Theory and Practice of Finding Eviction Sets

Pepe Vila[1,3], Boris Köpf[2] and José F. Morales[1]

[1]IMDEA Software Institute
[2]Microsoft Research
[3]Technical University of Madrid (UPM)

***Abstract*—Many micro-architectural attacks rely on the capability of an attacker to efficiently find small *eviction sets*: groups of virtual addresses that map to the same cache set. This capability has become a decisive primitive for cache side-channel, rowhammer, and speculative execution attacks. Despite their importance, algorithms for finding small eviction sets have not been systematically studied in the literature.**

In this paper, we perform such a systematic study. We begin by formalizing the problem and analyzing the probability that a set of random virtual addresses is an eviction set. We then present novel algorithms, based on ideas from threshold group testing, that reduce random eviction sets to their minimal core in linear time, improving over the quadratic state-of-the-art.

We complement the theoretical analysis of our algorithms with a rigorous empirical evaluation in which we identify and isolate factors that affect their reliability in practice, such as adaptive cache replacement strategies and TLB thrashing. Our results indicate that our algorithms enable finding small eviction sets much faster than before, and under conditions where this was previously deemed impractical.

I. Introduction

Attacks against the micro-architecture of modern CPUs have rapidly evolved from an academic stunt to a powerful tool in the hand of real-world adversaries. Prominent examples of attacks include side-channel attacks against shared CPU caches [1], fault injection attacks against DRAM [2], and covert channel attacks that leak information from speculative executions [3].

A key requirement for many of the documented attacks is that the adversary be able to bring specific cache sets into a controlled state. For example, *flush+reload* [1] attacks use special instructions to invalidate targeted cache content (like `clflush` on x86), for which they require privileged execution and shared memory space. Another class of attacks, called *prime+probe*, evicts cache content by replacing it with new content and can be performed without privileges from user space or from a sandbox.

The primitive used for replacing cache content is called an *eviction set*. Technically, an eviction set is a collection of (virtual) addresses that contains at least as many elements that map to a specific cache set as the cache has *ways*. The intuition is that, when accessed, an eviction set clears all previous content from the cache set. Eviction sets enable an adversary to (1) bring specific cache sets into a controlled state; and to (2) probe whether this state has been modified by the victim, by measuring latency of accesses to the eviction set.

Accessing a large enough set of virtual addresses is sufficient for evicting any content from the cache. However, such large eviction sets increase the time required for evicting and probing, and they introduce noise due to the unnecessary memory accesses. For targeted and stealthy eviction of cache content one hence seeks to identify eviction sets of *minimal size*, which is fundamental, for example, for

- fine-grained monitoring of memory usage by a concurrent process in timing attacks against last-level caches [4], [5];
- enforcing that memory accesses hit DRAM instead of the cache with high enough frequency to flip bits in rowhammer attacks [6];and
- increasing the number of instructions that are speculatively executed by ensuring that branch guards are not cached [3].

Computing minimal eviction sets is recognized as a challenging problem, equivalent to learning the mapping from virtual addresses to cache sets [4]. The difficulty of the problem is governed by the amount of control the adversary has over the bits of physical addresses. For example, on bare metal, the adversary completely controls the mapping to cache sets; on huge pages, it controls the mapping to cache sets within each cache slice, but not the mapping to slices; on regular 4KB pages, it only partially controls the mapping to sets within each slice; and on sandboxed or hardened environments it may not have any control over the mapping at all [7], [5].

Several approaches in the literature discuss algorithms for finding minimal eviction sets, see Section VII for an overview. These algorithms rely on a two-step approach in which one first collects a large enough set of addresses that is an eviction set, and then successively reduces this set to its minimal core. Unfortunately, these algorithms are usually only considered as a means to another end, such as devising a novel attack. As a consequence, they still lack an in-depth analysis in terms of complexity, real-time performance, correctness, and scope, which hinders progress in research on attacks and on principled countermeasures at the same time.

Our approach: In this paper we perform the first systematic study of finding minimal eviction sets as an algorithmic problem. In our study we proceed as follows:

- We give the first *formalization and analysis* of the problem of finding eviction sets. We study different variants of the problem, corresponding to different goals, for example, "evicting a specific cache set", and "evicting an arbitrary cache set". For these goals, we express the probability that a set of

virtual addresses is an eviction set as a function of its size. The function exhibits that a small set of virtual addresses is unlikely to be an eviction set, but that the likelihood grows fast with the set size. This analysis justifies the two-step approach taken in the literature for computing minimal eviction sets, and it exhibits favorable set sizes to start the reduction.

- We design *novel algorithms* for finding minimal eviction sets. The basis of our algorithms are tests from the literature [4] that use the cache side-channel as an oracle to determine whether a given set of virtual addresses is an eviction set. The key observation underlying our algorithms is that these tests can be seen as so-called *threshold group tests* [8]. This observation allows us to leverage ideas from the group testing literature for computing minimal eviction sets. We show that the resulting algorithm reduces an eviction set of size n to its minimal core using only $\mathcal{O}(n)$ memory accesses, which improves over the current $\mathcal{O}(n^2)$ state-of-the-art [9].
- We perform a rigorous *reliability analysis* of our algorithms on Intel's Haswell and Skylake microarchitectures. In our analysis, we identify ways to isolate the influence of TLBs and cache replacement policies. This allows us to exhibit conditions under which our algorithms are almost perfectly reliable, as well as conditions under which their reliability degrades.
- We carry out a *performance analysis* of our algorithms on Intel Skylake. Our analysis shows that the execution time of our algorithms indeed grows only linearly in practice, which leads to significant speed-up compared to the existing quadratic algorithms. While previous approaches rely on assumptions about the number of controlled physical bits (provided by huge and regular pages), our algorithms enable, for first time, computing eviction sets in scenarios without any control of the mapping from virtual addresses to cache sets, as in [7], [5].

Summary of contributions: Our contributions are both theoretical and practical. On the theoretical side, we formalize the problem of finding minimal eviction sets and devise novel algorithms that improve the state-of-the-art from quadratic to linear. On the practical side, we perform a rigorous empirical analysis that exhibits the conditions under which our algorithms succeed or fail. Overall, our insights provide a basis for principled countermeasures against, or paths for further improving the robustness of, algorithms for finding eviction sets.

We also include a tool for evaluating, on different platforms, all tests and algorithms presented in this paper:

https://github.com/cgvwzq/evsets

II. A Primer on Caching and Virtual Memory

In this section we provide the necessary background and notation used along the paper.

A. Caches

Caches are fast but small memories that bridge the latency gap between the CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *blocks*. Each block is cached as a whole in a cache *line* of the same size. When accessing a block, the cache logic has to determine whether the block is stored in the cache (a cache *hit*) or not (a cache *miss*). For this purpose, caches are partitioned into equally sized *cache sets*. The size or number of lines of cache sets is called *associativity* a (or ways) of the cache.

Cache Replacement Policies: Since the cache is much smaller than main memory, a *replacement policy* must decide which memory block to evict upon a cache miss. Traditional replacement policies include least-recently used (LRU), pseudo-LRU (PLRU), and first-in first-out (FIFO). In modern microarchitectures, replacement policies are often more complex and generally not documented. For example, recent Intel CPUs rely on replacement policies [10], [11] that dynamically adapt to the workload.

Cache Hierarchies: Modern CPU caches are organized in multiple levels, with small and fast lower-level caches per CPU core, and a larger but slower *last-level cache* (LLC) that is shared among different cores. The relationship between the content of different cache levels is governed by an inclusion policy. Intel caches, for instance, are usually *inclusive*. This means that the content of higher level caches (L1 and L2) is always a subset of the LLC's. In particular, blocks that are evicted from the LLC are also invalidated in higher levels. In this paper we focus on inclusive LLCs.

Mapping Memory Blocks to Cache Sets: The mapping of main memory content to the cache sets of a LLC is determined by the content's physical address. For describing this mapping, consider an architecture with n-bit physical addresses, cache lines of 2^ℓ bytes, and 2^c cache sets. The least significant ℓ bits of a physical address $y = (b_{n-1}, \ldots, b_0)$ form the *line offset* that determines the position within a cache line. Bits $(b_{c+\ell-1}, \ldots, b_\ell)$ of y are the *set index* bits that determine the cache set, and we denote them by $set(y)$. The most significant $n-\ell-c$ bits form the *tag* of y. See Figure 1 for a visualization of the role of address bits on a Intel Skylake machine.

Cache Slicing: Modern Intel CPUs partition the LLC into different 2^s many *slices*, typically one or two per CPU core. The slice is determined by an undocumented s-bit hash of the most significant $n - \ell$ bits of the address. With slicing, the c set index bits only determine the cache set *within* each slice.

The total cache size $|M| = 2^{s+c+\ell}a$ is then determined as the product of the number of slices, the number of cache sets per slice, the size of each line, and the associativity.

B. Virtual Memory

Virtual memory is an abstraction of the storage resources of a process that provides a linear memory space isolated from other processes and larger than the physically available resources. Operating systems, with help from the CPU's *memory management unit* (MMU), take care of the translation of virtual addresses to physical addresses.

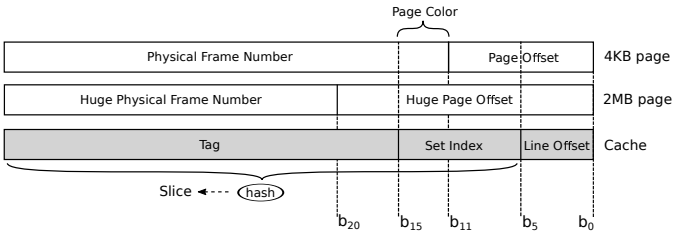


Fig. 1: Mapping from physical addresses to cache sets for Intel Skylake LLC, with 4 cores, 8 slices ($s = 3$), 1024 cache sets per slice ($c = 10$), lines of 64 bytes ($\ell = 6$), and associativity $a = 12$. The figure also displays page offsets and frame numbers for 4KB pages ($p = 12$) and 2MB huge pages ($p = 21$). The set index bits that are *not* part of the page offset determine the *page color*.

Virtual Address Translation: Physical memory is partitioned in *pages* of size 2^p. Common page sizes are 4KB (i.e. $p = 12$), or 2MB for *huge pages*[1] (i.e. $p = 21$).

We model the translation from virtual to physical addresses as a function pt that acts as the identity on the least significant p bits (named *page offset*) of a virtual address $x = (x_{48}, \ldots, x_0)$. That is, the virtual and physical addresses coincide on $(x_{p-1}, \ldots, x_0)$. pt maps the most significant $48-p$ bits, named *virtual page number* (VPN), to the *physical frame number* (PFN). We discuss how pt acts on these bits in Section III-C. Figure 1 includes a visualization of the page offsets and physical frame numbers for small and huge pages, respectively.

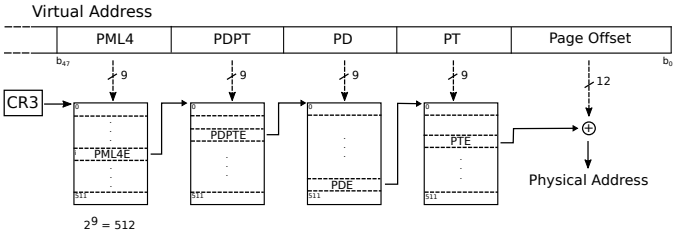


Fig. 2: Page walk on a 64-bit system with four levels of page tables: *PageMap Level 4*, *Page Directory Pointer*, *Page Directory*, and *Page Table* for 4KB pages, respectively. 2MB huge pages can be implemented by using a PD Entry directly as PT Entry. CPU's *Control Register 3* (CR3) points to the PML4 of the running process.

Implementing Virtual Address Translation: Operating systems keep track of the virtual-to-physical mapping using a radix tree structure called *page table* (PT) that is capable of storing the required information efficiently. Whenever a virtual address is accessed, the MMU traverses the PT until it finds the corresponding physical address. This process, also known as a *page walk*, is illustrated in Figure 2. The bits of the VPN are divided into 9-bit indexes for each level of the PT, which can store up to 512 entries (of 8 bytes each). To avoid performing a page walk for each memory access, each CPU core has a *translation lookaside buffer* (TLB) that stores the most recent translations. A page walk only occurs after a TLB miss.

[1]See Appendix A for a discussion of the availability of huge pages on different operation systems.

III. Eviction Sets

In this section we give the first formalization of eviction sets, and present tests that enable determining whether a given set of addresses is an eviction set. We then express the probability that a set of random addresses forms an eviction set as a function of its size. The development of this section forms the basis for the algorithms we develop, and for their evaluation.

A. Defining Eviction Sets

We say that two virtual addresses x and y are *congruent*, denoted by $x \simeq y$, if they map to the same cache set. This is the case if and only if the set index bits $set(\cdot)$ and slice bits $slice(\cdot)$ of their respective physical addresses $pt(x)$ and $pt(y)$ coincide. That is, $x \simeq y$ if and only if:

$$set(pt(x)) = set(pt(y)) \wedge slice(pt(x)) = slice(pt(y)) \quad (1)$$

Congruence is an equivalence relation. The equivalence class $[x]$ of x w.r.t. $\simeq$ is the set of virtual addresses that maps to the same cache set as x. We say that addresses are *partially congruent* if they satisfy the first term of Equation (1), i.e., they coincide on the set index bits but not necessarily on the slice bits.

We now give definitions of eviction sets, where we distinguish between two goals: In the first, we seek to evict a specific address from the cache. This is relevant, for example, to perform precise flushing in rowhammer attacks. In the second, we seek to evict the content of an arbitrary cache set. This is relevant, for example, for high bandwidth covert channels, where one seeks to control a large number of cache sets, but does not care about which ones.

Definition 1. We say that a set of virtual addresses S is

- an *eviction set for* x if $x \notin S$ and at least a addresses in S map to the same cache set as x:

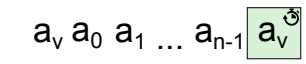
$$|[x] \cap\ S| \geq a$$

- an *eviction set (for an arbitrary address)* if there exists $x \in S$ such that $S \setminus \{x\}$ is an eviction set for x:

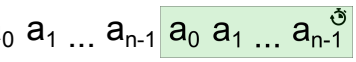
$$\exists x : |[x] \cap S| \geq a + 1$$

The intuition behind Definition 1 is that sequentially accessing all elements of an eviction set for x will ensure that x is *not* cached afterwards. Likewise, sequentially accessing $a + 1$ congruent elements will guarantee that at least one of them is being evicted.

For this intuition to hold, the cache replacement policy needs to satisfy a condition, namely that a sequence of a misses to a cache set evicts all previous content. This condition is satisfied, for example, by all permutation-based policies [12], which includes LRU, FIFO, and PLRU. However, the condition is only partially satisfied by modern (i.e.



Test 1: Eviction test for a specific address a_v: (1) Access a_v. (2) Access $S = \{a_0, \ldots, a_{n-1}\}$. (3) Access a_v. If the time for (3) is larger than a threshold, then S is an eviction set for a_v.



Test 2: Eviction test for an *arbitrary* address: (1) Access $S = \{a_0, \ldots, a_{n-1}\}$. (2) Access S again. If the overall time for (2) is above a threshold, S is an eviction set.

post Sandy Bridge) Intel CPUs. See Section VI for a more detailed discussion.

B. Testing Eviction Sets

Identifying eviction sets based on Definition 1 involves checking whether (1) holds. This requires access to bits of the physical addresses and cannot be performed by user programs. In this section we present tests that rely on a timing side-channel to determine whether a set of virtual addresses is an eviction set.

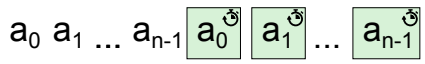
- Test 1 from [4], [5] enables user programs to check whether S is an eviction set for a specific virtual address a_v. The test relies on the assumption that a program can determine whether a_v is cached or not. In practice, this is possible whenever a program has access to a clock that enables it to distinguish between a cache hit or a miss.

Test 1 can also be used as a basis for testing whether a set S is an eviction set for an arbitrary address, by running $\textsc{Test}(S \setminus \{a_i\}, a_i)$, for all $a_i \in S$, and reporting any positive outcome. However, the number of memory accesses required for this is quadratic in the size of S.

- Test 2 is a more efficient solution that has been informally discussed in [5]. The idea is to iterate over all the elements of S twice, and measure the overall time of the second iteration. The first iteration ensures that all elements are cached. If the time for the second iteration is above a certain threshold, one of the elements has been evicted from the cache, showing that S is an eviction set. The downside of Test 2 is that it is susceptible to noise, in the sense that any delay introduced during the second iteration will result in a positive answer.
- We propose Test 3 as a variant of Test 2, drawing inspiration from Horn's proof-of-concept implementation of Spectre [13]. By measuring the individual time of *each* access instead of the overall access time one can (1) reduce the time window in which spurious events can pollute the measurements, and (2) count the exact number of cache misses in the second iteration. While this improves robustness to noise, it also comes with a cost in terms of the number of executed instructions.

C. The Distribution of Eviction Sets

In this section we analyze the distribution of eviction sets. More specifically, we compute the probability that a suitably



Test 3: Robust eviction test for an *arbitrary* address: (1) Access $S = \{a_0, \ldots, a_{n-1}\}$. (2) Access S again, measuring access time of each element. If the access times of more than a elements in (2) is above a threshold, S is an eviction set.

chosen set of random virtual addresses forms an eviction set, for different degrees of adversary control.

Choosing Candidate Sets: For explaining what "suitably chosen" means, we need to distinguish between the γ set index bits of the physical addresses that can be controlled from user space, and the $c - \gamma$ bits that cannot. The value of γ depends, for example, on whether we are considering huge or small pages.

Controlling set index bits from user space is possible because the virtual-to-physical translation pt acts as the identity on the page offset, see Section II-B. When trying to find a minimal eviction set, one only needs to consider virtual addresses that coincide on those γ bits.

The challenge is to find collisions on the $c - \gamma$ set index bits that can*not* be controlled from user space (the page color bits in Figure 1), as well as on the unknown s slice bits. In this section, we assume that the virtual-to-physical translation pt acts as a random function on those bits. This assumption corresponds to the worst case from an adversary's point of view; in reality, more detailed knowledge about the translation can reduce the effort for finding eviction sets [14].

Whenever we speak about "choosing a random set of virtual addresses" of a given size in this paper, we hence refer to choosing random virtual addresses that coincide on all γ set index bits under control. We now determine the probability of such a set to be an eviction set.

Probability of Colliding Virtual Addresses: We first compute the probability that two virtual addresses y and x that coincide on the γ user-controlled set index bits are actually congruent. We call this event a *collision* and denote it by C. As pt acts as a random function on the remaining $c - \gamma$ set index bits and s slice bits, we have:

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

The following example illustrates how removing adversary control increases the difficulty of finding collisions on common cache configurations.

Example 1. Consider the cache from Figure 1 with 8 slices (i.e. $s = 3$) of 1024 cache sets each (i.e. $c = 10$).

- With huge pages (i.e. $p = 21$), the attacker controls all of the set index bits, i.e. $\gamma = c$, hence the probability of a collision $P(C) = 2^{-3}$ is given by the number of slices.
- With pages of 4KB (i.e. $p = 12$), the number of bits under control is $\gamma = p - \ell = 6$, hence the probability of finding a collision is $P(C) = 2^{6-10-3} = 2^{-7}$.
- The limit case (i.e. $p = \ell = 6$) corresponds to an adversary that has no control whatsoever over the mapping of virtual

addresses to set index bits and slice bits – besides the fact that a virtual address always maps to the same physical address. This case corresponds to adding a permutation layer to all adversary-controlled bits (see, e.g. [7]) and is a candidate for a countermeasure that makes finding eviction sets intractable. For this case we obtain $P(C) = 2^{-10-3} = 2^{-13}$.

Probability of a Set to be an Eviction Set for x: We analyze the probability of a set of virtual addresses S to be an eviction set for a given address x. This probability can be expressed in terms of a binomially distributed random variable $X \sim B(N, p)$ with parameters $N = |S|$ and $p = P(C)$. With such an X, the probability of finding k collisions, i.e., $|S \cap [x]| = k$, is given by:

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

According to Definition 1, S is an eviction set if it contains at least a addresses that are congruent with x, see (1). The probability of this happening is given by:

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

Figure 3 depicts the distribution of sets to be an eviction set for x, based on the cache from Figure 1.

Probability of a Set to be an Eviction Set for an arbitrary address: We analyze the probability that a set S contains at least $a+1$ addresses that map to the same cache set. To this end, we view the problem as a cell occupancy problem.

Namely, we consider $B = 2^{s+c-\gamma}$ possible cache sets (or bins) with $N = |S|$ addresses (or balls) that are uniformly distributed, and ask for the probability of filling at least one set (or bin) with more than a addresses (or balls).

We model this probability using random variables $N_1, \ldots, N_B$, where N_i represent the number of addresses mapping to the i-th cache set, with the constraint that $N = N_1 + ... + N_B$. With this, the probability of having at least one set with more than a addresses can be reduced to the complementary event of all N_i being less or equal than a:

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

The right-hand side is a cumulative multinomial distribution, whose exact combinatorial analysis is expensive for large values of N and becomes unpractical for our purpose. Instead, we rely on a well-known approximation based on Poisson distributions [15] for calculating the probabilities.

Figure 3 depicts the distribution of sets to be an eviction set for an arbitrary address, based on the cache from Figure 1. We observe that the probability of the multinomial grows faster with the set size than the binomial distribution. This shows that a set is more likely an eviction set for an arbitrary address than it is for a specific address.

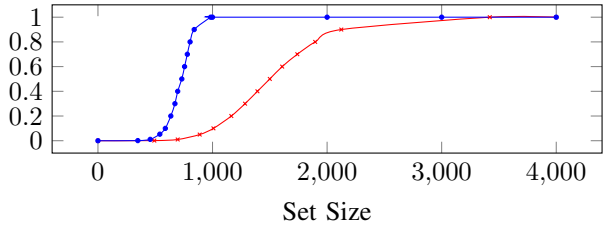


Fig. 3: Probability of random sets to be eviction sets as a function of their size, based on our theoretical models. We use $P(C) = 2^{-7}$ to represent an attacker with 4KB pages in the machine from Figure 1. The *blue-circle* line shows the multinomial model for an "arbitrary" eviction set. The *red-cross* line shows the binomial model for an "specific" eviction set.

Cost of Finding Eviction Sets: We conclude this section by computing the cost (in terms of the expected number of memory accesses required) of finding an eviction set of size N by repeatedly and independently selecting and testing candidate sets.

To this end, we model the repeated independent choice of eviction sets as a geometric distribution over the probability $p(N)$ that a candidate set of size N is an eviction set. The expectation $1/p(N)$ of this distribution captures the expected number of candidate sets that must be tested until we find an eviction set. Assuming that a test of a set of size N requires $\mathcal{O}(N)$ memory accesses, as in Section III-B, this yields an overall cost in terms of memory accesses for finding an initial eviction set of $\mathcal{O}(N/p(N))$.

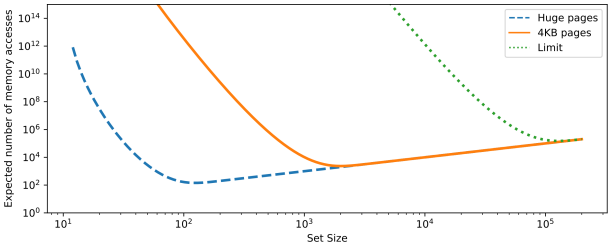


Fig. 4: Expected number of memory accesses for finding an eviction set as a function of its size. The *dashed blue* line represents $P(C) = 2^{-3}$, an attacker with huge pages (i.e. controls all $\gamma = 10$ set index bits). The *plain orange* line represents $P(C) = 2^{-7}$, an attacker with 4KB pages (i.e. controls $\gamma = 6$). The *dotted green* line represents $P(C) = 2^{-13}$, an attacker w/o any control over the set index bits (i.e. $\gamma = 0$).

Figure 4 depicts the cost function $N/p(N)$ for the adversaries from Example 1) for finding eviction sets for a specific address, and highlights the most favorable sizes for finding eviction sets. Since probability grows with set size, finding an eviction set of small size requires, in expectation, large number of trials. Once the probability stabilizes (i.e. the set is large enough), we start seeing the linear cost of the test.

IV. Algorithms for Computing Minimal Eviction Sets

The probability that a set of virtual addresses forms an eviction set depends on its size, on the cache settings (e.g., associativity and number of cache sets), and on the amount of control an adversary has over the physical addresses. In particular, a small set of random virtual addresses is unlikely to be an eviction set. This motivates the two-step approach for finding minimal eviction sets in which one (1) first identifies a large eviction set, and (2) then reduces this set to its minimal core.

Previous proposals in the literature rely on this two-step approach. In this section we first present the baseline reduction from the literature, which requires $\mathcal{O}(N^2)$ memory accesses. We then show that it is possible to perform the reduction using only $\mathcal{O}(N)$ memory accesses, which enables dealing with much larger initial eviction sets than before.

The main practical implication of this result is that finding minimal eviction sets from user (or sandboxed) space is faster than previously thought, and hence practical even without any control over the slice or set index bits. This renders countermeasures based on reducing adversary control over these bits futile.

A. The Baseline Algorithm

We revisit the baseline algorithm for reducing eviction sets that has been informally described in the literature. Its pseudocode is given as Algorithm 1.

Algorithm 1 receives as input a virtual address x and an eviction set S for x. It proceeds by picking an address c from S and tests whether $S \setminus \{c\}$ is still evicting x, see line 4. If it is *not* (notably the *if*-branch), c must be congruent to x and is recorded in R, see line 5. The algorithm then removes c from S in line 7 and loops in line 2.

Note that the eviction test TEST is applied to $R \cup (S \setminus \{c\})$ in line 4, i.e. all congruent elements found so far are included. This enables scanning S for congruent elements even when there are less than a of them left. The algorithm terminates when R forms a minimal eviction set of a elements, which is guaranteed to happen because S is initially an eviction set.

Proposition 1. Algorithm 1 reduces an eviction set S to its minimal core in $\mathcal{O}(N^2)$ memory accesses, where $N = |S|$.

The complexity bound follows because $|S|$ is an upper bound for the number of loop iterations as well as on the argument size of Test 1 and hence the number of memory accesses performed during each call to TEST.

The literature contains different variants of Algorithm 1[4], [5], [9]. For example, the variant presented in [9] always puts c back into S and keeps iterating until $|S| = a$. This still is asymptotically quadratic, but adds some redundancy that helps to combat errors.

If the quadratic baseline was optimal, one could think about preventing an adversary from computing small eviction sets by reducing or removing control over the set index bits, either by adding a mask via hardware [5] or a permutation layer via software [7] (see limit case in Example 1).

Algorithm 1 Baseline Reduction

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

```
1: R ← {}
2: while |R| < a do
3:     c ← pick(S)
4:     if ¬TEST(R ∪ (S \ {c}), x) then
5:         R ← R ∪ {c}
6:     end if
7:     S ← S \ {c}
8: end while
9: return R
```

B. Computing Minimal Eviction Sets for a Specific Address

We present a novel algorithm that performs the reduction of eviction sets to their minimal core in $\mathcal{O}(N)$ memory accesses, where $N = |S|$. This enables dealing with much larger eviction sets than the quadratic baseline and renders countermeasures based on hiding the physical address futile. Our algorithm is based on ideas from threshold group testing, which we briefly introduce first.

1) Threshold Group Testing: Group testing refers to procedures that break up the task of identifying elements with a desired property by tests on sets (i.e. *groups* of those elements). Group testing has been proposed for identifying diseases via blood tests where, by testing *pools* of blood samples rather than individual samples, one can reduce the number of tests required to find all positive individuals from linear to logarithmic in the population size [16].

Threshold group testing [8] refers to group testing based on tests that give a negative answer if the number of positive individuals in the tested set is at most l, a positive answer if the number is at least u, and any answer if it is in-between l and u. Here, l and u are natural numbers that represent lower and upper thresholds, respectively.

2) A Linear-Time Algorithm for Computing Minimal Eviction Sets: The key insight behind our algorithm is that testing whether a set of virtual addresses S evicts x (see Test 1) can actually be seen as a threshold group test for congruence with x, where $l = a - 1$ and $u = a$. This is because the test gives a positive answer if $|[x] \cap S| \geq a$, and a negative answer otherwise. This connection allows us to leverage the following result from the group testing literature for computing minimal eviction sets.

Lemma 1 ([8])**.** If a set S contains p or more positive elements, one can identify p of them using $\mathcal{O}(p \log |S|)$ threshold group tests with $l = p - 1$ and $u = p$.

Proof. The idea behind Lemma 1 is to partition S in $p + 1$ disjoint subsets $T_1, \ldots, T_{p+1}$ of (approximately) the same size. A counting argument (see Appendix C) shows that there is at

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}} ← split(S, a + 1)
    i ← 1
    while ¬TEST(S \ T_i, x) do
        i ← i + 1
    end while
    S ← S \ T_i
end while
return S
```

least one $j \in \{1, \ldots, p+1\}$ such that $S \setminus T_j$ is still an eviction set. One identifies such a j by group tests and repeats the procedure on $S \setminus T_j$. The logarithmic complexity is due to the fact that $|S \setminus T_j| = |S| \frac{p}{p+1}$, i.e. each iteration reduces the eviction set by a factor of its size, rather by than a constant as in Algorithm 1. □

Algorithm 2 computes minimal eviction sets based on this idea. Note that Lemma 1 gives a bound on the number of group tests. For computing eviction sets, however, the relevant complexity measure is the total number of memory accesses made, i.e. the sum of the sizes of the sets on which tests are performed. We next show that, with this complexity measure, Algorithm 2 is linear in the size of the initial eviction set.

Proposition 2. Algorithm 2 with Test 1 reduces an eviction set S to its minimal core using $O(a^2 N)$ memory accesses, where $N = |S|$.

Proof. The correctness of Algorithm 2 follows from the invariant that S is an eviction set and that it satisfies $|S| = a$ upon termination, see Lemma 1. For the proof of the complexity bound observe that the number of memory accesses performed by Algorithm 2 on a set S of size N follows the following recurrence.

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

for $N > a$, and $T(a) = a$. The recurrence holds because, on input S, the algorithm applies threshold group tests on $a+1$ subsets of S, each of size $N - \frac{N}{a+1}$. The overall cost for the split and the tests is $N \cdot a$. The algorithm recurses on exactly one of these subsets of S, which has size $N\frac{a}{a+1}$. From the Master theorem [17] it follows that $T(N) \in \Theta(N)$. □

See Appendix B for a direct proof that also includes the quadratic dependency on associativity.

C. Computing Minimal Eviction Set for an Arbitrary Address

The Algorithms presented so far compute minimal eviction sets for a specific virtual address x. We now consider the case of computing minimal eviction sets for an arbitrary address. This case is interesting because, as shown in Section III-C, a set of virtual addresses is more likely to evict any arbitrary address than a specific one. That is, in scenarios where the target address is not relevant, one can start the reduction with smaller candidate sets.

The key observation is that both Algorithm 1 and Algorithm 2 can be easily adapted to compute eviction sets for an arbitrary address. This only requires replacing the eviction test for a specific address (Test 1) by an eviction test for an arbitrary address (Test 3).

Proposition 3. Algorithm 1, with Test 3 for an *arbitrary* eviction set, reduces an eviction set to its minimal core in $\mathcal{O}(N^2)$ memory accesses, where $N = |S|$.

Proposition 4. Algorithm 2 with Test 3 reduces an eviction set to its minimal core in $\mathcal{O}(N)$ memory accesses, where $N = |S|$.

The complexity bounds for computing eviction sets for an arbitrary address coincide with those in Proposition 1 and 2 because Test 1 and Test 3 are both linear in the size of the tested set.

D. Computing Minimal Eviction Sets for Many Virtual Addresses

We now discuss the case of finding eviction sets for a large number of cache sets. For this we assume a given pool P of virtual addresses, and explain how to compute minimal eviction sets for all the eviction sets that are contained in P. For large enough P the result can be a set of eviction sets for all virtual addresses.

The core idea is to use a large enough subset of P and reduce it to a minimal eviction set S for an arbitrary address, say x. Use S to build a test $\textsc{Test}((S \setminus \{x\}) \cup \{y\}, x)$ for individual addresses y to be congruent with x. Use this test to scan P and remove all elements that are congruent with x. Repeat the procedure until no more eviction sets are found in P. With a linear reduction using Algorithm 2, a linear scan, and a constant number of cache sets, this procedure requires $\mathcal{O}(|P|)$ memory accesses to identify all eviction sets in P.

Previous work [9] proposes a similar approach based on the quadratic baseline reduction. The authors leverage the fact that, on earlier Intel CPUs, given two congruent physical addresses $x \simeq y$, then $x+\Delta \simeq y+\Delta$, for any offset $\Delta < 2^\gamma$. This implies that, given one eviction set for each of the $2^{c-\gamma}$ page colors, one can immediately obtain $2^\gamma - 1$ others by adding appropriate offsets to each address. Unfortunately, with unknown slicing functions this only holds with probability 2^{-s}, what increases the attacker's effort. Our linear-time algorithm helps scaling to large numbers of eviction sets under those conditions.

Another solution to the problem of finding many eviction sets has been proposed in [4]. This solution differs from the two-step approach in that the algorithm first constructs a so-called *conflict set*, which is the union of all minimal eviction sets contained in P, before performing a split into the individual minimal eviction sets. The main advantage of using conflict sets is that, once a minimal eviction set is found, the conflict set need not be scanned for further congruent addresses.

V. Evaluation

In this section we perform an evaluation of the algorithms for computing minimal eviction sets we have developed in Section IV. The evaluation complements our theoretical analysis along two dimensions:

Robustness: The theoretical analysis assumes that tests for eviction sets always return the correct answer, which results in provably correct reduction algorithms. In this section we analyze the robustness of our algorithms in practice. In particular, we study the influence of factors that are outside of our model, such as adaptive cache replacement policies and TLB activity. We identify conditions under which our algorithms are almost perfectly reliable, as well as conditions under which their reliability degrades. These insights can be the basis of principled countermeasures against, or paths forward for improving robustness of, algorithms for finding eviction sets.

Execution time: The theoretical analysis captures the performance of our algorithms in terms of the number of memory accesses. As for the case of correctness, the real execution time is influenced by factors that are outside of our model, such as the total number of cache and TLB misses, or the implementation details. In our empirical analysis we show that the number of memory accesses is in fact a good predictor for the asymptotic real-time performance of our algorithms.

A. Design of our Analysis

Implementation: We implemented the tests and algorithms described in Sections III-B and IV as a command line tool, which can be parameterized to find minimal eviction sets on different platforms. All of our experiments are performed using the tool. The source code is available at: https://github.com/cgvwzq/evsets.

Analyzed Platforms: We evaluate our algorithms on two different CPUs running Linux 4.9:

1) Intel i5-6500 4 x 3.20 GHz (Skylake family), 16 GB of RAM, and a 6 MB LLC with 8192 12-way cache sets. Our experiments indicate that only 10 bits are used as set index on this machine, we hence conclude that each core has 2 slices. Following our previous notation, i.e.: $a_{sky} = 12, c_{sky} = 10, s_{sky} = 3, \ell_{sky} = 6$.

2) Intel i7-4790 8 x 3.60GHz GHz (Haswell family), 8 GB of RAM, and a 8 MB LLC with 8192 16-way cache sets. This machine has 4 physical cores and 4 slices. Following our previous notation, i.e.: $a_{has} = 16, c_{has} = 11, s_{has} = 2, \ell_{has} = 6$. We emphasize that all experiments run on machines with user operating systems (with a default window manager and background services), default kernel, and default BIOS settings.

Selection of Initial Search Space: We first allocate a big memory buffer as a pool of addresses from where we can suitably chose the candidate sets (recall Section III-C). This choice is done based on the adversary's capabilities (i.e., γ), for example, by collecting all addresses in the buffer using a stride of $2^{\gamma+\ell}$, and then randomly selecting N of them. With this method, we are able to simulate any amount of adversary control over the set index bits, i.e. any γ with $\gamma < p - \ell$.

Isolating and Mitigating Interferences: We identify ways to isolate two important sources of interference that affect the reliability of our tests and hence the correctness of our algorithms:

- *Adaptive Replacement Policies:* Both Skylake and Haswell employ mechanisms to adaptively switch between undocumented cache replacement policies. Our experiments indicate that Skylake keeps a few *fixed* cache sets (for example, the cache set *zero*) that seem to behave as PLRU and match the assumptions of our model. Targeting such sets allows us to isolate the effect of adaptive and unknown replacement policies on the reliability of our algorithms.
- *Translation Lookaside Buffers:* Performing virtual memory translations during a test results in accesses to the TLB. An increased number of translations can lead to an increased number of TLB misses, which at the end trigger page walks. These page walks result in *implicit* memory accesses that may evict the target address from the cache, even though the set under test is *not* an eviction set, i.e. it introduces a false positive. TLB misses also introduce a noticeable delay on time measurements, what has been recently discussed in a concurrent work [18]. We isolate these effects by performing experiments for pages of 4KB on huge pages of 2MB, but under the assumption that, as for 4KB pages, only $\gamma = 6$ bits of the set index are under attacker control.

We further rely on common techniques from the literature to mitigate the influence of other sources of interference:

- For reducing the effect of *hardware prefetching* we use a linked list to represent eviction sets, where each element is a pointer to the next address. This ensure that all memory accesses loads are executed in-order. We further randomize the order of elements.
- For reducing the effect of *jitter*, we perform several time measurements per test and compare their average value with a threshold. In our experiments, $10 - 50$ measurements are sufficient to reduce the interference of context switches and other spurious events. More noisy environments (e.g. a web browser) may require larger numbers.

B. Evaluating Robustness

We rely on two indicators for the robustness of our tests and reduction algorithms:

- The *eviction rate*, which is the relative frequency of our tests returning true on randomly selected sets of fixed size.
- The *reduction rate*, which we define as the relative frequency of our reduction succeeding to reduce randomly selected sets of fixed size to a minimal eviction set.

Here, a reduction is successful if the elements it returns are congruent, i.e., they coincide on the set bits and on the slice bits. For this check we rely on the reverse engineered slice function for Intel CPUs [19].

With perfect tests (and hence correct algorithms), both the eviction rate and the reduction rate should coincide with the theoretical prediction given in Section III. Our analysis hence

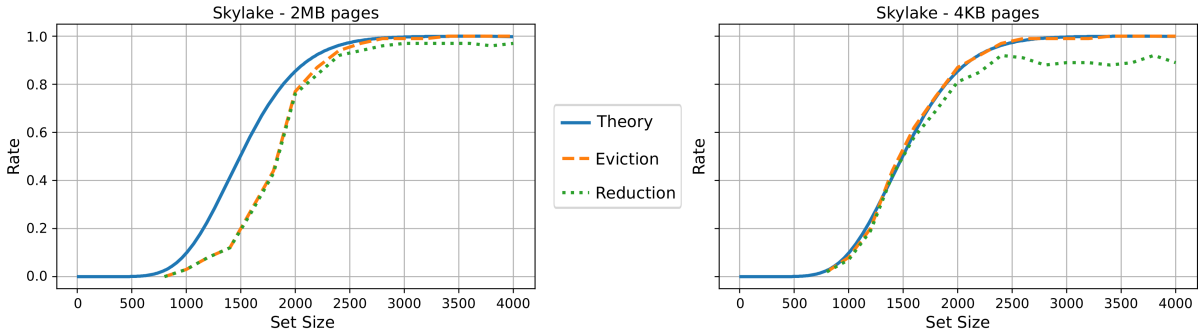
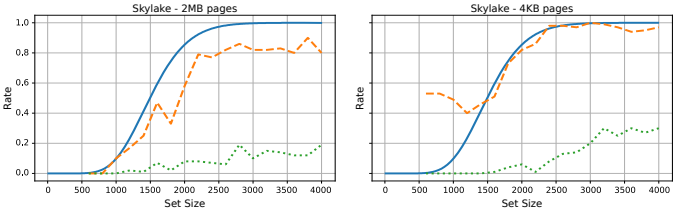
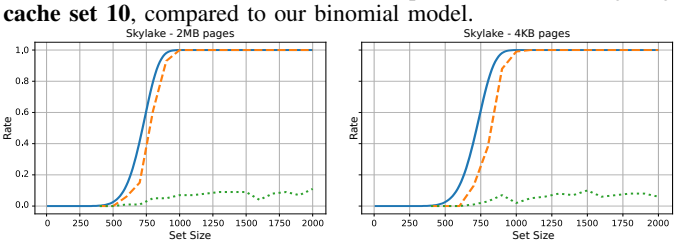


Fig. 5: Skylake: Eviction for specific address x on **cache set *zero***, compared to our binomial model. Each point is the average of 1000 reductions for sets of N randomly chosen addresses.

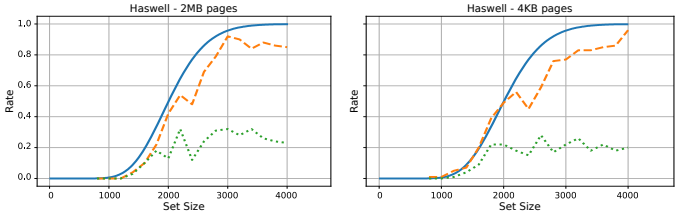


(a) Eviction and reduction rates for an specific address x targeting **cache set 10**, compared to our binomial model.

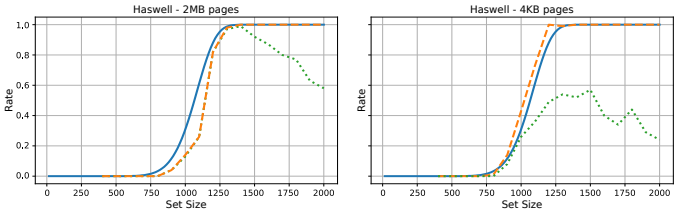


(b) Eviction and reduction rates for an arbitrary address, compared to our multinomial model.

Fig. 6: Experiments on Skylake. Each point is the average of 100 reductions for sets of N randomly chosen addresses.



(a) Eviction and reduction rates for an specific address x, compared to our binomial model.



(b) Eviction and reduction rates for an arbitrary address, compared to our multinomial model.

Fig. 7: Experiments on Haswell. Each point is the average of 100 reductions for sets of N randomly chosen addresses.

focuses on deviations of the eviction and reduction rate from these predictions.

Experimental Results: The experimental results for eviction and reduction for a specific address x are given in Figures 5 and 6a (for Skylake), and Figure 7a (Haswell). The results for arbitrary addresses are given in Figures 6b and 7b (for Skylake and Haswell, respectively). We highlight the following findings:

- *Analysis under idealized conditions* We first analyze our test and reduction algorithms under idealized conditions, where we use the techniques described in Section V-A to mitigate the effect of TLBs, complex replacement policies, prefetching, and jitter. Figure 5 illustrates that, under these conditions, eviction and reduction rates (Test 1 and Algorithm 2) closely match. Moreover, eviction and reduction rates closely match the theoretical prediction for small pages.

For huge pages, however, eviction and reduction rates remain below the theoretical prediction, see Figure 5. We attribute this to the fact that, using explicit allocations (see Appendix A), huge pages are chosen from a pre-allocated pool of physical pages that usually resides in a fixed *zone*. This limits the uniformity of the more significant bits and deviates from our uniform modeling.

- *Effect of the cache replacement policies*. Our experimental results show that the eviction and reduction rates decrease significantly on Haswell (Figure 7a), and on Skylake when targeting a cache set (Figure 6a) different from zero. The effect is also visible in the evaluation of algorithms for finding an arbitrary eviction set (see Figures 6b and 7b).

The decrease seems to be caused by two factors: the replacement policy of the targeted cache sets does not match our models; and the targeted cache set are influenced by accesses to other sets in the cache. We provide further evidence of this effect in Section VI.

- *Effect of TLB thrashing*. Virtual memory translations are more frequent with small pages than with huge pages, which shows in our experiments: The eviction rate lies above the theoretical prediction, in particular for large sets, which shows

the existence of false positives. In contrast, the reduction rate falls off. This is because false positives in tests cause the reduction to select sets that are *not* eviction sets, which leads to failure further down the path.

The effect is clearly visible in Figure 5, where we compare the results on small pages with those on huge pages for cache set zero on Skylake. We observe that the reduction rate on small pages declines for $N > 1500$, which, as Appendix D shows, coincides with the TLB capacity of Skylake of 1536 entries. The effect is also visible in Figure 7b , where we attribute the strong decline of the reduction rate after $N > 1000$ (Haswell's TLB capacity is 1024 entries) to implicit memory accesses having a greater chance to be an eviction set for Haswell's adaptive replacement policy. In the rest of figures the effect is overlaid with interferences of the replacement policy. However, Figure 6b shows that with large TLBs, and for most reasonable values of N, the effect of TLB thrashing is negligible.

C. Evaluating Performance

We evaluate the performance of our novel reduction algorithm and compare it to that of the baseline from the literature. For this, we measure the average time required to reduce eviction sets of different sizes to their minimal core. We first focus on idealized conditions that closely match the assumptions of the theoretical analysis in Section IV.

To put the performance of the reduction in context, we also evaluate the effort that is required for finding an initial eviction set to reduce. For this, we consider attackers with different capabilities to control the set index bits, based on huge pages ($\gamma = 10$), 4 KB pages ($\gamma = 6$), and with no control over the set index bits ($\gamma = 0$).

Together, our evaluation gives an account of how the performance gains of our novel reduction algorithm affect the overall effort of computing minimal eviction sets.

Experimental Results: The results of the evaluation of the reduction for a specific address on Skylake are given in Figure 8. We focus on cache set *zero* to mitigate the effect of the replacement policy, and we mitigate the influence of TLBs and prefetching as described in Section V-A.[2]

Each data point is based on the average execution time of 10 successful reductions. The sizes of the initial sets (x-axis) are chosen to depict the range where finding an initial eviction set does not require picking a too large number of candidate sets (depicted by the green bars). For a more systematic choice of the initial set size see the discussion below.

We highlight the following observations:

- The *slope* of the orange curve clearly illustrates the quadratic growth of execution time of the naive reduction, whereas the blue curve shows the linear growth of our novel algorithms. The *absolute values* account for constant factors such as the 50 time measurements per test, and the overhead due to metrics collection.

[2]In the *limit case* the stride of 64B makes inferences by prefetching prohibitive even with a randomized order, which is why we disable hardware prefetchers using `wrmsr -a 0x1a4 15`.

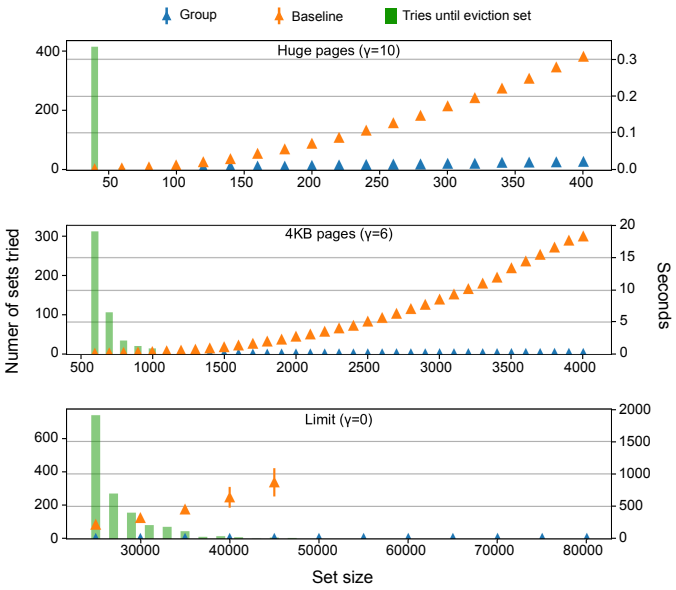


Fig. 8: The vertical green bars (left axis) depict the average number of times one needs to pick a set of addresses for finding an eviction set. Triangles (right axis) show time in seconds: *blue* depicts the average execution time of group test reductions; *orange* depicts the average execution time of baseline reductions. Different plots illustrate attackers with huge pages, 4KB pages, and w/o any control over the set index bits, respectively.

- For large set sizes, our novel reduction clearly outperforms the quadratic baseline. For example, for sets of size 3000, we already observe a performance improvement of a factor of 10, which shows a *clear practical advantage*. For small set sizes, this practical advantage seems less relevant. For such sizes, however, the number of repetitions required until find a real eviction set grows, as illustrated by the green bars. For the total cost of finding an eviction set, both effects need to be considered in combination.

Optimal Choice of the Initial Set Size: For evaluating the cost of first identifying and then reducing eviction sets, we rely on an expression for the overall number of memory accesses required for finding a minimal eviction set. This expression is the sum of the expected number $\frac{N}{p(N)}$ of memory accesses for finding an eviction set, see Section III-C, and of the memory accesses for the respective reductions N^2 and N, see Propositions 1 and 2. Based on this expression, we compute the optimal set sizes (from an attacker's perspective) for the linear and the quadratic reductions. We use these sizes as an approximation of the optimal use of each reduction algorithm in the overall pipeline, and we evaluate their execution time on sets of this size.

Table I shows the comparison of the linear and the quadratic reductions on sets of optimal size for three different attackers: with huge pages, with 4KB pages, and in the limit.

We highlight the following observations:

- For huge pages, computing eviction sets is cheap, and the

		Baseline		Group Testing	
Attacker	$P(C)$	N	Time	N	Time
HP ($\gamma = 10$)	2^{-3}	62	0.005s	62	0.004s
4KB ($\gamma = 6$)	2^{-7}	662	0.179s	862	0.023s
Limit ($\gamma = 0$)	2^{-13}	26650	218.651s	53300	1.814s

TABLE I: N shows the optimal set sizes for different attackers (γ bits) on Skylake ($a = 12$) using 50 time measurements per test. Time shows the average execution time of our implementations of Algorithm 1 (baseline) and Algorithm 2 (group testing) under ideal conditions.

linear-time reduction does not lead to a practical advantage.

- For small pages, the linear-time reduction improves the cost of computing eviction sets by a factor of more than 7. This is a significant advantage in practice, as it can make attacks more stealthy and robust against timing constraints.
- For the limit case, the linear-time reduction improves over the quadratic baseline by more than two orders of magnitude.

D. Performance in Practice

In this section we give examples of the performance benefits of our reduction algorithms in real-world scenarios, i.e., in the presence of TLB noise and adaptive replacement policies.

We implement two heuristics to counter the resulting sub-optimal reduction rates (see SectionV-A): *repeat-until-success*, where we pick a new set and start over after a failed reduction; and *backtracking*, where at each level of the computation tree we store the elements that are discarded, and, in case of error, go back to a parent node on which the test succeeded to continue the computation from there. For more details we refer to our open-source implementation.

For comparing the performance of the reduction algorithms in the context of these heuristics, we follow the literature and focus on initial set sizes that guarantee that the initial set is an eviction set with high probability. This is because a real-world attacker is likely to forgo the complications of repeatedly sampling and directly pick a large enough initial set.

The following examples provide average execution times (over 100 samples) for different attackers on randomly selected target cache sets. Skylake ($a = 12$) using 10 time measurements per test.

- For finding eviction sets with *huge pages*, previous work [20] suggests an initial set size of $N = 192$ which, according to our binomial model (see Section III-C), yields a probability of sets to be evicting close to 1. For this size, the baseline reduction takes 0.014 seconds, while the group-testing reduction takes 0.003 seconds, i.e. our algorithm improves the baseline by a factor of 5.
- For finding minimal eviction sets with *4KB pages*, previous work [9] suggests an initial set size of $N = 8192$, which amounts to the size of LLC times the number of slices. We choose an initial set size of $N = 3420$ for our comparison, which according to our model provides a probability of being an eviction set close to 1. For this N, the baseline reduction takes 5.060 seconds, while the group-testing reduction takes 0.245 seconds, i.e. our algorithm improves the baseline by a factor of 20. Finding *all* minimal eviction sets (for a fixed offset) within this buffer[3] requires more than 100 seconds with the baseline algorithm. With group testing, the same process takes only 9.339 seconds, i.e. it improves by a factor of 10.

E. Summary

In summary, our experiments show that our algorithms improve the time required for computing minimal eviction sets by factors of 5-20 in practical scenarios. Moreover, they show that finding minimal eviction sets from virtual (or sandboxed) memory space is fast *even without any control* over the slice or set index bits, rendering countermeasures based on masking these bits futile.

VI. A Closer Look at the Effect of Modern Cache Replacement Policies

There are several features of modern microarchitectures that are not captured in our model and that can affect the effectiveness of our algorithms, such as adaptive and randomized replacement policies, TLBs, prefetching, etc. The evaluation of Section V shows that the influence of prefetching can be partially mitigated by an adversary, and that the influence of TLBs is not a limiting factor in practice. The role of the cache replacement policy is less clear.

In this section, we take a closer look at the role of modern cache replacement policies in computing minimal eviction sets. As discussed in Section II, post Sandy Bridge architectures boast replacement policies with features such as adaptivity or thrashing-resistance. With such features, accessing a set of a addresses that are congruent with $[x]$ is neither necessary nor sufficient for evicting x, which introduces a two-sided error (false positives and false negatives) in our tests for congruence. We first explain the key mechanisms that lead to this error, before we experimentally analyze its influence on Skylake and Haswell.

A. Adaptive Replacement Policies

Adaptive cache replacement policies [21] dynamically select the replacement policy depending on which one is likely to be more effective on a specific load. For this, they rely on so-called *leader sets* that implement different policies. A counter keeps track of the misses incurred on the leaders and adapts the replacement policy of the *follower sets* depending on which leader is more effective at the moment. There are different ways for selecting the leaders: a *static* policy in which the leader sets are fixed; and a *rand-runtime* policy that randomly selects different leaders every few millions instructions.

A previous study indicates that the replacement mechanism used in Ivy Bridge is indeed adaptive, with static leader sets [11]. To the best of our knowledge, there is no detailed study of replacement mechanisms on more recent generations of Intel processors such as Haswell, Broadwell, or Skylake, but there are mentions of high-frequency policy switches on

[3]We empirically observe that on Skylake, this size is sufficient to contain eviction sets for most of the 128 different cache sets for a fixed offset.

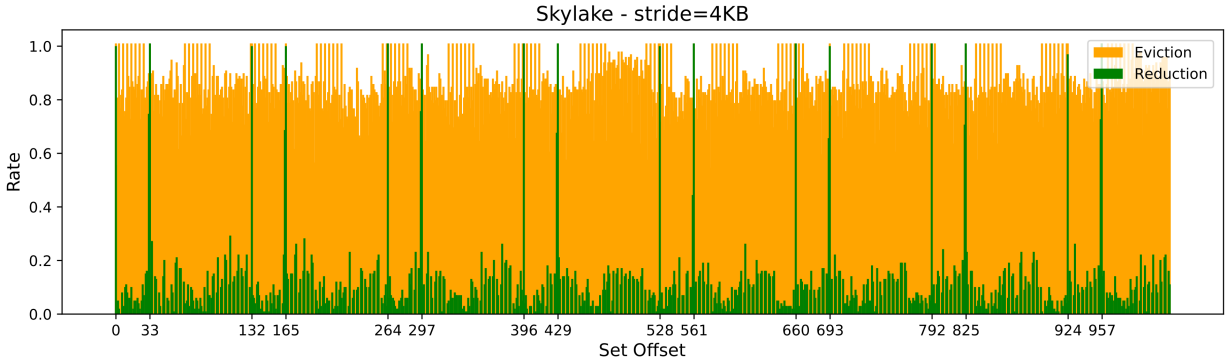


Fig. 9: Skylake's eviction and reduction rates per set index. With a stride of 4KB and a total of 4000 addresses (most of them non-congruent). The number of sets in-between two leaders is either 32 or 98. We rely on huge pages to precisely control the target's set index bits.

Haswell and Broadwell CPUs as an obstacle for prime+probe attacks [9].

We perform different experiments to shed more light on the implementations of adaptivity in Skylake and Haswell, and on their influence on computing minimal eviction sets. To this end, we track eviction and reduction rates (see Section V) for each of the set indexes individually

1) on arbitrary eviction sets
2) on eviction sets in which all addresses are *partially* congruent.

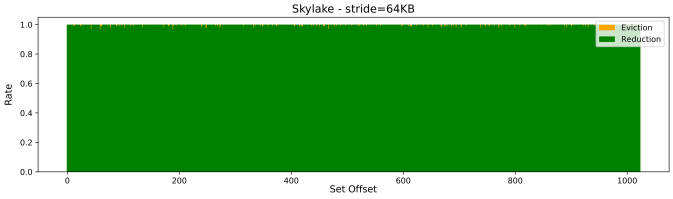
In the second case, the reduction uses only addresses belonging to a single cache set per slice. Assuming independence of cache sets across slice, a comparison with the first case allows us to identify the influence across cache sets. For both experiments we rely on huge pages in order to precisely control the targeted cache set and reduce the effect of the TLB, see Section V-A.

B. Evaluating the Effect of Adaptivity

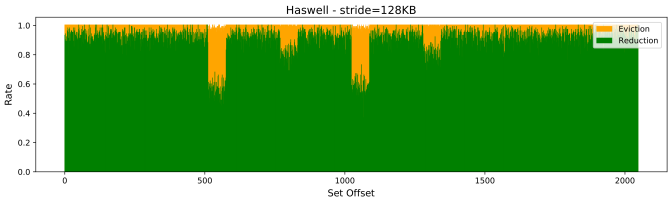
The results for reducing arbitrary eviction sets on Skylake are given in Figure 9, the results for Haswell are given in Figure 10. We focus on initial eviction sets of size $N = 4000$ (but observe similar results for other set sizes). We highlight the following findings:

Fig. 10: Haswell's eviction and reduction rates per set index. With a stride of 4KB and a total of 4000 addresses (most of them non-congruent).

- Skylake seems to implement an adaptive replacement mechanism with static leader sets, much like Ivy Bridge. In particular, we identify a subset of 16 (out of 1024 per slice) sets where the reduction rate is consistently above 95% and where tests consistently evict the target address according to our model (i.e. without false positives and false negatives). On the follower sets the reduction rate quickly falls off despite a high eviction rate, indicating that the test can produce both false positives and false negatives.
- In contrast to Skylake, on Haswell we are not able to identify any leader sets with consistently high reduction rates, which suggests a dynamic leader selection policy is in place.



(a) Skylake's eviction and reduction rates per set index, based on a stride of 64KB (only partially congruent addresses).



(b) Haswell's eviction rate and reduction rate per set index, based on a stride of 128KB (only partially congruent addresses).

Fig. 11: Eviction rate and reduction rate per set index for initial sets of 4000 partially congruent addresses.

The results of our reductions on partially congruent eviction sets on Haswell and Skylake are given in Figure 11. They show that eviction and reduction rates are close to the predicted optimum. This improves over the reduction rate in Figure 9 and 10, and indicates a strong interference on the eviction test when accessing neighboring cache sets. In particular, we observe that the robustness of the reduction increases with the proportion of partially congruent addresses in the initial eviction set.

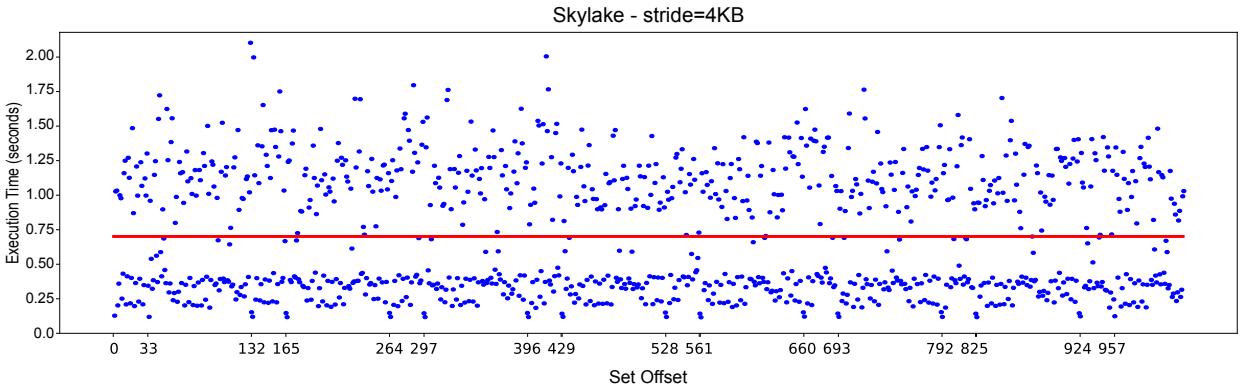


Fig. 12: Skylake's total execution time per set index using backtracking and repeat-until-success. Average time over 100 samples, all of them successful. Stride of 4KB (simulate adversary) and initial set of 4000 addresses (most of them non-congruent). The lowest execution times (below 0.12s), correspond to sets with higher reduction rate. Horizontal line shows the overall average execution time.

Finally, Figure 12 depicts the average execution time, including the overhead of the backtracking heuristic, of finding a minimal eviction set for each cache set index. A lower reduction rate implies a higher number of errors, and hence more backtracking steps and a longer execution time. This effect is visible when comparing with Figure 9: cache sets with the highest reduction rates have the lowest execution times.[4]

C. Future Work

A more detailed analysis of the algorithmic implications of adaptive cache replacement policies is out of the scope of this paper. However, we briefly describe some ideas for future work:

- Controlling the policy. A better understanding of adaptivity mechanisms could be exploited to augment the number of followers that behave deterministically, and hence facilitate the reduction. For instance, once we find eviction sets for a leader set on Skylake, a parallel thread could produce hits on that set (or misses on another leader), ensuring that it keeps the lead.
- Group testing. Work on *noisy* group testing [22], or threshold group testing with gap [8], can provide useful tools for dealing with the uncertainty about the exact behavior of modern microarchitectures.

VII. Related Work

Computing minimal, or at least small, eviction sets provides an essential primitive for removing or placing arbitrary data in the cache, which is essential for LLC cache attacks (Prime+Probe [4], Evict+Reload [23], etc.), for DRAM fault attacks (such as Rowhammer [24], [2], which break the separation between security domains), for memory-deduplication attacks (such as VUSION [25]), as well as for the recent Meltdown [26] and Spectre [3] attacks (which use the cache to leak data across boundaries and to increase the number of speculatively executed instructions).

Gruss et al. [6] already identified *dynamic* and *static* approaches for finding eviction sets. The dynamic method uses timing measurements to identify colliding addresses without any knowledge about the LLC slicing hash function or the physical addresses; whereas the static method use the reverse engineered hash and (partial) information about the physical addresses to compute eviction sets. In practice, most attacks [20] rely in a hybrid approach, producing a partially congruent set of addresses with static methods, and pruning or reducing the results with a dynamic method (mostly variants of Algorithm 1). We review some of the most relevant approaches:

Fully static, without slicing: In CPUs without slicing (such as ARM) it is possible to find eviction sets directly using the information from the pagemap interface. Lipp et al. [27] explore how to perform Prime+Probe, Evict+Reload, and other cross-core cache attacks on ARM. Fortunately, Google patched Android in March 2016 [5], and now it requires *root* privileges to disclose physical addresses, difficulting the task of finding eviction sets.

Static/Dynamic with huge pages: Liu et al. [4] and Irazoqui et al. [5], in their seminal work on attacks against LLC, rely on 2MB huge pages to circumvent the problem of mapping virtual addresses into cache sets. They are the first to propose this method.

Gruss et al. [6] present the first rowhammer attack from JavaScript. To achieve this, they build eviction sets thanks to 2MB huge pages (provided by some Linux distributions with *transparent huge pages* support, see Appendix A).

On the other hand, more sophisticated cache attacks from Intel's SGX [28] rely on the predictable physical allocation of large arrays within SGX enclaves, and on the information extracted from another side-channel in DRAM row's buffers.

[4]The plot also shows two different clusters of execution times, for which we currently lack a satisfying explanation.

[5]Android patch: https://source.android.com/security/bulletin/2016-03-01

Sandboxed environments without huge pages: Oren et al. [9] present an extension to Liu et al.'s work, carrying out the first cache attack from JavaScript, where regular 4KB pages are used and pointers are not directly available. It exploits the knowledge of browser's page aligned allocation for large buffers to construct an initial set with identical page offset bits. Then they leverage the clever technique described in Section IV-D for further accelerating the process of finding other eviction sets.

Dedup Est Machina [14] also implements a JavaScript rowhammer attack, but this time targeting Microsoft Edge on Windows 10. Interestingly, they can not rely on large pages, since Microsoft Edge does not explicitly request them. However, they discover that the Windows kernel dispenses pages for large allocations from different physical memory pools that frequently belong to the same cache sets. Thereby, they are able to efficiently find eviction sets (not minimal) by accessing several addresses that are 128KB apart (and often land in the same cache set).

Horn's [13] breaks virtual machine isolation using a heuristic to find small eviction sets by iterating over Test 3 several times, and discarding all elements that are always *hot* (i.e. always produce cache hits). While this heuristic performs extremely well in practice, its asymptotic cost is quadratic on the set size.

Finally, a more recent work on cache attacks from portable code [18] (PNaCl and WebAssembly) discusses the problem of finding eviction sets on regular 4KB pages and how to partially deal with TLB thrashing.

In contrast to these approaches, our work is the first to consider adversaries with less than 12 bits of control over the physical addresses, it formalizes the problem of finding eviction sets, and provides new techniques that might *enable purely dynamic approaches*.

Reverse engineering of slicing functions: Modern CPUs [6] with LLC slicing use proprietary hash functions for distributing blocks into slices, which lead to attempts to reverse engineer them. These works are based on: 1) allocating and identifying sets of colliding addresses [30], [19]; and 2) reconstructing the slice function using the hamming distance [31], or solving systems of equations [32], between these addresses. Even though we now know the slice hash function for several microarchitectures, and Maurice et al. [20] leverage it to speed up the finding of eviction sets with huge pages, we believe that its use on real attacks is hindered by constrained environments with scarce information about the physical addresses.

Thrashing/scanning resistant replacement policies: Modern replacement policies such as insertion policies [33] or DRRIP [10], are known to perform better than PLRU against workloads causing scanning or thrashing. However, they also make eviction less reliable, and fall outside our current models (see Section III). Howg [11] proposes a *dual pointer chasing* to mitigate these effects; and Gruss et al. [6] generalize the approach with *eviction strategies*, which are access patterns over eviction sets that increase the chance of eviction under some unknown modern policies. Both approaches are orthogonal to our in that they already assume the possession of eviction sets.

Set index randomization: Concurrent work proposes some new designs for randomized caches [34], [35], where cache sets are indexed with a keyed function that completely voids any attacker control over the physical address bits. A key result of these proposals is that they make cache-based attacks, and specially finding small eviction sets, more difficult. Their security analysis, however, considers quadratic attackers; it will be interesting to see how it is affected by our linear-time algorithm.

[6] According to Intel's Architecture Reference Manual [29] (see *2.4.5.3 Ring Interconnect and Last Level Cache*), Sandy Bridge is the first generation with slicing.

VIII. Conclusion

Finding small eviction sets is a fundamental step in many microarchitectural attacks. In this paper we perform the first study of finding eviction sets as an algorithmic problem.

Our core theoretical contribution are novel algorithms that enable computing eviction sets in linear time, improving over the quadratic state-of-the-art. Our core practical contribution is a rigorous empirical evaluation in which we identify and isolate factors that affect their reliability in practice, such as adaptive replacement strategies and TLB thrashing.

Our results demonstrate that our algorithms enable finding small eviction sets much faster than before, enabling attacks under scenarios that were previously deemed impractical. They also exhibit conditions under which the algorithms fail, providing a basis for research on principled countermeasures.

Acknowledgments

We thank Trent Jaeger, Pierre Ganty, and the anonymous reviewers for their helpful comments. This work was supported by a grant from Intel Corporation, Ramón y Cajal grant RYC-2014-16766, Spanish projects TIN2015-70713-R DEDETIS and TIN2015-67522-C3-1-R TRACES, and Madrid regional project S2013/ICE-2731 N-GREENS.

References

[1] 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)*, (San Diego, CA), pp. 719–732, USENIX Association, 2014.

[2] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," in *Black Hat*, 2015.

[3] 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.

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks Are Practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, (Washington, DC, USA), pp. 605–622, IEEE Computer Society, 2015.

[5] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, (Washington, DC, USA), pp. 591–604, IEEE Computer Society, 2015.

[6] Daniel Gruss and Clémentine Maurice and Stefan Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, Springer, 2016.
[7] D. G. Michael Schwarz, Moritz Lipp, "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks," in *Network and Distributed System Security Symposium 2018 (NDSS'18)*, 2018.
[8] P. Damaschke, *Threshold Group Testing*, pp. 707–718. Springer Berlin Heidelberg, 2006.
[9] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications," in *CCS*, ACM, 2015.
[10] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," *SIGARCH Comput. Archit. News*, vol. 38, pp. 60–71, June 2010.
[11] H. Wong, "Intel Ivy Bridge Cache Replacement Policy," January 2013.
[12] A. Abel and J. Reineke, "Reverse engineering of cache replacement policies in intel microprocessors and their evaluation," in *IEEE International Symposium on Performance Analysis of Systems*, 2014.
[13] J. Horn, "Reading privileged memory with a side-channel." https://googleprojectzero.blogspot.com.es/2018/01/reading-privileged-memory-with-side.html, 2018.
[14] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pp. 987–1004, 2016.
[15] B. Levin, "A representation for multinomial cumulative distribution functions," *The Annals of Statistics*, pp. 1123–1126, 1981.
[16] R. Dorfman, "The detection of defective members of large populations," *The Annals of Mathematical Statistics*, vol. 14, no. 4, pp. 436–440, 1943.
[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," *Growth*, vol. 5, no. A8, p. B1, 1989.
[18] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code." Cryptology ePrint Archive, Report 2018/119, 2018. https://eprint.iacr.org/2018/119.
[19] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, November 2015.
[20] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Rmer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS, The Internet Society, February 2017.
[21] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," *SIGARCH Comput. Archit. News*, vol. 34, pp. 167–178, May 2006.
[22] S. Cai, M. Jahangoshahi, M. Bakshi, and S. Jaggi, "Efficient algorithms for noisy group testing," *IEEE Transactions on Information Theory*, vol. 63, pp. 2113–2136, April 2017.
[23] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 897–912, USENIX Association, 2015.
[24] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 361–372, IEEE Press, 2014.
[25] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure Page Fusion with VUsion," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pp. 531–545, 2017.
[26] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *CoRR*, vol. abs/1801.01207, 2018.
[27] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 549–564, USENIX Association, 2016.
[28] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, *Malware Guard Extension: Using SGX to Conceal Cache Attacks*, pp. 3–24. Springer International Publishing, 2017.
[29] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 2018.
[30] "L3 cache mapping on Sandy Bridge CPUs." http://lackingrhoticity.blogspot.com.es/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html, 2015.
[31] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 191–205, IEEE Computer Society, 2013.
[32] G. I. Apecechea, T. Eisenbarth, and B. Sunar, "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors," *IACR Cryptology ePrint Archive*, vol. 2015, p. 690, 2015.
[33] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer, "Adaptive insertion policies for high performance caching," in *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pp. 381–391, 2007.
[34] D. Trilla, C. Hernández, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pp. 98:1–98:6, 2018.
[35] M. K. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *IEEE/ACM International Symposium on Microarchitecture - MICRO 2018*, 2018.
[36] "The linux kernel archives: Transparent huge pages." https://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2017.

APPENDIX

A. Huge Pages

Modern operating systems implement support for large buffers of virtual memory to be mapped into contiguous physical chunks of 2MB (or 1GB), instead that of regular 4KB. These large chunks are called huge pages. On one hand, huge pages save page walks when traversing arrays of more than 4KB, improving performance. On the other hand, they increase the risk of memory fragmentation, what might lead to wasting resources.

On Linux systems, huge pages can be demanded explicitly or implicitly:

- Explicit requests are done by passing special flags to the allocation routine (e.g. flag `MAP_HUGETLB` to the `mmap` function). In order to satisfy these requests, the OS pre-allocates a pool of physical huge pages of configurable size (which by default is 0 in most systems).
- Implicit requests are referred as *transparent huge pages* [36]. THPs are implement with a kernel thread that, similarly to a garbage collector, periodically searches for enough contiguous 4KB virtual pages that can be re-mapped into a free 2MB chunk of contiguous physical memory (reducing PTs size). THP can be configured as: `always`, meaning that all memory allocations can be re-mapped; `never`, for disabling it; and `madvise`, where the programmer needs to signal preference for being re-mapped via some flags (note that this is not a guarantee).

On other systems huge pages are implement differently, but the effect is generally the same. For instance, BSD's documentation refers to them as *super pages*, while Windows calls them *large pages*.

Interestingly, memory allocations in modern browsers are not backed by huge pages unless the system is configured with THP set to `always`. Hence, relying on them for finding eviction sets is not feasible in most default systems.

B. Proof of Proposition 2

In the worst case, we access $a+1$ different a-subsets groups of size $\frac{n}{a+1}$ each, and safely discard $\frac{n}{a+1}$ elements that are not part of the minimal eviction set. We first express recurrence (2) as a summation

$$T(n) = an + an(\frac{a}{a+1}) + an(\frac{a}{a+1})^2 + ... + an(\frac{a}{a+1})^k$$

Our termination condition is $n(\frac{a}{a+1})^k < a$, meaning that we already have an minimal eviction set. By using the logarithm we can set the exponent of the last iteration as $k = \log_{a/(a+1)} a/n$, which allows us to define the function as a geometric progression

$$T(n) = a\sum_{i=1}^{i=k} n(\frac{a}{a+1})^{i-1} = \frac{an(1-(\frac{a}{a+1})^{log_{a/(a+1)}\, a/n})}{1-\frac{a}{a+1}}$$

The logarithm in the exponent cancels out, and we obtain

$$T(n) = \frac{an(1-\frac{a}{n})}{1-\frac{a}{a+1}} = a(n-a)(a+1) = a^2n + an - a^3 - a^2$$

For simplicity we ignore the effect of the ceiling operator required in a real implementation, where n is always an integer. It can be shown that this error is bounded by a small factor of our k, so we consider it negligible.

C. Pidgeonhole Principle

Proposition 5. Let $|S \cap P| \geq a$ and let $T_1, \ldots, T_{a+1}$ be a partition of S. Then there is $i \in \{1, \ldots, a+1\}$ such that $|(S \setminus T_i) \cap P| \geq a$

Proof. When $|S \cap P| = a$, one of the $a+1$ blocks of the partition must have an empty intersection with P, say T_j. Then $|(S \setminus T_j) \cap P| = |S \cap P| = a$.

When $|S \cap P| > a$, assume for contradiction that

$$\forall i \in \{1, \ldots, a+1\} \colon\ |(S \setminus T_i) \cap P| < a\ . \tag{3}$$

We introduce the following notation: $p_i = |T_i \cap P|$ and $x = |S \cap P|$. With this, (3) can be reformulated as

$$\forall i \colon x - p_i < a$$

Summing over all i we obtain

$$(a+1)x - x = ax < (a+1)a\ ,$$

which contradicts the assumption that $x > a$. □

D. Intel's TLBs

Modern CPUs have very distinct TLBs implementations. In particular, modern Intel CPUs implement different buffers for data (dTLB) and instructions (iTLB), a second level TLBs (sTLB) with larger capacity, and different TLBs for each PT level.

Table II shows a summary of TLB parameters for Haswell and Skylake families:

	Haswell	Skylake
iTLB 4K	128 entries; 4-way	128 entries; 8-way
iTLB 2M/4M	8 entries; full	8 entries; full
dTLB 4K	64 entries; full	64 entries; 4-way
dTLB 2M/4M	32 entries; 4-way	32 entries; 4-way
dTLB 1G	4 entries; 4-way	4 entries; full
sTLB 4K/2M	1024 entries; 8-way	1536 entries; 4-way
sTLB 1G	-	16 entries; 4-way

TABLE II: TLB implementation information for Haswell and Skylake microarchitectures. Extracted from the Intel's Architectures Optimization Manual [29].