地址的概率可以规约未所有 N_i 都小于或等于 a 的互补事件:

$$P(\exists i | N_i > a) = 1 - P(N_1 \leq a, ..., N_B \leq a)$$

3.2 计算最小驱逐集算法

3.2.1 基准方法

伪代码如算法 1 所示。

算法 1 的输入是虚拟地址 x 和针对地址 x 的驱逐集 S。它通过从 S 中选择一个地址 c 并测试 $S\backslash\{c\}$ 是否仍然将 x 驱逐。如果不是（尤其是 if 分支），则 c 必须与 x 同余，并在第 5 行中记录在 R 中。然后在第 7 行从 S 中移除 c 并在第 2 行循环。注意，驱逐测试 TEST 在第 4 行应用于 $R\,(S\backslash\{c\})$，即迄今为止找到的所有同余元素都包括在内。这使得即使剩下的同余元素不足 a 个，也可以扫描 S 以查找它们。该算法在 R 形成 a 个元素的最小驱逐集时终止，这是因为 S 最初是一个驱逐集。

Proposition 1. *算法 1 将驱逐集 S 减少到其最小核心需要 $O(N^2)$ 次内存访问，其中 $N = |S|$。*

Algorithm 1 Baseline Reduction

In: S=candidate set, x=victim address
Out: R=minimal eviction set for v

1: $R \leftarrow \{\}$
2: **while** $|R| < a$ **do**
3: $\quad c \leftarrow pick(S)$
4: $\quad$ **if** $\neg$TEST$(R \cup (S \setminus \{c\}), x)$ **then**
5: $\quad\quad R \leftarrow R \cup \{c\}$
6: $\quad$ **end if**
7: $\quad S \leftarrow S \setminus \{c\}$
8: **end while**
9: **return** R

3.2.2 针对特定地址的最小驱逐集

该算法基于阈值组测试的思想。

阈值组测试：组测试是一种通过对集合（即这些元素的组）进行测试来分解识别具有期望属性的元素的过程。阈值组测试是基于测试的组测试，如果被测试集中的阳性个体数量最多为 l，则给出否定答案，如果数量至少为 u，则给出肯定答案，并且如果在 l 和 u 之间则给出任何答案。在这里，l 和 u 是表示下限和上限阈值的自然数。

用于计算最小驱逐集的线性时间算法：测试一组虚拟地址 S 是否将 x 逐出（参见 Test 1），实际上可以被视为与 x 的一致性的阈值组测试，其中 l = a - 1，u = a。这是因为如果 $|[x] \cap S| \geq a$，则测试给出肯定答案，否则给出否定答案。

Lemma 1. *如果集合 S 包含 p 个或更多阳性元素，则可以使用 $O(p\ log\ |S|)$ 个阈值组测试来识别其中的 p 个元素，其中 $l = p - 1$，$u = p$。*

证明. 引理 1 的思想是将 S 划分为 p + 1 个大致相同大小的不相交子集 $T_1, ..., T_{p+1}$。一个计数论证表明，至少存在一个 $j \in \{1, ..., p+1\}$，使得 $S\backslash T_j$ 仍然是一个驱逐集。通过组测试识

别这样的 j，并在 $S\backslash T_j$ 上重复该过程。对数复杂性是由于 $|S\backslash T_j| = |S|\frac{p}{p+1}$，即每次迭代将驱逐集的大小减小一个因子，而不是像 Algorithm 1 中那样减小一个常数. □

Algorithm 2 基于这个思想计算最小的驱逐集。请注意，引理 1 给出了对组测试次数的界限。然而，对于计算驱逐集，相关的复杂性度量是所进行的测试的集合大小之和，即内存访问的总数。

Proposition 2. *使用 Test 1 的 Algorithm 2 将一个大小为 N 的驱逐 S 减小到其最小核心，其内存访问次数为 $O(a^2N)$，其中 $N = |S|$。*

证明. Algorithm 2 的正确性来自于 S 是一个驱逐集，并且在终止时满足 $|S| = a$ 的不变性，参见引理 1，算法 2 在大小为 N 的集合 S 上执行的内存访问次数遵循以下递推关系：

$$T(N) = T(N\frac{a}{a+1}) + N \cdot a \tag{2}$$

对于 N>a，且 T(a)=a。这个递推关系成立，因为在输入 S 的情况下，该算法对 S 的 a + 1 个子集中的每个应用阈值组测试，每次测试的大小为 $N - \frac{N}{a+1}$ 。拆分和测试的总体成本是 $N \cdot a$。算法对 S 的这些子集中的一个进行递归，其大小为 $N\frac{a}{a+1}$。根据主定理，得出 $T(N) \in \Theta(N)$ □

Algorithm 2 Reduction Via Group Testing

In : S=candidate set, x=victim address
Out : R=minimal eviction set for x

while $|S| > a$ **do**
  $\{T_1, ..., T_{a+1}\} \leftarrow split(S,\ a+1)$
  $i \leftarrow 1$
  while $\neg$TEST$(S \setminus T_i,\ x)$ **do**
    $i \leftarrow i + 1$
  end while
  $S \leftarrow S \setminus T_i$
end while
return S

3.2.3 针对任意地址的最小驱逐集

算法 1 和算法 2 都可以轻松地适应计算任意地址的驱逐集。这只需要将特定地址的逐出测试（Test 1）替换为任意地址的逐出测试（Test 3）。

Proposition 3. *算法 2 采用 Test 3，可以在 O(N) 的内存访问中将驱逐集减少到其最小核心，其中 $N = |S|$。*

计算任意地址的驱逐集的复杂性界与命题 1 和 2 中的相同，因为 Test 1 和 Test 3 都与被测试集的大小成线性关系。

3.2.4 对于大量虚拟地址计算所有最小驱逐集

假设存在一个给定的虚拟地址池 P，并解释如何计算包含在 P 中的所有驱逐集的最小驱逐集。对于足够大的 P，结果可以是所有虚拟地址的驱逐集的集合。

核心思想是使用 P 的一个足够大的子集，并将其减少到任意地址 x 的最小驱逐集 S。使用 S 构建测试 $TEST((S\backslash\{x\})\cup\{y\},x)$，以测试单个地址 y 是否与 x 同余。使用此测试扫描 P 并删除与 x 同余的所有元素。重复此过程，直到在 P 中找不到更多驱逐集。通过使用算法 2 进行线性减少、线性扫描和常量数量的缓存集，此过程需要 O(|P|) 的内存访问以识别 P 中的所有驱逐集。

4 复现细节

4.1 与已有开源代码对比

复现工作参考部分源码，包括内存地址映射模块，地址解析模块等。在此基础上，增加地址序列池功能，减小算法中资源损耗等（创新）。

本工作实现了计算最小驱逐集的基准方法（创新），与论文中的改进算法进行对比，证明本文所提的算法是正确的。

4.2 实验环境搭建

华为云耀服务器：2 核 2 GiB
操作系统：Ubuntu 22.04 server 64bit
gcc 11.4.0 / GNU make 4.3

4.3 界面分析与使用说明

4.3.1 make

将源文件生成目标文件，再链接成为一个共享库（动态链接库）

4.3.2 make clean

删除所有.c 目标文件，以及.so 共享库（动态链接库）

4.3.3 命令行参数

例如：
sudo ./evsets -b 3000 -c 6 -s 8 -a g -e 2 -n 12 -o 4096 -r 10 -t 95 -C 0

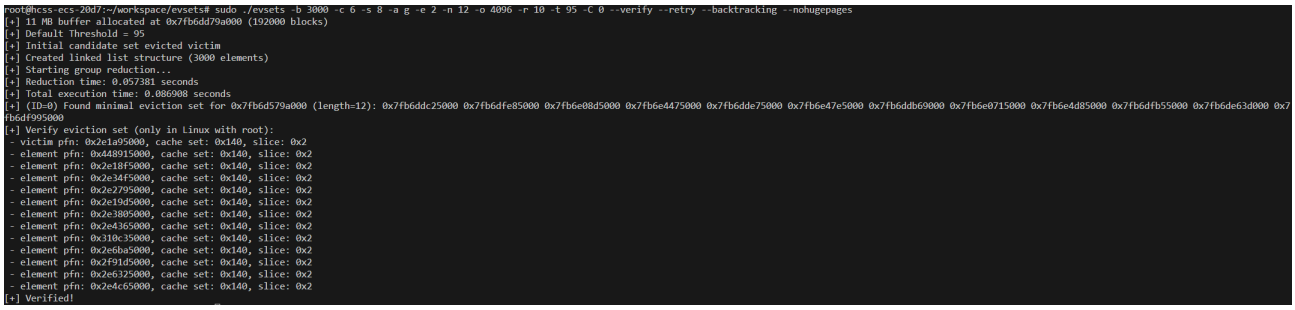
表 1. 命令行的参数设置

-b	N	number of lines in initial buffer
-t	N	threshold in cycles
-c	N	cache size in MB
-s	N	number of cache slices
-n	N	cache associativity
-o	N	stride for blocks in bytes
-a	n|o|g|l	search algorithm
-e	0|1|2|3|4	eviction strategy: 0-haswell, 1-skylake, 2-simple
-C	N	page offset
-r	N	numer of rounds per test
-q	N	ratio of success for passing a test

4.4 创新点

- 首次对找到驱逐集的问题进行了形式化和分析，将一组虚拟地址是驱逐集的概率表示为其大小的函数。该函数表明，一个小的虚拟地址集不太可能是一个驱逐集，但随着集合大小的增加，可能性增长迅速。
- 设计了用于找到最小驱逐集的新算法，仅使用 O(n) 内存访问将大小为 n 的驱逐集减少到其最小核心，这改进了当前 $O(n^2)$ 的最新技术

5 实验结果分析



命令参考 4.3.3，详细信息包括总的执行时间（0.086908s），减少所花时间（0.57381s），最小驱逐集的虚拟地址集（12 个，a = 12）

6 总结与展望

本文将寻找驱逐集作为一个算法问题进行研究。核心理论贡献是新颖的算法，能够在线性时间内计算逐出集，改进了二次时间复杂度的现有技术水平。

未来的研究可以致力于以下几个方向：

- **算法优化：**进一步优化计算最小驱逐集的算法，以提高实时性能、准确性和适用范围。可以探索新的技术和方法，以解决现有算法可能存在的局限性。
- **实验验证：**进行更多实验，验证算法在不同硬件和环境条件下的可靠性和性能。这将有助于确保算法的鲁棒性，并更好地理解其在实际应用中的表现。
- **安全对策：**开发基于算法研究的实际安全对策，以应对现有和潜在的攻击。这可以包括设计硬件或软件防御机制，以减轻驱逐集攻击可能带来的风险。
- **开源共享：**将研究成果开源，促进学术界和工业界的合作。通过共享代码和数据，可以加速领域内的研究进展，并使更多研究人员能够参与到解决这一问题的努力中。

参考文献

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cacheside-channel attacks are practical. pages 605–622, Washington, DC, USA, 2015.

[3] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. 2015.

[4] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.