Non-Fusion Based Coherent Cache Randomization Using Cross-Domain Accesses

Kartik Ramkrishnan
University Of Minnesota, Twin Cities
Minneapolis, Minnesota, USA

Stephen McCamant
University Of Minnesota, Twin Cities
Minneapolis, Minnesota, USA

Antonia Zhai
University Of Minnesota, Twin Cities
Minneapolis, Minnesota, USA

Pen-Chung Yew
University Of Minnesota, Twin Cities
Minneapolis, Minnesota, USA

ABSTRACT

Randomization has proven to be a effective defense against conflict-based side-channel attacks in a shared cache. It improves security by assigning a unique randomization scheme to each security domain, e.g., though a different hashing function. However, if two domains have shared data, the domains must be fused in order to guarantee correctness (i.e., data coherence). Such *domain fusion* significantly reduces the effectiveness of randomization and weakens its security protection.

We propose *randomization with sharing* (RAWS), which enables secure cross-domain accesses while enforcing cache coherence (and thus data coherence). Based on RAWS, we design a *non-fusion based inter-domain coherence protocol* (NF-IDCP). NF-IDCP enables cache coherence by looking up and flushing multiple cache lines associated with shared-writable data during their cross-domain accesses. Furthermore, NF-IDCP uses *constant-delay banking* to securely reduce the latency of the cache line flushes. We also use a secure tag-based filter (STF) to reduce flush costs, for example, by explicitly storing the exact cache locations to be flushed.

The security evaluation shows that conflict attacks on the optimized NF-IDCP structures cannot leak conflict observations at a meaningful rate. Attack simulations using CacheFX demonstrate that domain fusion significantly retards the protection provided by randomization schemes. Performance overhead of SPECrate 2017 and PARSEC 3.0 benchmarks is evaluated on ZSim, a microarchitectural simulator. To study the performance impact on realistic workloads, such as Firefox, Chromium and X Server, we use a cache simulator built on top of PANDA, a full-system emulator. Across all configurations, the average performance overhead is less than 5%, and the hardware overhead is less than 3% compared to a domain-fused randomization.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3645011

KEYWORDS

Cache, Side-channel, Randomization, Sharing, Coherence

ACM Reference Format:
Kartik Ramkrishnan, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2024. Non-Fusion Based Coherent Cache Randomization Using Cross-Domain Accesses. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3634737.3645011

1 INTRODUCTION

Side-channel attacks, especially on the last-level cache, have continued to grow in significance due to their applicability in a wide-range of modern processors [9, 18]. One of the most important classes of side-channel attacks is the *conflict-based* side-channel, which attempts to leak information about access patterns of security-critical memory addresses via a cache interference pattern [23, 25, 33, 35, 38, 52, 57, 58, 73, 75] across security domains (mutually distrusting processes, parts of processes, threads, virtual machines or other).

Researchers have developed a novel countermeasure to conflict-based side-channel attacks (see §2), known as *address-set randomization* [31, 40, 50, 53, 60, 62, 63, 69], which mitigates the ability of attackers to carry out conflict-based attacks, especially via the shared last-level cache. This allows victim domains to scatter their cache lines in random cache sets selected via a cryptographic hashing of the cache line address, thus mitigating conflict-based attacks. Randomization schemes protect two classes of memory addresses, namely, *private* addresses and *shared-read-only* addresses. The former refers to addresses that are not reachable by more than one security domain, and the latter refers to addresses that are reachable by two or more security domains, but only by using reads. The third class includes *shared-writable* memory addresses. They are unprotectable by default because their information can be leaked between domains without relying on side-channels merely by observing the results of reads/writes.

State-of-the-art randomization schemes use a different randomization function for each security domain (see §2.2). For *shared-writable* addresses, aliasing will occur in the shared last-level cache (LLC), which means that each security domain will not know the whereabouts of those cache lines used by other security domains. Thus, there will be many uncoordinated per-domain write accesses leading to a loss of global coherence invariants, such as *single-write-multiple-read* (SWMR) and *data-invariance* (see §2.2.1). To prevent aliasing, randomization schemes assume that the privileged system software can tell them which addresses are shared-writable, so that

all domains can use a common hashing function on them. However, existing system software may not have such functionality across domains (see §2.2.2).

In the absence of software support, the only recourse is to use *domain fusion*, where programs in different security domains are re-assigned to the same security domain. This allows them to use the same randomization function and eliminates aliasing. However, it gives up on *occupancy-attack resilience* and *flush-attack resilience* (see §2.2), two prominent security benefits of randomization schemes. While flush-attack resilience can be recovered using orthogonal techniques like First-Time-Miss [40, 49], there is no known way to recover occupancy-attack resilience. The above domain fusion problem has significant practical implications because many programs such as Google-Chrome [10], Chromium [14], Firefox [11] and X Server [13] have deep ties with shared-memory *inter-process communication* (IPC), including IPC between untrusting processes (see §2.2.4). Thus, the following question arises:

Can randomization be augmented to support non-fusion based data coherence in order to retain its security benefits?

To eliminate aliasing, we propose a global cache coherence principle, *randomization with sharing* (RAWS). The main idea of RAWS is to perform *cross-domain* accesses to find all copies of cache lines in the LLC. In Figure 1, we illustrate why multi-domain randomization is unable to support cache coherence, and how RAWS can help. The left-half of the figure shows the state-of-the-art (SOTA) non-fused randomization. An access to the shared cache is carried out by *domain 1* for a cache line address that is *dirty* and is being accessed by *domain 2* (highlighted in red). It is different from the location determined by *randomization function 1* (random 1) because domain 2 uses a different randomization function, leading to a loss of global coherence via aliasing. In the right-half of the figure, we show how RAWS augments non-fused randomization to carry out *two* lookups instead of one, by also employing *randomization function 2* (random 2). Thus, the dirty cache line used by *domain 2* can be found, and the cache coherence protocols can be made aware of each other. For a secure and practical use of the RAWS principle in a generalized target environment, we propose to use cross-domain cache flushes due to their ready availability in most caches. For reducing the number of cross-domain flushes, we propose to use banking/parallelization or filtering (see §2.3).

Accordingly, we leverage existing *invalidation-based* coherence protocols, and trigger *flushes* to maintain coherence invariants across processor groups used by different domains (see §3.1). For maintaining inter-domain SWMR, each cache line is augmented with an inter-domain coherence protocol *read-write bit* (NF-IDCP-RW), which is set if there is a possibility of a *write* by the domain. Setting the NF-IDCP-RW bit will trigger a flush in the coherence protocols of other domains if they have a cache line corresponding to the same cache line address, thus preventing concurrent reads or writes from other domains to the cache line address. This is as required by the inter-domain SWMR invariant. If there is no cache line in the reader domain (i.e., a read miss), then it will trigger a flush in the coherence protocols of other domains to writeback the content of any dirty NF-IDCP-RW cache line with the same cache line address. It can then proceed with the read of the latest data content (data invariance). We present a generic state-diagram of a coherence protocol that highlights the NF-IDCP flushes in §3.

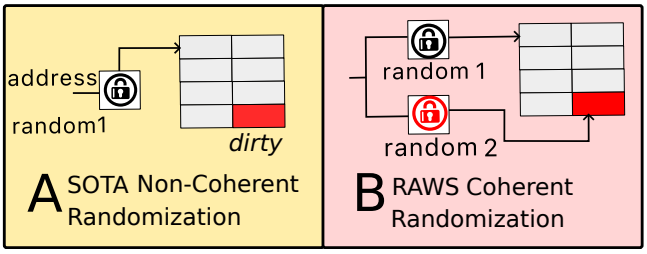


Figure 1: RAWS enables multi-domain lookup in order to facilitate global cache coherence in randomization schemes.

To efficiently support the above inter-domain coherence actions, we propose two optimizations to enhance the performance securely. The first optimization is *parallelization* of the cross-domain accesses (lookups or flushes), which will need to be done for the two invariants. This can be achieved using existing approaches such as banking [20, 51, 78] (see §4.1). The second optimization is to use a *secure tag-based filter* (STF) to reduce the number of cross-domain lookups for flushes. For the SWMR flushes, the STF keeps a counter, which is pointed to by all the corresponding cache lines in the LLC. Upon the need of flushing, it sets the invalid bit in this counter and defers the actual flushes for later. The flushes will be done off the critical path and the counter will be decremented to track their progress, finally, de-allocated when all the cache lines are flushed. For the data invariance flushes, the STF will store the SDID of any cache line that has the IDCP-RW state. Hence, it will take only one lookup to locate those cache lines for flushing. A crucial security issue is that any set of cache line addresses in the cache should always be represented in the STF, which requires it to behave like a fully-associative structure. We achieve this fully-associative behavior using a combination of a main storage region and an overflow storage region. We discuss the STF filter in §4.2.

We implement NF-IDCP randomization schemes on each cache slice of a multicore processor. Our security evaluation of constant-time banking and STF via simulation shows that they are not helpful for an attacker (see §5.1). We also analyze eviction-set, occupancy and flush-based attacks and conclude that the attacker is not aided by the added NF-IDCP mechanism (see §5.2). Lastly, we demonstrate via attack simulation on encryption libraries that domain fusion significantly improves the effectiveness of occupancy-based attacks (more than 70% more effective) compared to non-fused randomization (see §5.2).

For the performance evaluation, we simulate two different scenarios. The first scenario is highly intensive in terms of shared data usage, which stresses the flush-actions of NF-IDCP and magnifies their performance overheads. We use PARSEC 3.0 [77] and real-world workloads that include Firefox, Chromium and X Server (see §6.2) workloads. The infrastructure uses ZSim [55] and PANDA-based [28] simulators, respectively, for the above scenario. The second scenario has a low amount of data sharing, in which we should ideally retain most of the performance of non-coherent randomization. We simulate the SPEC2017Rate [24] benchmarks on ZSim for the second scenario. The performance overhead in all cases is less than 5% compared to the domain-fused randomization schemes (see §6).

Contribution Summary: To the best of our knowledge, this is the *first* work that systematically addresses the cross-domain data sharing issue for cache randomization schemes. We make the following contributions.

(1) A new approach, *Randomization-With-Sharing* (RAWS), which enables global cache coherence in randomization schemes that allows cross-domain accesses without using domain fusion.
(2) A new non-fusion based inter-domain coherence protocol (NF-IDCP) that enables cache coherence among the multiple cache coherence protocols running in a randomized cache.
(3) A security evaluation via analysis and simulation that shows NF-IDCP can recover the security guarantees of the randomization schemes by preventing domain fusion.
(4) A performance evaluation which shows that the NF-IDCP-enhanced randomized cache has less than 5% overhead compared to domain-fused randomization. Hardware overheads are less than 3%.

The rest of the paper is organized as follows: §2 discusses side-channel attacks, state-of-the-art randomization strategies and domain fusion. §3 shows how NF-IDCP enables inter-domain coherence. §4 presents constant-latency banking and secure tag-based filters (STF) to optimize NF-IDCP. §5 performs a security evaluation of NF-IDCP. §6 does the performance evaluation on PARSEC 3.0 (significant amount of shared addresses), real-world workloads and the SPECrate 2017 (no shared addresses) benchmarks. §7 has discussion/related work and §8 concludes the paper.

2 SIDE-CHANNELS, RANDOMIZATION SCHEMES AND RAWS

In this section, we present relevant background for this paper, which includes conflict-based side-channel attacks (§2.1), randomization schemes (§2.2), coherence protocols (§2.2.1) and data sharing (§2.2.4). This will set the stage for the introduction of the randomization-with-sharing principle (RAWS) that enables the use of cross-domain accesses to maintain coherence. We present the targeted environment in which we intend RAWS to function, and its security, performance and hardware overhead (§2.3). We furthermore propose a secure non-fusion based inter-domain coherence protocol (NF-IDCP) as a way to apply the RAWS technique with minimal modifications to existing randomization schemes (§2.4). Lastly, we present background about banking (§2.4.1) and fully-associative structures (§2.4.2) as two approaches that can be adapted to help us securely reduce the latency and contention of NF-IDCP flushes.

2.1 Conflict-Based Side-Channels

A very important class of cache side-channels are the *conflict-based* side-channel attacks, which have been shown to be capable of leaking secret encryption keys. There are three major kinds of such attacks: the *conflict-set attacks*, the *occupancy-based attacks*, and the *shared memory flush-based attacks*.

The most general set-conflict attack is the PRIME + PROBE attack [33]. In this attack, the attacker attempts to evict all of the addresses in a cache set using an eviction set [38, 64]. This eviction set contains many addresses that map to the same set (PRIME). It then waits to see whether its cache lines are evicted by further accesses from the victim. Lastly, it tries to access its cache lines in the cache set, and look for cache misses (due to their higher access time). If any cache miss is detected, it means that there must have been victim accesses in the intervening time period (PROBE). There are also other variants of PRIME + PROBE that can speed up the rate of leakage [34, 35, 44, 45, 70, 71]. Some attacks use a cache set's replacement states to augment conflict-based attacks [23]. The occupancy attacks [58, 59] do not target any particular set, but try to observe the total evictions of a randomly selected pool of cache lines over a period of time. Instead of evictions, flushing instructions can also be used to explicitly eliminate cache lines, and to speed up conflict attacks [60].

The above conflict-set and occupancy attacks target either private memory or shared read-only memory. For shared read-only memory, cache line flushes can also be used to directly eliminate the corresponding cache lines for a faster rate of attack [32, 74].

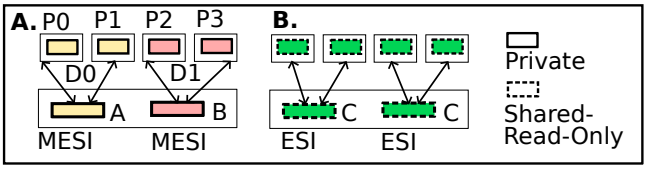


Figure 2: The per-domain coherence protocol used by existing randomization scheme MIRAGE (see §2.2).

2.2 Randomization Schemes Have A Domain Fusion Problem

Randomization schemes use address encryption to protect caches against conflict-based attacks [37, 46, 47, 50, 60, 62, 68]. The most secure randomization schemes [31, 53, 69] concatenate the address with the security domain ID before the encryption to create per-domain set mappings. Such address encryption can scatter addresses across the cache sets in a domain-sensitive manner. It substantially increases the difficulty of finding security-critical cache lines, thus mitigating conflict-based attacks. Randomization schemes, such as SassCache [31], can also limit the *coverage* of the cache for different security domains, making it impossible for attackers to touch victim cache lines. Randomization schemes are transparent to the applications and can scale to tens of security-domains. Randomization schemes can also protect memory addresses that are either *private* (accessible to only one security domain), or *read-only shared* (accessible to multiple security domains). Randomization does not protect *shared-writable* addresses (accessible to multiple security domains) due to its ability of direct (i.e., non-side-channel) transmission of information.

Some of the most recent CVEs related to conflict-based side-channel attacks are CVE-2023-32691 [7], CVE-2023-26557 [6], CVE-2023-26566 [5], CVE-2023-25000 [3] and CVE-2023-25332 [4]. We observe that randomization defenses won't work properly against web-browser based attackers [58], due to a domain fusion issue that we describe in §2.2.3. Hence, we need to augment randomization to function without domain fusion.

2.2.1 Per-Domain Coherence Protocol: We briefly describe the well known MESI [61] protocol in which there are four states, M,

E, S and I. They are short for *modified*, *exclusive*, *shared* and *invalid*. The *modified* state means that the cache line has the latest copy of the data and the copy in memory is stale. *Exclusive* means that the data is clean and that there is only one CPU that is currently using the cache line. *Shared* means that multiple CPUs may be using the cache line, and that the copy is clean (same as memory). *Invalid* means that the cache line does not have any usable data.

In the coherence protocols for an *inclusive* cache, there is always a copy of the cache line in the last-level cache if there are to be copies in the private caches. This cache line will be tagged with coherence metadata, such as the list of processor cores that may have a private copy of the cache line (also known as sharers). *Non-inclusive* caches will have a separate directory with the coherence metadata, instead of associating it with each cache line. To the best of our knowledge, state-of-the-art randomization schemes have all been implemented on *inclusive* caches [31, 53, 69], so our discussion is accordingly carried out from that viewpoint. However, it should also be possible to apply similar techniques, if a randomization scheme is implemented on a *non-inclusive* cache.

We show an example where a state-of-the-art randomization scheme, such as MIRAGE [53], ScatterCache [69] or SassCache [31] execute per-domain coherence protocols, in Figure 2 . There are two domains in this example, D_0 and D_1. Each domain has access to two cores. D_0 has access to P_0 and P_1. D_1 has access to P_2 and P_3. There are two scenarios presented here. In Scenario A, we have cache lines corresponding to private addresses A and B (solid outlined boxes). Scenario A's per-domain MESI protocol maintains coherence among P_0, P_1 and the LLC, each of which can have a copy of the cache line. Similarly, in Scenario B, we have a cache line C corresponding to a *shared read-only* address (dotted outlined boxes). Scenario B's per-domain protocol maintains coherence among P2, P3 and the LLC. The coherence states will be only ESI in this case because there will never be any write to shared read-only addresses.

2.2.2 Software Coherence: State-of-the-art randomization schemes, such as ScatterCache [69] and SassCache [31], speculate that privileged software can be augmented to transparently detect shared-writable addresses, which can be used to prevent domain fusion. To satisfy the above requirement, we need to make non-trivial changes to several data structures and subsystems in the operating system (OS) or other privileged software that can affect the shared-writable status of a page. This includes the metadata for each page frame to store security domain identifiers (SDIDs), the page-table contents (to store associated SDIDs) and memory related system calls (to update SDID metadata). The extra metadata may significantly increase the total memory consumption of the system, and changes to the software subsystems may significantly affect performance due to the extra page-tracking operations. Given the above complexity, it is also unknown whether transparency can be guaranteed in all cases, or whether application modifications are needed. Therefore, a thorough design and evaluation will be necessary to determine the feasibility of the software-based approach, which is beyond the scope of the paper. We have instead taken the orthogonal hardware route towards supporting cache coherence, which has the advantage of simplifying the software developers' life because it is transparent to their applications.

```
//gles2_cmd_decoder.cc//
//Consume GL commands from shared buffer//
5906   ... DoCommandsImpl( ... ) {
         -
5913     CommandBufferEntry* cmd_data =
5914     static_cast<CommandBufferEntry*>(buffer);
         -
5918     while (...
5919            ...) {
5920           const unsigned int size =
              cmd_data->value_header.size;
              - //Security checks, processing
5988     }
         -
6007   }

//gles2_cmd_format_autogen.h
//Shared region cast as struct TexImage3D
8537   struct TexImage3D{
         -
8549     void Init(GLenum _target,//commands
8550              GLint _level,
                 -
8558             uint32_t _pixels_shm_offset){
8559          SetHeader();
8560          target = _target;
             - //Write other fields
8570     }
         -
8590     uint32_t target; //buffer field
         -
8601   };
```

Listing 1: A simplified code snippet showing command buffer usage in a Chromium server process.

2.2.3 Domain Fusion: The code snippet of Listing 1 shows an example from the Chromium code base [1]. The privileged server process helps to run/emulate OpenGL commands received from a client, for which it implements a function called `DoCommandsImpl` whose argument is a pointer to a shared ring-buffer (also known as a 'command buffer'). The client is one of the renderer processes associated with the browser tabs. It may contain malicious code and therefore runs at a lower privilege level [2]. The top half of the code (lines 5906-6007) corresponds to the server process, which extracts commands from the command buffer. It carries out a security check for its validity, and then processes the commands in a `while` loop (lines 5918-5988).

The latter part of the code (lines 8537-8601) shows how the client modifies the command buffer content. The command buffer is cast as a structure (line 8537), and an `Init` function (lines 8549-8570) is invoked to modify the content of the buffer (8590-8601). One of the updated fields, `target`, is shown on line 8601. The update to it is shown on line 8560. The field `target` should be visible to the server process as part of the commands it receives from the client.

If a randomization scheme is applied, then the domains of the client and server would need to be fused. As mentioned in §1, fusion of the security domains leads to a lower security level. This is mainly due to the loss of protection against occupancy attacks and the loss

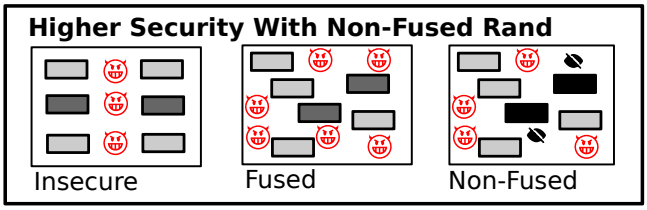


Figure 3: Randomization has resistance to occupancy attacks, but domain fusion eliminates this security benefit. Dark cache lines are security-sensitive cache lines, which are all hidden from the attacker (see §2.2).

of protection for the shared-read-only addresses. Figure 3 shows the security issue created due to domain fusion for occupancy-based attacks. We observe that the attackers can evict security-critical cache line addresses in the fused domain, whereas they are hidden in randomization without domain fusion because they use different randomization functions. The attacker can no longer flush out security-sensitive cache lines in the victim's domain.

First-Time-Miss (FTM): To prevent flush-based attacks, it has been suggested to disable user-space *clflush*-like instructions, or to add First-Time-Miss [40, 49] style defenses if different security domains are fused. In a First-Time-Miss defense, an injected timing-delay makes it impossible to detect cache lines shared between different security domains.

2.2.4 Other Data Sharing Examples. As discussed in §1, data sharing is very important in many applications such as Chromium. It also exists in browsers that are related to the Chromium code base such as Google Chrome [10]. To motivate our work further, here are some other popular applications that can also benefit from the proposed non-fused randomization. For example, Firefox uses shared memory IPC communication between untrusting processes, such as the privileged browser process (parent) and the unprivileged content process (child) [8, 11]. XServer [19] uses a pixel buffer [13], which is shared between a server and potentially untrusted clients. Qube OS [17] uses shared memory ring-buffers to enable communication among different virtual-machines (VMs) in the system. The PulseAudio server may also use a shared ring-buffer [16] to receive data from client processes using the `enable-memfd` option. The `memfd_create` system call is provided by Linux to facilitate *sealable* data sharing between untrusting processes [12].

2.3 RAWS To The Rescue

Targeted Software and Hardware Environment: All code running at a higher privilege level than userspace, e.g., in supervisor mode or above, is considered to be secure. The privileged software assigns different security domains to processes, virtual machines, threads and others with security domain identifiers (SDIDs) of its choosing. Different security domains do not trust each other but may yet share writable-data, as we had pointed out in §2.2.4. The hardware environment is a multicore processor with private and shared caches. Different security domains are assigned to different cores with private caches, but there is no restriction on concurrent use of the shared caches. The above environment is a good target because most of the data sharing examples that we mentioned in §2.2.4 do appear in such an environment. We can imagine that a laptop user could be running a web-browser with many concurrent tabs, or that it could be running on a virtual desktop on cloud server. It is also possible that the other applications could run in a cloud based scenario or a laptop/desktop personal computing scenario. In all these cases, our solution is applicable for a more secure system environment.

Targeted Properties of the Cache : Any modifications to the randomized cache should be *practical* and *secure*. We prescribe four properties to this end.

- $\mathcal{P}_1$: The extra inter-domain coherence actions should *transparently* restore coherence invariants.
- $\mathcal{P}_2$: The latency and contention overheads due to the added cache accesses should be as low as possible so that performance is not affected.
- $\mathcal{P}_3$: The hardware overheads need to be as low as possible so that our solution can be deployed in a practical setting.
- $\mathcal{P}_4$: Attack resilience of randomization schemes should be restored, in particular, against the three kinds of attacks discussed in §2.1.

Out Of Scope: All attacks that randomization is not intended to protect are beyond the scope of this work. This includes leakages through shared-writable memory addresses [36], stateless attacks [65, 66, 75] or non-cache side-channels [43]. For these attacks, orthogonal defense techniques can be used in conjunction with randomization [41, 42]. While beyond the scope of this work, we briefly discuss how some of these methods could be used in conjunction with our NF-IDCP solution (see §7).

2.4 Our Approach To Realizing The Targeted Cache Properties

We have already mentioned that a possible solution to domain fusion is to use the RAWS principle of cross-domain accesses. More specifically, we would like to do it using cache line flushes due to their availability in existing coherence protocols, which are usually invalidation based. Thus, it satisfies our first requirement of transparently supporting data coherence, property $\mathcal{P}_1$. However, doing so *securely* is complicated by the use of additional flushes. It is known that adding flushes in the LLC can help speed up the rate of conflict-based attacks (see §2.1). Fortunately, a straightfoward solution is that flushed cache lines persist in the cache, and behave like normal cache lines from a replacement perspective although their contents cannot be used. This will help us achieve our security property $\mathcal{P}_4$ by preventing new conflict-set attacks from becoming possible. We present the NF-IDCP protocol in the context of a generic invalidation-based coherence protocol in §3.

Näive application of RAWS can suffer from a higher latency and contention due to the need to lookup a significant number of cache locations concurrently during each cache access. Hence, we propose additional supporting hardware structures, which can either enable parallel accesses (see §2.4.1), or cut down on the number of flushes on the critical path (see §2.4.2). This will help us to achieve our performance targets $\mathcal{P}_2$. Towards this end, we present some background concepts that are helpful in the above pursuits, namely, *constant-delay banking* and *tag-based filtering*.

2.4.1 Constant-Delay Banking. Banking has long been used in memories to improve memory bandwidth [20]. Here, the cache is divided up into several *mini-caches* (i.e., banks), which can be accessed in parallel. These mini-caches are effectively connected by a bus, so that request and messages can be broadcast to all of them in parallel. We can leverage this infrastructure to carry out parallel lookups to multiple cache locations. The overheads of banking should be low because there is no additional storage overhead. The key change we make is to make the access time *constant*. Otherwise, the banking may reveal set mapping information that is otherwise hidden in the randomization. Hence, constant-delay banking can also help us to achieve the targeted properties $\mathcal{P}_3$ and $\mathcal{P}_4$. We discuss constant-delay banking in §4.

2.4.2 Tag-Based Filters . One way to filter out unnecessary cache lookups is to maintain metadata in the LLC about the cache line addresses represented inside. The metadata is tagged with the cache line address, and can tell us which cache locations need to be flushed precisely, or they can contain counters, which can help track deferred flushes. The above metadata approach makes sense if there are enough domains present to justify the extra costs. For example, for a 32-domain system, there will be 32 bits per cache line due to firt-time-miss (FTM) protections on domain-fused randomization. These bits can be eliminated because we prevent domain fusion. It helps to offset the hardware overhead of adding the above tag-based filter, helping us to satisfy the $\mathcal{P}_3$ hardware overhead requirement. One of the key security issues with the above tag-based filter is that the metadata needs to be housed in a fully-associative structure to prevent set over-subscriptions. We can achieve that by using a set-associative structure for the main region, an additional overflow region for tags that do not fit into the main region, thus helping our security target of $\mathcal{P}_4$. We discuss the performance enhancing functionality of the STF in §4.2.1, and the fully-associative housing in §4.2.2.

3 SECURELY ENABLING INTER-DOMAIN COHERENCE USING CACHE LINE FLUSHES

.

In this section, we present a state-machine for NF-IDCP that maintains cache coherence among security domains. It allows only one domain to have the read-write permission to a cache line at any given time (see §3.1, §3.2). We lastly show that this is a sufficient guarantee for the cache coherence invariants (see §3.3).

3.1 Coherence Protocol NF-IDCP Transitions

The **invalid** state ***NF-IDCP-I*** indicates that there is no valid cache line for that address in the security domain. This could either be because there is no copy of the cache line for that address, or because there are copies of the cache line but none of them has valid data in them. The **read-write** state ***NF-IDCP-RW*** indicates that only one per-domain protocol will have a copy of the cache line in the shared LLC, and in its private caches. Other security domains will not have any cache line copy in the shared LLC, or their private caches. The cache line may be *dirty* because writes may have occurred to the cache line. In the **read-only** state ***NF-IDCP-RO***, it is possible that multiple domains may have copies of the cache line in LLC and the their private caches, but none of them is dirty (i.e., written to).

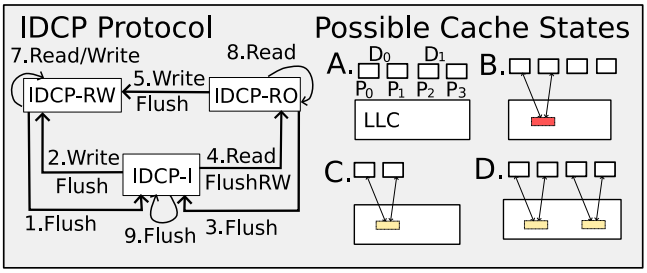


Figure 4: The states and transitions of the NF-IDCP state machine (left). An example (right) that shows how it maintains cache coherence between two security domains (see §3.2).

3.2 NF-IDCP Transitions

Figure 4 shows how the NF-IDCP protocol works. It includes three NF-IDCP states, namely, **NF-IDCP-I**, **NF-IDCP-RW** and **NF-IDCP-RO**. The transitions among the states are shown in the left half of the figure. For simplicity, we will not use the NF-IDCP prefix while describing the transitions.

Flush Transitions:

1. RW—>I: The transition occurs whenever there is a flush or an eviction on the cache line address in the RW state.

3. RO—>I: An eviction or a flush occurred on an RO cache line.

9. I—>I: This transition occurs if there is a flush or an eviction of the cache line, but the security domain does not have a valid copy of the cache line.

Transitions To RW Status:

2. I—>RW: The transition occurs when a *write-miss* on the cache line address occurs. The cache line copies in the other domains are flushed. As the result, it is the only domain that has the cache line with the *RW* state.

5. RO—>RW: If a domain already has the cache line(s) in the RO state, then a flush of the copies in all other-domain happens. After its completion, the cache line transitions to RW and the write can occur.

7. RW—>RW: This transition occurs if there are reads or writes from the same domain to an RW cache line.

Transitions To RO Status:

4. I—>RO: Upon a *read-miss*, all cache lines with the RW state are flushed from the cache. Then, an RO cache line can be fetched into the cache by the domain carrying out the read. Since no other dirty cache line is present, the data is up-to-date with the latest write.

8. RO—>RO: This transition occurs if there are reads to a cache line that a domain already has a copy of.

In the right half of Figure 4, it shows the different coherence states a cache line may be in. There are two security domains: D_0 and D_1. Each has access to two cores, P_0, P_1 for D_0 and P_2, P_3 for D_1. In Part A, it shows the I state as its initial state. No valid cache line is present in the LLC or elsewhere. Part B refers to a security domain having a cache line in the RW state. There is only one copy of the cache line in the LLC (red color), and there may be multiple copies in the private caches associated with the security domain D_0. The cache lines are allowed to be dirty. In Parts C and D, they show the two possibilities of the RO state. There may be one copy

of the cache line in the LLC (Part C), or there may be two copies of the cache line in the LLC (Part D). Both cache lines in Part D are clean (yellow color). Note that the MESI coherence protocol is run between the cores assigned to the same domain for their cache line copies in LLC.

3.3 Maintaining Coherence Invariants

We layout the argument below regarding why the SWMR and data-invariance are satisfied by the NF-IDCP.

A. Only One Domain Can Have RW Lines: The RW cache lines are only available to one security domain at a time. In order to prove this, we consider the lifetime of an RW cache line. An RW cache line is fetched into LLC when there is a *write miss* on the cache line (transition 2), or when there is a *write* to an RO cache line (transition 5). In both cases, all cache lines used by other domains are flushed out. Furthermore, when any other domain tries to access the cache line, they can only get the cache line once it has been flushed out from the domain with RW permission (transitions 2 and 4). Therefore, only one domain can have the RW cache line at any given time.

B. RO and RW Per-Domain Protocols Have The Latest Data: If a security domain has RO or RW cache lines, it means that it has the latest copies of those cache lines. When they are fetched, they are the latest copies because all other dirty copies must have been flushed out to the main memory (transitions 2 and 4). If any other domain tries to do a *write*, then it needs to remove the RO or RW cache lines being used by a different domain. Thus, the above property holds.

SWMR Property: If a *write* occurs, it will either be a cache miss (I), or happen on a cache line in the RO or the RW state. In the first two cases (I->RW or RO->RW transitions), there will be a flush of the cache lines in all other domains before the write can happen. This guarantees that no other domain has a copy of the cache line when the write takes place. Hence, SWMR property is maintained because only one domain has the write permission. In the third case (RW->RW), we already know from the *property A* that no other domain has a copy. Therefore, SWMR is maintained at the *inter*-domain level. *Intra*-domain SWMR will be maintained by the per-domain coherence protocol.

Data Invariance: If a *read* occurs on an already-present cache line, it could be associated with an I->RO, RO->RO or RW->RW transition. Since we know by the *prop*erty B that the obtained RO or RW cache lines always have the latest copy of the data, we satisfy the required data invariance property. In Appendix §B, we present a simple implementation of NF-IDCP on top of a MESI protocol for two domains, and the simulation results using the coherence protocol verification tool Murphi [27].

Takeaway 1

The NF-IDCP approach maintains coherence in the absence of domain fusion using cache line flushes, thus maintaining coherence invariants and the security guarantees of randomization.

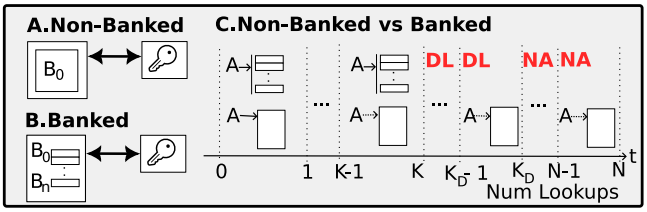


Figure 5: NF-IDCP with banked cache accesses. The banked accesses are faster to complete compared to a non-banked cache. (see §4.1).

4 SECURELY REDUCING INTER-DOMAIN COMMUNICATION LATENCY

In this section, we discuss ways to improve the performance of the NF-IDCP schemes, as per our targeted property of low performance overhead (see §2.3). The first optimization is secure parallelization and the second optimization is secure tag-based filtering (STF). The parallelization strategy (see §4.1) uses constant-delay banking to increase the number of concurrent accesses to the cache securely, which is useful for faster cross-domain flushes.

The STF approach (see §4.2) uses tagged metadata to reduce flush latency. It stores counters to facilitate lazy flushes of the RO cache lines in the LLC. It also stores precise domain-information to locate the RW cache lines in the LLC. The above metadata can be housed in two fully-associative structures, Dir_0 and Dir_1, for immunity against conflict-based attacks.

Lastly (see §4.2.2), we discuss the hardware design of the fully-associative structures inside the STF. It has a main region, which houses tags in a set-associative structure. It also has an overflow region, which hosts any tags that do not fit into the main region due to set oversubscription. Also, cache lines are moved out of the overflow region back into the main region, if there is space in the main region. Hence, the STF works like a fully-associative cache, which is secure against conflict-based attacks.

4.1 Secure Parallel Accesses

Caches already support well-established banking techniques to increase parallel accesses (see §2.4.1). In banking, the cache is divided into multiple sections known as *banks*, which can be accessed in parallel. The use of parallel cross-domain accesses improves the performance by reducing access latency. However, we do not want attackers to learn the relative locations of sets in different banks by measuring the access time to the cache, hence, a *constant latency* is imposed. The imposed constant latency should be the *expected* maximum access time as observed over a large number of accesses.

Figure 5 shows the scheme of our secure banked cache. Part A shows the non-banked cache, which has a single bank B_0. Part B shows n banks, $B_0 ... B_n$, with $\frac{1}{n}$ of the total sets in each bank. In both the parts, the encryption key in a box signifies the randomization functions that are associated with the cache.

In Part C, we also show an example lookup of the non-banked and the banked cache, by an address A. The domain for A can arbitrary (not shown). We assume that there are N domains in

this example, so there will N lookups, each using a different randomization function for each domain. In the first time step (0-1), the banked cache will access all n banks in parallel. Thus, upto n locations can be accessed in parallel. However, the non-banked cache can access only one location. This goes on K times, where $K < N$. At this point in the sequence, the banked cache would have completed all of its accesses to the N different locations. However, the non-banked cache has only completed K accesses. The next interesting point in the sequence is at the K_D^{th} access, which is the expected maximum time for the banked cache. At this time, the banked cache can securely declare the cache access to N locations as completed. Between time K and K_D, the banked cache will be idle. The extra delays for the constant time banking are indicated by the red DL letters. However, the non-banked cache has only accessed K_D locations. Hence, it will now continue the access sequence until it completes N accesses to the cache. The banked cache has already completed its lookups indicated by the red NA letters.

4.2 Secure Tag-Based Filters (STF)

Hardware Modifications: The hardware modifications are shown in Figure 6. The left half of the first row, A_0, shows the NF-IDCP cache. The NF-IDCP cache contains a randomization function (box with key inside). The cache lines have tag (white), data (black) and SDID (grey) fields. The right half, B_0, shows all the added hardware for the STF optimization. STF structures Dir_0 and Dir_1 are added to the hardware. Dir_0 has counters (dash-dot outlined box). Dir_1 has pointers, SDIDs and tags (light grey box, white box dark grey box). To facilitate the STF optimizations, the NF-IDCP cache is modified so that each tag also holds pointers to the appropriate Dir_0 counters (dark grey box). Below, we shall discuss how the above hardware is used (§4.2.1). The STF structures are themselves randomized so that they behave in a fully associative manner.

4.2.1 How STF Works. In the Dir_0 structure, we keep counters with valid/invalid bits, corresponding to the RO cache lines in the LLC. The valid RO cache lines will maintain a pointer to the counter. A valid counter is incremented every time a new domain gets a copy of the RO cache line from the LLC. It is decremented during cache evictions. When a NF-IDCP flush occurs that needs to remove *all* RO lines due to writes by a domain, we set the bit to invalid thus making it an invalid counter. The RO cache lines can be flushed "lazily" in future accesses, and each flush will decrement the corresponding counter. There can exist multiple valid RO lines pointing to a valid counter, and also multiple groups of RO lines, which are being lazily flushed/evicted, pointing to their corresponding invalid counters. The valid counter location for that cache line address is stored in the corresponding Dir_1 entry.

In the second row of Figure 6, we show an example, contrasting how regular NF-IDCP flushes and lazy NF-IDCP flushes work. In the left-hand half of the second row (example A_1), there are three cache line copies in the shared cache that correspond to the different security domains. Steps 2, 3 and 4 are needed to carry out the flushes. In the right half of the second row (example B_1), the initial state in Step 1 has four copies of the cache line, which are used in different security domains. The cache lines are all clean initially. Hence, they all point to the same valid counter in Dir_0. In Step 2, we reset the pointer associated with the cache line written to because it is no longer an RO cache line (marked in red). In Step 3, the invalid bit is set for the counter associated with the remaining RO lines (marked in mesh gradient). In both examples A_1 and B_1, invalidation is sent to remove all private copies.

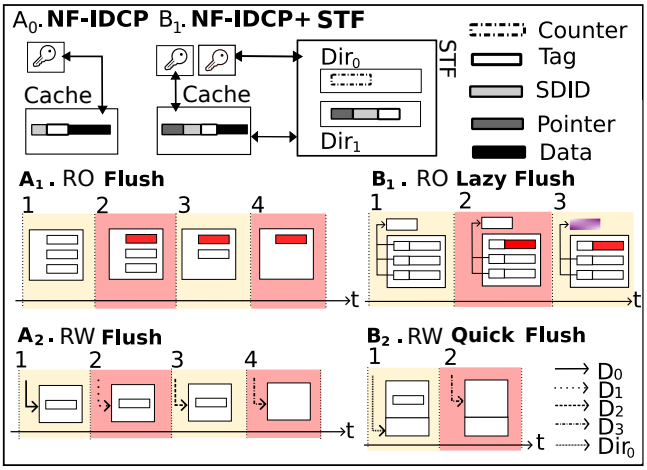


Figure 6: The secure tag-based filter for optimizing NF-IDCP, using lower latency cache evictions (see §4.2).

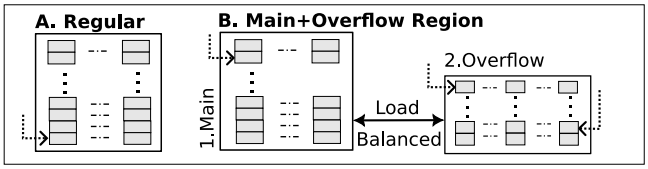


Figure 7: A fully-associative structure using a main region and an overflow region (see §4.2.2).

Quick Flushes Of RW Lines: We first lookup the Dir_1 structure while doing an NF-IDCP flush of RW cache lines upon a cache miss. It tells us which domain currently has the RW cache line (if any). Then, we need to flush the RW cache line that location. In the third row of Figure 6, we show how the flush for an RW cache line is done in the unoptimized NF-IDCP and in the STF-optimized version of NF-IDCP. In the left half of the third row (example A_2), we show that it takes four steps (Step 1, Step 2, Step 3 and Step 4) to find the RW cache line and flush it. In the right side example (B_2), Dir_1 is looked up (Step 1), and the SDID corresponding to the RW cache line address is determined. In the next step (Step 2), the RW cache line is flushed from the LLC.

Eliminating Tag Duplication: In the above setup, tags are duplicated in the Dir_1 and the cache. This can be eliminated by having a pointer from the cache to Dir_1 instead, thus saving on space.

4.2.2 Fully Associative Housing For STF. One way to create a fully associative structure is to use two regions, a *main* region and an *overflow* region. The main region has a set-associative structure. The overflow region also has a set-associative structure but with a

higher associativity. Alternately, it can use a strategy like MIRAGE to lower its associativity.

For each access, the main region and the overflow region are both accessed simultaneously. A domain-insensitive randomization function is used for each cache line address to determine the the respective sets to be accessed. This does not cause any security issue, because the STF housing merely houses the Dir_0 or Dir_1 metadata for the cache line addresses in the main cache, and does *not* drive any conflict misses. Metadata can be retrieved from either region. As the cache adds and removes cache line addresses, corresponding metadata will be added or removed. We can also move any metadata from the overflow region into main region if space becomes available there.

Figure 7 shows the hardware diagram for a set-associative structure and a fully-associative cache structure. In Part A, it is a regular set-associative cache. This can host the metadata that the STF requires. In the Part B on the right-hand side, there are two regions, the main region (1) and the overflow region (2). The overflow region is a more highly associative region, but smaller in size. These two regions are connected, indicated by the bold double-arrow line. The set to be accessed is determined by a randomization function on the cache line address (not shown).

Takeaway 2

The two optimizations to NF-IDCP, namely *parallelization* and *STF*, securely reduce the latency of the cache-flush operations generated by IDCP, at a low hardware overhead.

5 SECURITY ANALYSIS AND EVALUATION

As specified in (see §2.3), NF-IDCP should restore the security guarantees of randomization schemes by preventing domain fusion. Our security argument for the above has three parts. First, we carry out stress tests on the parallelization optimization at different banks and domain counts, to determine maximum expected latencies . Second, we stress the fully-associative structures in the STF using a random access pattern to see if we can trigger any set over-subscriptions (see §5.1.1) . Third, we determine that a randomization scheme that uses NF-IDCP has the same attack resilience to conflict attacks (discussed in §2.1) as the underlying randomization scheme. For the above, we use a generalized representation of a conflict attack as a sequence of random variables that represent the accesses and hit/miss observations of an attacker. Then, we show that the joint distribution of the accesses and the observations are not affected by the NF-IDCP flushes. We discuss these three security issues in §5.2.

Furthermore, we show that resilience against occupancy attacks is maintained at a high level due to identical coverage for any attacker, regardless of whether the NF-IDCP mechanism is switched on or switched off. We argue that flush attacks on shared read-only addresses are mitigated due to the inability of an attacker to do cross-domain flushes on read-only shared addresses. Finally, we carry out extensive simulations and show that domain fusion substantially reduces the attack resilience of randomization (see §5.2.3).

Table 1: Bank Contention Simulation Results. Each measured Max Contention in column 3 corresponds to the number of banks in column 1 of the same row, for the number of domains shown in column 2.

Banks	Domains	Max Cont	Accesses
8,16,32,64,128	8	7,7,7,7,6	40 trillion
8,16,32,64,128	16	13,11,9,8,8	40 trillion
8,16,32,64,128	32	20,15,12,11,9	40 trillion

5.1 NF-IDCP Optimization Security

5.1.1 Parallelization Security . To stress the parallelization optimization, we carry out 40 trillion random accesses on a parallelized cache slice. Table 1 shows experimental results for the total contention on each bank. Each of the max-contention numbers was determined using an experiment with over a trillion cache accesses. Each row in the first column of Table 1 lists 5 different LLC's with different number of banks, 8, 16, 32, 64, 128. The second column shows the total number of domains involved in the system. This corresponds to the total number of accesses for the NF-IDCP flushes. Each row in the third column lists the maximum number of accesses that happen to the same bank for a single address. This is related to the worst-case latency for carrying out the NF-IDCP flushes. The last column corresponds to the duration of the experiment in terms of the number of accesses to the multi-banked cache slice.

Observations: As expected, a larger bank count significantly reduces the maximum observed contention. Therefore, for each of these combinations of bank counts and domain counts, the imposition of the NF-IDCP flush-operation latency corresponding to several sequential accesses, similar to max contention, will guarantee no timing side-channels due to the banking, at least for the duration of our experiment. Since we have carried out 40 trillion accesses for each combination of domain counts and bank counts, this means that we can conservatively estimate that the cache slice will need to be flushed after that many accesses. However, the magnitude of leakage due to this (say, one reset every 40 trillion accesses), is negligible compared to the leakage rate needed for a successful recovery of a secret key (empirically, one observation every few thousands or tens of thousands of cycles [72, 74]).

STF Security: The STF's security guarantee is connected to the probability that an attacker will be able to oversubscribe a set in the overflow region. We carried out a simulation of a trillion random accesses on a regularly sized cache slice, and found that the maximum subscription on the overflow region was 48, for an overflow region that was 12.5% the size of the main region. Hence, a 48-way associative overflow region is sufficient to prevent any resets in that number of accesses. This reset rate is at least several orders of magnitude less than the rate at which randomization leaks information.

5.2 No Improvement To Conflict-Based Attacks

The resilience against conflict-based attacks for an NF-IDCP randomization scheme needs to be identical to the underlying randomization scheme, according to the targeted security property $\mathcal{P}_4$.

We discuss the above in the context of the three conflict attacks introduced in §2.

5.2.1 Conflict-Set Attack Resilience. A conflict-set attack (as discussed in §2) can be generalized in the context of an NF-IDCP cache, as an attacker making hit/miss measurements using a sequence of accesses that can be reads, writes and flushes. The above flushes may affect hit/miss timing measurements. However, we have already argued that the NF-IDCP flushes do not help a conflict attacker, because they have no effect from a replacement standpoint (see §2.4). Therefore, an attacker cannot generate new conflict patterns by engaging the NF-IDCP.

More concretely, we can represent any conflict-set attack as a sequence of random variables $S_0^{'}, A_0^{'}, OB_0^{'} ... S_i^{'}, OB_i{'}, A_i^{'} ... S_n^{'}, OB_n{'}, A_n^{'}$, for a sequence of length n. The $S_i^{'}$ variables represent the cache state, which is a set of pairs consisting of cache locations and the cache line addresses therein. The $A_i^{'}$ variables represent an access to a cache line address from a particular security domain, which could be a read, write or flush. The $OB_i^{'}$ variables represent an observation about a hit, an observation about a miss or no observation (for flushes).

We also consider an analogous attacker that uses only reads, without writes or flush instructions, to mimic the above attack. The access sequence can be represented as $S_0^*, A_0^*, OB_0^* ... S_i^*, OB_i^*, A_i^* ... S_n^*, OB_n^*, A_n^*$, for a sequence of length n. The random variable S_i^* represents the cache locations and cache line addresses therein, similar to $S_i^{'}$. A_i^* represents reads, pseudo-writes and pseudo-flushes. The pseudo-writes cause a read to be executed. Pseudo-flushes do nothing. OB_i^* is the *adjusted* observation. They can be cache hits or misses for reads and pseudo-writes. There is no observation for pseudo-flushes. The cache hits can be adjusted by the attacker to be recorded as cache misses, based on previous pseudo-writes and pseudo-flushes. We can show via inductive strategy that the joint probability distribution $p_{A_0^{'}, OB_0^{'} ... A_n^{'}, OB_n^{'}}$ is exactly the same as $p_{A_0^*, OB_0^* ... A_n^*, OB_n^*}$. Thus, the attackers have the same power because there is equal probability of recording any sequence of accesses and timing observations. In Appendix §A, we show the details of the inductive proof of the above intuitive assertion.

5.2.2 Occupancy Attack Resilience. As we discussed in §2.2, SassCache can control the *coverage* of different security domains. This enables occupancy attack protections. Cache lines can *partially isolated, fully isolated* or *not isolated* at all. *Fully isolated* means that an attacker's security domain has no chance of evicting the cache line. *Partially isolated* means that the attacker has the possibility of evicting the cache line in some, but not all skews. *Not isolated* means that in all skews, the attacker has a chance of evicting the cache line of the victim.

The main security benefits of SassCache occur because the probability that a security-sensitive cache line is fully isolated or partially isolated is *high* (greater than 99.999%), and the fraction of cache lines that aren't isolated is *low* (less than 0.0001%), for the default coverage settings of 39% for the attacker. Consider a SassCache that has NF-IDCP mechanisms available. SassCache divides the cache into k mini-caches, or skews) in the system. In each skew, the attacker has access to a random subset of the cache line locations, 39% by default. A security sensitive cache line will need to be mapped to one of those locations to be evictable by the attacker. Since this mapping is decided by the underlying randomization function, and not the NF-IDCP, therefore, the isolation property for any skew is not affected by NF-IDCP. If domain fusion occurred due to the absence of NF-IDCP, then the victim cache lines would all be evictable by the attacker, substantially reducing resilience against attack (see Figure 12 in Appendix §C).

Shared Read-Only Memory Flush Attack Resilience: Due to the tagging of cache lines with an SDID (see §4.2), cross-domain flushes via *clflush* instructions will never happen in any NF-IDCP enhanced cache. Thus, we have robust resilience against the above attacks.

5.2.3 Demonstrating Negative Effects Of Domain Fusion: We implemented SassCache inside CacheFX [30], a popular tool for evaluating the security of randomization. We configure SassCache so that there are two security domains, one for the attacker and the other for the victim. We used a cache with 2048 cache lines to model a significant size while keeping computational costs as economical as possible (100 billion attacker accesses simulated). The criteria for the resilience of the randomization, is to measure how often an attacker can distinguish two different cryptographic keys, only based on side-channel observations. In our experiment, we ran AES [22] encryption and RSA [74] encryption inside CacheFX. Each attack runs 200,000 encryptions, alternating between two randomly chosen encryption keys. The attacker observes the cache state via eviction-set attack (same as conflict-set attack) or occupancy attack, and tries to make a distinction between the two keys based on the differences in the observed eviction pattern. The entire process was repeated more than 300 times for each of the keys, and a *success rate* was measured, which is the fraction of attempts where the attacker could distinguish the two keys.

Table 2 summarizes the result of the attack experiments. The first column shows the the four attacks simulated. Each attack either used AES or RSA (SquareMult) encryption algorithms, and used occupany attack or eviction-set strategy. The percent of attacks that could successfully distinguish the keys in domain-fused SassCache are shown in the second column. In non-fused randomization (third column), the success rate of the attacker is 70% higher than non-fused SassCache. In the fourth column we consider a SassCache configuration, where the two domains have their coverages constrained within two different cache halves, using simple scale-and-shift of the index function (see Appendix §D.). The attack success is reduced to zero due to no cross-domain evictions.

We extensively evaluate the resistance to eviction-set and occupancy attacks of other randomization schemes (CEASER, CEASER-S, ScatterCache, MIRAGE); non-fused SassCache enables the best security guarantees (see Appendix §C).

6 PERFORMANCE EVALUATION

Using ZSim [55], we carry out a performance evaluation of our scheme using PARSEC 3.0 workloads, which has significant data sharing. The processor configuration used for simulation mainly consists of a Nehalem-like out-of-order core coupled with a three-level cache hierarchy. The L1I cache is 32KB in size (4-way assoc), the L1D is 32KB in size (4-way assoc) and the L2 cache is 256KB in size. The shared LLC consists of one slice per-core, where the size of each cache slice is 2MB with a 16-way associativity. We also

Table 2: Side-Channel Attacks To Distinguish Two Encryption-Keys. The columns refer to different Sass-Cache configuration and the rows refer to different kinds of attacks. The cells contain percentage of attacks which succeeded out of 300 trials.

Attack Type	Sass (f)	Sass (nf)	Sass-P (nf)
Eviction-Set, AES	18%	7.5%	0%
Eviction-Set, RSA	100%	15%	0%
Occupancy, AES	100%	22%	0%
Occupancy, RSA	100%	22%	0%

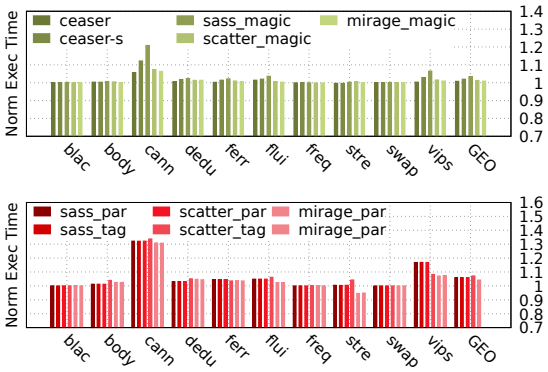
simulate real-world Firefox, Chromium and X Server workloads on a similar cache configuration, using a PANDA [28] based simulator. We also present SPECRate2017 results in an online Appendix [15].

An *insecure* cache is devoid of any side-channel protection. The *ceaser* and *ceaser-s* strategies are fused-domain as discussed in §2. The next three schemes used are *sass_magic*, *scatter_magic* and *mirage_magic*, which are the same as SassCache, ScatterCache and MIRAGE, but with an ideal zero-overhead coherence mechanism. That means, each write to a cache line address transmits the written contents to cache lines used by all domains instantaneously. Next, there are three configurations: *sass_par*, *scatter_par* and *mirage_par*. They are the coherent versions of the same via NF-IDCP strategy and parallelization optimization. Finally, there are three approaches: *sass_tag*, *scatter_tag* and *mirage_tag*. These three approaches are the versions of the above randomizations with NF-IDCP and the added STF optimization. Since we want to be conservative on the performance overheads, we add a latency of 8 cycles for the parallelization optimization, and a latency of 4 cycles for the tag-based filter optimization on the critical cache hit path.

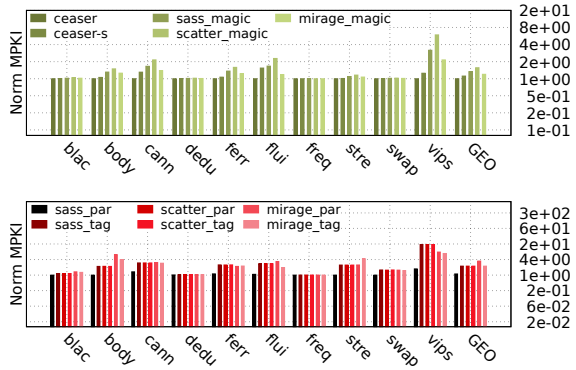
6.1 PARSEC 3.0 (with Significant Data Sharing)

The PARSEC 3.0 benchmarks represent a wide range of sharing patterns that are used by multi-threaded applications. In theory, any two domains that use shared memory could potentially use such access patterns. Therefore, we use PARSEC 3.0 benchmarks as one possible way to evaluate the performance of the NF-IDCP-enhanced cache. In our 8-core simulations, we use them to run 8 threads in parallel, each assigned to a different security domain. The key performance metric is the *total execution time*, which indicates how many cycles it took the multicore processor to complete the workload. To better understand the performance, we also record supporting figures, such as the MPKI (misses per kilo instruction) and the LLC access fraction.

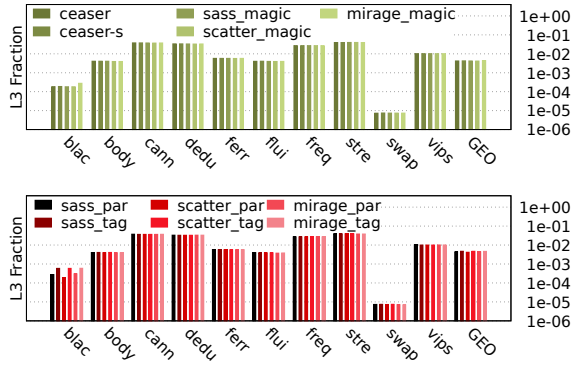
Performance Results: On average, we see less than 5% performance degradation compared to domain-fused randomization, for NF-IDCP randomization. In two cases, *canneal* and *vips*, there is upto 30% and 10% performance degradation, respectively. This overhead is mainly due to a lack of cache hits on shared-writable memory cache lines in the LLC. Figure 8 shows the performance results for the PARSEC 3.0 benchmarks. Part (a) of the figure shows the performance results for the benchmarks normalized to the *insecure* configuration, and the geometric mean of the same is also shown. Part (b) of the figure shows the normalized MPKI numbers



(a) The total execution time for the PARSEC 3.0 benchmarks simulated for twelve cache configurations (see §6.1)



(b) The MPKI (misses-per-kilo-instruction) for PARSEC 3.0 benchmarks simulated for twelve cache configurations (see §6.1).



(c) The fraction-of-accesses going to the LLC for PARSEC 3.0 benchmarks simulated for twelve cache configurations (see §6.1). GEO is short for geometric mean.

Figure 8: The performance results and the supporting data for PARSEC 3.0 obtained from simulations on ZSim.

for the simulations, and Part (c) shows the fraction of memory accesses that went to the LLC. We generally see a correlation where a higher MPKI results in lower performance. However, this effect is tempered by the LLC access fraction. Generally, a lower LLC access

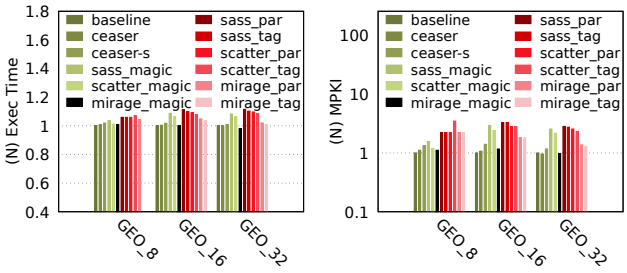
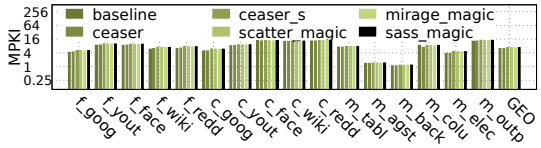
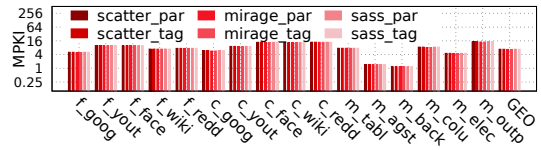


Figure 9: The average trends in performance and MPKI for PARSEC 3.0 using 16 domains/16 cores and 32 domains/32 cores.



(a) The MPKI results of a cache simulation for real-world workloads, for the first six configurations (see §6.2)



(b) The MPKI results of a cache simulation for real-world workloads, for the last six configurations (see §6.2)

Figure 10: Simulations on real-world workloads show minimal impact of NF-IDCP on cache MPKI.

fraction leads to a lower MPKI. The performance overhead of PARSEC 3.0 remains less than 5% for 16-core and 32-core workloads also (see Figure 9).

6.2 Real-World Workloads (Low Data Sharing)

Firefox and Chromium are configured to run a single tab. The content/renderer process for that tab is configured to run in one security domain. The other processes are all configured to run in another domain. X Server is configured to use a client MuPDF [39], which decrypts PDF files before display.

On our PANDA-based cache simulator, we find that performance degradation in terms of MPKI is less than 0.5 (see Figures §10a and §10b), for simulations of 1 billion memory accesses. This can be attributed to the relatively low proportion of shared memory accesses used by Firefox, Chromium and X Server (less than 2%).

7 DISCUSSION AND RELATED WORK

Hardware Overheads: We estimate the hardware overhead of the system using a standard cache design exploration tool, CACTI 7.0 [20]. The main hardware overheads are due to parallelization overheads and STF overheads. The parallelization overhead is less than 3% upto 32-banks. The STF overhead is less than 8% for 32 domains compared to an insecure cache. This is only 3% more than the cost of an FTM-enhanced domain-fused scheme, which has more than 5% storage overhead.

Energy Overheads: Majority of the extra energy consumption is due to the extra tag-lookups in the LLC for cache flushes. Based on CACTI 7.0 results, the dynamic energy cost for the twelve NF-IDCP configurations we simulated is less than 2% for the SPECRate2017, PARSEC 3.0 and real-world workloads. The static energy cost is increased by 3%, similar to the hardware overhead increase for the STF configuration. It is not increased significantly for the multi-banked configuration.

Cache Partitioning And Other Randomization Techniques: A wide-array of cache-partitioning techniques have been proposed in recent literature targeting the security use cases [21, 29, 36, 41, 48, 54, 56, 67, 76]. We can augment all the existing cache partitioning strategies using our approach. We evaluated a partitioned version of SassCache in Appendix §D. Phantom Cache [62] randomizes within a small number of sets instead of randomizing across the entire cache. Song et.al [60] propose new reallocation techniques to reduce the overheads of re-keying in randomization. HybCache [26] uses cache randomization and partitioning in a hybrid manner. NewCache [37] and RPCache [68] are applied to the private caches rather than the shared caches. MIRAGE [53] uses load-balancing for fully associative caching.

8 CONCLUSIONS

We identified the important problem of *domain fusion* due to data sharing between different security domains. In existing randomization schemes, domain fusion forces different security domains to use the same randomization function, thus reducing the security level significantly.

We introduce a new randomization-with-sharing approach, RAWS, which enables non-fusion based data coherence using randomized cross-domain accesses. We develop a secure inter-domain coherence protocol (NF-IDCP) using RAWS. It uses cache flushes to restore coherence without relying on domain fusion. We integrate NF-IDCP into existing state-of-the-art randomization schemes (ScatterCache, MIRAGE and SassCache) using secure parallelization and tag-based filter optimizations.

We perform a security evaluation using CacheFX attack simulations and a probabilistic analysis, and a performance evaluation on a wide-range of benchmarks, including SPECRate 2017, PARSEC 3.0 and real-world workloads. In all analysis and evaluations, we demonstrate that the security breaches induced by domain fusion are eliminated by NF-IDCP. Across all evaluation, the average performance overhead is less than 5% and the hardware overhead is less than 3%.

ACKNOWLEDGMENTS

This research was supported in part by NSF Grant CNS-2106771.

REFERENCES

[1] [n. d.]. codebase. https://www.chromium.org/developers/design-documents/gpu-command-buffer. Accessed: 2023-07-15.
[2] [n. d.]. commandbuffer. https://www.chromium.org/developers/design-documents/gpu-command-buffer. Accessed: 2023-07-15.

[3] [n. d.]. cve-2023-25000. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-25000. Accessed: 2023-12-12.

[4] [n. d.]. cve-2023-25332. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-25332. Accessed: 2023-12-12.

[5] [n. d.]. cve-2023-26556. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26556. Accessed: 2023-12-12.

[6] [n. d.]. cve-2023-26557. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26557. Accessed: 2023-12-12.

[7] [n. d.]. cve-2023-32691. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-32691. Accessed: 2023-12-12.

[8] [n. d.]. firefoxSharing. https://blog.mozilla.org/attack-and-defense/2021/01/27/effectively-fuzzing-the-ipc-layer-in-firefox/. Accessed: 2023-07-15.

[9] [n. d.]. intelsidechannel. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/securing-workloads-against-side-channel-methods.html. Accessed: 2023-07-15.

[10] [n. d.]. ipc-sniffer. https://github.com/tomer8007/chromium-ipc-sniffer. Accessed: 2023-07-10.

[11] [n. d.]. IPDL. https://firefox-source-docs.mozilla.org/ipc/ipdl.html. Accessed: 2023-07-10.

[12] [n. d.]. memfdSharing. https://man7.org/linux/man-pages/man2/memfd_create.2.html. Accessed: 2023-07-15.

[13] [n. d.]. mit-shm. https://en.wikipedia.org/wiki/MIT-SHM. Accessed: 2023-07-10.

[14] [n. d.]. mojo-ipc. https://chromium.googlesource.com/chromium/src/+/refs/tags/72.0.3586.1/ipc/. Accessed: 2023-07-10.

[15] [n. d.]. onlineAppendix. https://github.com/kartikram3/RAWS-Supplementary. Accessed: 2023-12-12.

[16] [n. d.]. pulseaudio. https://man.archlinux.org/man/pulseaudio.1.en. Accessed: 2023-07-15.

[17] [n. d.]. qubes. https://www.qubes-os.org/. Accessed: 2023-07-15.

[18] [n. d.]. sidechannel1. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/security-best-practices-side-channel-resistance.html. Accessed: 2023-07-15.

[19] [n. d.]. xserver. https://en.wikipedia.org/wiki/X.Org_Server. Accessed: 2023-07-15.

[20] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[21] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 213–224.

[22] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 201–215.

[23] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. RELOAD+ REFRESH: Abusing cache replacement policies to perform stealthy cache attacks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1967–1984.

[24] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[25] Yun Chen, Lingfeng Pei, and Trevor E Carlson. 2021. Leaking Control Flow Information via the Hardware Prefetcher. *arXiv preprint arXiv:2109.00474* (2021).

[26] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 451–468.

[27] David L Dill. 1996. The Mur ϕ verification system. In *Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8*. Springer, 390–393.

[28] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. 1–11.

[29] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. 2018. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 104–117.

[30] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. 2023. Cachefx: A framework for evaluating cache security. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 163–176.

[31] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. 2022. Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1101–1115.

[32] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[33] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 897–912.

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. 353–364.

[35] Sowoong Kim, Myeonggyun Han, and Woongki Baek. 2022. DPrime+ DAbort: A High-Precision and Timer-Free Directory-Based Side-Channel Attack in Non-Inclusive Cache Hierarchies using Intel TSX. IEEE.

[36] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.

[37] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. 2016. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro* 36, 5 (2016), 8–16.

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.

[39] Miles Arthur Munson and Jesse S Cross. 2011. *Deep PDF parsing to extract features for detecting embedded malware*. Technical Report. Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA ....

[40] Divya Ojha and Sandhya Dwarkadas. 2021. TimeCache: Using Time to Eliminate Cache Side Channels when Sharing Software. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 375–387.

[41] Hamza Omar, Brandon D'Agostino, and Omer Khan. 2020. OPTIMUS: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks. *IEEE Trans. Comput.* 69, 11 (2020), 1558–1570.

[42] Hamza Omar and Omer Khan. 2020. IRONHIDE: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 111–122.

[43] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. {DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks. In *25th {USENIX} security symposium ({USENIX} security 16)*. 565–581.

[44] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic analysis of randomization-based protected cache architectures. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 987–1002.

[45] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. [n. d.]. Prime+ Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. ([n. d.]).

[46] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.

[47] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 360–371.

[48] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.

[49] Kartik Ramkrishnan, Stephen McCamant, Pen Chung Yew, and Antonia Zhai. 2020. First Time Miss: Low Overhead Mitigation for Shared Memory Cache Side Channels. In *49th International Conference on Parallel Processing-ICPP*. 1–11.

[50] Kartik Ramkrishnan, Antonia Zhai, Stephen McCamant, and Pen Chung Yew. 2019. New attacks and defenses for randomized caches. *arXiv preprint arXiv:1909.12302* (2019).

[51] Jude A Rivers, Gary S Tyson, Edward S Davidson, and Todd M Austin. 1997. On high-bandwidth data cache design for multi-issue processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 46–56.

[52] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. 2021. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1090.

[53] Gururaj Saileshwar and Moinuddin Qureshi. 2021. {MIRAGE}: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[54] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*. 57–68.

[55] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news* 41, 3 (2013), 475–486.

[56] Brian C Schwedock and Nathan Beckmann. 2020. Jumanji: The Case for Dynamic NUCA in the Datacenter. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 665–680.

[57] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 131–145.

[58] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. {Prime+ Probe} 1,{JavaScript} 0: Overcoming Browser-based {Side-Channel} Defenses. In *30th USENIX Security Symposium (USENIX Security 21)*. 2863–2880.

[59] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust website fingerprinting through the cache occupancy channel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 639–656.

[60] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. 2021. Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 955–969.

[61] Daniel Sorin, Mark Hill, and David Wood. 2011. *A primer on memory consistency and cache coherence*. Morgan & Claypool Publishers.

[62] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2020. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization.. In *NDSS*.

[63] Thomas Unterluggauer, Austin Harris, Scott Constable, Fangfei Liu, and Carlos Rozas. 2022. Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 13–24.

[64] Pepe Vila, Boris Köpf, and José F Morales. 2019. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–54.

[65] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. [n. d.]. MeshUp: Stateless Cache Side-channel Attack on CPU Mesh. ([n. d.]).

[66] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. 2021. Volcano: Stateless Cache Side-channel Attack by Exploiting Mesh Interconnect. *arXiv preprint arXiv:2103.04533* (2021).

[67] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[68] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*. 494–505.

[69] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. Scattercache: Thwarting cache attacks via cache set randomization. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 675–692.

[70] Zihan Xue, Jinchi Han, and Wei Song. 2023. CTPP: A Fast and Stealth Algorithm for Searching Eviction Sets on Intel Processors. (2023).

[71] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 888–904.

[72] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. 2019. Secdir: a secure directory to defeat directory side-channel attacks. In *Proceedings of the 46th International Symposium on Computer Architecture*. 332–345.

[73] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are coherence protocol states vulnerable to information leakage?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 168–179.

[74] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 719–732.

[75] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7, 2 (2017), 99–112.

[76] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. Coloris: a dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 381–392.

[77] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3. 0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News* 44, 5 (2017), 1–16.

[78] Zhaomin Zhu, Koh Johguchi, Hans Jürgen Mattausch, Tetsushi Koide, Tai Hirakawa, and Tetsuo Hironaka. 2003. A novel hierarchical multi-port cache. In *ESSCIRC 2004-29th European Solid-State Circuits Conference (IEEE Cat. No. 03EX705)*. IEEE, 405–408.

A MODELLING CONFLICT-SET ATTACKS AS A SEQUENCE OF RANDOM VARIABLES

A.1 Setting Up The Attack Scenario

We reiterate the scenario that was mentioned in §5.2.1. We can represent any conflict-set attack as a sequence of random variables $S_0^{'}, A_0^{'}, OB_0^{'}, ...S_i^{'}, OB_i{'}, A_i^{'}...S_n^{'}, OB_n{'}, A_n^{'}$. The $S_i^{'}$ variables represent the cache state, which is a set of pairs consisting of cache locations and the cache line addresses therein. The $A_i^{'}$ variables represent an access to a cache line address from a particular security domain, which could be a read, write or flush. The $OB_i^{'}$ variables represent an observation about a hit, an observation about a miss or no observation (for flushes). We also consider an analogous attacker that uses only reads, and not writes nor flush instructions, to mimic the above attack. The access sequence can be represented as $S_0^*, A_0^*, OB_0^*, ...S_i^*, OB_i^*, A_i^*...S_n^*, OB_n^*, A_n^*$, for a sequence of length n. The random variable S_0^* represents the cache locations and cache line addresses therein, similar to $S_i^{'}$. A_i^* represents reads, pseudo-writes and pseudo-flushes. The pseudo-writes cause a read to be executed. Pseudo-flushes do nothing. OB_i^* is the *adjusted* observation. They can be cache hits or misses for reads and pseudo-writes. There is no observation for pseudo-flushes. The cache hits can be adjusted by the attacker to be recorded as cache misses, based on previous pseudo-writes and pseudo-flushes. We can show via inductive strategy that the joint probability distribution $p_{A_0^{'}, OB_0^{'}...A_n^{'}, OB_n^{'}}$ is exactly the same as $p_{A_0^*, OB_0^*...A_n^*, OB_n^*}$, for a sequence of length n. Thus, the attackers have the same power because there is equal probability of recording any sequence of accesses and timing observations.

A.2 Inductive Argument

A.2.1 Base Case. Let us consider the base case. We want to prove that $p_{S_0^{'}, A_0^{'}, OB_0^{'}}$ is identical to $p_{S_0^*, A_0^*, OB_0^*}$. We start building the above joint distributions one random variable at a time. The probability distributions $p_{S_0^{'}}(s_0)$ and $p_{S_0^*}(s_0)$ are the same because we start from an empty cache for both attackers. Next, the conditional probability distributions of the access for the first attacker is $p_{A_0^{'}|S_0^{'}}(a_0|s_0)$, which could be reads, writes or flushes on a particular address. $p_{A_0^*|S_0^*}(a_0|s_0)$ is the same because the reads-only attacker has no further information than the first attacker in determining what accesses to carry out. The resultant joint distribution $p_{S_0, A_0}(s, a)$ is the product $p_{A_0^{'}|S_0^{'}}(a_0|s_0)$ * $p_{S_0^{'}}(s_0)$. Similarly, the joint distribution $p_{S_0^*, A_0^*}(s_0, a_0)$ is the product $p_{A_0^*|S_0^*}(a_0|s_0)$ * $p_{S_0^*}(s_0)$. Since the corresponding product terms are the same in both the above expressions, therefore, the resultant joint distributions $p_{S_0, A_0}(s, a)$ and $p_{S_0^*, A_0^*}(s, a)$ are the same. Let us also consider the next joint distributions $p_{S_0^{'}, A_0^{'}, OB_0^{'}}(s_0, a_0, ob_0)$ and $p_{S_0^*, A_0^*, OB_0^*}(s_0, a_0, ob_0)$. Each can be written as the product of two terms $p_{OB_0^{'}|S_0^{'}, A_0^{'}}(ob_0|s_0, a_0)$ * $p_{S_0^{'}, A_0^{'}}(s_0, a_0)$ and $p_{OB_0^*|S_0^*, A_0^*}(ob_0|s_0, a_0)$ * $p_{S_0^*, A_0^*}(s_0, a_0)$. The first terms of the above products (conditional distribution terms) are the same because, for each initial state s_0, if the same operation a_0 is performed (read, write or flush), then it will be a miss for read or write and it will be no observation for flush. This is because the initial state s_0 is empty. The second terms of the products are

also the same, as we had shown above. Therefore, the overall joint distribution is also the same. This proves the base case.

A.2.2 Inductive Hypothesis And General Case. The inductive hypothesis is that two joint distributions will be the same upto the i^{th} term in the sequence. Thus, the two probability distributions $p_{S'_0,...S'_i,A'_i,OB'_i}$ and $p_{S^*_0,...S^*_i,A^*_i,OB^*_i}$ will be identical. Let us call this distribution $\mathcal{D}$.

A.2.3 General Case. The general case we would like to prove is that the joint distributions are the same upto the $i+1^{th}$ part of the sequence. $p_{S'_0,...S'_{i+1}\ A'_{i+1},OB'_{i+1}}$ and $p_{S^*_0,...S^*_{i+1},A^*_{i+1},OB^*_{i+1}}$ need to be the same. Let us now add one random variable at a time, similar to the base case.

Adding S_{i+1} ***To The Sequence:*** First, let us add the random variable S_{i+1}. We need to prove that $\mathcal{K}_1 : p_{S'_0,...OB'_i,S'_{i+1}}$ and $\mathcal{K}_2 : p_{S^*_0,...\ OB^*_i,S^*_{i+1}}$ are equal. In order to prove the above, we can split the above terms into two subterms, a baseline distribution and a conditional distribution. Hence, $\mathcal{K}_1$ can be represented as $p_{S'_{i+1}|S_{0'}\ ...,S'_i,A'_i,OB'_i} * \mathcal{D}$. Similarly, $\mathcal{K}_2$ can be represented as $p_{S^*_{i+1}|S_{0^*}\ ...,S^*_i,A^*_i,OB^*_i} * \mathcal{D}_0$. Therefore, to prove that $\mathcal{K}_1 = \mathcal{K}_2$, we only need to prove that the two conditional distributions above are equal. Let us consider the two analogous conditional distributions $p_{S'_{i+1}|S'_0,...\ S'_i,A'_i,OB'_i}$ and $p_{S^*_{i+1}|S^*_0,...S^*_i,A^*_i,OB^*_i}$. Consider the i^{th} tag state s_i and the i^{th} address access, a_i. Exhaustively, there are three possibilities, cache hit, cache miss and flush. Since the replacement function is identical for both attackers, therefore, the conditional distributions, upon a cache miss for a_i, will be the same. If a_i is a cache hit on s_i or if a_i is a cache flush, then the new state s_{i+1} will always be the same as s_i for both attackers, because neither of those operations change the tag state. Therefore, in all cases, we shall get the same values for the conditional joint distributions. Therefore, $\mathcal{K}_1 = \mathcal{K}_2 = \mathcal{K}$.

Adding A_{i+1} ***To The Sequence:*** We are adding in the next random variable A_{i+1} to the sequence. We would like to prove that the joint probability distributions $\mathcal{L}_1 : p_{S'_0,...OB'_i,S'_{i+1},\ A'_{i+1}}$ and $\mathcal{L}_2 : p_{S^*_0,...\ OB^*_i,S^*_{i+1},A^*_{i+1}}$ are equal. Hence, $\mathcal{L}_1$ can be represented as $p_{A'_{i+1}|S_{0'}\ ...,S'_i,A'_i,OB'_i,S'_{i+1}} * \mathcal{K}$. Similarly, $\mathcal{L}_2$ can be represented as $p_{A^*_{i+1}|S^*_0\ ...,S^*_i,A^*_i,OB^*_i,S^*_{i+1}} * \mathcal{K}$. Therefore, to prove that $\mathcal{L}_1 = \mathcal{L}_2$, we only need to prove that the two conditional distributions above are equal. We note that for both the attackers, the marginal distributions are identical. The conditional distribution values for any given subsequence $a_0, ob_0 ... a_i, ob_i$, will be identical for both attackers, because they are both carrying out the same attack strategy. Therefore, $\mathcal{L}_1 = \mathcal{L}_2 = \mathcal{L}$.

Adding OB_{i+1} ***To The Sequence:*** We are adding in the next random variable OB_{i+1} to the sequence. We would like to prove that the joint probability distributions $\mathcal{M}_1 : p_{S'_0,...OB'_i,S'_{i+1},\ A'_{i+1},OB'_{i+1}}$ and $\mathcal{M}_2 : p_{S^*_0,...\ OB^*_i,S^*_{i+1},A^*_{i+1},OB^*_{i+1}}$ are equal. Hence, $\mathcal{M}_1$ can be represented as $p_{OB'_{i+1}|S_{0'}\ ...,S'_i,A'_i,OB'_i,S'_{i+1},A'_{i+1}} * \mathcal{L}$. Similarly, $\mathcal{M}_2$ can be represented as $p_{OB^*_{i+1}|S^*_0\ ...,S^*_i,A^*_i,OB^*_i,S^*_{i+1},A^*_{i+1}} * \mathcal{L}$. Therefore, to prove that $\mathcal{M}_1 = \mathcal{M}_2$, we only need to prove that the two conditional distributions above are equal.

We make the observation ob_{i+1}, which depends on a_{i+1}, s_i. If it is a read or write, then the observation will be cache hit or cache miss for the first attacker. First, let us consider that the first attacker had a cache hit on the access a_{i+1}. For the second attacker, the corresponding actions would have been read or pseudo-write. Since the tag states they are operating on are the same (s_i), therefore, they should get the same observation ob_{i+1}. Next, let us consider that the first attacker had a cache miss on the access a_{i+1}. This could be because the tag was not present, or it could be because the tag was invalidated. To determine the valid/invalid state of the tag, we look backwards in the sequence. If there are any flushes (due to writes or flush instructions) after the last read/write to the address, then it is invalid. Otherwise, the tag is in a valid state. For the second attacker, if the tag was not present, it will see a cache miss. If the tag was present, the attacker would still adjust the observation to a cache miss, based on the backwards look on the sequence. Hence, the observation ob_{i+1} is always the same for the two attackers. Hence, the conditional distributions are the same, leading us to the result $\mathcal{M}_1 = \mathcal{M}_2 = \mathcal{M}$.

Extracting The Marginal Distribution: Based on the above, if the attack went on for n steps, we can extract the marginal distributions $p_{A'_0,OB'_0...A'_n,OB'_n}$ and $p_{A^*_0\ ,OB^*_0...A^*_n,OB^*_n}$, which should also be identical. Each sequence in the distribution has the exact same probability of occurring, and thus the two attackers have the same attack power.

B COHERENCE PROTOCOL VERIFICATION

We implement a four-processor system using the programming language provided by the Murphi [27]. Two processors are assigned to one domain, and the other two processors to another domain. There is a single cache line for domain 1 in the LLC and upto one cache line in each of its private caches. Similarly, there is a single cache line in the LLC for domain 2 and upto one cache line in each of its private caches.

In the above system, each cache line holds one of the four coherence states, M, E, S or I. They also hold many other transient states that are used during transitions between M, E, S and I. There is also a single NF-IDCP-RW bit for each cache line address. This is set whenever there is a cache write to a domain. We also model flushes whenever there are read or write misses, as per the NF-IDCP protocol of §3.

The baseline MESI protocol (without the NF-IDCP flushes) was exhaustively verified using Murphi. This confirms that the two coherence invariants of SWMR and data invariance are always maintained. This was a relatively quick verification (a few minutes) due to the small size of the protocol and small number of cache lines. The MESI + NF-IDCP procotol is larger due to double the core count and two LLC cache lines instead of one. This exponentially increases the state space. Hence, we need to use a *simulation* approach to detect errors. Murphi underwent four days of simulation to find errors. After testing more than 45 billion rules at the time of writing this document, there was no error found, indicating a significant degree of stability of our implementation. We were able to also run an exhaustive verification algorithm for > 100 GB worth of states until the memory was exhausted by the Murphi.

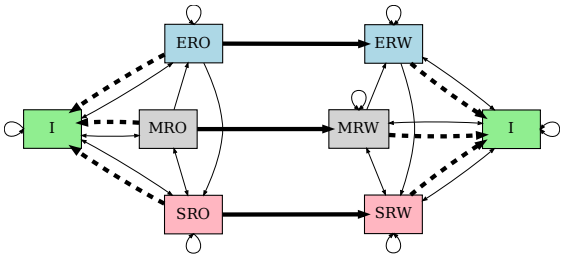


Figure 11: A simplified diagram showing the a MESI-NF-IDCP coherence protocol. The important transitions that are used for inter-domain coherence are highlighted in bold solid or bold dotted style (see Appendix §B).

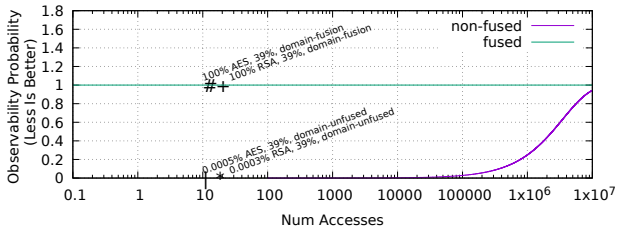
B.1 Important Transitions

B.1.1 How MESI-NF-IDCP Works. The functionality of MESI-NF-IDCP is very similar to the baseline MESI protocol, but with the additional NF-IDCP transitions explained in §3. Let us consider the lifetime of a cache line. Initially, a cache line corresponding to the cache line address is not present in the cache, or it may be in the invalid state. Next, the cache line is fetched, either due to a read or a write. If it is fetched due to a read, the cache line flushes out all the other cache lines in the system, if any have NF-IDCP-RW bit set. Then the domain that did the read transitions between the MESI states as normal in a per-domain protocol. Note that the M state is possible but the new contents will not be written immediately. Instead, there will be a transition where all the cache line copies used by that domain will set their NF-IDCP-RW. Then only, the actual value will be written into the cache line. At some point, if another domain then does a read or a write, then the cache line will be removed. If the NF-IDCP-RW bit is never set (the domain does only reads), then the cache lines may still be flushed due to a write by another domain.

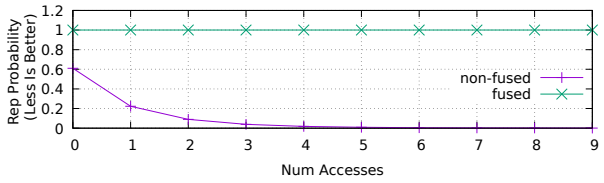
Figure 11 shows the important transitions in the MESI implementation of the protocol for any one of the involved domains. The four MESI states have been expanded into 8 states, depending upon the value of the NF-IDCP bit, which can be either NF-IDCP-RW (RW for short) or NF-IDCP-RO (RO for short). The main transitions related to NF-IDCP are highlighted in bold. The bold solid lines show the transition regarding a change from NF-IDCP-RO state to NF-IDCP-RW state. The bold dashed lines are the NF-IDCP flushes which are brought on due to either a read or a write by another domain. NF-IDCP-RO cache lines are flushed due to a write by another domain. NF-IDCP-RW cache lines are flushed due to a read or a write by another domain. The thin lines are part of the vanilla MESI protocol.

C DEMONSTRATING A LOSS OF ATTACK RESILIENCE DUE TO DOMAIN FUSION

We discuss a couple of analytical results regarding domain fusion in SassCache. Then, we present our argument about how Flush + Reload attacks will become a problem if there is domain fusion.



(a) Observability probability of SassCache under conditions of domain fusion and no domain fusion. (see Appendix §C)



(b) Repeated-Eviction probability of SassCache under conditions of domain fusion and no domain fusion. (see Appendix §C)

Figure 12: Observability and Repeated-Eviction probability in fused and non-fused SassCache.

Lastly, we present extensive simulation results which demonstrate that domain fusion significantly lowers conflict-attack resilience.

Observability Probability: In Figure 12a, we show that the *Observability* probability of the security-sensitive cache lines, is extremely high for the real-world workloads. It is 0.0005% for AES encryption, which has only 10 security sensitive cache lines related to T-table lookups. It is 0.0003% for RSA, which also has less than 20 security sensitive cache lines, such as those related to modular exponentiation [31]. These points are marked on the lower curve. However, the observability probability rises to 100% if domain-fusion is engaged (line at $y = 1$). This means, attacker has a much higher change to evict security-sensitive cache lines in domain-fused randomization [31]. In Figure 12b, the *Repeated-Eviction* probability is also extremely high. In non-fused SassCache, cache lines usually get hidden after a few evictions by the attacker, with a high probability (bottom line). However, domain fusion makes it so that there is no chance of a cache line getting hidden even after an arbitrary number of evictions (line at $y = 1$).

Flush + Reload Resilience: Figure 13 shows that expected measurements of a Flush + Reload attacker (see §2.1) are all cache misses. Due to tagging of cache lines with security domain ID (SDID), there is no chance for an attacker to flush a cache line used by another security domain. This transforms all measurements of a Flush + Reload [74] attack attacker into cache misses. In a domain-fused randomization, this guarantee is lost.

Attack Experiments: Figure 14 carries out attacks defined in §5.2.3 on a wide range of randomization configurations. We have *ceaser_rand* [46] and *ceasers_rand* [47], which necessarily have domain fusion engaged, being fused randomizations. We also have *sasscache* [31] where each domain has a random coverage of the cache, *assoc_rand* (MIRAGE [53], fully associative randomization) and *scattercache [69]* (like SassCache, but all domains have full

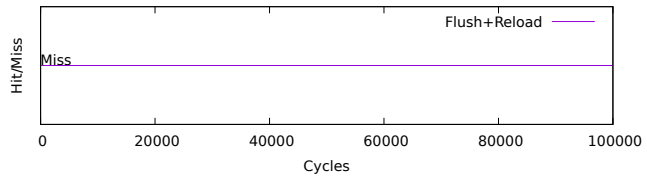


Figure 13: The attacker's measurements in a Flush+Reload attack on a non-fused randomization scheme. (see Appendix §C)

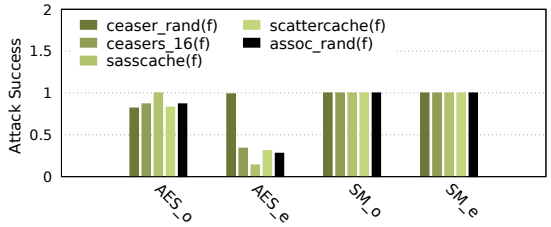
coverage). We carry out four kinds of attacks, which are represented as AES_o, AES_e, SM_o and SM_e. AES_o has a victim carrying out AES encryptions and an attacker carrying out occupancy attacks. AES_e has a victim carrying out AES encryptions and an attacker carrying out eviction-set attacks. SM_o has a victim carrying out Square-Multiply based encryption (like RSA encryption) and an attacker carrying out occupancy attacks. Lastly, SM_e has a victim carrying out Square-Multiply based encryption and an attacker carrying out eviction-set attacks.

Figure 14a shows the *success rate* (see §5.2.3), when operating in domain-fused mode (f). There is no attack resilience for the Square-Multiply algorithm (both *SM_e* and *SM_o*), which has almost a 100% attack success rate. In the AES_e attack, there is some resilience against conflict-based attacks, except for *ceaser_rand*, which has almost 100% attack success. In the AES_o attack, the attack success rate is close to 100%. In Figure 14b, in non-fused mode (nf), only *sasscache* shows significant attack resilience across the board. All other schemes have substantially less attack resilience. This tells us that the best randomization is *sasscache* and we should always run it without domain fusion. *sasscache-p* is a modified version of SassCache where there is no overlap between the attacker and victim domain, and so there is no possibility of attack success (see Appendix §D).

D PERFORMANCE OF PARTITIONED SASSCACHE

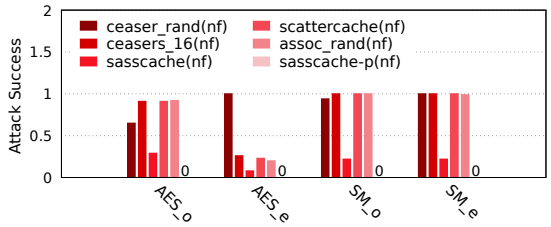
We can optionally modify the SassCache indexing function, so that each domain maps to a different range of cache sets. This should not have a significant latency effect, because it will only need a shift and an addition operation to carry out the above transformation. Consider an example where there are two security domains. Then, the regular SassCache configuration will choose 39% of cache lines randomly to be used by one domain and another 39% randomly for the other domain. All we'll need to do, is to divide the set index by 2 (one right shift) for the first domain. For the second domain, we will shift and also add a constant whose value is the number of sets divided by 2. In this setup, there will be no overlap. Similarly, if there are N domains, we can shift the index obtained by vanilla SassCache right by $log(N)$ bits and add the value $domain\_id * (numSets/N)$, to get a partitioned indexing function.

Figure 15 shows the PARSEC 3.0 performance results for a partitioned version of SassCache (1/8th for each security domain). Performance overheads are high for *canneal*, due to a lack of cache



(a) The attack results for fused randomizations shows that attack success is the lowest for SassCache overall and that there is no occupancy attack resilience (see Appendix §C).

.



(b) The attack results of non-fused randomizations shows that SassCache has maximum eviction and occupancy attack resilience (see Appendix §C).

Figure 14: The attack simulation results using CacheFX shows that non-fused and partitioned SassCache are the most secure randomization candidates.

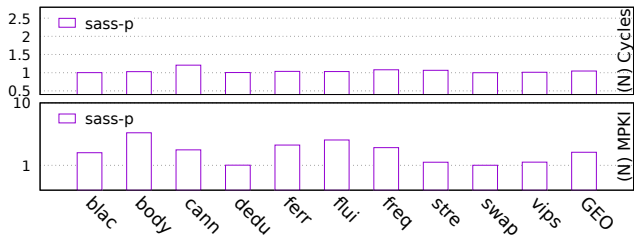


Figure 15: Performance of *sass-p* for PARSEC 3.0 workloads, normalized to a baseline insecure cache. We see that the performance is comparable to existing systems.

hits on shared-writable memory. The average performance overheads are less than 5%, thus indicating that SassCache running in partitioned mode can also be a reasonable solution.

Received 21 August 2023; revised 20 Dec 2023; accepted 17 Jan 2024