Gaussian Elimination of Side-Channels: Linear Algebra for Memory Coloring

Jana Hofmann*
jana.hofmann@mpi-sp.org
MPI-SP
Bochum, Germany

Cédric Fournet
fournet@microsoft.com
Azure Research, Microsoft
Cambridge, UK

Boris Köpf
boris.koepf@microsoft.com
Azure Research, Microsoft
Cambridge, UK

Stavros Volos
svolos@microsoft.com
Azure Research, Microsoft
Cambridge, UK

Abstract

Memory coloring is a software-based technique to ensure microarchitectural isolation between trust domains sharing a CPU. Prior coloring schemes target individual microarchitectural components and thus provide only partial solutions. In this paper, we provide theoretical foundations and practical algorithms to infer comprehensive coloring schemes for modern cloud CPUs.

To this end, we first formulate the requirements for effective memory coloring schemes in a set-theoretic model, including definitions for simultaneous isolation of shared components and uniform utilization of private components. We then algebraically characterize these requirements for microarchitectural components that are indexed by linear functions, which is the prevalent case in today's CPUs. Based on this, we develop efficient algorithms for computing multi-resource coloring schemes from linear indexing functions, and for reverse-engineering unknown linear indexing functions under minimal assumptions.

In a case study, we use our algorithms to compute coloring schemes for recent Intel CPUs, and we show how to design indexing functions that maximize the number of supported trust domains.

CCS Concepts

• **Security and privacy → Formal security models**; **Side-channel analysis and countermeasures**; *Hardware reverse engineering*.

Keywords

memory coloring; side-channels; reverse engineering

ACM Reference Format:
Jana Hofmann, Cédric Fournet, Boris Köpf, and Stavros Volos. 2024. Gaussian Elimination of Side-Channels: Linear Algebra for Memory Coloring. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690263

*Work carried out while at Azure Research, Microsoft.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690263

1 Introduction

Cloud CPUs can host multiple tenants, each running within their own *trust domain* that must remain isolated from the others. However, trust domains often share memory and microarchitectural components, such as branch predictors, buffers, and caches. Their access patterns can then be observed from other trust domains, giving rise to side-channel attacks that break isolation [17, 18, 22, 23, 32, 40, 41, 44]. The usual way to prevent this problem is to partition microarchitectural components, so that each trust domain is assigned its own physically isolated part of each microarchitectural component. There are two types of microarchitectural components that need to be partitioned across trust domains: core-local and off-core components.

- *Core-local* components, such as CPU buffers, branch predictors, L1/L2 caches, and prefetchers are private to each core. These components can be easily partitioned using exclusive *core scheduling* [1, 13], which guarantees that a physical core never runs threads from different trust domains at the same time.
- *Off-core* components, such as the L3 cache, cache coherence directories, and DRAM, are shared across all cores, and hence they require other hardware or software isolation mechanisms. On the hardware side, the L3 cache can often be partitioned using, e.g., Intel's cache allocation technology or AMD's platform quality of service extensions, but there are no corresponding mechanisms for directories or DRAM banks. On the software side, the prevalent technique accounting for these components is resource partitioning via *memory coloring* [2, 5, 14, 16, 19, 25, 28, 31, 43].

Memory Coloring. Memory coloring works by assigning a *color* to each system physical address, and by using the memory manager (in the hypervisor or in the host operating system) to ensure that each color is assigned to at most one trust domain [5, 34].

A suitably chosen memory coloring scheme can achieve microarchitectural isolation by indirectly partitioning shared microarchitectural components. Components (e.g., L3 cache or DRAM) are collections of separate hardware *resources* (e.g., L3 cache sets or DRAM banks) such that access patterns to one of these resources cannot be observed from the others. A component's *indexing function* maps each system physical address to one of its resources. To achieve isolation, it thus suffices to choose a coloring scheme that assigns different colors to addresses mapped to different resources.

Memory coloring is attractive as it does not require changes to hardware and applications: Since coloring is based on system physical addresses, VMs and applications can still be assigned contiguous guest physical and virtual memory. Thus, coloring implemented by the virtualization layer is transparent to VMs and tenants.

Challenges. So far, memory coloring has mostly been used to partition microarchitectural components individually. To achieve full microarchitectural isolation, one must solve three challenges:

- *Simultaneous Isolation:* When trust domains share multiple microarchitectural components, the coloring scheme must partition all of them simultaneously.
- *Uniform Utilization:* A coloring scheme should not inadvertently partition private (core-local) resources, as this would lead to under-utilization and performance loss.
- *Reverse-Engineering:* Coloring depends on the knowledge of the components' indexing functions, which are often undocumented for commercial CPUs, so we need efficient ways to infer them.

Approach. In this paper, we provide theoretical foundations and practical algorithms for tackling these three challenges. We proceed in several steps.

- We provide a set-theoretic *model* for memory partitioning. This includes definitions of the isolation and utilization requirements. When considering multiple components at once, we show that these definitions compose to a well-defined constraint system.
- We design *efficient algorithms* for computing coloring schemes that satisfy all isolation and utilizations requirements. They build on algebraic characterizations of these requirements, and reduce to well-known algebraic tools, such as Gaussian elimination. Our algorithms apply to indexing functions that are (at least partially) *linear*, i.e., definable in terms of XORs of address bits, which is the prevalent case in today's CPUs. A key feature of our algorithms is that they operate on the *kernel* (also called nullspace) of the indexing functions, which means that we do not need to know the exact implementation of the function in hardware.
- We perform a *case study*, where we apply our theory and algorithms to analyze the colorability of caches, coherence directories, and DRAM on consumer-grade and server-class Intel CPUs. Our results show that coloring is often possible, but the number of available colors varies. Using the example of a recent Intel server-class CPU, Sapphire Rapids, we showcase how we would have designed its indexing functions with the explicit goal of maximizing the number of isolated trust domains.
- To make our approach applicable to black-box CPUs, we develop novel algorithms for *linearity testing* and *reverse-engineering* of linear indexing functions. We leverage that, for coloring purposes, it suffices to infer the kernel of the functions; this enables us to complete reverse-engineering using a single eviction set and without accessing the function's output value. Both algorithms are based on the simplest algebraic primitive: testing for congruence, i.e., whether two addresses are mapped to the same output value. Our simulation shows that it only takes a few hundred congruence checks to infer various existing indexing functions, which significantly improves efficiency over state-of-the-art reverse-engineering algorithms for linear functions.

Contributions. To summarize, our contributions are as follows.

- We present a theory of microarchitectural resource partitioning using memory coloring. This includes definitions for simultaneous isolation of shared components and uniform utilization of private components, together with algebraic characterizations for the case of linear indexing functions.
- We develop algorithms for computing coloring functions that guarantee the isolation and utilization requirements for linearly indexed components.
- We show how to design indexing functions with the explicit goal of maximizing the number of colors.
- We define novel algorithms for checking linearity of black-box indexing functions and for reverse-engineering linear functions.

2 Modelling Memory Coloring

We start by formally modelling memory coloring on a set-theoretic level as partition of the physical address space. Next, we define the requirements on such a partition, namely isolation with respect to shared components and uniform utilization of private components. We also show how memory coloring composes for hierarchically organized microarchitectures.

2.1 Partitions

A *partition* P of a set S is a collection of disjoint non-empty subsets of S whose union covers S, that is, $P = \{C_i \mid i \in I\}$ with $C_i \cap C_j = \emptyset$ for all $i \neq j$ and $\bigcup_{i \in I} C_i = S$. We refer to each subset C_i of the partition as a *class*. The set I is the set of class *indices*.

The different partitions of a set can be partially ordered as follows: P is finer than Q (and Q coarser than P), written $P \sqsubseteq Q$, if each class of P is a subset of a class of Q. Hence, the finest partition of S is $\{\{s\} \mid s \in S\}$, whereas the coarsest partition is $\{S\}$.

For two partitions P and Q of the same set, the *join* $P \sqcup Q$ is the finest partition that is coarser than both P and Q, and the *meet* $P \sqcap Q$ is the coarsest partition that is finer than both P and Q.

Example 1. *Consider the following two partitions of the set of binary words of length 3:*

$$\begin{aligned} P = \{&\{x \mid x_0 = 0, x_1 = 0\}, \{x \mid x_0 = 0, x_1 = 1\},\\ &\{x \mid x_0 = 1, x_1 = 0\}, \{x \mid x_0 = 1, x_1 = 1\}\}\\ Q = \{&\{x \mid x_0 = 0, x_2 = 0\}, \{x \mid x_0 = 0, x_2 = 1\},\\ &\{x \mid x_0 = 1, x_2 = 0\}, \{x \mid x_0 = 1, x_2 = 1\}\} \end{aligned}$$

We use x_i to refer to bit i (starting with bit x_0). The join of P and Q keeps their common split along the value of bit x_0:

$$P \sqcup Q = \{\{x \mid x_0 = 0\}, \{x \mid x_0 = 1\}\}$$

Their meet $P \sqcap Q$ is a set of 8 sets, one for each valuation of the bits x_0, x_1, x_2.

It is often convenient to define partitions as preimages of total functions $f : S \to I$, where each preimage of f forms a class, i.e., $x \in C_i$ if and only if $f(x) = i$. We denote the resulting partition by $P_f = \{f^{-1}(i) \mid i \in I\}$.

While a function uniquely determines a partition, the converse is not true, i.e. a partition can be defined by several different functions.

Example 2. *Functions $f_1(x) = x_0$ and $f_2(x) = \neg x_0$ both describe partition $P \sqcup Q$ from Example 1.*

2.2 Memory Coloring

Memory coloring describes the process of partitioning the memory into classes called *colors*. Each color is assigned to a *trust domain* (e.g., a tenant in the cloud). The goal of memory coloring is to prevent information flow between trust domains without impacting their individual performance. We model the memory M as the set $\mathbb{F}_2^n$ of n-bit physical addresses, where $n = 48$ is common.

Architectural Partitioning. When partitioning memory, we usually cannot partition the space arbitrarily but have to take certain architectural constraints into account. The memory is already partitioned into pages,[1] and pages should not be split between colors. For 4KiB-sized pages, partitioning into pages can be defined as

$$P_{4K} = \{\{x \mid x_{47} = 0, \dots, x_{12} = 0\}, \dots, \{x \mid x_{47} = 1, \dots, x_{12} = 1\}\}.$$

The partition describes that addresses agreeing on all but the 12 least significant bits $x_0, \dots, x_{11}$ belong to the same page. We use the refinement relation to express that a partition P should assign addresses of the same page to the same color: $P_{arch} \sqsubseteq P$.

Microarchitectural Partitioning. Another natural partitioning of the memory is described by the CPU's microarchitectural *components* (e.g., caches). Each component consists of a set of *resources* R (e.g., cache sets). Components can thus be represented with their indexing function $f : M \to R$ that maps each physical address to one of the component's resources.

Example 3. *The cache set indexing function of a typical L3 cache projects on address bits x_6 to x_{16}:*

$$f_{L3}(x) = x_6,\ x_7,\ x_8,\ x_9,\ x_{10},\ x_{11},\ x_{12},\ x_{13},\ x_{14},\ x_{15},\ x_{16}$$

Function f_{L3} is of type $\mathbb{F}_2^{48} \to \mathbb{F}_2^{11}$, i.e., it has 11 output bits. Each output $o \in \mathbb{F}_2^{11}$ refers to one cache set.

2.3 Isolation Through Coloring

We consider a threat model where different trust domains run on the same CPU, but *not* on the same core, which is the common case in cloud computing. This means that core-local microarchitectural components, such as the L1 and L2 caches, are *private* to each trust domain, while off-core microarchitectural components, such as the last-level cache (typically L3), are *shared* between the trust domains.

When two trust domains share a component, this can lead to information leakage, as visualized in Figure 1 (a). If they share a cache set, for example, one trust domain can observe which cache lines are evicted by the other domain's computation using a timing side-channel. Our goal is to prevent such side-channels with the help of memory coloring.

Definition 1. *A memory partition P is* isolating *for a shared component f iff f maps addresses of distinct colors to distinct resources:*

$$\forall C_1, C_2 \in P.\, \{f(x) \mid x \in C_1\} \cap \{f(x) \mid x \in C_2\} = \emptyset$$

Note that Definition 1 is is equivalent to $P_f \sqsubseteq P$.

Example 4. *Partition P_f defined by $f = x_{14},\ x_{15},\ x_{16}$ provides eight colors. It is isolating for f_{L3}, because addresses with different values of bits x_{14}, x_{15}, x_{16} are mapped to different cache sets.*

[1]Systems support multiple page sizes: 4KiB, 2MiB, and 1GiB

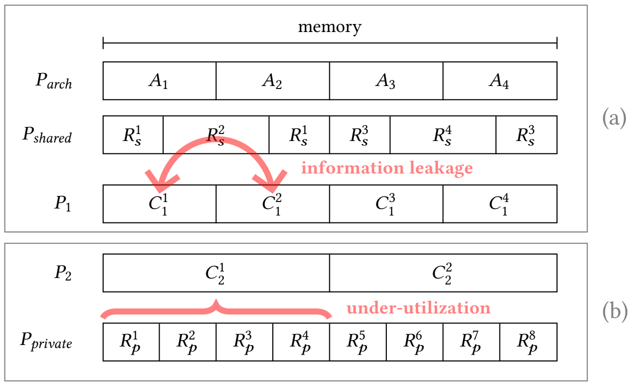

Figure 1: Visualization of information leakage (a) and under-utilization (b). It sketches the partitions induced by an architectural requirement (P_{4K}), as well as a shared and private microarchitectural component (P_{shared}, $P_{private}$). Memory partition P_1 defines a color for each P_{4K} class, which respects the architectural requirements of P_{4K}, but is *not* isolating for P_{shared} (information leakage between colors C_1^1 and C_1^2 through share resource R_s^2). Memory partition P_2 respects P_{4K} and is isolating for P_{shared}. P_2 under-utilizes the private component, however, as it only uses half its classes.

2.4 Uniform Utilization Despite Coloring

We call microarchitectural components that are not shared between trust domains private. Since trust domains run on separate cores, core-local buffers, predictors, and core-local (L1/L2) caches are by construction private, no matter how we partition memory.

While private components are not susceptible to side-channels, they are still affected by memory coloring: When we restrict a trust domain to a set of addresses, this may lead to under-utilization of the private resources, depending on their indexing function. This is visualized in Figure 1 (b).

Example 5. *Let $f = x_{14},\ x_{15},\ x_{16}$ be defined as in Example 4, and let f_{L2} be the cache set indexing function of the L2 cache:*

$$f_{L2}(x) = x_6,\ x_7,\ x_8,\ x_9,\ x_{10},\ x_{11},\ x_{12},\ x_{13},\ x_{14}$$

A trust domain that is assigned color $C = \{x \mid x_{14} = x_{15} = x_{16} = 0\}$ would only utilize half of the L2 cache, as none of C's addresses is mapped to the cache sets with $x_{14} = 1$.

Our second goal is to define partitions that do not lead to under-utilization.

Definition 2. *A memory partition P* uniformly utilizes *a component f, short $P \perp P_f$, if*

$$\forall C_1 \in P, C_2 \in P_f.\ \frac{|C_1 \cap C_2|}{|C_1|} = \frac{|C_2|}{|M|}$$

The definition ensures that the portion of a color mapped to resource r is equal to the portion of the entire memory mapped to r. Note that $\perp$ is commutative (for a proof see Appendix A).

Uniform utilization has a direct connection to probabilistic *independence*. Consider a probability distribution $p\colon \mathbb{F}_2^n \to [0, 1]$, and write $p(A)$ for the probability of subsets ("events") $A \subseteq \mathbb{F}_2^n$.

With this, Definition 2 is equivalent to $p(A \cap C) = p(A)p(C)$, or (equivalently) $p(A \mid C) = p(A)$. This means that restricting random memory accesses to a color C does not change their probability of being mapped to a specific cache set.

2.5 Memory Coloring as a Constraint System

To summarize, our goal is to partition memory into colors that are isolating for all shared components and uniformly utilizing for all private components. Formally, given architectural constraints P_{arch} (e.g., the page constraint P_{4K}), shared components $f_1, \ldots, f_n$, and private components $g_1, \ldots g_m$, we want to find the finest memory partition P such that

(1) $P_{arch} \sqsubseteq P$
(2) $P_{f_i} \sqsubseteq P$ for all f_i
(3) $P_{g_i} \perp P$ for all g_i

The trivial partition $P = \{M\}$ satisfies these requirements, but it supports only a single trust domain. To maximize the number of trust domains, we seek for the finest partition that satisfies the requirements. Note that this partition need not be unique: For $f(x) = x_1, x_2$ and $g(x) = x_1 \oplus x_2$ (where $\oplus$ is the XOR operator), both $h(x) = x_1$ and $h(x) = x_2$ satisfy $P_f \sqsubseteq P_h$ and $P_g \perp P_h$.

2.6 Coloring and Composition

Many microarchitectural components are organized hierarchically. For example, typical last-level caches are composed of *slices*, and each slice contains the same number of cache sets. Similarly, DRAM (dynamic random access memory) is organized into channels, dual inline memory modules (DIMMs) per channel, ranks per DIMM, and banks per rank.

We model this by defining local indexing functions $f_i : M \to R_i$ for each level, and we represent the entire component using the global indexing function $f : M \to R_1 \times \ldots \times R_m = (f_1, \ldots, f_m)$. The following proposition establishes how isolation and utilization properties extend from the local to the global indexing function.

Proposition 1. *Let $f = (f_1, \ldots, f_m)$ and P be a partition with $P_{f_i} \sqsubseteq P$ for some f_i. Then*

- $P_f \sqsubseteq P$
- $P_{f_j} \perp P_{f_i}$ *implies* $P_{f_j} \perp P$

The first bullet point shows that if P is isolating for one indexing function, it is also isolating for the entire function. This is because P_f intersects the preimages of all f_i, and therefore $P_f \sqsubseteq P_{f_i} \sqsubseteq P$.

The second bullet point shows that, whenever a partition P_i uniformly utilizes a component f_j, then any partition P that is coarser than P_i will also uniformly utilize f_j. This statement is targeted towards hierarchical structures, where indexing functions are often independent: For example, the cache slice selection is usually independent of the cache set selection within the slice, to achieve uniform utilization of all cache sets. Then, if we compute a coloring function based on cache set indexing, every trust domain will uniformly utilize all slices. The proof for Proposition 1 is given in the appendix (Appendix A), as are all other proofs of this paper.

Proposition 1 has two valuable implications. First, if we do not know the entire indexing function (e.g., we only know the cache set indexing function but not the slicing function), we can compute isolating and uniformly utilizing partitions based on only part of the function. Second, if the indexing function is only partly linear, it is sufficient to compute partitions based on the linear part.

3 Coloring with Linear Functions

We develop algorithms to automatically infer memory partitions that satisfy the requirements described in Section 2.5. Our algorithms apply to the important class of linear indexing functions. Most indexing functions are linear, e.g., cache set indexing functions, Intel's DRAM indexing functions (for channels, ranks, and banks) [23, 36], many cache slicing functions [6, 20], and the indexing function of cache coherence directories [40]. Furthermore, non-linear indexing functions are often at least partially linear. As Proposition 1 shows, if we infer coloring functions based on the linear part of a function, we obtain a partition that is isolating and uniformly utilizing also for the non-linear parts.

3.1 Linear Functions

The set $\mathbb{F}_2^n$ of physical addresses forms a vector space over the field $\mathbb{F}_2$, where addition corresponds to bitwise XOR ($\oplus$) and multiplication corresponds to bitwise AND (&). In particular, for $x, y \in \mathbb{F}_2^n$, $x + y = x - y$. In this paper, we use three ways to represent linear functions over $\mathbb{F}_2$: (1) as output bits defined as XOR of input bits, (2) as matrices, and (3) in terms of the kernel.

XOR Representation. In a linear function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$, each of the m output bits is defined as $f_i(x) = a_0^i \& x_0 \oplus \ldots \oplus a_{n-1}^i \& x_{n-1}$. Since all scalars $a_0^i, \ldots, a_{n-1}^i$ are either 0 or 1, f_i can be represented by a chain of XORs of some of x's address bits. Examples 3 and 5 in Section 2 use this representation.

Matrix Representation. The matrix representation of a function f is obtained by applying f to all standard unit vectors of the domain $\mathbb{F}_2^n$ and using the result as columns: $A_f = (f(e_0), \ldots, f(e_{n-1}))$.

Example 6. *Let function $f : \mathbb{F}_2^3 \to \mathbb{F}_2^2$ define two output bits as follows: $f(x) = x_0,\ x_1 \oplus x_2$. The matrix representation of f is*

$$A_f = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

The ith row of A_f indicates which bits of x are XORed in output bit i.

Kernel Representation. For the the third representation, recall that the *kernel* of f is defined as the set of addresses for which f maps to 0, i.e., $ker(f) = \{x \mid f(x) = 0\}$. The kernel characterizes the preimages of a linear function as follows: Two addresses x and y are in the same preimage of f iff their difference is in the kernel, i.e., $f(x) = f(y) \equiv f(x-y) = 0$. Thus, two functions with the same kernel define the same partition: $P_f = P_g$ iff $ker(f) = ker(g)$. Also note that $P_f \sqsubseteq P_h$ iff $ker(f) \subseteq ker(h)$. The relationship between P_f and $ker(f)$ can be concisely expressed by

$$P_f = \mathbb{F}_2^n / ker(f)\ ,$$

where the *quotient* V/W of two vector spaces $W \subseteq V$ is defined as the set of equivalence classes of the equivalence relation $x \equiv y$ iff $x - y \in W$. While the kernel uniquely defines P_f, recall from Example 2 that it does not uniquely define f.

3.2 Computing the Join of Partitions

As a first step, we describe how to compute a partition that respects our architectural requirements and guarantees isolation for shared components (see Requirements 1 and 2 in Section 2.5). Given architectural constraints P_{arch} and shared microarchitectural components $f_1, \ldots, f_n$, our goal is to compute a function h such that $P_{arch} \sqsubseteq P_h$ and $P_{f_i} \sqsubseteq P_h$ for all f_i. Since P_h should provide as many colors as possible, this is equivalent to computing $P_h = P_{arch} \sqcup P_{f_1} \sqcup \ldots \sqcup P_{f_n}$. We describe how to compute h such that $P_h = P_{f_1} \sqcup P_{f_2}$ for two arbitrary linear functions f_1 and f_2, which generalizes to computing the join for multiple components.

In general, computing the join of two partitions boils down to computing the transitive closure of overlapping pairs of classes and is not straightforward. Fortunately, for the case of linear functions, there is a direct connection between the join of partitions of two functions and the (vector-space) sum of their kernels:

Proposition 2. *Let f_1, f_2 be linear functions defined on $\mathbb{F}_2^n$. Then*

$$P_{f_1} \sqcup P_{f_2} = \mathbb{F}_2^n / _{ker(f_1)+ker(f_2)}$$

Proposition 2 directly translates into an algorithm to compute the join for linear functions: compute the kernel of functions f_1 and f_2, and find a function whose kernel is the sum of vector spaces $ker(f_1)$ and $ker(f_2)$. Both steps crucially rely on computing a basis of the kernel of a linear function. This can be done using standard algorithms built on Gaussian elimination [38]. We denote this algorithm as $\textsc{basis}(ker(\cdot))$. The pseudocode for computing $P_{f_1} \sqcup P_{f_2}$ is given in Algorithm 1.

Algorithm 1: Computing the join of partitions

```
Input  : Linear functions $f_1$ and $f_2$
Output : Linear function $h$ given as matrix $A_h$ such that
         $P_h = P_{f_1} \sqcup P_{f_2}$
$v_1, \ldots, v_j = \textsc{basis}(ker(f_1))$;
$v_{j+1}, \ldots, v_t = \textsc{basis}(ker(f_2))$;
$u_1, \ldots, u_l = \textsc{basis}(ker((v_1, \ldots, v_t)^T))$;
return $(u_1, \ldots, u_l)^T$;
```

The first two calls to $\textsc{basis}(ker(\cdot))$ obtain bases of $ker(f_1)$ and $ker(f_2)$, respectively. We combine these to a (not necessarily linearly independent) set of vectors $v_1, \ldots, v_t$ spanning $ker(f_1) + ker(f_2)$. The remaining step is to find a function h whose kernel is the space spanned by $v_1, \ldots, v_t$, i.e., we look for a matrix A_h such that $A_h v_i = 0$ for all $1 \leq i \leq t$. Put differently, if $B = (v_1, \ldots, v_t)$ is the matrix whose columns are $v_1, \ldots, v_t$, we need to solve $A_h B = 0$. This is equivalent to $B^T A_h^T = 0$. Thus, a basis of the kernel of B^T exactly describes the rows of A_h, which define the output bits of h.

Lemma 1. *Algorithm 1 computes A_h with $P_h = P_{f_1} \sqcup P_{f_2}$.*

We provide a formal correctness argument in Appendix B. For the rest of the paper, to simplify notation, we write $f_1 \sqcup f_2$ for the function h with $P_h = P_{f_1} \sqcup P_{f_2}$. For a complexity analysis, note that Gaussian elimination is in time $O(n^3)$. Therefore, Algorithm 1 is also in time $O(n^3)$.

Importantly, Algorithm 1 is applicable even if we only know the kernel of the indexing functions. In that case, we would simply skip the first two steps. We leverage this fact in Section 5, where we develop a linearity check and reverse-engineering algorithm based solely on the kernel representation of a function.

Example 7. *In the simple case that the functions' output bits do not use XORs, the join describes the intersection of the output bits. For example, for $f_1(x) = x_0, x_1, x_2$ and $f_2(x) = x_2, x_3, x_4$, we obtain $(f_1 \sqcup f_2)(x) = x_2$.*

3.3 Computing Uniformly Utilizing Partitions

The next step is to refine the coloring function h computed in the last subsection such that it uniformly utilizes all private components. Since P_h is the finest partition that respects architectural constraints and provides isolation for shared resources (Requirements 1 and 2 from Section 2.5), we can only achieve uniform utilization by merging colors, i.e., removing output bits from h. Our algorithm relies on the following proposition, which states that a partition defined by function h is uniformly utilizing component g if the row vectors of A_h and A_g are linearly independent of each other.

Proposition 3. *Let h, g be linear functions defined on $\mathbb{F}_2^n$. Then the following are equivalent*

(1) $P_h \perp P_g$

(2) $rank\left(\begin{bmatrix} A_g \\ A_h \end{bmatrix}\right) = rank\left(A_h\right) + rank\left(A_g\right)$

Proposition 3 gives rise to Algorithm 2 for computing uniformly utilizing partitions.

Algorithm 2: Computing uniformly utilizing partitions

```
Input  : $h$ as matrix $A_h$ with rows $v_1, \ldots, v_m$, and $g$ as a
         matrix $A_g$
Output : $h'$ as matrix $A_{h'}$ such that $P_h \sqsubseteq P_{h'}$ and $P_{h'} \perp P_g$
$A_{h'} = []$
for $i = 1$ to $m$ do
    if $rank\left(\begin{bmatrix} A_g \\ A_{h'} \\ v_i \end{bmatrix}\right) == rank\left(\begin{bmatrix} A_g \\ A_{h'} \end{bmatrix}\right) + 1$ then
        $A_{h'} = \begin{bmatrix} A_{h'} \\ v_i \end{bmatrix}$;
return $A_{h'}$;
```

The algorithm iteratively selects rows from A_h, in each step maintaining the invariant that $rank\left(\begin{bmatrix} A_g \\ A_{h'} \end{bmatrix}\right) = rank\left(A_g\right) + rank\left(A_{h'}\right)$. The rank equality check can be implemented using Gaussian elimination by testing if v_i is linearly independent of $\begin{bmatrix} A_g \\ A_{h'} \end{bmatrix}$.

With Proposition 3, correctness of the algorithm follows from a simple inductive argument and the fact that the rows of the initial matrix A_h are linearly independent (guaranteed by Algorithm 1), i.e., if A_h is a $m \times n$ matrix, then $rank\left(A_h\right) = m$.

Lemma 2. *Algorithm 2 computes $A_{h'}$ with $P_h \sqsubseteq P_{h'}$ and $P_{h'} \perp P_g$.*

Guaranteeing $P_h \sqsubseteq P_{h'}$ is important to ensure that $P_{h'}$ still satisfies the isolation requirement for shared components. We also note that the algorithm might produce different (valid) solutions depending on the order of the rows in A_h. As for the join, to simplify notation, we may write $h \perp g$ instead of $P_h \perp P_g$.

Algorithm 2 inherits an upper bound of $O(n^3)$ from Gaussian elimination for the first rank check. We can store the result in upper echelon form so that each of the $O(n)$ subsequent checks for independence of v_i requires only $O(n^2)$ steps, leading to an overall $O(n^3)$ bound.

Example 8. *In the simple case that the functions' output bits do not use XORs, Algorithm 2 achieves uniform utilization by eliminating bits that are shared between both functions. For $h(x) = x_0, x_1, x_2$ and $g(x) = x_0$, for example, we obtain $h'(x) = x_1, x_2$.*

3.4 Design Patterns for Indexing Functions

We conclude this section by exploring how to *design* indexing functions with coloring requirements in mind. Here, we give high-level patterns that maximize the number of colors obtained with Algorithms 1 and 2. In the next section, we apply these insights in a case study of modern Intel CPUs.

Assume we need to design three indexing functions f_1, f_2, and g, where f_1 and f_2 index shared microarchitectural components and g indexes a private component. Our goal is to design these functions such that (1) they adhere to given systems constraints, (2) they maximize the number of colors provided by $f_1 \sqcup f_2$, and (3) we guarantee that $(f_1 \sqcup f_2) \perp g$.

Goal 1: Adhering to Systems Constraints. The components have a fixed size, which means that f_1, f_2, and g must have fixed ranks. We focus on system constraints that require specific address bits to be used in the functions. A common pattern found across many CPUs is that the lower significant bits of physical addresses are used for indexing of caches, as this ensures that linear memory traversals are spread out over all cache sets. This can be achieved by either using the requested bits as singleton output bits (as in the definition of f_{L3} in Example 3) or by XORing them with other address bits.

Goal 2: Maximizing the Number of Colors. Our second goal is to maximize the number of colors provided by $f_1 \sqcup f_2$. Proposition 2 only gives an indirect handle on the size of the join because it requires reasoning about the kernels of f_1 and f_2. To reason more directly about the output bits of the functions, we connect the kernel of a function f with its *rowspace*. The rowspace $row(A_f)$ of a matrix A_f is the vector space spanned by its rows. From the definition of the rowspace it follows that $ker(f) = row(A_f)^\perp$, where $(\cdot)^\perp$ denotes the orthogonal complement with respect to the dot product.

It holds that $(U \cap V)^\perp = U^\perp + V^\perp$, which directly gives us the following proposition.

Proposition 4. $ker(f) + ker(g) = (row(A_f) \cap row(A_g))^\perp$

Propositions 2 and 4 together imply that in order to increase the number of classes of $P_{f_1} \sqcup P_{f_2}$, we need to increase the overlap of the rowspaces of A_{f_1} and A_{f_2}.

An immediate consequence is that a larger number of shared output bits increases the number of colors. Adhering to the systems constraints of using certain address bits, this can be achieved by XORing output bits of both functions f_1 and f_2.

Example 9. *Consider again functions $f_1(x) = x_0, x_1, x_2$ and $f_2 = x_2, x_3, x_4$ from Example 7. If we replace output bit x_0 in f_1 and output bit x_4 in f_2 by their sum $x_0 \oplus x_4$, we obtain functions*

$$f_1'(x) = x_0 \oplus x_4, \quad x_1, \quad x_2 \qquad f_2'(x) = x_2, \quad x_3, \quad x_0 \oplus x_4$$

with $(f_1' \sqcup f_2')(x) = x_0 \oplus x_4, x_2$. We thus increase the number of colors from two to four while still using all address bits of the systems constraints and without changing the rank of the functions.

Sharing output bits are a sufficient condition for increasing the intersections of the rowspaces, but not a necessary one, as demonstrated by the following example.

Example 10. *Consider the functions*

$$f_1(x) = x_1, \quad x_2 \oplus x_3, \quad x_3 \oplus x_4 \qquad f_2(x) = x_2 \oplus x_5, \quad x_3 \oplus x_5, \quad x_4 \oplus x_5$$

We have $(f_1 \sqcup f_2)(x) = x_2 \oplus x_3, x_2 \oplus x_4$ admitting four colors, even though f_1 and f_2 do not share any output bit. This is because $x_2 \oplus x_3$ and $x_2 \oplus x_4$ are both in the rowspaces of A_{f_1} and A_{f_2}.

Goal 3: Ensuring Uniform Utilization. Our third goal is to achieve uniform utilization of component g when coloring with $f_1 \sqcup f_2$. Proposition 3 gives us a direct handle on this problem: We need to increase the number of output bits in g and $f_1 \sqcup f_2$ that are independent of each other. This can be achieved, for example, by XORing address bits to g's output bits that are neither present in g nor in $f_1 \sqcup f_2$. To achieve the same effect for $f_1 \sqcup f_2$, we need to add address bits to f_1 and f_2 such that they carry over to $f_1 \sqcup f_2$.

Example 11. *Consider f_1, f_2 and g defined as follows:*

$$f_1(x) = x_0 \oplus x_1 \qquad f_2(x) = x_0 \oplus x_1 \oplus x_2, \quad x_2$$
$$(f_1 \sqcup f_2)(x) = x_0 \oplus x_1 \qquad g(x) = x_0, x_1$$

$f_1 \sqcup f_2$ is not uniformly utilizing g, as $x_0 \oplus x_1$ is linearly dependent on x_0 and x_1. If we add bit x_3 to g, we obtain $g'(x) = x_0 \oplus x_3, x_1$ and $(f_1 \sqcup f_2) \perp g'$. Alternatively, if we added x_3 to the output bits of f_1 and f_2 that generate $x_0 \oplus x_1$, we would obtain the same effect: $f_1'(x) = x_0 \oplus x_1 \oplus x_3$ and $f_2'(x) = x_0 \oplus x_1 \oplus x_2, x_2 \oplus x_3$ results in $(f_1' \sqcup f_2')(x) = x_0 \oplus x_1 \oplus x_3$, which is independent of x_0 and x_1.

4 Case Study: Coloring Intel CPUs

In this section, we apply the theory, algorithms, and design patterns from Section 3 to infer memory coloring schemes for three generations of Intel CPUs.

4.1 Background: Microarchitecture of Intel CPUs

Intel CPUs feature multiple cores with private L1 and L2 caches. All cores share a sliced L3 cache, where the number of cache slices typically matches the number of cores. DRAM is shared among all cores; it is organized hierarchically into channels, Dual Inline Memory Modules (DIMMs), ranks, and banks. Each bank comprises multiple rows and includes a row buffer, which holds the row that has been accessed most recently within a bank.

Different CPUs use different techniques for keeping private caches coherent. The traditional technique is to use an *inclusive* L3 cache. Inclusivity means that every line present in one of the L2 caches is also present in the L3 cache, which allows the L3 to track the global memory state. With the trend towards non-inclusive L3

$$\begin{aligned}
f^1_{slice}(x) &= x_6 \oplus x_{10} \oplus x_{12} \oplus x_{14} \oplus x_{16} \oplus x_{17} \oplus x_{18} \oplus x_{20} \oplus x_{22} \oplus x_{24} \oplus x_{25} \oplus x_{26} \oplus x_{27} \oplus x_{28} \oplus x_{30} \oplus x_{32} \oplus x_{33} \oplus x_{35} \oplus x_{36} \\
f^2_{slice}(x) &= x_7 \oplus x_{11} \oplus x_{13} \oplus x_{15} \oplus x_{17} \oplus x_{19} \oplus x_{20} \oplus x_{21} \oplus x_{22} \oplus x_{23} \oplus x_{24} \oplus x_{26} \oplus x_{28} \oplus x_{29} \oplus x_{31} \oplus x_{33} \oplus x_{34} \oplus x_{35} \oplus x_{37} \\
f^3_{slice}(x) &= x_{10} \oplus x_{11} \oplus x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18} \oplus x_{19} \oplus x_{20} \oplus x_{21} \oplus x_{22} \oplus x_{27} \oplus x_{28} \oplus x_{30} \oplus x_{31} \oplus x_{32} \oplus x_{33} \\
f^4_{slice}(x) &= x_9 \oplus x_{12} \oplus x_{16} \oplus x_{17} \oplus x_{19} \oplus x_{21} \oplus x_{22} \oplus x_{23} \oplus x_{25} \oplus x_{26} \oplus x_{27} \oplus x_{29} \oplus x_{31} \oplus x_{32} \oplus x_{33} \oplus x_{34} \oplus x_{35}
\end{aligned}$$

Figure 2: Cache slice function of Intel Core i9-9980HK (Coffee Lake) [6]. The four bits can be used to address 16 slices.

(1) Skylake i5-6200U, 4GiB DRAM: $f_{bank}(x) = x_6 \oplus x_{13}, \quad x_{14} \oplus x_{16}, \quad x_{15} \oplus x_{17}$

(2) Coffee Lake i5-9400, 8GiB DRAM: $f_{bank}(x) = x_6 \oplus x_{13}, \quad x_{14} \oplus x_{17}, \quad x_{15} \oplus x_{18}, \quad x_{16} \oplus x_{19}$

(3) Coffee Lake i5-9400, 16GiB DRAM: $f_{bank}(x) = x_7 \oplus x_{14}, \quad x_{15} \oplus x_{19}, \quad x_{16} \oplus x_{20}, \quad x_{17} \oplus x_{21}, \quad x_{18} \oplus x_{22}$

Figure 3: DRAM indexing functions in various Intel machines [4, 36].

caches, the emerging technique is to use a *cache coherence directory*, which is an inclusive cache that only keeps track of the tags (not the data) of the lines cached across all L2 caches [30]. Directories are sliced and indexed similarly to the L3 cache [40].

In the following, we use memory coloring (at 4K-page granularity) to prevent the sharing of cache sets and DRAM banks. We also rely on cache allocation technology (CAT) for partitioning the L3 cache. Compared to memory-based coloring, CAT is a hardware-based technique that partitions the cache along its *ways*, not its sets. With CAT enabled, a thread can only cause evictions in the ways assigned to it, but it can still read from the entire cache. Thus, CAT is not isolating for shared memory.

4.2 Coloring the Past: Coffee Lake & Skylake(-X)

We first investigate how shared resources can be partitioned on Coffee Lake and Skylake machines with inclusive L3 caches. We consider two scenarios: one where we rely on coloring to partition the shared L3 and another where we rely on hardware support, i.e., CAT. We then investigate resource partitioning on Skylake-X, which features a non-inclusive L3 cache.

4.2.1 Partitioning the L3 Cache using Coloring. First, we aim to simultaneously color the L3 cache and DRAM only using memory coloring. The full cache set selection function $f_{L3full} = (f_{slice}, f_{L3})$ is composed of the slice selection function f_{slice} (depicted in Figure 2) and the set indexing function f_{L3} (defined in Example 3). For DRAM, we consider the functions displayed in Figure 3. Assuming f_{bank} to be function (1) in Figure 3, the following function jointly colors the two resources at 4K-page granularity.

$$h(x) = (f_{4K} \sqcup f_{L3full} \sqcup f_{bank})(x) = x_{14} \oplus x_{16}$$

This function also satisfies $h \perp f_{L2}$ for the set indexing function function $f_{L2}(x) = x_6, \ldots x_{14}$ of a typical 8-way 256KiB L2 cache. This shows that joint coloring of L3 and DRAM is possible – albeit with only one coloring bit, i.e., two trust domains. The other DRAM bank functions displayed in Figure 3 are not suited for coloring.

4.2.2 Partitioning the L3 Cache Using CAT. As a way to obtain more colors, we now partition the L3 ways using CAT, which effectively turns it into a private resource. Thus, when partitioning DRAM using coloring,[2] we need to make sure that each color uniformly utilizes all L3 cache sets. That is, we now seek a coloring function h such that $P_{f_{bank}} \sqsubseteq P_h$, $h \perp f_{L3}$, and $h \perp f_{L2}$. From Proposition 1, we know that it is enough to guarantee $h \perp f_{L3}$, as this implies $h \perp f_{L3full}$. For bank indexing function (3) in Figure 3, we obtain

$$h(x) = x_{15} \oplus x_{19}, \quad x_{16} \oplus x_{20}, \quad x_{17} \oplus x_{21}$$

On Coffee Lake i5-9400, partitioning the L3 cache with CAT and DRAM using coloring thus supports eight trust domains.

4.2.3 Partitioning with a Non-Inclusive L3. Skylake-X is one of the first Intel microarchitectures with a non-inclusive L3 cache and a dedicated cache coherence directory. Reverse-engineering efforts [40] revealed that the directory relies on the same slicing function as the L3 cache. It also has the same number of sets, indicating that the set indexing function is the same as well.

This means that partitioning via CAT is ineffective, as it would not affect the directory, requiring that the L3 cache and directory are partitioned using coloring. This may result in fewer colors, if at all, as in Section 4.2.1. While this is bad news for coloring on past microarchitectures, Section 4.3 shows that current and future microarchitectures can be more amenable to coloring.

4.3 Coloring the Future: Designing Indexing Functions for Sapphire Rapids

We now apply the insights described in Section 3.4 to design hypothetical indexing functions for a 60-core Xeon Platinum 8490H, Intel's most recent server-class CPU. That is, we seek indexing functions that maximize the number of colors while satisfying the constraints laid out in Section 2.5.

We first define an indexing function template based on public information about the CPU's microarchitecture (see [10]) and on hypotheses informed by common design principles. We then complete the template with coloring and utilization constraints in mind.

4.3.1 Indexing Function Templates. The CPU features 4 chiplets (aka *quadrants*), each with a 16-node mesh interconnect.

[2]Coloring also avoids sharing memory (for which CAT is not isolating) across trust domains as each color and its memory pages are private to one trust domain.

DRAM Banks. One node per quadrant hosts the memory controller with two channels. We assume that each memory channel hosts 2 DIMMs with 2 ranks each. In DDR5, a rank consists of up to 32 banks. To address 8 Channels × 2 DIMMs × 2 ranks × 32 banks, i.e., 1024 DRAM banks in total, we require 10 bits. We assume that the indexing function is composed of the following components.

- The CPU features a *sub-NUMA clustering mode*, where the memory controller in each quadrant is responsible for a contiguous chunk of physical memory. Assuming a total memory size of 512GiB (i.e., chunk sizes of 128GiB), we add output bits x_{37}, x_{38} to the function, which select one of the 4 quadrants.
- Each chunk is partitioned into 64B-sized smaller chunks, which are interleaved between the two channels. We therefore use x_6 to index the channel.
- The remaining 7 bits are typically chosen above the 4K-page offset, i.e., from bit x_{12} and above.

This yields the following template for DRAM bank indexing:

$$f'_{bank}(x) = x_6,\ x_{37},\ x_{38},\ \text{and 7 bits } x_i \text{ with } i \geq 12$$

Caches and Directory. The remaining 15 nodes per quadrant each host one core and one slice of the non-inclusive L3 cache and the cache coherence directory. The slices are determined by the two quadrant bits and (since we have 15 nodes) a non-linear function f'_{slice}. We hypothesize that the L2 caches, L3 cache slices, and directory slices each feature 2048 sets and thus require 11 bits for indexing. This matches the cache size specifications, under the assumption that L2 has associativity 16 and L3 has associativity 15, as often seen in Intel microarchitectures. As is common in most caches, we use the least significant bits for such indexing. This yields the following templates for the cache indexing functions:

$$f'_{L2}(x) = x_6, \dots, x_{16}$$
$$f'_{L3}(x) = x_6, \dots, x_{16}, x_{37}, x_{38},\ \text{and a non-linear component } f'_{slice}$$

We further assume $f'_{dir} = f'_{L3}$, which matches prior Intel microarchitectures [40].

4.3.2 *Optimizing for Coloring.* We complete the templates with two coloring constraints in mind.

- To ensure a large number of colors, we maximize the overlap between the bits in f_{bank} and $f_{L3} = f_{dir}$.
- To ensure uniform utilization of the private L2, we minimize the overlap of $f_{bank} \sqcup f_{L3}$ with f_{L2}.

Fortunately, we can achieve both goals simultaneously. We add some of the free bits $x_{17}, \dots, x_{36}$ to the indexing function of the L3 cache and the directory to guarantee uniform utilization of L2 cache. We implicitly also consider the 4K-page architectural constraint, i.e., we do not modify those output bits that use address bits below 12. We mark the newly introduced bits with the lighter color.

$$f_{L3}(x) = f_{dir}(x) = x_6,\ \dots,\ x_{11},\quad x_{37},\quad x_{38},$$
$$x_{12} \oplus x_{17},\quad x_{13} \oplus x_{18},\ \dots,\ x_{16} \oplus x_{21}$$

We do not modify the non-linear component. For the bank indexing function, we recycle output bits of f_{L3} to instantiate five of the seven free bits in the bank indexing function.

$$f_{bank}(x) = x_6,\quad x_{37},\quad x_{38},$$
$$x_{12} \oplus x_{17},\quad x_{13} \oplus x_{18},\ \dots,\ x_{16} \oplus x_{21},\quad x_{22},\quad x_{23}$$

Overall, this yields a coloring function $h = f_{L3} \sqcup f_{bank} \sqcup f_{4K}$ defined by

$$h(x) = x_{12} \oplus x_{17},\quad x_{13} \oplus x_{18},\ \dots,\ x_{16} \oplus x_{21},\quad x_{37},\quad x_{38}$$

with $h \perp f_{L2}$ and h supporting $2^7 = 128$ colors. That means that we can support at least as many colors as there are cores.

4.3.3 *Maxing out Colors.* The number of banks (1024 in our setting) puts a natural limit on the number of colors. Achieving this limit is within reach. For example, when the number of cores per quadrant is divisible by 2 (e.g., 12, 14, or 16) one could index the slice using one to four additional linear bits, which are amenable to coloring if page offset bits are excluded from the slicing function. Alternatively, one could double the number of L3 and directory sets (e.g., by reducing their associativity), which would yield an additional linear bit that is independent of the L2 indexing function. With such changes, the limit of 1024 colors is readily achievable.

5 Inferring Linear Indexing Functions

We now switch perspective from hardware vendors, who design indexing functions, to cloud providers, who wish to provide isolation across trust domains using a memory coloring scheme. In the absence of information about indexing functions (which are usually proprietary and not shared by the hardware vendors), cloud providers need to first reverse-engineer the functions before inferring the memory coloring scheme.

We develop a reverse-engineering algorithm (Section 5.4) for linear functions that capitalizes on the observation from Section 3.2 that knowing the *kernel* is sufficient for computing a coloring scheme. Like this, it avoids computing the complete truth table of the function, which significantly improves efficiency over state-of-the-art reverse-engineering algorithms.

Before reverse-engineering, we test the unknown indexing function for linearity, for which we design a novel algorithm (Section 5.3). Both algorithms are based on the simplest algebraic primitive: a test for *congruence* (i.e., if $f(x) = f(y)$). Like this, we forgo common assumptions in state-of-the-art reverse-engineering approaches, including the need to know the rank of the function or to observe its output values.

5.1 State of the Art

We first recapitulate existing approaches for reverse-engineering linear indexing functions. They can be categorized into two classes.

5.1.1 *Linear Algebraic Approaches.* Most algebraic approaches require access to the output values $f(x)$ for chosen addresses x [6, 8, 11, 20, 40], and they assume f has a known rank. Using the access to output values, the subset of address bits occurring in an output bit can by identified by simply checking if flipping x_j changes the value of f_i [6, 20]. If yes, then $f_i = x_j \oplus f'_i$.

The output value for an address is most reliably obtained using performance counters [6, 8, 20]. If performance counters are not available, timing side-channels are used to group a large number of addresses into congruence classes. Each class is associated with one output value, and it is assumed that all possible values are captured [6, 11, 40]. This approach carries the risk of "guessing" the wrong output value, which could result in inferring a non-linear function even if the actual function is linear; we further discuss this in Section 5.3. This approach also requires a large number of timing measurements to find witnesses for all output values.

In contrast to past algebraic work, we use a kernel-centered approach that in particular does not require a group of addresses for every possible output value. A concurrent reverse-engineering approach [34] follows a similar idea. It infers the indexing function by observing how it acts on a basis of the address space. This requires generating numerous eviction sets, which can be computationally expensive [33]. The algorithm presented in this paper requires only a single eviction set for reverse-engineering the kernel.

5.1.2 *Brute-Force Approaches.*

Non-algebraic approaches also rely on grouping a large number of addresses into congruence classes. The function is then either inferred manually [9, 26, 42] or by searching the function space for a solution respecting these classes [7, 23, 36]. To reduce the search space, these approaches often rely on domain knowledge, e.g., knowledge about used address bits [9, 23], or assuming a specific function shape [36, 39].

5.2 Congruence Tests and Eviction Sets

Our algorithms are centered around congruence tests. Two addresses x, y are *congruent* with respect to f if $f(x) = f(y)$. For linear functions, congruence tests can be used to check if an address is in the kernel of f. To do so, fix an address x and compute an *eviction set* for x. An eviction set for a microarchitectural component (e.g., L3 cache, DRAM) is a sufficiently large set of addresses that map to the same resource (e.g., L3 cache set, DRAM bank) [33]. Accessing all addresses in the set will evict all previous data (e.g., cache lines, DRAM row) from the resource. Address y is in the kernel of f iff $x + y$ is evicted by x's eviction set, i.e., if $f(x) = f(x + y)$. Using this approach, we can test any address for kernel membership using a single eviction set.

The complexity of generating an eviction set and testing for eviction greatly varies across components and is impacted by their microarchitecture, e.g., their replacement policy.

Caches. The size of an eviction set is the number of cache ways, e.g., 16 for a typical L3 cache. For inclusive caches with a least-recently-used replacement policy, eviction set generation is known to be quite efficient with a complexity of $O(ws \log s)$, where w is the number of ways and s is the number of cache sets [15]. To test if y is evicted by an eviction set, one just accesses all addresses in the set and then measures the time of accessing y. If the access time is above a certain threshold, y was evicted.

For non-inclusive caches, generating an eviction set and checking for congruence is still feasible, but considerably harder. This is because the cache line is allocated only after being evicted from lower-level caches; and the replacement policy may depend on the coherence state of the cache line [40].

DRAM Banks. The DRAM row buffer holds the DRAM row that has been most recently accessed within a DRAM bank, i.e., an eviction set consists of a single address. Congruence checking is based on distinguishing row hits and row misses [23]. To check congruence of x and y, first find an address z that maps to the same bank as x but a different row. This is observable because of the higher access latencies needed to close and open the rows. Then, repeatedly access x and y, and z and y. If x and y map to the same bank, at least one of the pairs will cause high latencies.

5.3 Testing for Linearity

In this section, we investigate the linearity of an unknown indexing function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$, where the rank m is unknown as well.

We say f is *congruent to linear* if there *is* a linear function g that generates the same preimages as f, i.e. $P_f = P_g$. For example, the function $f : \mathbb{F}_2 \to \mathbb{F}_2$ with $f(x) = \neg x$ is not linear (because $f(0) \neq 0$) – but it is congruent to linear because $g(x) = x$ is linear and has the same preimages as f. This generalization is natural in our context because colors could be computed based on g.

We show that checking if an indexing function is congruent to linear reduces to performing congruence tests. Using this observation, we deduce a general algorithm for testing congruence-to-linearity.

Lemma 3. *A function f is congruent to linear iff*

$$\forall x, y, e_i.\, f(x) = f(y) \Rightarrow f(x + e_i) = f(y + e_i),$$

where e_i is a standard unit vector.

Proof Sketch. Intuitively, this proposition holds because any preimage class C can be represented by a vector v, i.e., $C = [v] = \{v + a \mid a \in ker(f)\}$. Thus, if $w = v + e_i$ for some standard unit vector e_i, then adding e_i to any element of C results in a vector from $[w]$. Appendix C details the full proof. □

To avoid cumbersome terminology and whenever there is no room for confusion, we use *congruence to linear* and *linear* interchangeably in the rest of the section.

Algorithm for Testing Linearity. Lemma 3 directly translates to an algorithm that tests if an indexing function is linear. First, create an arbitrary eviction set E for an arbitrary address x that is as close to minimal as possible. Then, for some $i \in [0..1-n]$, flip bit i in all addresses contained in E, resulting in set E'. Test if E' is an eviction set for $x + e_i$. If not, then Lemma 3 is not satisfied and the function is definitely not linear. Repeat the process for all indices i and possibly more eviction sets E.

Complexity. In the worst case, f behaves mostly linearly, except that two addresses are swapped between two preimage classes, making the function non-linear. Finding these classes and the addresses within may require eviction sets that cover all addresses in $\mathbb{F}_2^n$, resulting in a worst-case complexity that is exponential in n, i.e., comparable to building the complete truth table.

In Section 5.6, we evaluate this algorithm in a simulation on common functions and find that only a handful of eviction sets are needed for this approach to determine that a function is linear. This shows that in practice, the check for linearity does not incur the worst-case complexity.

5.4 Reverse-Engineering the Kernel

In the following, we assume that the check for linearity was successful. We now aim to reverse-engineer the kernel of f. The naive algorithm is to test all 2^n addresses for kernel membership using congruence tests, which is prohibitive for an address length of 48

Algorithm 3: Reverse-engineering the kernel of a linear indexing function f.

```
Input  : Minimal eviction set $E$ for some address $x$
Output : Linear function $h$ such that $P_h = P_f$
$K$ = BASIS($\{a + b \mid a, b \in E\}$);
while True do
    $h$ = toFunction($K$);
    let $U$ s.t. $h(U) = im(h)$ and $U$ is minimal;
    if $\exists u \in U$ s.t. $x + u$ is evicted by $E$ then
        $K = K \cup \{u\}$;
    else
        return $h$;
```

bits. To reduce the search space, we observe that if we have found a strict subspace $K \subset ker(f)$ of the real kernel, then any space U that extends K to $\mathbb{F}_2^n$ must contain a kernel vector not yet in K. If h is a function with $ker(h) = K$, then we can choose U as a set of vectors that h maps to a basis of its image.

Proposition 5. *Let $h : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a linear function and $U \subseteq \mathbb{F}_2^n$ be a minimal subspace with $h(U) = im(h)$. Then $ker(h) \oplus U = \mathbb{F}_2^n$.*

Above, $\oplus$ is the direct sum of two vector spaces, $im(h)$ is the image of h, and $h(U)$ is the set $\{h(u) \mid x \in U\}$. Since $im(h)$ is spanned by the column vectors of A_h, U can be chosen as a subset of the standard unit vectors. We have chosen U such that it's dimension $dim(U)$ equals $n - dim(K)$. Thus, every kernel vector we find halves the remaining search space. Algorithm 3 implements this idea.

The algorithm takes as input an eviction set E for an arbitrary address x. The set E should be minimal, i.e., it should only evict addresses from one cache set. We first construct an initial set of kernel vectors obtained by collecting all sums of addresses in the eviction set (if $a, b \in E$, then $f(a) = f(b)$ and thus $f(a + b) = 0$). This set spans a vector space that is a subspace of $ker(f)$. We store in K a basis for this space. The next step generates a function h with $ker(h) = K$. toFunction(K) can be implemented as in Algorithm 1 as $\text{BASIS}(ker(K^T))^T$. Next, we compute a vector space U according to Proposition 5. We traverse U to check if it contains an address u that is in the kernel of f. If yes, we add it to K . As K spans a vector space that is disjoint from U (by definition of the direct sum), u is linearly independent of the vectors in K. If there is no such u, $ker(f) = K$, and we can terminate. To ensure that we never check the same vector twice for kernel membership, our implementation of this algorithm caches all vectors u that fail the eviction test for the next iteration of the loop (not displayed in Algorithm 3).

Complexity. Since the cost of implementing a congruence check depends on the target resource, we describe the algorithm's complexity with respect to the number of congruence tests needed. For relevant ranges of the dimension n, other operations like checking linear dependency are negligible in comparison.

In the worst case, if the eviction set E contains only one vector and $dim(ker(f)) = 1$, we have to search the entire address space for that single non-zero kernel vector. Then, the complexity is exponential in n. In Section 5.6, we evaluate this algorithm in a simulation for common cache and DRAM indexing functions, and find that it infers the correct kernel with at most a few hundred congruence tests, even when starting with a small eviction set. This is because in practice, a random eviction set of size k is likely to contain close to k independent initial kernel vectors (with $k < n - m$). The first space U is then of size 2^{n-k}, the next one of size 2^{n-k-1}, and so on, until we reach a space of size 2^m, and we confirm that we found the entire kernel. Usually, m is significantly smaller than n, making the kernel quite large. We therefore quickly find the next independent vector when searching the subspaces U.

This algorithm can also be applied if the function is partially linear and we know the non-linear part. However, it requires more than one eviction set. We describe the details in Appendix C.2.

5.5 Refining the Search Strategy

Algorithm 3 does not define how to search the subspace U for the next kernel basis vector. In practice, it makes sense to first check the standard unit vectors for kernel membership. Using this strategy, we first find all address bits that are not used by the function: If bit i is not used by f, then the ith standard unit vector is in the kernel of f. This approach quickly reduces the domain of the space we still have to search. For example, if f only uses 20 of the 48 address bits, then 48 congruence checks will give us the first 28 basis vectors of the kernel. As a result, the remaining space to search now has a dimension of only 20.

Next, we search for all kernel vectors that have exactly two bits set to 1. That way, we find all functions that use every address bit at most once. We call these functions simple.

Definition 3. *A linear function is f* simple *if its matrix representation A_f has at most one 1 in each column.*

Examples for simple functions are typical cache set indexing functions that use only singleton bits and Intel's DRAM bank functions depicted in Figure 3. Our refined search strategy exploits the fact that the kernel of simple functions is spanned by vectors with at most two bits set to 1.

Proposition 6. *If f is simple, then $ker(f)$ is defined by a basis B such that in each $v \in B$, at most two bits are set to 1.*

5.6 Experiments

We evaluate the linearity testing and the reverse-engineering algorithm in a simulation environment implemented in Python. This enables us to evaluate the algorithms not only with various previously reported functions but also with randomly generated ones, which better highlights the boundaries of the algorithms.

5.6.1 Linearity Testing. We implement the bit-flip algorithm described in Section 5.3. By construction, the algorithm does not produce false negatives (linear functions labeled as non-linear). We therefore investigate the false-positive rate for various non-linear functions with domain $\mathbb{F}_2^{48}$. The algorithm is parametric in the number of eviction sets and in the size of each of the sets. We construct eviction sets as random collections of addresses that evaluate to the same output. For each eviction set, we test bit flips for all 48 bits.

Random Functions. We first evaluate the algorithm on randomly generated non-linear functions. The functions are generated with

Rank	# Eviction Sets	Error Rate					
		Fully Non-Linear			Almost Linear		
		s = 5	s = 10	s = 20	s = 5	s = 10	s = 20
2	1	1%	0.6%	0.3%	5%	3.5%	1.9%
	5	0%	0%	0%	0.2%	0%	0%
	10	0%	0%	0%	0.1%	0%	0%
4	1	0%	0%	0%	3.6%	1.4%	1.3%
	5	0%	0%	0%	0.2%	0%	0.1%
	10	0%	0%	0%	0%	0%	0%
8	1	0%	0%	0%	25.5%	18.9%	13.4%
	5	0%	0%	0%	2.1%	0.4%	0.6%
	10	0%	0%	0%	0%	0%	0%
16	1	0%	0%	0%	40.0%	31.6%	27.5%
	5	0%	0%	0%	5.1%	3.2%	2.7%
	10	0%	0%	0%	1.4%	1.2%	0.5%

Table 1: Simulation of linearity check on random non-linear functions. s indicates the size of the eviction set.

Type	Function	Rank	# Evict. Sets	Error Rate	
				s = 5	s = 10
Cache Slice	Intel Skylake, 20 cores [6, 21]	5	1	0.1%	0%
			5	0%	0%
	Intel Skylake, 24 cores [6, 21]	5	1	3.5%	0.1%
			5	0%	0%
DRAM	AMD Zen+, 16 GiB [12]	5	1	34.6%	24.3%
			5	0.5%	0.1%
	AMD Zen2, 16 GiB [12]	5	1	25.9%	12.3%
			5	0%	0%
	AMD Zen4, 32 GiB [12]	6	1	44.5%	41.5%
			5	1.1%	1.0%

Table 2: Simulation of linearity testing on previously reported non-linear functions. s indicates the eviction set size.

an AST depth of at most 5 using ¬, ∧, ∨, ⊕ as logical operators. Our results are reported in Table 1. In a first set of experiments (Fully Non-Linear), we generate random functions of varying rank and check their linearity using an increasing number and size of eviction sets. In the second set of experiments (Almost Linear), we increase the difficulty by using functions in which only one of the output bits is non-linear, the others are linear. We test each configuration on 1000 randomly generated functions.

We see that our check reliably detects fully non-linear functions; the error rate is less than 1% as long as the eviction set has 10 elements. For almost-linear functions, detecting linearity gets naturally harder with an increasing rank and we see that a single eviction set is clearly not sufficient. If we generate at least 5 eviction sets with 10 or more addresses, the error rate is less than 5%. For 10 eviction sets of size 20 each, we obtain an error rate of only 0.5% even for functions with rank 16.

Existing Functions. We also evaluate the check on previously reverse-engineered non-linear indexing functions; results are given in Table 2. The first two non-linear functions form the base sequence of slicing functions in Intel Skylake processors [21]. We use a recently reverse-engineered simplified formulation of the functions [6]. Both consist of 2 linear and 3 non-linear output bits. The last two functions are DRAM bank indexing functions used in AMD Zen+, Zen2, and Zen4 machines [12]. These functions first subtract a constant offset from the address and then behave like

Type	Rank	Eviction Set Size	# Congruence Checks		
			avg	min	max
Fully Random	2	1	104	50	171
		10	83	48	115
	4	1	255	123	447
		10	214	91	383
	8	1	4756	984	9552
		10	3148	1080	7853
	12	1	66k	20k	139k
		10	55k	16k	138k
	16	1	876k	260k	2359k
		10	680k	122k	1718k
Simple Random	12	1	4274	4196	4383
		10	4221	4162	4288
	16	1	66k	66k	66k
		10	66k	66k	66k

Table 3: Simulation of reverse-engineering random functions.

a linear function. Computing constant offsets can be formulated as non-linear Boolean functions (see Appendix C.3). All functions under test are detected as non-linear with an error rate ≤ 1% when using at least 5 eviction sets.

Discussion. Our evaluation shows that non-linearity can be very reliably detected with the help of 5 to 10 eviction sets. Constructing 10 eviction sets is easily feasible on all currently used microarchitectural components. This is a sterile simulation environment, of course, and there will be noise on a real system. Our results make us confident, though, that a sufficient number of repetitions will yield similarly good results even with noise present.

5.6.2 *Reverse-Engineering.* To evaluate the reverse-engineering algorithm, we count the number of congruence checks needed to determine the kernel. As before, we test the approach on randomly generated functions (Table 3) as well as previously reverse-engineered indexing functions (Table 4).

Random Functions. We generate linear indexing functions of varying rank where each output bit XORs a random subset of address bits. We consider initial eviction set sizes of 1 (corresponding to the eviction set size of DRAM row buffers) and 10 (an under-approximation of typical cache eviction set sizes). We test each configuration on 100 different functions and repeat each experiment 10 times. As expected, the number of required congruence tests grows exponentially with the rank of the function. A larger eviction set size decreases the number of needed tests, but not hugely. The experiments confirm that the average complexity of the approach is indeed significantly below the worst-case complexity of 2^n (remember that $n = 48$). To evaluate the impact of our search strategy, we also generate random simple functions according to Definition 3. We see that simple functions are found with significantly fewer congruence checks.

Existing Functions. Finally, we evaluate our approach on a range of previously reverse-engineered linear indexing functions. To simulate actual eviction sets, we use an eviction set size 1 for DRAM functions and 16 for L3 cache slicing and cache set functions. For Intel machines, DRAM indexing functions are typically linear with a small number of XORs (functions (1) - (3) are the ones provided

Type	CPU	Rank	Evict. Set Size	Avg #Congr. Checks
DRAM	(1) Intel Coffee Lake, 16 GiB [4, 36]	5	1	89
	(2) Intel Coffee Lake, 8 GiB [4, 36]	4	1	69
	(3) Intel Skylake, 4GiB [4, 36]	3	1	58
	(4) Intel Haswell [23]	7	1	201
	(5) AMD Zen+, 16GiB (linear part) [12]	5	1	231
	(6) AMD Zen4, 16GiB (linear part) [12]	6	1	153
Cache Slice	(7) Intel Coffee Lake [6]	4	16	260
	(8) Intel Ivy Bridge to Haswell, 8 cores [20]	3	16	131
	(9) Intel Ivy Bridge to Haswell, 4 cores [20]	2	16	78
f_{L3}	(10) all CPUs	11	16	43

Table 4: Simulation of reverse-engineering previously reported indexing functions.

in Figure 3). For the recently reverse-engineered AMD DRAM indexing functions (functions (5) and (6)), we tested our algorithm on the linear part, simulating the assumption that the previously applied bit-shift is known. This is realistic as the bit-shift is caused by the fixed ranges for the memory-mapped I/O devices, which can be inferred from the system's primary PCI memory mapping (see [12]). Intel's cache slice functions usually XOR a larger number of address bits (see Figure 2, which is function (7) in Table 4.) Lastly, we also tested the L3 cache set indexing function, which consists of singleton output bits. For all these functions, we need at most a few hundred congruence checks to determine their kernel.

Discussion. Our experiments demonstrate that inferring linear functions requires only a single eviction set and a few hundred congruence tests. This is especially useful for reverse-engineering DRAM functions, where current approaches perform congruence checks for thousands of addresses to enable a brute-force search that does not provide any formal guarantees (see the overview in Section 5.1). Our contributions are on the algorithmic side and our experiments were conducted in a simulation. We leave an evaluation on real hardware to future work.

6 Related Work

Cache Coloring and Beyond. Memory coloring has a long history with different but related use cases. Early uses of cache coloring aimed at better single-tenant performance, e.g., to achieve a cache-friendly layout of virtual memory pages, or to speed up virtual-to-physical address translation [2, 14, 31]. Coloring has later been used for performance predictability and worst-case execution time analysis of multi-core systems [37]. Cache partitioning for performance isolation and predictability for cloud workloads has been achieved with coloring [16, 25] or with hardware mechanisms such as CAT [27]; DRAM bank coloring for performance isolation was explored in [43].

The first security applications of memory coloring aimed to prevent side-channel attacks on a shared cache without considering other shared resources [5, 28]. A recent security application is the mitigation of RowHammer by coloring DRAM subarrays [19]. To our knowledge, the only attempt at simultaneous coloring of two resources is [29], which partitions the L3 cache and DRAM banks for performance isolation. It relies on the intersection of the indexing functions' output bits to be non-empty, which is a stricter requirement than the join. Our work is the first to propose thorough theoretical foundations and practical algorithms for comprehensive multi-resource memory coloring.

Resource-Exclusive VMs. Concurrent work [34] introduces *resource isolation contracts*, which combine constraints on compute scheduling with memory coloring schemes to achieve microarchitectural isolation between VMs in the cloud. Using the algorithms from Section 3, we can confirm that the coloring schemes inferred in [34] are indeed isolating and uniformly utilizing.

Reverse-Engineering Indexing Functions. For an overview of related work on reverse-engineering linear indexing functions, we refer to Section 5.1. For non-linear functions, there only exists very few systematic approaches. These works exploit system knowledge about subtracted offsets [12] or need to know the indexing function's output for a large number of addresses [6, 21].

Further Off-Core Resources. Finally, we remark that there are microarchitectural components that we do not consider in this paper. Some are not amenable to software-based partitioning, such as the staging buffer, which is shared across all cores and accessed during execution of specific instructions [24]. Others are potential candidates for an extension of our coloring schemes. An example is the mesh interconnect, for which recent approaches use smart placements to prevent leakage through latencies [3, 35].

7 Conclusion

We have presented theoretical foundations and practical algorithms for designing comprehensive memory coloring schemes for cloud CPUs. Our algorithms simultaneously achieve isolation with respect to shared resources while guaranteeing that private resources are fully utilized. We have further presented design patterns for indexing functions that maximize the number of available colors. Lastly, we have shown how to reverse-engineer the algebraic kernel of linear indexing functions under minimal assumptions.

Concurrent work [34] shows how modern hypervisors can leverage multi-resource coloring schemes to provide strong microarchitectural isolation between guest VMs. Since the associated performance overhead is small, such an approach provides practical and principled protection against cross-VM side-channel attacks.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful suggestions and comments. We also thank our colleagues at Microsoft, in particular Oleksii Oleksenko, for their feedback.

References

[1] Azure. 2022. Hyper-V HyperClear. https://techcommunity.microsoft.com/t5/virtualization/hyper-v-hyperclear-mitigation-for-l1-terminal-fault/ba-p/382429. Accessed: 2024-08-31.
[2] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. 1996. Compiler-directed page coloring for multiprocessors. *ACM SIGPLAN Notices* 31, 9 (1996), 244–255.
[3] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. 2022. Don't mesh around: Side-channel attacks and mitigations on mesh interconnects. In *USENIX Security Symposium*. USENIX Association.
[4] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized many-sided rowhammer attacks from JavaScript. In *USENIX Security Symposium*. USENIX Association.

[5] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time protection: the missing OS abstraction. In *14th EuroSys Conference*. ACM.
[6] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. 2024. Efficient and generic microarchitectural hash-function recovery. (2024).
[7] Martin Heckel and Florian Adamsky. 2023. Reverse-engineering bank addressing functions on amd cpus. https://dramsec.ethz.ch/papers/revengamd.pdf. Accessed: 2024-08-31.
[8] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. 2020. Reliable reverse engineering of Intel DRAM addressing using performance counters. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE.
[9] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Symposium on Security and Privacy (S&P)*. IEEE.
[10] Intel. 2022. 4th Gen Intel Xeon Processor Scalable Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html.
[11] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. Systematic reverse engineering of cache slice selection in Intel processors. In *Euromicro Conference on Digital System Design (DSD)*. IEEE Computer Society.
[12] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer attacks on AMD zen-based platforms. In *USENIX Security Symposium*. USENIX Association.
[13] The Linux Kernel. 2023. Core Scheduling. https://www.kernel.org/doc/html/next/admin-guide/hw-vuln/core-scheduling.html. Accessed: 2024-08-31.
[14] Richard E. Kessler and Mark D. Hill. 1992. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)* (1992).
[15] T. Kessous and N. Gilboa. 2024. Prune+PlumTree - finding eviction sets at scale. In *Symposium on Security and Privacy (S&P)*. IEEE.
[16] Hyoseung Kim and Ragunathan Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *13th International Conference on Embedded Software (EMSOFT)*. ACM.
[17] Zili Kou, Sharad Sinha, Wenjian He, and Wei Zhang. 2022. Attack directories on ARM Big.LITTLE processors. In *41st International Conference on Computer-Aided Design (ICCAD)*. ACM.
[18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Symposium on Security and Privacy (S&P)*. IEEE.
[19] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. 2023. Siloz: Leveraging DRAM isolation domains to prevent inter-VM rowhammer. In *Symposium on Operating Systems Principles (SOSP)*. ACM.
[20] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses (RAID)*. Springer.
[21] John D McCalpin. 2021. *Mapping addresses to l3/cha slices in intel processors*. Technical Report. https://repositories.lib.utexas.edu/server/api/core/bitstreams/7c5a5389-d124-41e0-bcbe-b5e95c03fc9b/content Accessed: 2024-08-31.
[22] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. cache attacks and countermeasures: the case of AES. In *CT-RSA*.
[23] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *USENIX Security Symposium*. USENIX Association.
[24] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. Crosstalk: Speculative data leaks across cores are real. In *Symposium on Security and Privacy (S&P)*. IEEE.
[25] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. 2009. Resource Management for Isolation Enhanced Cloud Services. In *Workshop on Cloud Computing Security*. ACM.
[26] Mark Seaborn. 2015. L3 cache mapping on Sandy Bridge CPUs. https://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html. Accessed: 2024-08-31.
[27] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. 2021. Provisioning differentiated last-level cache allocations to VMs in public clouds. In *Symposium on Cloud Computing (SoCC)*. ACM.
[28] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *International Conference on Dependable Systems and Networks Workshops*. IEEE/IFIP.
[29] Noriaki Suzuki, Hyoseung Kim, Dionisio De Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. 2013. Coordinated bank and cache coloring for temporal protection of memory accesses. In *International Conference on Computational Science and Engineering*. IEEE.
[30] Simon M. Tam, Harry Muljono, Min Huang, Sitaraman Iyer, Kalapi Royneogi, Nagmohan Satti, Rizwan Qureshi, Wei Chen, Tom Wang, Hubert Hsieh, Sujal Vora, and Eddie Wang. 2018. SkyLake-SP: A 14nm 28-Core Xeon processor. In *IEEE International Solid-State Circuits Conference*.
[31] George Taylor, Peter Davies, and Michael Farmwald. 1990. The TLB slice – a low-cost high speed address translation mechanism. In *Annual International Symposium on Computer Architecture*. IEEE.
[32] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES implemented on computers with cache. In *CHES*.
[33] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and practice of finding eviction sets. In *Symposium on Security and Privacy (S&P)*. IEEE.
[34] Stavros Volos, Cédric Fournet, Jana Hofmann, Boris Köpf, and Oleksii Oleksenko. 2024. Principled microarchitectural isolation on cloud CPUs. In *Conference on Computer and Communications Security (CCS)*. ACM.
[35] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. 2022. MeshUp: Stateless cache side-channel attack on CPU mesh. In *Symposium on Security and Privacy (S&P)*. IEEE.
[36] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. 2020. DRAMDig: A knowledge-assisted tool to uncover DRAM address mapping. *Design Automation Conference (DAC)*.
[37] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. 2013. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*. IEEE.
[38] Wikipedia. 2024. Kernel (Linear Algebra). https://en.wikipedia.org/wiki/Kernel_(linear_algebra). Accessed: 2024-08-31.
[39] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One bit flips, one cloud flops: Cross-VM row aammer attacks and privilege escalation. In *USENIX Security Symposium*. USENIX Association.
[40] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *Symposium on Security and Privacy (S&P)*. IEEE.
[41] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*. USENIX Association.
[42] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel last-level cache. Cryptology ePrint Archive, Paper 2015/905. https://eprint.iacr.org/2015/905
[43] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE.
[44] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for It: Bridging the gap between microarchitectural and architectural side channels. In *USENIX Security Symposium*. USENIX Association.

A Proofs of Section 2

Proposition 7. *$P_1 \perp P_2$ iff $P_2 \perp P_1$.*

Proof. Let $C_1 \in P_1, C_2 \in P_2$. We have

$$\frac{|C_1 \cap C_2|}{|C_2|} = \frac{|C_1|}{|M|} \quad \equiv \quad |C_1 \cap C_2| = \frac{|C_1| \cdot |C_2|}{|M|}$$

and thus

$$\frac{|C_1 \cap C_2|}{|C_1|} = \frac{|C_1| \cdot |C_2|}{|C_1| \cdot |M|} = \frac{|C_2|}{|M|}$$

□

Proposition 1. *Let $f = (f_1, \dots, f_m)$ and P be a partition with $P_{f_i} \sqsubseteq P$ for some f_i. Then*

- *$P_f \sqsubseteq P$*
- *$P_{f_j} \perp P_{f_i}$ implies $P_{f_j} \perp P$*

Proof. The first bullet point follows simply from $P_f \sqsubseteq P_{f_i}$ (by definition of f). Transitively, we obtain $P_f \sqsubseteq P_{f_i} \sqsubseteq P$. For the second bullet point, assume $P_{f_j} \perp P_{f_i}$ and let $C_j \in P_{f_j}$ and $C \in P$. We show $P \perp P_{f_j}$ using commutativity of $\perp$.

$$\frac{|C \cap C_j|}{|C_j|} = \frac{|(C^1 \cup \ldots \cup C^n) \cap C_j|}{|C_j|}$$

where each $C^i \in P_{f_i}$ because $P_{f_i} \sqsubseteq P$

$$= \frac{\left|C^1 \cap C_j\right| + \ldots + \left|C^n \cap C_j\right|}{\left|C_j\right|}$$

because $C^1, \ldots, C^n$ are pairwise disjoint

$$= \frac{|C^1|}{|M|} + \ldots + \frac{|C^n|}{|M|} \quad \text{because } P_{f_i} \perp P_{f_j}$$

$$= \frac{|C|}{|M|}$$

□

B Proofs of Section 3

Proposition 2. *Let f_1, f_2 be linear functions defined on $\mathbb{F}_2^n$. Then*

$$P_{f_1} \sqcup P_{f_2} = \mathbb{F}_2^n /_{ker(f_1)+ker(f_2)}$$

Proof. Any partition P defines an equivalence relation $\sim_P$ with $x \sim_P y$ iff there is a $C \in P$ such that $x, y \in C$. We write $\sim_\sqcup$ for the equivalence relation defined by $P_{f_1} \sqcup P_{f_2}$ and $\sim_/$ for the one defined by $\mathbb{F}_2^n /_{ker(f_1)+ker(f_2)}$. We show $x \sim_\sqcup y$ iff $x \sim_/ y$.

By the definition of the join as finest relation that refines both P_{f_1} and P_{f_2}, we have $x \sim_\sqcup y$ iff there exist $z_0, \ldots, z_n$ such that

$$x \sim_{f_1} z_0 \sim_{f_2} z_1 \sim_{f_1} \ldots \sim_{f_1} z_n \sim_{f_2} y$$

As argued in Section 3.1, $x \sim_{f_1} x'$ iff $x + x' \in ker(f_1)$, which is equivalent to $x' = x + a$ for some $a \in ker(f_1)$. Thus, $z_0 = x + a_0$ for $a \in ker(f_1)$, $z_1 = x + a_0 + a_1$ for $a_1 \in ker(f_2)$. Continuing the argument, $y = x + a_0 + a_1 + \ldots + a_n$, where each a_i is either in $ker(f_1)$ or in $ker(f_2)$. By the definition of the sum of vector spaces, $a_0 + \ldots + a_n \in ker(f_1) + ker(f_2)$. This is equivalent to $y \in x + (ker(f_1) + ker(f_2))$ and $x \sim_/ y$. □

Lemma 1. *Algorithm 1 computes A_h with $P_h = P_{f_1} \sqcup P_{f_2}$.*

Proof. Let $V = ker(f_1) + ker(f_2)$. With Proposition 2, our aim is to compute a function h with $ker(h) = V$. By definition of the sum of vector spaces, $v_1, \ldots, v_t$ spans V. If we find A with $Aw = 0$ for exactly all linear combinations w of $v_1, \ldots, v_t$, then A defines h. Equivalently, $A \cdot (v_1, \ldots, v_t) = 0$ (where 0 is the 0-matrix). Let $B = (v_1, \ldots, v_t)$. Since $A \cdot B = 0$ iff $B^T \cdot A^T = 0$, we can choose A^T as the basis $u_1, \ldots, u_l$ of the kernel of B^T. Consequently, we have $A = (u_1, \ldots, u_l)^T$. Then, by construction, $ker(f_1) + ker(f_2) \subseteq ker(A)$. It remains to show $ker(f_1) + ker(f_2) = ker(A)$. To do so, it is known to be sufficient to show $dim(ker(f_1) + ker(f_2)) = dim(ker(A))$, where dim is the dimension of a vector space. Let's recapitulate the following known algebraic facts:

- For any linear function $g : W^n \longrightarrow W^m$, we have $n = dim(im(g)) + dim(ker(g))$, where $im(g)$ is the image of a function.
- Furthermore, $dim(im(g)) = rank\left(A_g\right) = rank\left(A_g^T\right)$, where the rank of a matrix is the number of linearly independent column vectors of a matrix.

With that we get:

$$\begin{aligned} & dim(ker(A)) \ = \ n - dim(im(A)) \\ & = n - rank\left(A^T\right) = \ n - dim(ker(B^T)) && \text{by definition of } A \\ & = n - (n - rank\,(B)) \\ & = dim(ker(f_1) + ker(f_2)) && \text{by definition of } B \end{aligned}$$

□

Proposition 3. *Let h, g be linear functions defined on $\mathbb{F}_2^n$. Then the following are equivalent*

(1) $P_h \perp P_g$

(2) $rank\left(\begin{bmatrix} A_g \\ A_h \end{bmatrix}\right) = rank\,(A_h) + rank\,(A_g)$

Proof. First observe that for any f, $rank\left(A_f\right) = dim(im(f))$. With the dimension formula $n = dim(im(f)) + dim(ker(f))$, we get $dim(ker(f)) = n - rank\,(f)$. Also, $\mathbb{F}_2^n$ is a finite vector space of cardinality 2^n. Finally, remember that for linear f, for any $C \in P_f$, $|C| = |ker(f)|$.

Now let $A \in P_h, B \in P_g$. By definition, $A \cap B \in P_{(h,g)}$. We obtain

$$\begin{aligned} & \frac{|A \cap B|}{|B|} = \frac{|A|}{2^n} \\ & \equiv 2^{dim(ker((h,g)))} \cdot 2^n = 2^{dim(ker(h))} \cdot 2^{dim(ker(g))} \\ & \equiv n + dim(ker((h,g))) = dim(ker(h)) + dim(ker(g)) \\ & \equiv n + (n - rank\,((h,g))) = (n - rank\,(h)) + (n - rank\,(g)) \\ & \equiv rank\,((h,g)) = rank\,(h) + rank\,(g) \end{aligned}$$

□

C Additional Material Section 5

C.1 Proofs

Lemma 3. *A function f is congruent to linear iff*

$$\forall x, y, e_i.\, f(x) = f(y) \Rightarrow f(x + e_i) = f(y + e_i),$$

where e_i is a standard unit vector.

Proof. We show an equivalent claim, namely that f is congruent to linear iff

$$\forall x, y, d.\, f(x) = f(y) \Rightarrow f(x + d) = f(y + d), \qquad (1)$$

where d is an arbitrary vector. These two statements are equivalent: One direction is trivial (d can be instantiated with e_i). The other direction can be obtained by repeated application of Equation (1). We now show both directions of the statement.

- If there is a linear g with $P_f = P_g$, then Equation (1) holds for g because it is linear and transfers to f because $P_f = P_g$.
- Assume Equation (1) holds. To show that there is a linear function g with $P_f = P_g$, we show that f has a proper kernel, i.e. there is a vector space K such that $k \in K$ iff $\forall x.\, f(x) = f(x+k)$. We define K by fixing an arbitrary vector v and setting $K = \{k \mid f(v) = f(v+k)\}$. We show that K is a vector space, i.e., closed under scalar multiplication and addition:

– K is closed under scalar multiplication because $0 \in K$.

– K is closed under addition: $\forall x, y \in K.\, x + y \in K$. From $f(v) = f(v+x)$ it follows $f(v+y) = f(v+x+y)$ because of Equation (1). From $f(v) = f(v + y)$ thus follows $f(v) = f(v + x + y)$ and thus $x + y \in K$.

Finally, we show $k \in K$ iff $\forall x.\, f(x) = f(x + k)$. This holds because $f(v) = f(v + k)$ iff $f(v + (v + x)) = f(v + k + (v + x))$ (because of Equation (1)) iff $f(x) = f(x + k)$.

□

Proposition 5. *Let $h : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^m$ be a linear function and $U \subseteq \mathbb{F}_2^n$ be a minimal subspace with $h(U) = im(h)$. Then $ker(h) \oplus U = \mathbb{F}_2^n$.*

Proof. We need to show two things. First, we have to show that $K \cap U = \emptyset$, to justify the application of the direct sum operator. Second, we need to show that $K + U = \mathbb{F}_2^n$.

- Show: $K \cap U = \emptyset$. Assume there was a u with $u \in K$ and $u \in U$. Let $u_1, \ldots, u_m$ be a basis of U. Thus, there is a non-empty set $\{u_{i_1}, \ldots, u_{i_l}\} \subseteq \{u_1, \ldots, u_m\}$ such that $u = u_{i_1} + \ldots + u_{i_l}$. As $h(u) = 0$, we have $h(u_{i_1}) + \ldots + h(u_{i_l}) = 0$. But then either $h(u_{i_j}) = 0$ or $h(u_{i_j}) = h(u_{i_k})$ for some $j, k \in [1, \ldots, l]$. In either case, this breaks the assumption that U is a *minimal* space such that $h(U) = im(h)$.
- Show: $K + U = \mathbb{F}_2^n$. $K + U \subseteq \mathbb{F}_2^n$ follows trivially as both K and U are subspaces of $\mathbb{F}_2^n$. It is a known fact that $dim(h) = dim(ker(h)) + dim(im(h))$. Since $dim(im(h)) = dim(U)$, and $K \cap U = \emptyset$, $dim(K) + dim(U) = n$ and equality follows.

□

Proposition 6. *If f is simple, then $ker(f)$ is defined by a basis B such that in each $v \in B$, at most two bits are set to 1.*

Proof. Let $X_{unused} \subseteq x_1, \ldots, x_n$ be all address bits that are not used in f. Let furthermore $bits(r) \subseteq x_1, \ldots, x_n$ be all bits that are set to 1 in a row r of A_f. Similarly, $bits(a) \subseteq x_1, \ldots, x_n$ denotes all bits set to 1 in an address a. Since f is simple, $a \in ker(f)$ iff either

- a has exactly one bit x_i set and $x_i \in X_{unused}$, or
- for each row r in A_f, $|bits(r) \cap bits(a)|$ is even.

Thus $ker(f)$ is spanned by the set that contains all

- standard vectors e_i with $x_i \in X_{unused}$ and
- addresses a with exactly i, j set in a and for some row r of A_f, $x_i, x_j \in bits(r)$.

□

C.2 Combination with Non-linear Functions

As afterthought, we discuss how we might be able to reverse-engineer the linear output bits of an indexing function for the case that other bits of the function are non-linear. Concretely, we consider the case that the matrix representation of $f = (f_{nl}, f_l)$, where f_{nl} is a non-linear function and f_l is a linear function. The question is: Can we still reverse-engineer f_l, still assuming that we cannot determine the value of $f(x)$ and do not know its rank m? We discuss two scenarios. In the first, we assume that f_{nl} is fully known. We show that we can adapt Algorithm 3, but establish that we might need more than one eviction set. In the second scenario, f is entirely unknown. For this case, we illustrate the inherent difficulties of the problem on examples.

Known Non-linear Part. Assume we know f_{nl} and want to reverse-engineer f_l using Algorithm 3. We need to determine the kernel of f_l. Since f_l forms only a subset of the output bits of f, we might have $f(x + y) \neq f(x)$ even if $y \in ker(f_l)$. For some other x', however, we might have $f(x' + y) = f(x')$. To illustrate this issue, consider the following example.

Example 12. *Let $f_{nl} = x_1 \vee x_2$ and $f_l = x_2 \oplus x_3$. For $y = x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, we have $y \in ker(f_l)$, but $f(x) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \neq \begin{pmatrix} 0 \\ 0 \end{pmatrix} = f(x+y)$. For $x' = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, on the other hand, $f(x') = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = f(x' + y)$. In particular, there is no x s.t., for all $y \in ker(f_l)$, $f(x) = f(x + y)$.*

The above example illustrates that in order to apply Algorithm 3, we need to create an eviction set for more than one address x. The number of eviction sets needed can be computed from the non-linear function f_{nl}, resulting in the following adaption of Algorithm 3:

(1) Compute for each address x the set $S_x = \{y \mid f_{nl}(x) = f_{nl}(x + y)\}$.

(2) Pick a minimal set X of addresses s.t, $\bigcup_{x \in X} S_x = \mathbb{F}_2^n$. Compute an eviction set for each $x \in X$.

(3) To test if $y \in ker(f_l)$, check congruence with the address x for which $y \in S_x$ holds. Since $y \in S_x$, we know that $f_{nl}(x+y) = f_{nl}(x)$, so $x + y$ is evicted by x's eviction set iff $y \in ker(f_l)$.

We leave an estimate for the minimal number of eviction sets needed for this approach for future work.

Unknown Non-linear Part. Finally, we point out the challenges in reverse-engineering f_l if f_{nl} is unknown. The following example illustrates that there might be several possibilities to split a non-linear function f into a non-linear part f_{nl} and a linear part f_l.

Example 13. *Let $f = (f_1, f_2)$ with $f_1 = (x_1 \vee x_2) \oplus x_3$ and $f_2 = (x_1 \vee x_2) \oplus \neg x_4$. Individually, both f_1 and f_2 are non-linear functions. However, there exist functions g and h that both have the same preimages as f:*

$$g_1 = (x_1 \vee x_2) \oplus x_3, \quad x_3 \oplus x_4$$
$$h_1 = (x_1 \vee x_2) \oplus \neg x_4, \quad x_3 \oplus x_4$$

The above example demonstrates that we might be able to factor out linear parts of a function even if individually, all output bits are non-linear. It is unclear, though, how to extract linear output bits from non-linear functions. The example shows that we cannot search for "linear" input bits: In h, bit x_4 is not used by the non-linear function h_1 and could be called linear; in g, this would be bit x_3. h and g have the same preimages, though, and should therefore be considered equivalent. We leave the question of how to reverse-engineer linear parts of a non-linear function for future work.

C.3 Implementing Offsets with Boolean Functions

Assume we want to subtract a constant $c = 2^i$ from an address x. If bit x_i is 1, we can just flip it to 0. If it is 0, however, we need to flip it to 1, and attempt to flip bit $i + 1$ from 1 to 0 instead. Here, again, if bit x_{i+1} is 0, we flip it to 1 and continue our search for the first non-zero bit. This approach can be expressed with the following non-linear function f, which takes the value of bit x_j and returns its value in the address $x - c$. For a linear function g, if we replace every occurrence of x_j by $f(x_j)$, we obtain a non-linear function that first subtracts c and then applies g.

$$\begin{aligned} f(x_j) &= x_j && \text{if } j < i \\ &= \neg x_i && \text{if } j = i \\ &= (x_i \vee \ldots \vee x_{j-1}) \oplus \neg x_j && \text{if } j > i \end{aligned}$$

D Artifact

The artifact of this paper reproduces our linearity testing and reverse-engineering results. Specifically, the reviewers confirmed the numbers reported in Tables 1 to 4 in Section 5.6.