Taming the Linux Memory Allocator for Rapid Prototyping

Ruiyi Zhang, Tristan Hornetz, Lukas Gerlach, and Michael Schwarz

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract. Microarchitectural attacks pose an increasing threat to system security. They enable attackers to extract sensitive information such as cryptographic keys, website usage patterns, or keystrokes. Software-level defenses, such as constant-time implementations, mitigate some attack vectors but impose significant challenges on developers. Operating-system-level mitigations, such as page coloring and memory isolation, address these threats but require intricate kernel modifications and time-consuming workflows, making prototyping new defenses complex.
In this paper, we present MAPAlloc (Microarchitectural Prototyping Allocator), a flexible, cross-architecture framework for rapidly prototyping memory allocation-based defenses and attacks on Linux systems. Using a simple domain-specific language, MAPAlloc allows for precise control over physical memory allocation on x86, ARMv8, and RISC-V. MAPAlloc enables quick implementation and evaluation of mitigations such as page coloring and novel techniques like layered page coloring, increasing the number of cache colors from 32 to 256 on modern CPUs. We demonstrate MAPAlloc's versatility through case studies that prevent Prime+Probe and DRAMA attacks and reverse-engineer the AMD Zen 4 complex cache-indexing function for use in layered page coloring. Additionally, we prototype a Prime+Probe attack with an incomplete non-linear slice function from previous work by limiting the physical memory using MAPAlloc. Without MAPAlloc, such defense and attack prototypes require complicated modifications of the Linux kernel, making them hard to develop and test. Thus, MAPAlloc is an essential framework for simplifying research in microarchitectural security.

1 Introduction

Microarchitectural attacks are increasingly becoming a significant and practical threat. These attacks not only threaten security by inferring cryptographic keys [6, 31] but also threaten privacy, such as in website fingerprinting [55, 68], keystroke logging [52, 62], and compromising the privacy guarantees of differential privacy algorithms [25]. Academia and industry have dedicated substantial effort to addressing this issue, targeting both hardware and software levels. While hardware mitigations look promising, retrofitting them to existing hardware CPUs is often not possible [49, 63, 51]. Moreover, it typically takes years until mitigations appear in hardware, if they appear at all [56, 12]. A more

flexible approach involves software-level defenses against microarchitectural attacks. Constant-time implementation [23], commonly found in cryptographic libraries [41, 64, 46, 59], can avoid a broad class of microarchitectural attacks, including ones that are based on timing or memory access patterns. However, constant-time code is challenging to implement, making it impractical for all but the most high-risk targets. Moreover, it shifts the burden to the developer instead of fixing it at a more central point, such as the operating system.

Consequently, previous works proposed multiple mitigations on the operating-system level [54, 29, 4, 32]. These mitigations influence memory allocation to better isolate attacker and victim applications. Page coloring [5] assigns non-overlapping cache sets to mutually untrusted applications, preventing eviction-based cache attacks such as Prime+Probe and Evict+Reload. CATT [4] mitigates the effect of Rowhammer on the kernel by separating kernel and user-space memory such that these memory regions never end up in adjacent DRAM rows. Similarly, ZebRAM [32] modifies the allocator to have guard rows in DRAM, making Rowhammer flips ineffective. Such mitigations all require complex changes to the kernel, requiring considerable expertise in kernel data structures and functionality. Additionally, as microarchitectural attacks typically require native code execution, the workflow of compiling and rebooting into a new kernel for tests is time-consuming and tedious.

In this paper, we present MAPAlloc (Microarchitectural Prototyping Allocator), a rapid-prototyping framework that provides extensive control over the *physical* memory allocation of the Linux kernel. MAPAlloc supports an expressive Domain-Specific Language (DSL) for memory constraints to enable precise per-process control over the physical pages a process can access. MAPAlloc can easily provide proof-of-concept implementations for mitigations such as page coloring by simply specifying the cache-set function using the DSL and the process to which the restriction should be applied. Moreover, MAPAlloc is implemented as a cross-architecture kernel module, supporting x86, ARMv8, and RISC-V CPUs. Thus, MAPAlloc can be loaded at runtime on a wide range of systems. MAPAlloc is designed for rapid prototyping, allowing the evaluation of defense strategies such as emulating hardware coloring effects. However, it is not a production-ready mitigation. Any real defense could similarly bypass or modify the memory allocator, just as MAPAlloc does. Therefore all mitigations prototyped using MAPAlloc will work in real systems.

To demonstrate the flexibility of MAPAlloc, we demonstrate how it can use DRAM addressing functions [45] to separate DRAM rows and cache-set functions to isolate cache sets [54], quickly reproducing known mitigations. In case studies, we show that our prototyped mitigation using MAPAlloc effectively prevents Prime+Probe and DRAMA attacks. Additionally, we introduce *layered page coloring*, an improved variant of page coloring that combines cache sets with cache slices to provide more flexibility in assigning non-overlapping cache sets. We introduce a graph-based algorithm to calculate the number of available colors when combining arbitrary linear and non-linear microarchitectural hash functions. We show that on modern CPUs, layered page coloring can increase

the number of colors from 32 to 256. Further, we reverse-engineer the cache set-indexing function on AMD Zen 4 CPUs, showing that this complex function can also be used for layered page coloring on the L2 and L3 caches. Thus, layer page coloring is a viable defense on modern CPUs that could realistically be implemented in the Linux kernel.

As a side effect, MAPAlloc can also simplify microarchitectural attack prototyping. For example, existing non-linear cache-slice functions for modern CPUs have only been reverse-engineered for systems with at most 4 GB of DRAM [15]. However, by limiting memory allocations to the first 4 GB of physical memory using MAPAlloc, attacks can still be tested on systems with more memory. Similarly, MAPAlloc can reduce the time it takes Rowhammer attacks to find exploitable rows by limiting rows to a specific DIMM or even bank. Such artificial constraints on the system can ease attack prototyping, as shown in previous work [11]. Note that MAPAlloc does not enhance real attacks, as it operates as a privileged Linux kernel module. It only assists in prototyping by applying controlled memory constraints.

Contributions. The main contributions of this work are:

1. We present MAPAlloc, a generic framework that allows for customizable refinement of physical page allocations across different architectures, enabling researchers to quickly prototype mitigations (such as page coloring) and attacks.
2. We introduce layered page coloring with linear and non-linear microarchitectural hash functions, showing that such a combination can increase the number of security domains for page coloring.
3. We reverse-engineer the complex set addressing and cache-slice functions on AMD Zen 4 and demonstrate their use in page coloring.
4. In three case studies, we demonstrate the use cases of MAPAlloc by building page coloring for the last-level cache and the DRAM and artificially limiting the physical memory to prototype Prime+Probe with an incomplete non-linear cache-slice function.

Structure. The paper is organized as follows. Section 2 provides relevant background. Section 3 presents the design and implementation of our frameworks. In Section 4, we evaluate MAPAlloc by building prototypes for (layered) page coloring against eviction-based attacks and DRAMA attacks, and we prototype a Prime+Probe attack with an incomplete non-linear cache-slice function. Section 5 evaluates the functionality and performance of MAPAlloc. We discuss related works and limitations in Section 6. Section 7 concludes.

Availability. Our framework is available at `https://github.com/cispa/MAPAlloc`.

2 Background

2.1 Memory Allocators

Memory allocators are critical components of modern operating systems, responsible for managing the allocation and deallocation of memory in units called pages, each typically 4 kB in size. The main allocator in Linux, known as the buddy allocator [8], groups free memory pages into buddies of various sizes, which are powers of two. For example, an order-0 buddy is one 4 kB page, while an order-3 buddy is 8 contiguous 4 kB pages (32 kB). The allocator uses free lists to keep track of available memory blocks for each size. When a process requests memory, the allocator can split larger blocks into smaller ones or merge adjacent blocks to form larger ones, reducing fragmentation. Memory allocation requests include flags, known as Get Free Pages (GFP) flags [8], which specify additional information like whether the memory is for user or kernel space. These flags help the allocator decide how to fulfill the request efficiently. Additionally, the per-CPU page (PCP) allocator [9] manages free pages for each CPU, speeding up memory allocation by reducing contention. Together, these mechanisms ensure efficient memory management in the Linux kernel.

2.2 Cache Eviction and Attacks

Modern processors use cache memory to store frequently accessed data and instructions, significantly improving performance by reducing the time needed to access data from the main memory. Unlike the larger 4 kB pages in main memory, the cache is organized into cache sets, each containing multiple 64-byte chunks, known as cache lines. Despite their speed, caches have limited size, making cache eviction necessary to create space for new entries when the cache is full. Well-known cache side-channel attacks, such as Prime+Probe [44] and Evict+Reload [18], exploit the cache eviction process to infer sensitive information. In Prime+Probe, the attacker fills cache sets with their own data and later measures access times to determine which cache lines were evicted by the victim's memory accesses, revealing the victim's access patterns. Conversely, Evict+Reload relies on shared memory. The attacker evicts a specific cache line by filling the cache set, waits for the victim to access it, and then measures the reload time to infer whether the victim accessed the memory.

2.3 DRAMA and Rowhammer

Unlike side channel attacks that target CPU caches, DRAMA attacks [45] target internal caching structures of DRAM modules called row buffers. DRAM is structured into multiple components, including channels, DIMMs, ranks, and banks, with the row being the smallest independently accessed unit. Each bank consists of numerous rows, with a corresponding row buffer for read and write operations. When data is accessed, the corresponding row is loaded into the row buffer, which can then be read. Loads into the row buffer have variable latency

depending on the row buffer state. If the row currently loaded is already cached in the row buffer, access times are typically 50-70 cycles faster. By measuring these access times, an attacker can monitor the state of the row buffer and implicitly the victim's memory access patterns. Similar to cache sets, a row can contain memory from different applications.

While DRAMA attacks can leak victim memory access patterns, Rowhammer attacks [30, 50, 48] can corrupt data. Similar to DRAMA attacks, Rowhammer operates on the DRAM. However, in contrast to DRAMA, Rowhammer exploits the physical properties of the DRAM cells. As DRAM cells are dynamic memory, they leak charge over time and must be refreshed regularly to prevent memory corruption. By repeatedly accessing (hammering) a row of memory, the power leakage of DRAM cells is accelerated, and bit flips, typically in the adjacent rows, are induced. These bit flips can corrupt data, allowing attackers to manipulate the memory content of other processes. While multiple mitigations on hardware [12, 38, 3], software [4, 2] and combined [27] levels have been proposed, Rowhammer remains challenging to mitigate. This is due to the closed nature of DRAM modules and memory controllers and the fact that Rowhammer is a physical phenomenon that even differs between identical setups [16]. Multiple previous mitigation techniques, such as increasing refresh rates and ECC DRAM, have been shown to be only partially effective [12].

2.4 Page Coloring

Page coloring is a memory allocation technique employed for performance [67] or security [54] purposes. It allows for assigning "colors", which are labels, to memory pages. The color of a page can be determined by arbitrary attributes, such as physical address or allocating process. These colors can then serve as an additional attribute during allocation to distribute memory pages more precisely. For example, page coloring can help to optimize cache utilization by considering the cache sets as colors. Thus, pages with different colors can be used for frequently accessed memory, as the non-overlapping cache sets prevent mutual eviction. This increases the cache hit rate and, thus, an application's performance. In the context of security, page coloring can be used to prevent side-channel attacks by isolating microarchitectural elements between processes. For example, if pages from processes in different security domains are mapped to disjoint cache sets, this prevents eviction-based cache side channels between the processes.

3 Framework

In this section, we present a high-level overview of MAPAlloc and its practical implementation for memory allocation manipulation. MAPAlloc allows researchers to influence the *physical* memory allocation for a specific application using a simple DSL. Once configured, MAPAlloc allocates pages only with the physical addresses that fulfill the constraints given by the DSL. We evaluate the proof-of-concept program on x86, ARMv8, and RISC-V CPUs.

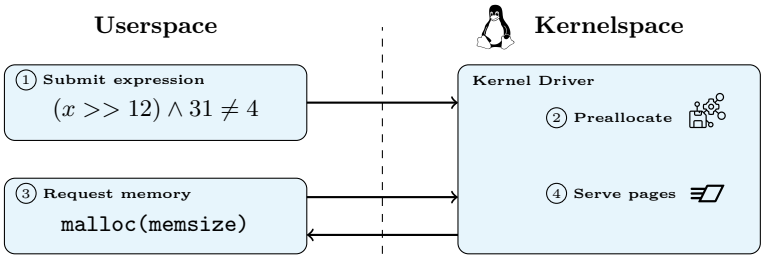

Fig. 1: The design of MAPAlloc. A client requests a memory policy, which the kernel driver preallocates. Afterward, the client can request memory allocations from this pool.

3.1 Design

MAPAlloc consists of two core components: a DSL specifying constraints on physical addresses and a custom allocator that allocates pages that fulfill these constraints. While MAPAlloc has to run in the kernel, it exposes a user-space interface for providing the DSL-defined constraints. Constraints are on a per-process basis, specified via the process ID.

Figure 1 shows the overview of MAPAlloc. After providing DSL-defined constraints to MAPAlloc, MAPAlloc preallocates physical pages fulfilling the constraints in kernel mode. These preallocated pages are kept in a pool to quickly provide them to the application if needed. Hence, allocating pages with MAPAlloc does not require costly searches for fitting pages, as the pool decouples this search from the allocation. As a result, allocations are always fast, independent of the constraint's complexity. We allow for the configuration of this pool size to ensure an ideal tradeoff between memory overhead and allocation speed.

To ensure that all memory allocations are served via MAPAlloc and its pool, MAPAlloc hooks the Linux kernel's memory allocation functions. This captures all memory allocations, including explicit allocations via `mmap` and `brk/sbrk` and implicit allocations triggered by page faults. Constraints can also apply to already allocated pages. For example, since user-space applications never directly interact with physical pages, the operating system (or, in this case, MAPAlloc) can transparently migrate existing pages to ones that meet specific constraints. MAPAlloc provide APIs to enforce migration by iterating through the relevant pages, updating the respective page-table entries, and flushing the TLB to ensure coherence.

3.2 DSL

We provide a lightweight DSL for specifying constraints. The current proof-of-concept implementation supports an arbitrary number of constraints involving arithmetic operations, bit operations, and comparisons. These constraints are combined using logic operations. The DSL's grammar is shown in the appendix in Figure 5. The advantage of a DSL is that constraints can be complex and

can be extended with functions in a backward-compatible manner. MAPAlloc parses the grammar to apply the constraints to the memory allocations. While the current implementation does not yet support stateful constraints, this is not a limitation of the design. Future contributions are encouraged to extend the DSL with this capability.

3.3 Implementation

To make the usage of MAPAlloc as simple as possible, we implement it as a Linux kernel module instead of a kernel patch. While this reduces the flexibility of MAPAlloc and impacts the performance, it greatly enhances the maintainability. Furthermore, using kernel modules is often possible when a custom kernel is difficult to install, such as on ARM and RISC-V development boards or smartphones. The kernel module is written in C and hooks the Linux kernel's memory management functions to manipulate memory allocations. We do not rely on architecture-specific functions, making MAPAlloc compatible with a wide range of architectures. We successfully verify its functionality on x86, ARMv8, and RISC-V.

Our implementation hooks the `__alloc_pages` function of the Linux kernel. This function is the lowest-level function responsible for returning a physical page. It handles all single-page allocations, both implicit and explicit allocations. Thus, we only have to hook here to cover the majority of cases for getting a user-accessible physical page. We use a `kretprobe` to hook into this function before it returns. Due to our implementation as a kernel module, we can neither change nor replace the function generically. Thus, we have to resort to such a probe.

We opt for the return probe, as this allows using the results of the kernel function while being able to change the return value. In the handler of the return probe, we have access to the page allocated by the kernel. We can either directly use this page if it fulfills the constraints, replace it with a page from our pool, or reject it, forcing the kernel to allocate a new page.

Additionally, we hook `vm_mmap_pgoff`, which is used for `mmap` functionality. Hooking this function allows for intercepting explicit multi-page allocations, replacing them with pages that fulfill the constraints.

3.4 Usage

Listing 1 shows a sample usage of MAPAlloc. The main workflow is to create a new process via `fork`, provide the DSL-based constraints to MAPAlloc for the new process, and then replace the constrained process with the target process via `exec*`. This workflow ensures that all pages of the process fulfill the constraints. While MAPAlloc can also be applied to a running process, it only affects the pages allocated after applying MAPAlloc. However, we can also support this use case by migrating existing physical pages of the target process to ones that fulfill the constraints.

```
const char* constraint = "((x >> 12) & 31 >= 1) && "
                         "((x >> 12) & 31 <= 4)";
if(mapalloc_init()) {
  printf("Error initializing MAPAlloc, did you load the module?\n");
  return 1;
}
pid_t pid;
if((pid = fork()) != 0) {
  mapalloc_constrain(pid, constraint);
  execve([...])
}
```

Listing 1: Example usage of MAPAlloc for starting a new process that only receives physical pages mapping to a subset of cache sets.

$$
\begin{aligned}
i(a) = {} & a_6,\ a_7,\ a_8,\ a_9 \oplus a_{28} \oplus a_{29},\ a_{10} \oplus a_{27} \oplus a_{30},\ a_{11} \oplus a_{26} \oplus a_{31} \\
& a_{12} \oplus a_{25} \oplus a_{32},\ a_{13} \oplus a_{24} \oplus a_{33},\ a_{14} \oplus a_{23} \oplus a_{34},\ a_{15} \oplus a_{22} \oplus a_{35},\ a_{16} \oplus a_{21} \oplus a_{36} \\
h(a) = {} & a_{17},\ a_{18},\ a_{19}
\end{aligned}
$$

Fig. 2: Set index function i and slice hash function h for EPYC 9124 (Zen 4). As both functions do not share bits, they can be trivially combined. The total number of combined colors is the product of the colors provided by set and slices.

4 Case Studies

In this section, we demonstrate different use cases for MAPAlloc. We demonstrate that MAPAlloc makes it easy to prototype page coloring, both for cache sets and DRAM rows. Additionally, we introduce layered page coloring for non-linear microarchitectural hash functions, showing that combining microarchitectural hash functions enables finer-grained coloring. We also show that MAPAlloc helps prototyping attacks by artificially limiting the physical address space, allowing the use of incomplete cache-slice functions.

4.1 Eviction Set Prevention

In this case study, we focus on the color attributes of the cache slice and the cache sets. With MAPAlloc, we evaluate the page coloring prototype against eviction-based cache side-channel attacks.

The cache set index function is typically an identity function over parts of the physical address. For a cache with S sets, bits $a_{6 \ldots 6+\log_2(S)}$ of the physical address a determine the set index. While an attacker can control bits within a page, the bits $a_{12 \ldots 6+\log_2(S)}$ can still contribute to page coloring. Coloring based on the cache set is trivial to implement using MAPAlloc, as the constraint is a simple bitmask on the physical address.

Table 1: The result of our coloring isolation. *Color Num* is the number of colors provided by using set partition or slice partition, which is not controllable by an attacker who can control bits 6-12. *EVC_ Time* indicates how long an attacker needs to build eviction sets via the tool evsets [60]. ✗ indicates that eviction set building failed.

CPU	Coloring Element	Color Num	EVC_Time	After coloring
Intel Core i3-5010U	Cache Set	64	16 s	✗
AMD EPYC 7252	Cache Set	64	30 s	✗
AMD EPYC 9124	Cache Set	32	✗	✗
AMD EPYC 9124	Cache Slice	8	✗	✗

Layered Page Coloring As regular page coloring is limited to 32 or 64 colors on current x86 CPUs, we introduce layered page coloring, combining multiple indexing functions to create more colors. One such additional indexing function is the cache-slice function that partitions the cache into multiple slices. However, the cache set and slice functions have overlapping bits, making this combination non-trivial. The increased number of colors depends on the number of overlapping bits used in both functions and how the attacker-controllable bits $a_{6\ldots11}$ affect the slice hash function. In such cases, a linear algebra-based approach based on the kernel of the hash function can be used [19]. We reimplement this approach, which allows efficient computations of memory partitions based on multiple *linear* hash functions. On AMD EPYC 9124, the cache-index function i and slice hash h do not interfere with each other, as shown in Figure 2. Therefore, it is trivial to compute a good partition that splits both cache sets and slices into different colors.

As the case of non-linear function has yet to be solved generically, we model the problem using graphs to calculate the number of available colors. While our approach scales exponentially, the problem we solve is small enough to be practical. We refer to the combination of cache slice and cache set as *extended set*. We evaluate which extended set the attacker can reach for all physical addresses when controlling the page offset. The address and the reachable extended set are connected nodes in a graph. On the full graph, we calculate the number of components, i.e., parts of the graph that are not connected. This number is equivalent to the number of colors available for page coloring. Additionally, each component contains the extended set for a color that can be used in MAPAlloc. We calculate the number of available colors for an Intel Coffee Lake with 6 slices based on the non-linear slice function provided by Gerlach et al. [15]. This non-linear slice function doubles the number of usable colors to 64.

To evaluate the page coloring as a prototype mitigation, we use a global variable as the target. We allocate a 128 MB memory buffer and select candidate addresses from this buffer to construct an eviction set, using the eviction-set generation from Vila et al. [60]. Afterward, we calculate the color of the target variable based on the cache sets or cache slices function. We assume an L3

Prime+Probe attacker can access all the remaining page colors except for the one the target variable has. Hence, we deploy this memory constraint to avoid allocating addresses with the same color and repeat the eviction set construction. We evaluate this policy on various machines, as shown in Table 1. The tool fails to find eviction sets on AMD EPYC 9124, where effective eviction construction is naturally complex, as the non-inclusive L3 cache is divided into 256 partitions (colors). All test machines run Ubuntu 22.04. The Intel machine uses the Linux kernel 5.15.0, and the AMD EPYC machines use the Linux kernel 6.8.0.

Under the mitigation policy of MAPAlloc, our results show that a Prime+Probe attacker cannot build eviction sets for the target set on any test machine. We validate the effectiveness of this mitigation by repeating the building process 100 times. None of these repetitions found an eviction set. On AMD EPYC 9124, each eviction address must have the same color as the target, chosen from 256 possible colors. As a proof-of-concept to demonstrate the isolation of coloring, we successfully build eviction sets by refining allocations to reside in the same slice and set as the victim variable.

4.2 Mitigating DRAMA and Rowhammer

In this section, we demonstrate the use of page coloring to prevent DRAMA and Rowhammer attacks. Our approach to preventing DRAMA uses reverse-engineered DRAM mapping functions to implement a page coloring scheme that avoids row conflicts between victim and attacker.

Layered page coloring can be employed, extending our approach to mitigate DRAMA or Rowhammer and cache-based side-channel attacks.

DRAMA. DRAMA attacks [45] exploit the conflict of the DRAM row buffer. The row buffer is shared within a DRAM bank; the bank to which a memory address belongs is determined by the DRAM mapping function. This function uses implementation-dependent physical address bits to map to a channel, DIMM, rank, and bank. We use open-source tooling [21] to reverse-engineer the mapping functions on Intel CPUs with different memory configurations. Similar to the cache-slice function, if bits within the range of a 4 kB page are used in the mapping function, MAPAlloc cannot rely on them to create coloring. Therefore, we only impose constraints on bits higher than 12.

Our goal in protecting against DRAMA attacks is to create two disjoint pools of addresses D and N. It must hold that for DRAM bank b

$$\forall a \in D, \text{bank}(a) = b \quad \text{and} \quad \forall a \in N, \text{bank}(a) \neq b$$

If these constraints are fulfilled, addresses in D and N map to different banks, thereby preventing row buffer conflicts.We can achieve the desired coloring by first allocating D, which is possible as we know the DRAM mapping function, and then blocking all addresses mapping to b for further allocations. In practice we choose $|N| = 4\,\text{kB}$ and $|D| = 1\,\text{MB}$. To evaluate our coloring scheme, we test it on multiple different machines as listed in Table 2. Our results show that independent of the DRAM mapping function, we can prevent row buffer conflict

Table 2: The result of our DRAM banks coloring isolation. ✗ denotes that the attacker fails to create row buffer conflicts.

CPU	Banks Mapping	Color Num.	After Coloring
Intel Core i3-5010U	$a_{14} \oplus b_{17}$, $a_{15} \oplus b_{18}$, $a_{16} \oplus b_{19}$	8	✗
Intel Core i9-9980HK	$a_{14} \oplus b_{18}$, $a_{15} \oplus b_{19}$, $a_{16} \oplus b_{20}$, $a_{17} \oplus b_{21}$	16	✗

for the address set N via coloring. Therefore, an attacker cannot mount DRAMA attacks on addresses in N, as they cannot access addresses aliasing to the same row buffer.

Rowhammer. Page coloring mitigations against Rowhammer have been proposed in previous work [37, 32]. One mitigation idea implemented in [32] is to add *guard rows* next to rows one wants to protect against Rowhammer. More precisely, one computes the row index r_v of victim data via its physical address and then adds n guard rows at $r_v \pm n$. The number of guard rows depends on a DRAM-dependent parameter called the blast radius [33]. The blast radius is the maximal distance from a hammered row at which bit flips are reliably observed. MAPAlloc can easily add the guard rows by adding constraints that only allow sensitive rows to be n rows away from attacker-controlled rows. If implemented naively, this approach drastically reduces the available memory depending on the choice of n. However, the memory overhead can be reduced if the guard rows are filled with data that is not sensitive to bit flips. Furthermore, as MAPAlloc is implemented in the kernel, we can transparently apply error correction to the guard rows. Errors in the guard rows can be detected and corrected by the error correction code, while sensitive rows are protected by the fact that they are out of the blast radius. Similar layered mitigations using error correction [26, 10] have already been explored in related work.

A related approach [37] proposes subarray groups to partition the DRAM more efficiently. However, we do not implement this approach because MAPAlloc currently does not support hypervisor-based applications, and subarray groups only make sense in a hypervisor context.

4.3 Prime+Probe with Incomplete Non-linear Slice Function

While linear cache-slice functions have been thoroughly explored and reverse-engineered [53, 39, 24, 36, 20], non-linear cache-slice functions remain limited in availability. Although they have been reverse-engineered for some CPUs [22, 66, 40, 15], the results are often incomplete because the functions only work for systems with limited memory. For example, the cache-slice functions for Intel Coffee Lake and Alder Lake as reported by Gerlach et al. [15] only work for systems where the highest physical address does not have any bits above bit 31 set. Thus, this limits the applicability to systems with less than 8 GB of memory.

For prototyping attacks, MAPAlloc can be used to artificially limit the available physical memory of the system. Thus, MAPAlloc ensures that the incomplete slice function is still valid for all allocated pages. We evaluate this by

mounting a Prime+Probe attack on an Intel Xeon E-2176M (Coffee Lake) with 6 cores with the slice function of Gerlach et al. [15]. We rely on MAPAlloc to restrict the physical memory for attacker and victim to 2 GB to ensure we can always apply the slice function.

We target the AES T-Table implementation in OpenSSL. To align with prior research, we use OpenSSL 1.0.1e [47, 14, 17, 34]. We base our implementation on the one from Gruss et al. [17] but replace the cache-slice function and apply MAPAlloc to restrict the memory. With these adaptions, we recover, on average, more than 97 % of the key correctly. This result shows that MAPAlloc helps to test microarchitectural attacks that would otherwise only work if the system is physically changed, i.e., if DRAM is physically removed.

5 Evaluation

In this section, we evaluate the functionality and performance of MAPAlloc across architectures.

5.1 Functionality

We test the functionality of MAPAlloc on various microarchitectures among x86, ARM, and RISC-V architectures. Table 3 lists the machines used for testing the proof-of-concept constraint outlined in Listing 1. The slice functions and DRAM addressing functions used in case studies are determined via open-source reverse-engineering tools [45, 15].

In our tests, we first deploy the memory policy to constrain the page color for upcoming allocations. Then, we allocate memory via `mmap`, global variables, and heap variables to verify if their physical addresses fulfill the specified constraints. Our results demonstrate that MAPAlloc can manage memory allocation and maintain page coloring on all tested machines.

5.2 Performance

In the following section, we evaluate the performance of MAPAlloc.

Pre-Allocation When initializing, MAPAlloc pre-allocates pages in kernel mode to avoid expensive search operations at runtime. The cost of this operation primarily depends on the number of pages we consider and check against our constraints. In our implementation, this number is set to 87.5 % of the system's unused memory. For a freshly booted system, the pre-allocation time depends mainly on the amount of available DRAM. On an AMD EPYC 9124 with 16 GB of DRAM, pre-allocation takes roughly 4.5 s. On the Intel Core i3-5010U with only 8 GB of DRAM, it takes about 1.8 s. Since pre-allocating pages with MAPAlloc is a one-time initialization effort, it does not directly influence the runtime of user applications. We confirm this by running the popular 7-zip LZMA data compression benchmark [42] alongside another coloring process that pre-allocates pages. After 10 iterations, the average impact remains at 0.4 %.

Table 3: CPUs tested for MAPAlloc.

CPU	μarch	Release	Last-level Cache	Known Functions	
				Slice	DRAM
Intel Core i5-2520M	Sandy Bridge	2011	Inclusive	●	●
Intel Core i3-5010U	Broadwell	2015	Inclusive	●	●
Intel Xeon E3-1505M	Skylake	2015	Inclusive	●	●
Intel Xeon E-2176M	Coffee Lake	2018	Inclusive	●	●
Intel i3-8130U	Kaby Lake R	2018	Inclusive	●	●
Intel Celeron N4500	Jasper Lake	2021	Non-inclusive	●	●
AMD EPYC 7252	Rome	2019	Non-inclusive	●	○
AMD EPYC 9124	Genoa	2023	Non-inclusive	◐	○
Broadcom BCM2711	Cortex-A72	2019	Inclusive	N/A	○
Allwinner D1	RISC-V C906	2021	N/A	N/A	○

The icons represent the knowledge of a function. ● : known or found by open-sourced tools [45, 15], ○ : unknown, ◐ : discovered in this work

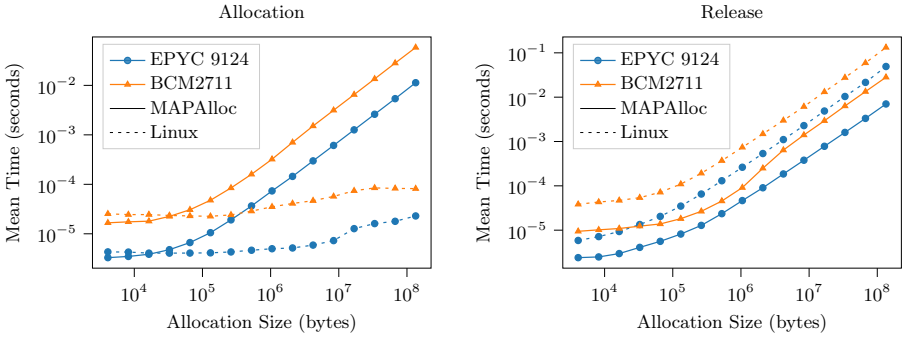

Fig. 3: Mean time to allocate and release memory blocks of different sizes with MAPAlloc and Linux ($n = 4096$ per sample, Linux v6.8.0). Less is better.

Memory Allocation To assess the performance of our memory allocator, we measure the mean time required for allocating and releasing differently-sized mappings. The results of this evaluation for the AMD EPYC 9124 (x86_64) and the Broadcom BCM2711 (ARMv8) are shown in Figure 3. For small allocations of 8 or fewer pages, MAPAlloc's page allocator performs slightly better than Linux. This is likely due to the low complexity of our implementation as compared to Linux's allocation routines. However, our prototype implementation does not implement bulk allocations but instead remaps every page individually. Furthermore, MAPAlloc zeroes the pages during initialization instead of during cleanup. Hence, we observe significantly longer allocation times than with Linux for larger memory blocks, with the time growing roughly proportionally to the allocation size. On the other hand, releasing pages with MAPAlloc is consistently faster than with Linux, up to 7 times faster. Note that when freeing pages, MA-

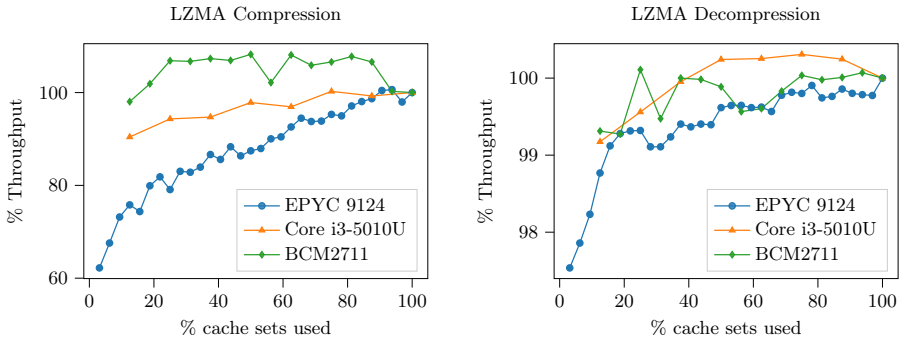

Fig. 4: Mean LZMA de/compression throughput with limited cache access, normalized by baseline performance (7-zip v23.01, 1 thread)

PAlloc only clears the user page table entries and adds the pages back to the pool. In contrast, Linux might run more expensive management tasks, such as attempting to merge smaller page buddies into larger ones.

Microarchitectural Effects One of the applications we propose for MAPAlloc is studying the runtime effects of page coloring schemes on software. To demonstrate MAPAlloc's effectiveness for this purpose, we investigate the 7-zip compression benchmark while preventing access to parts of the cache. See Figure 4 for the results of this evaluation. As expected, we observe a decline in throughput on the AMD EPYC 9124 and Intel Core i3-5010U, as we reduce the number of available cache sets. This effect is most pronounced in the compression benchmark. For example, restricting the available cache sets to only 512 out of 16384 on the AMD EPYC 9124 degrades the compression throughput to 62.2 % of the throughput with the full cache. While we also observe performance degradation in the decompression benchmark, this is less significant, with the throughput being reduced to only 97.5 %. Hence, we conclude that LZMA compression on x86_64 significantly benefits from a large cache, whereas this is not necessarily the case with LZMA decompression.

6 Discussion

In this section, we discuss the applicability to virtual machines, hardware mechanisms with similar capabilities, implementation limitations, and related work.

6.1 Virtual Machines

Our implementation of MAPAlloc is limited to non-virtualized environments. MAPAlloc neither supports running in the hypervisor to constrain virtual machines nor inside virtual machines. Conceptually, nothing hinders implementing

MAPAlloc in a hypervisor to apply constraints to entire virtual machines. However, similar approaches for virtual machines have already been implemented by prior work [61]. Therefore, we leave integrating support for arbitrary memory constraints into such hypervisors for future work.

Running MAPAlloc inside a virtual machine is more challenging. While MAPAlloc does run inside a virtual machine, it is functionally inoperable. The reason is that physical addresses inside a virtual machine, so-called guest physical addresses, are not actual physical addresses. Guest physical addresses are virtual addresses on the host translated to real physical addresses using extended page tables. Thus, the virtual machine does not know physical addresses. Constraints enforced by MAPAlloc in virtual machines degenerate to constraints on virtual addresses and can not be used to prototype defenses or attacks. This limitation can be overcome by adding a hypervisor part to MAPAlloc that could provide the actual physical address to MAPAlloc running inside the virtual machine.

6.2 Hardware Mitigations

Numerous hardware-based isolation techniques offer an orthogonal alternative to software approaches like MAPAlloc. For instance, Intel's Cache Allocation Technology (CAT) partitions the cache structure among different threads, providing isolation. Intel CAT was considered as a potential mitigation against cache side-channel attacks [35]. However, recent work [65, 43] has demonstrated its inefficiency. On AMD EPYC server CPUs, the new cache range reservation feature [1] enables the hypervisor to lock specific L3 cache ways for a designated system memory range. Although this feature was not originally intended for security purposes, it effectively isolates the specified system memory range from the rest of the memory in the L3 cache. Configuring this range involves two model-specific registers shared across an entire core complex, which limits its application to multiple processes. Ultimately, neither Intel nor AMD provides a hardware feature that mitigates DRAM attacks like DRAMA.

6.3 Related Work

Related work on partitioning memory, also called cache coloring, has been explored for a wide range of applications. Initially, cache coloring aimed to improve performance by optimizing cache utilization [28, 7, 58]. These approaches ensure a good cache hit rate for frequently used memory pages. Security applications of memory coloring have been explored by Hofmann et al. [19]. These applications mainly include mitigating cache-based side channel attacks [67, 54]. Similar to cache coloring, separating DRAM banks into multiple domains has been proposed as a Rowhammer mitigation in previous work [37, 32].

6.4 Implementation Limitations

In the following, we discuss limitations specific to our implementation of MAPAlloc. We note that overcoming these limitations is merely an engineering effort.

Corner Cases. Our implementation of MAPAlloc considers the common cases of memory allocation in the Linux kernel. However, due to the complexity of the Linux kernel, we are likely missing corner cases that are not handled. As MAPAlloc is not designed as a security mechanism but rather as a rapid prototyping framework, we consider such corner cases as unproblematic.

Exempted Pages. While MAPAlloc can enforce constraints on most pages, there are some pages where constraints cannot be enforced. We identify two categories of such pages. First, kernel pages mapped to the user space, such as vDSO [13]. As we do not enforce constraints on kernel pages, these pages are entirely out of scope for MAPAlloc. Second, implicit or explicit shared memory, if not all applications sharing the memory have compatible constraints. Such shared memory can also happen without a developer knowing, e.g., if the operating system uses page deduplication [57].

7 Conclusion

We introduced MAPAlloc, a versatile, cross-architecture framework for efficiently prototyping defenses and attacks on Linux that require control over physical memory allocations. Using a simple domain-specific language, MAPAlloc enables precise control over physical memory allocation across x86, ARMv8, and RISC-V architectures. MAPAlloc enables rapid implementation and evaluation of mitigations, such as page coloring. Additionally, MAPAlloc allows for quickly extending such techniques, as we show with layered page coloring, which significantly expands the number of cache colors from 32 to 256 on modern CPUs. We achieve this increase in isolation domains by reverse-engineering AMD Zen 4's cache indexing function and combining it with our reverse-engineered cache-slice function. By simplifying and accelerating the prototyping process, MAPAlloc bridges the gap between theoretical concepts and practical implementations, helping to improve the robustness of system security.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their valuable feedback and suggestions that helped improve the paper.

References

1. AMD64 Architecture Programmer's Manual (2024)
2. Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T.: ANVIL: Software-based protection against next-generation Rowhammer attacks. ACM SIGPLAN Notices (2016)
3. Bennett, T., Saroiu, S., Wolman, A., Cojocar, L.: Panopticon: A complete in-dram rowhammer mitigation. In: Workshop on DRAM Security (DRAMSec) (2021)

4. Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R.: CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In: USENIX Security Symposium (2017)
5. Bray, B.K., Lunch, W.L., Flynn, M.J.: Page allocation to reduce access time of physical caches (1990), http://i.stanford.edu/pub/cstr/reports/csl/tr/90/454/CSL-TR-90-454.pdf
6. Brumley, D., Boneh, D.: Remote Timing Attacks Are Practical. In: USENIX Security Symposium (2003)
7. Bugnion, E., Anderson, J.M., Mowry, T.C., Rosenblum, M., Lam, M.S.: Compiler-directed page coloring for multiprocessors. ACM SIGPLAN Notices (1996)
8. Corbet, J.: Some kernel memory-allocation improvements (2015), https://lwn.net/Articles/658081/
9. Corbet, J.: Remote per-CPU page list draining (2022), https://lwn.net/Articles/884448/
10. Dio, A.D., Koning, K., Bos, H., Giuffrida, C.: Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In: NDSS (2023)
11. Easdon, C., Schwarz, M., Schwarzl, M., Gruss, D.: Rapid Prototyping for Microarchitectural Attacks. In: USENIX Security (2022)
12. Frigo, P., Vannacci, E., Hassan, H., van der Veen, V., Mutlu, O., Giuffrida, C., Bos, H., Razavi, K.: TRRespass: Exploiting the Many Sides of Target Row Refresh. In: S&P (2020)
13. Frysinger, M.: vdso(7) — linux manual page (2024)
14. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. Journal of Cryptographic Engineering (2016)
15. Gerlach, L., Schwarz, S., Faroß, N., Schwarz, M.: Efficient and Generic Microarchitectural Hash-Function Recovery. In: S&P (2024)
16. Gerlach, L., Thomas, F., Pietsch, R., Schwarz, M.: A Large-Scale Rowhammer Reproduction Study Using the Blacksmith Fuzzer. In: ESORICS (2023)
17. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA (2016)
18. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium (2015)
19. Hofmann, J., Fournet, C., Köpf, B., Volos, S.: Gaussian elimination of side-channels: Linear algebra for memory coloring. In: ACM CCS (2024)
20. Hund, R., Willems, C., Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P (2013)
21. IAIK: DRAMA Reverse-Engineering Tool and Side-Channel Tools (2016), https://github.com/IAIK/drama
22. Inci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898 (2015)
23. Intel Corporation: Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations (2020), https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html
24. Irazoqui, G., Eisenbarth, T., Sunar, B.: Systematic reverse engineering of cache slice selection in intel processors. In: Euromicro Conference on Digital System Design (2015)
25. Jin, J., McMurtry, E., Rubinstein, B.I.P., Ohrimenko, O.: Are We There Yet? Timing and Floating-Point Attacks on Differential Privacy Systems. In: S&P (2022)

26. Juffinger, J., Lamster, L., Kogler, A., Eichlseder, M., Lipp, M., Gruss, D.: Csi: Rowhammer-cryptographic security and integrity against rowhammer. In: IEEE S&P (2022)
27. Juffinger, J., Lamster, L., Kogler, A., Eichlseder, M., Lipp, M., Gruss, D.: Csi: Rowhammer-cryptographic security and integrity against rowhammer. In: IEEE S&P (2023)
28. Kessler, R.E., Hill, M.D.: Page placement algorithms for large real-indexed caches. TOCS (1992)
29. Kim, T., Peinado, M., Mainar-Ruiz, G.: StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: USENIX Security Symposium (2012)
30. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: ISCA (2014)
31. Kocher, P.C.: Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO (1996)
32. Konoth, R.K., Oliverio, M., Tatar, A., Andriesse, D., Bos, H., Giuffrida, C., Razavi, K.: ZebRAM: Comprehensive and compatible software protection against rowhammer attacks. In: OSDI (2018)
33. Lang, Z., Jattke, P., Marazzi, M., Razavi, K.: Blaster: Characterizing the blast radius of rowhammer. In: Workshop on DRAM Security (DRAMSec) (2023)
34. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium (2016)
35. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: HPCA (2016)
36. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P (2015)
37. Loughlin, K., Rosenblum, J., Saroiu, S., Wolman, A., Skarlatos, D., Kasikci, B.: Siloz: Leveraging dram isolation domains to prevent inter-vm rowhammer. In: SOSP (2023)
38. Marazzi, M., Solt, F., Jattke, P., Takashi, K., Razavi, K.: Rega: Scalable rowhammer mitigation with refresh-generating activations. In: S&P (2023)
39. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID (2015)
40. McCalpin, J.D.: Mapping addresses to l3/cha slices in intel processors. Tech. rep. (2021)
41. OpenSSL: OpenSSL: The Open Source toolkit for SSL/TLS (2019), `http://www.openssl.org`
42. Pavlov, I.: 7-zip (2023), `https://7-zip.org/`, v23.01
43. Pawel Wieczorkiewicz and Rodrigo Branco and Ben Lee: On the Effectiveness of Intel's CAT as a Side-Channel Mitigation Technology (2024), `https://langsechq.gitlab.io/spw24/papers/LangSec2024-Branco-CAT-paper.pdf`
44. Percival, C.: Cache Missing for Fun and Profit. In: BSDCan (2005)
45. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium (2016)
46. Pornin, T.: BearSSL: A smaller SSL/TLS library (2022), `https://www.bearssl.org`

47. Purnal, A., Turan, F., Verbauwhede, I.: Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS (2021)
48. Qiao, R., Seaborn, M.: A New Approach for Rowhammer Attacks. In: International Symposium on Hardware Oriented Security and Trust (2016)
49. Qureshi, M.K.: CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: IEEE MICRO (2018)
50. Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H.: Flip feng shui: Hammering a needle in the software stack. In: USENIX Security Symposium (2016)
51. Saileshwar, G., Qureshi, M.: MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design. In: USENIX Security Symposium (2021)
52. Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., Mangard, S.: KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS (2018)
53. Seaborn, M.: L3 cache mapping on Sandy Bridge CPUs (April 2015), `http://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html`, retrieved on June 26, 2015
54. Shi, J., Song, X., Chen, H., Zang, B.: Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In: DSN-W (2011)
55. Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., Yarom, Y.: Robust Website Fingerprinting Through The Cache Occupancy Channel. In: USENIX Security Symposium (2019)
56. Sun, K., Branco, R., Hu, K.: A New Memory Type Against Speculative Side Channel Attacks (2019)
57. Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Memory Deduplication as a Threat to the Guest OS. In: EuroSys (2011)
58. Taylor, G., Davies, P., Farmwald, M.: The tlb slice—a low-cost high-speed address translation mechanism. In: ISCA (1990)
59. The Mbed TLS Contributors: Security (2024), `https://mbed-tls.readthedocs.io/en/latest/project/long-term-plans/#security`
60. Vila, P., Köpf, B., Morales, J.: Theory and Practice of Finding Eviction Sets. In: S&P (2019)
61. Volos, S., Fournet, C., Hofmann, J., Köpf, B., Oleksenko, O.: Principled microarchitectural isolation on cloud cpus. In: ACM CCS (2024)
62. Weber, D., Thomas, F., Gerlach, L., Zhang, R., Schwarz, M.: Indirect Meltdown: Building Novel Side-Channel Attacks from Transient Execution Attacks. In: ESORICS (2023)
63. Werner, M., Unterluggauer, T., Giner, L., Schwarz, M., Gruss, D., Mangard, S.: ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: USENIX Security Symposium (2019)
64. wolfSSL: wolfSSL: Embedded TLS Library (2023), `https://www.wolfssl.com/`
65. Yan, M., Sprabery, R., Gopireddy, B., Fletcher, C., Campbell, R., Torrellas, J.: Attack directories, not caches: Side channel attacks in a non-inclusive world. In: S&P (2019)
66. Yarom, Y., Ge, Q., Liu, F., Lee, R.B., Heiser, G.: Mapping the Intel Last-Level Cache. Cryptology ePrint Archive, Report 2015/905 (2015)
67. Ye, Y., West, R., Cheng, Z., Li, Y.: Coloris: a dynamic cache partitioning system using page coloring. In: PACT (2014)
68. Zhang, R., Kim, T., Weber, D., Schwarz, M.: (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In: USENIX Security (2023)

⟨*expression*⟩ ::= ⟨*term*⟩
  | ⟨*expression*⟩ `||` ⟨*term*⟩

⟨*term*⟩ ::= ⟨*factor*⟩
  | ⟨*term*⟩ `&&` ⟨*factor*⟩

⟨*factor*⟩ ::= ⟨*bitwise-term*⟩
  | ⟨*factor*⟩ `|` ⟨*bitwise-term*⟩

⟨*bitwise-term*⟩ ::= ⟨*bitwise-factor*⟩
  | ⟨*bitwise-term*⟩ `^` ⟨*bitwise-factor*⟩

⟨*bitwise-factor*⟩ ::= ⟨*bitwise-shift*⟩
  | ⟨*bitwise-factor*⟩ `&` ⟨*bitwise-shift*⟩

⟨*bitwise-shift*⟩ ::= ⟨*arithmetic*⟩
  | ⟨*bitwise-shift*⟩ `«` ⟨*arithmetic*⟩
  | ⟨*bitwise-shift*⟩ `»` ⟨*arithmetic*⟩

⟨*arithmetic*⟩ ::= ⟨*term1*⟩
  | ⟨*arithmetic*⟩ `+` ⟨*term1*⟩
  | ⟨*arithmetic*⟩ `-` ⟨*term1*⟩

⟨*term1*⟩ ::= ⟨*factor1*⟩
  | ⟨*term1*⟩ `*` ⟨*factor1*⟩
  | ⟨*term1*⟩ `/` ⟨*factor1*⟩
  | ⟨*term1*⟩ `%` ⟨*factor1*⟩

⟨*factor1*⟩ ::= ⟨*primary*⟩
  | `~` ⟨*factor1*⟩

⟨*primary*⟩ ::= ⟨*literal*⟩
  | `(` ⟨*expression*⟩ `)`

⟨*relational-expression*⟩ ::= ⟨*expression*⟩ `==` ⟨*expression*⟩
  | ⟨*expression*⟩ `!=` ⟨*expression*⟩
  | ⟨*expression*⟩ `<` ⟨*expression*⟩
  | ⟨*expression*⟩ `>` ⟨*expression*⟩
  | ⟨*expression*⟩ `<=` ⟨*expression*⟩
  | ⟨*expression*⟩ `>=` ⟨*expression*⟩

⟨*literal*⟩ ::= ⟨*number*⟩
  | ⟨*boolean*⟩

⟨*number*⟩ ::= ⟨*digit*⟩+

⟨*boolean*⟩ ::= `true`
  | `false`

⟨*digit*⟩ ::= `0-9`

Fig. 5: Grammar for the memory constraint DSL.