HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments

Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi,
Technische Universität Darmstadt

https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

HYBCACHE: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments

Ghada Dessouky, Tommaso Frassetto, Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{ghada.dessouky, tommaso.frassetto, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

Modern multi-core processors share cache resources for maximum cache utilization and performance gains. However, this leaves the cache vulnerable to side-channel attacks, where inherent timing differences in shared cache behavior are exploited to infer information on the victim's execution patterns, ultimately leaking private information such as a secret key. The root cause for these attacks is mutually distrusting processes sharing the cache entries and accessing them in a deterministic and consistent manner. Various defenses against cache side-channel attacks have been proposed. However, they suffer from serious shortcomings: they either degrade performance significantly, impose impractical restrictions, or can only defeat certain classes of these attacks. More importantly, they assume that side-channel-resilient caches are required for the entire execution workload and do not allow the possibility to selectively enable the mitigation only for the security-critical portion of the workload.

We present a generic mechanism for a flexible and soft partitioning of set-associative caches and propose a hybrid cache architecture, called HYBCACHE. HYBCACHE can be configured to selectively apply side-channel-resilient cache behavior only for isolated execution domains, while providing the non-isolated execution with conventional cache behavior, capacity and performance. An isolation domain can include one or more processes, specific portions of code, or a Trusted Execution Environment (e.g., SGX or TrustZone). We show that, with minimal hardware modifications and kernel support, HYBCACHE can provide side-channel-resilient cache *only* for isolated execution with a performance overhead of 3.5–5%, while incurring no performance overhead for the remaining execution workload. We provide a simulator-based and hardware implementation of HYBCACHE to evaluate the performance and area overheads, and show how HYBCACHE mitigates typical access-based and contention-based cache attacks.

1 Introduction

For decades now, upcoming processor generations are being augmented with novel performance-enhancing capabilities. Performance and security of processor architectures and microarchitectures are considered exclusively independent design metrics, with architects primarily focused on the more tangible performance benefits. However, the recent outbreak of micro-architectural cross-layer attacks [4–6, 18, 19, 22, 42, 44, 46, 47, 50, 56, 59, 68, 70, 79], has demonstrated the critical and long-ignored effects of micro-architectural performance optimizations on systems from a security standpoint. It is becoming evident how performance and security are at conflict with each other unless architects address the design trade-off early on and not as an afterthought.

One prominent performance feature and the subject of a wide range of recent architectural attacks is the use of caches and cache-like structures to provide orders-of-magnitude faster memory accesses. The intrinsic timing difference between a cache hit and miss is one of various *side channels* that can be exploited by an adversary process via a carefully crafted side-channel attack to infer the memory access patterns of a victim process [23, 25–29, 34, 35, 38, 54, 61, 71, 77, 78]. Consequently, the adversary can leak unauthorized information, such as a private key, hence violating the confidentiality and isolation of the victim process.

Cache Side-Channel Attacks. In earlier years, cache side-channel attacks have been shown to compromise cryptographic implementations [8, 54, 61, 78]. More recently, attack variants such as *Prime + Probe* [34, 38, 54, 61] and *Flush + Reload* attacks [29, 78] are being demonstrated on a much larger scale. They have been shown to bypass address space layout randomization (ASLR) [23, 25], infer keystroke behavior [26, 27], or leak privacy-sensitive human genome indexing computation [11], whereby millions of platforms using various architectures have been shown vulnerable to such attacks. The attacks require an adversary to orchestrate particular cache evictions of target memory addresses of interest and

after a time interval measure its own memory access latencies or observe relevant computation and profile how it has been affected. This enables the adversary to deduce the victim's memory access patterns and infer dependent secrets. Cache side-channel attacks have been shown to exploit core-specific caches as well as shared last-level caches across different cores or virtual machines [27, 38, 54]. Even hardware-security extensions and trusted execution environments (TEEs) such as Intel SGX [13, 33] and ARM TrustZone [7] are not immune to these attacks. While they do not claim cache side-channel security, recent cache side-channel attacks targeting SGX [11, 21, 60, 66] and TrustZone [49, 80] have been shown to compromise the acclaimed privacy and isolation guarantees of these security architectures, thus undermining their very purpose.

Existing Cache Defenses. To defeat cache side-channel attacks, there has been extensive research on techniques to identify and mitigate information leaks in a software's memory access patterns [16, 17, 45]. However, mitigating these leaks efficiently for arbitrary software (beyond cryptographic implementations) remains impractical and challenging. Alternatively, hardware-based and software approaches have been proposed to modify the cache organization itself to limit cache interference across different security domains. Examples include modifying replacement and leveraging inclusion policies [39, 76], as well as approaches that rely on cache partitioning [24, 40, 41, 51, 72, 73, 82], and randomization/obfuscation-based schemes [52, 53, 63, 69, 73] to randomize the relation between the memory address and its cache set index.

While strict cache partitioning is the intuitive approach to provide complete cache isolation and non-interference between mutually distrusting processes, it remains highly impractical and prevents efficient cache utilization. On the other hand, randomization-based approaches make the attacks computationally much more difficult by randomizing the mapping of memory addresses to cache sets. However, existing schemes either require complex management logic, impose particular restrictions, rely on weak cryptographic functions, or mitigate only some classes of cache side-channel attacks. Most importantly, all of the aforementioned schemes are designed to provide side-channel cache protection for the entire code execution, which is actually not required in practice.

Our Goals. We observe that usually the majority of the code is not security-critical. Typically, a small portion of the code is security-critical and requires cache-based side-channel resilience. Moreover, this security-critical portion of the code is often already running in an isolated environment, such as in a TEE or in an isolated process. In these cases, a trusted component, namely the processor hardware or microcode or the operating system kernel, enforces this isolation. We aim to leverage and extend this existing isolation mechanism to also selectively enable side-channel resilience for the caches *only* for the portion of the code that needs it, without reducing the cache performance for the remaining non-isolated code. In doing so, we practically address the persistent performance-security trade-off of caches by providing the system administrator with a "tuning knob" to configure by balancing and isolating the workload as required. Consequently, s/he can tune the resulting cache side-channel resilience, utilization, and performance, while guaranteeing no performance overhead is incurred on the non-isolated portion of the code execution. Only the isolated (usually the minority) portion is subject to a reasonable reduction in cache capacity and performance – the cost of increased security guarantees.

To achieve this flexible and hybrid cache behavior, we introduce HYBCACHE, a generic mechanism that protects isolated code from cache side-channel attacks without reducing the cache performance for the remaining non-isolated code. In HYBCACHE, isolated execution only uses a pre-defined (small) number of cache ways[1] in each set of a set-associative cache. It uses these ways fully-associatively, while for eviction random victim cache lines are selected to be replaced by new ones, thus breaking the set-associativity and removing the root cause of access leakage. Non-isolated execution uses all cache ways set-associatively as usual, without any performance overhead. While isolated and non-isolated execution may compete for the use of some ways in the cache, the random replacement policy and fully-associative mapping used by the isolated execution prevent leaking information about the accessed memory locations (and their cache set mapping) to the non-isolated execution, thus making the precomputation and construction of an eviction set impossible. Moreover, HYBCACHE flexibly supports multiple, mutually distrusting isolated execution domains while preserving the above security guarantees individually for each domain.

HYBCACHE is architecture-agnostic, and can be seamlessly integrated with any isolation mechanism (TEEs or inter-process isolation); the definition of the isolation domains and the distribution of the workload is left up to the system administrator. HYBCACHE is backward compatible by design; it provides conventional set-associative caches for the workload if the side-channel resilience feature is not supported.

Contributions. The main contributions of this paper are as follows.

- We present HYBCACHE, the first cache architecture designed to provide flexible configuration of cache side-channel resilience by selectively enabling it for isolated execution without degrading the performance and available cache capacity of non-isolated execution.
- We evaluate the performance overhead of a simulator-based implementation of HYBCACHE and show that it is less than 5% for the SPEC2006 benchmarks suite,

[1] *Ways* are different available entries in a cache set to which a particular memory address can be allocated.

and estimate the memory and area overheads of a cycle-accurate hardware implementation of HYBCACHE.
- We show – through our security analysis – how breaking set-associative mapping and shared cache lines between mutually distrusting isolation domains (which are the root causes for typical cache side-channel attacks besides the intrinsic cache sharing and competition) mitigates typical contention-based and access-based cache attacks.

2 Cache Organization, Attacks and Defenses

We briefly present the typical cache organization, as well as recent cache side-channel attacks that are within the scope of our work, and limitations of existing defenses.

2.1 Cache Organization

Cache Structure. Caches are typically arranged in a hierarchy of fastest/closest/smallest to slowest/furthest/largest levels of cache, respectively L1, L2, and L3 cache/last-level-cache (LLC). Each core incorporates its L1 and L2 caches and shares the LLC with other on-chip cores. A cache consists of the storage of the actual cached data/instructions and the *tag* bits of their corresponding memory addresses. Cache memory is organized into fixed-size memory blocks, called *cache lines* each of size B bytes. Set-associative caches are organized into S sets of W ways each (called a W-way set-associative cache) where each way can be used to store a cache line. A single cache line can only be allocated to only one of the cache sets, but can occupy any of the ways within this cache set. The least significant $log_2\ B$ bits are the *block offset* bits that indicate which byte block within the B-Byte cache line is requested. The next $log_2\ S$ bits are the *index* bits used to locate the correct cache set. The remaining most significant bits are the *tag* bits for each cache line.

In a set-associative cache, once the cache set of a requested address is located, the *tag* bits of the address are matched against the tags of the cache lines in the set to identify if it is a cache hit. If no match is found, then it is a miss at this cache level, and the request is sent down to the next lower-level cache in the hierarchy until the requested cache line is found or fetched from main memory (cache miss). However, in a fully-associative cache, a cache line can be placed in any of the cache ways where the entire cache serves as one set. No index bits are required, but only $log_2\ B$ block offset bits and the rest of the bits serve as tag bits.

Eviction and Replacement. Due to set-associativity and limited cache capacity, cache contention and capacity misses occur where a cache line must be evicted in favor of the new cache line. Which cache line to evict depends on the replacement policy deployed, some of which include First-in-First-Out (FIFO), Least-Recently-Used (LRU), pseudo-LRU, Least-Frequently-Used (LFU), Not-Recently-Used (NRU), random and pseudo-random replacement policies. In practice, approximations to LRU (pseudo-LRU) and random replacement (pseudo-random) are usually deployed.

2.2 Cache Side-Channel Attacks

Cache side-channel attacks pose a critical threat to trusted computing and underlie more proliferating side-channel attacks such as the Spectre [44] and Meltdown [50] variants. Different classes of these attacks have been demonstrated on all platforms and architectures ranging from mobile and embedded devices [49] to server computing systems [34, 54, 81]. They have also been shown to undermine the isolation guarantees of trusted execution environments, like Intel SGX [11, 21, 60, 66] and ARM TrustZone [49, 80]. Such attacks have been shown to infer both fine-grained and coarse-grained private data and operations, such as bypassing address space layout randomization (ASLR) [23, 25], inferring keystroke behavior [26, 27], or leaking privacy-sensitive human genome indexing computation [11], as well as RSA [54, 81] and AES [10, 34] decryption keys.

Cache side-channel attacks exploit the inherent leakage resulting from the timing latency difference between cache hits and misses. This is then used to infer privacy/security-critical information about the victim's execution. In an offline phase, the attacker must first identify the target addresses of interest (by means of static and dynamic code analysis of the victim program) whose access patterns leak the desired information about the victim's execution, such as a private encryption key. In an online phase, the attacker measures the timing latency of its memory accesses or the victim's computation time to infer the desired information.

To demonstrate how a simple cache attack works, consider the pseudo-code of the Montgomery ladder implementation for the modular exponentiation algorithm shown in Algorithm 1. Modular exponentiation is the operation of raising a number b to the exponent e modulo m to compute $b^e \bmod m$ and is used in many encryption algorithms such as RSA. Leaking the exponent e may reveal the private key. As shown in Algorithm 1, the operations performed for each of the exponent bits directly correspond to the value of the bit. If the exponent bit is a zero, the instruction in Line 5 is executed. If the exponent bit is a one, the instruction in Line 9 is executed. An attacker that can observe or deduce these execution patterns can thus disclose the value of each corresponding exponent bit, and eventually recover the encryption key [78, 81]. S/he, however, needs to identify the target addresses that need to be observed (the addresses of the instructions in Lines 5 and 9 in this example) in the victim program and accordingly construct the eviction set. The eviction set is a collection of addresses that are mapped to the same specific cache set to which the target addresses are also mapped. The attacker uses this eviction set to evict the contents of the whole set in the cache, and therefore guarantee to successfully evict the target

addresses from the caches. Consequently, s/he measures the timing latency of its own memory accesses after a time interval to deduce whether the victim has accessed these target addresses.

Algorithm 1: Montgomery Ladder RSA Implementation

```
   Input: base b, modulo m, exponent e = (e_{n-1}...e_0)_2
   Output: b^e  mod  m
1  R_0 ← 1; R_1 ← b;
2  for i from n-1 downto 0 do
3      if e_i = 0 then
4          R_1 ← R_0 × R_1 mod m;
5          R_0 ← R_0 × R_0 mod m;
6      end
7      if e_i = 1 then
8          R_0 ← R_0 × R_1 mod m;
9          R_1 ← R_1 × R_1 mod m;
10     end
11 end
12 return R_0;
```

The online phase of these attacks consists of three main steps: *Eviction*, *Waiting* and *Analysis*. The attacker uses the eviction set to *evict* the victim's target addresses from the cache. Next, the attacker *waits* an interval of time to allow the victim to access the target addresses. Then the attacker measures and *analyzes* its access time measurements to determine if the victim has accessed the target addresses. This is repeated as many times as the attacker requires to collect sufficient traces to recover the exponent bits.

The different techniques used by the attacker to perform the eviction can be classified into two main approaches, either access-based or contention-based. In access-based attacks such as Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], and Flush + Prefetch [25], the attacker accesses the target addresses directly by flushing them out of the cache using the dedicated *clflush* instruction [2] and possibly exploiting timing leakage from the execution of the *clflush* instruction [26]. This invalidates the lines containing these addresses and writes them back to memory. Evict + Reload [27] attacks have also been shown which do not require the *clflush* instruction, but instead evict specific cache sets by accessing physically congruent addresses. These attacks are only feasible in case of shared memory pages between the attacker and victim, usually in the form of shared libraries. Otherwise, an attacker resorts to contention-based attacks such as Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], Evict + Time [23, 61], alias-driven attacks [28], and indirect Memory Management Unit (MMU)-based cache attacks [71], where s/he constructs an eviction set and uses it to trigger and exploit a cache contention in the same cache set as the target addresses, thus evicting cache lines containing the target addresses from the pertinent cache set.

The waiting interval should be selected and synchronized such that the victim is expected to access the target address at least once before the attacker analyzes the collected observations. By analyzing the collected observations, the attacker determines whether the target address was indeed accessed by the victim. This is achieved by different techniques depending on the attack approach, either the adversary measures the overall time needed by the victim process to perform certain computations [8, 10], or probes the cache with eviction sets and profiles cache activity to deduce which memory addresses were accessed [34, 38, 54, 77, 78], or accesses target memory addresses and measures the timing of these individual accesses [29, 61]. Alternatively, the adversary can also read values of addresses from the main memory to see whether cache lines that contain cacheable target addresses have been evicted to memory [28].

Cache-collision timing attacks exploit cache collisions that the victim experiences due to its cache utilization, e.g., after a sequence of lookups performed by a table-driven software implementation of an encryption scheme, such as AES [10]. These attacks are out of scope in this work since they are not common, are specific to certain software implementations, and can only be mitigated by adapting the implementation or locking the relevant cache lines after pre-loading them.

2.3 Limitations of Existing Defenses

To mitigate these attacks, software-based countermeasures and modified cache architectures have been proposed in recent years, which we cover in depth in the Related Work (Section 8). These can be classified into two main paradigms: 1) applying cache partitioning to provide strict isolation, or 2) applying randomization or noise to make the attacks computationally impractical. However, all proposed countermeasures to date either impact performance significantly, require explicit programmer's annotations, are not seamlessly compatible with existing software requirements such as the use of shared libraries, are architecture-specific, or do not defend against all classes of attacks. Most importantly, all existing defenses apply their side-channel cache protection for the entire execution workload.

In practice, cache side-channel resilience is only required for the security-critical (usually smaller) portion of the workload that is allocated to execute in isolation. Thus, non-isolated execution should not suffer any resulting performance costs. To address this in this work, we propose a modified hybrid cache microarchitecture that enables side-channel resilience only for the isolated portion of execution, while retaining the conventional cache behavior and performance for the non-isolated execution.

3 Adversary Model and Assumptions

To provide side-channel-resilient cache accesses for only security-critical isolated execution, we propose a hybrid *soft* partitioning scheme for set-associative memory structures.

In this work, we apply it to caches and call it HYBCACHE. HYBCACHE aims to provide cache-based side-channel resilience to the security-critical or privacy-sensitive workload that is allocated to one or more **I**solated Execution **D**omains (I-Domains), while maintaining conventional cache behavior for non-critical execution that is allocated to the **N**on-**I**solated Execution **D**omain (NI-Domain). HYBCACHE assumes an adversary capable of mounting the attacks described in Section 2.2 and is designed to mitigate them.

Furthermore, the construction of HYBCACHE is based on the following assumptions:

A1 Security-critical code that requires side-channel resilience is already allocated to an isolated component, like a process or a TEE (enclave).

A recent trend in the design of complex applications, like web browsers, is to compartmentalize them using multiple processes. As an example, all major browsers spawn a dedicated process for every tab [43] and some even use a dedicated process to better isolate privileged components [58]. Similarly, the widespread availability of TEEs, like SGX, encourages developers to encapsulate sensitive components of their code in protected environments.

A2 Isolated execution is the minority of the workload.

Isolation works best when the isolated component is as small as possible, thus reducing the attack surface. This complies with the intended usage of TEEs like SGX where only small sensitive components of the code would be allocated to the TEE. Hence, we assume only the minority of the workload needs to be isolated. HYBCACHE still provides the same security guarantees if the majority of the workload is isolated, but the performance of the isolated execution would suffer.

A3 Sensitive code only uses writable shared memory for I/O (if at all), and access patterns to this shared memory do not leak any information.

Isolated code should focus on processing some local data, while I/O needs should be limited to copying the input(s) into the isolated component, and copying the output(s) out of the component. Both of these procedures just access the data sequentially; thus, the access patterns during I/O do not depend on the data and does not leak any information.

A4 The attacker is not in the same I-Domain as the victim.

HYBCACHE is designed to isolate mutually distrusting I-Domains and thus, we must assume the attacker and the victim are not in the same I-Domain. Note that, as a consequence of A3, if a process handles sensitive data and has multiple threads, they must all be in the same I-Domain, since they share the entire address space. In cases where isolation between threads sharing the same address space is also required, HYBCACHE can, in principle, provide intra-process isolation as discussed later in Section 7.

4 Hybrid Cache (HYBCACHE)

We systematically analyzed existing contention-based and access-based cache attacks in the literature (Section 2.2) to identify their common root causes (besides the intrinsic sharing of cache entries and latency difference between a cache hit and miss). Cache side-channel attacks are, by nature, very specific to the victim program and may exploit attack-specific features such as the side-channel leakage of the *clflush* [26] or prefetch instructions [25]. Nevertheless, each one of these attacks is primarily caused by one or both of the following root causes: shared memory pages (and cache lines) between mutually distrusting code, and deterministic and fixed set-associativity of cache structures, which enables targeted cache set contention by pre-computed eviction sets.

4.1 Requirements Derivation

In light of the above, HYBCACHE should provide side-channel resilience between different isolation domains with respect to their cache utilization. An adversary process sharing the cache with a victim process should not be able to distinguish which memory locations a victim accesses. Nevertheless, we emphasize that the only approach to enforce complete non-interference between different domains is by strict static cache partitioning, such that no cache resources are shared, and thus zero information leakage occurs. On the other hand, this is impractical, and results in inefficient cache utilization from a performance standpoint. Our key objective in this work is to practically address and accommodate this persistent performance/security trade-off of cache structures by providing sufficiently strong cache side-channel-resilience, such that practical and typical cache side-channel attacks become effectively infeasible without necessarily enforcing complete non-interference. Additionally, we desire that this security guarantee is run-time configurable, such that it is only in effect when required.

This builds on our insight that it is neither practical nor required to provide cache side-channel resilience for all the code in the workload. This additional security guarantee is only required for security-critical execution, which is a minority of the workload (Assumption A2), and usually isolated in a Trusted Execution Environment (TEE) (Assumption A1). Thus, we require to provide a cache architecture that provides non-isolated execution with conventional cache utilization (with no performance costs), and simultaneously side-channel-resilient cache utilization (with a tolerable performance degradation) only for the smaller portion of the execution workload that is security-sensitive and isolated. We also require that our architecture is portable, can be easily deployed, and is backward compatible when a system does not support it. We summarize these requirements below:

R1 Strong side-channel resilience guarantees between the isolated and non-isolated execution domains, sufficient to

thwart typical contention-based and access-based cache attacks

R2 Dynamic and scalable cache isolation between multiple different isolation domains

R3 Addressing the cache performance/security trade-off by configuring the non-isolated/isolated workload balance (compliant with how TEEs are intended and designed to be used) such that the performance of the non-isolated execution workload is not degraded

R4 Usability: backward-compatible, architecture-agnostic, no usage restrictions and no code modifications required

Next, we present the high-level construction of HYBCACHE in Section 4.2 and its microarchitecture in more detail in Sections 4.3 and 4.4.

4.2 High-Level Idea

In HYBCACHE, a subset of the cache, named *subcache*, is reserved to form an orthogonal isolated cache structure. Specifically, $n_{isolated}$ *cache ways* within the conventional *cache sets* form the *subcache*. While these *subcache* ways are available for the NI-Domain to utilize, the I-Domains are restricted to utilize *only* these *subcache* ways. However, the I-Domains utilize this *subcache* in a fully-associative way and using a random-replacement policy. In doing so, all mutually distrusting processes executing in the I-Domains can share the *subcache* without leaking information on the actual memory locations they access. Since these *subcache* ways are not reserved exclusively for isolated execution and can also be utilized by non-isolated execution with least priority, the NI-Domain still retains unaltered cache capacity usage and non-degraded performance.

The key purpose of HYBCACHE, unlike existing defenses, is to selectively enable side-channel-resilient cache utilization only for the I-Domains. Hence, only the isolated execution is subjected to the resulting performance overhead, while still maintaining conventional cache behavior and performance for the NI-Domain, as outlined in Requirement R3. We describe next the architecture of HYBCACHE and how it achieves this.

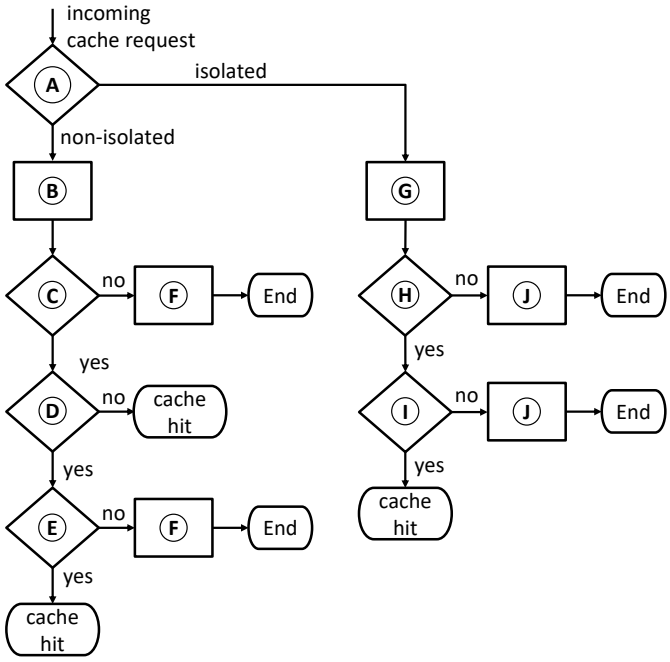
4.3 Controller Algorithm

HYBCACHE modifies how memory lines are mapped to cache entries for the I-Domains. $n_{isolated}$ ways (at least a way in each set) of the conventional set-associative cache are designated to the orthogonal *subcache*. Cache lines are mapped fully-associatively to the *subcache* entries and evicted and replaced in the *subcache* using a random replacement policy. This means that a given memory line can be cached in any of the $n_{isolated}$ entries. This breaks the deterministic link between memory addresses and their corresponding cache locations, thus defeating an attacker that attempts to infer the victim's memory accesses by triggering and observing contention in a particular cache set.

Figure 1 illustrates how the HYBCACHE controller manages cache requests. HYBCACHE supports multi-core processors with simultaneous multithreading (SMT) and assumes that each process is assigned an `IsolationDomainID` (`IDID`) that identifies whether the process is in an I-Domain (and which isolation domain) or in the NI-Domain. Any incoming cache request is accompanied by the `IDID` of the issuing process. In Ⓐ, HYBCACHE controller queries the `IDID` of the cache request and the request is serviced accordingly. If it is in the NI-Domain, the complete cache is queried conventionally using the set index and tag bits of the requested address to locate the cache set and line respectively (Ⓑ & Ⓒ). If a match is found, the controller checks whether the cache line was found in one of the *subcache* ways in Ⓓ. Recall that these ways are not reserved exclusively for isolated execution, i.e., they can be used by non-isolated execution but with least priority in case a cache set becomes over-utilized. Therefore, if a matching cache line is found in one of these ways, the controller checks whether it was cached by an isolated or non-isolated process (Ⓔ). The requesting process can only hit and access the cache line if that line was placed by a process in the NI-Domain. Otherwise, it is not allowed to hit on it.

Checks in the controller are implemented to occur in parallel, i.e., all cache hits are generated in the same number of clock cycles (as well as cache misses), to eliminate respective timing side channels. In case of a cache miss, the memory block is fetched from main memory and cached in Ⓕ. The eviction and replacement are performed according to the deployed policy. All ways are available for eviction, including the *subcache* ways to provide the NI-Domain execution with unaltered cache capacity. However, the usage of the *subcache* ways by the I-Domains is considered while recording the recency of accesses to the cache ways to make it least likely to evict a line from one of the *subcache* ways if it is recently used by an I-Domain process.

If the cache request is issued by an I-Domain process, it is serviced by querying only the *subcache* (Ⓖ). The *subcache* deploys fully-associative mapping, and is thus queried by a lookup of all the ways using the (cache line address bits - block offset bits) as tag bits (Ⓗ) and simultaneously querying that the line belongs to an I-Domain (since these ways may also be used by the NI-Domain) and that it was placed by a process with the same `IDID` (Ⓘ). Otherwise, a cache miss occurs. Disallowing I-Domain processes from hitting on cache lines originally placed by processes in other I-Domains provides dynamic isolation between an unlimited number of mutually distrusting processes that share memory. In case of a miss, any of the *subcache* ways is randomly selected and its cache line is evicted and replaced by the memory block fetched from main memory (Ⓙ). The random replacement policy considers all *subcache* ways equally, even those occupied by the NI-Domain cache lines.



(A) Is the process issuing the request in isolated or non-isolated execution mode?

(B) Query cache set-associatively using set index and tag bits to locate the way with requested memory block

(C) Is way with matching tag found?

(D) Is it one of the *subcache* ways?

(E) Is **line-IDID** = non-isolated (all-zero)?

(F) Cache miss: Evict and replace (via LRU/pseudo-LRU policy) cache line (including these occupying *subcache* ways) by memory block fetched from main memory

(G) Query the $n_{isolated}$ ways of *subcache* fully-associatively using the requested cache line address as tag for lookup

(H) Is way with matching **tag** found?

(I) Is way occupied by a line with matching **line-IDID**?

(J) Cache miss: Randomly replace and evict any of the cache lines occupying the *subcache* ways (irrespective of **line-IDID** of the cache lines)

FIGURE 1: HYBCACHE controller policy

4.4 Hardware Microarchitecture

Figure 2 shows how HYBCACHE could be applied for a conventional cache hierarchy of a multi-core processor. The cache capacity available for the NI-Domain execution is unaltered, i.e., the conventional set-associative cache with all its sets and ways can be utilized by the NI-Domain.

At each cache level, way-based partitioning is used to reserve at least a way in each set (gray ways in Figure 2). These ways, combined, form the orthogonal *subcache* that the I-Domain execution is restricted to use. However, these *subcache* ways are *not* used exclusively by the I-Domain execution, i.e., the NI-Domain execution may use these ways in case a corresponding set is fully utilized and the least-recently-used (LRU) replacement algorithm requires to evict a cache line from a *subcache* way in this set. This ensures that the NI-Domain execution is provided with unaltered cache capacity and does not suffer performance degradation.

The *subcache* is fully-associative and deploys random replacement policy, i.e., a given memory block is always equally likely to be cached in any of the available ways. This breaks set-associativity and provides randomization-based dynamic isolation between different I-Domains while allowing flexible sharing of the *subcache* depending on the run-time utilization requirements of the isolated execution domains. Using the *subcache* fully-associatively further maximizes the utilization of its limited hardwired capacity.

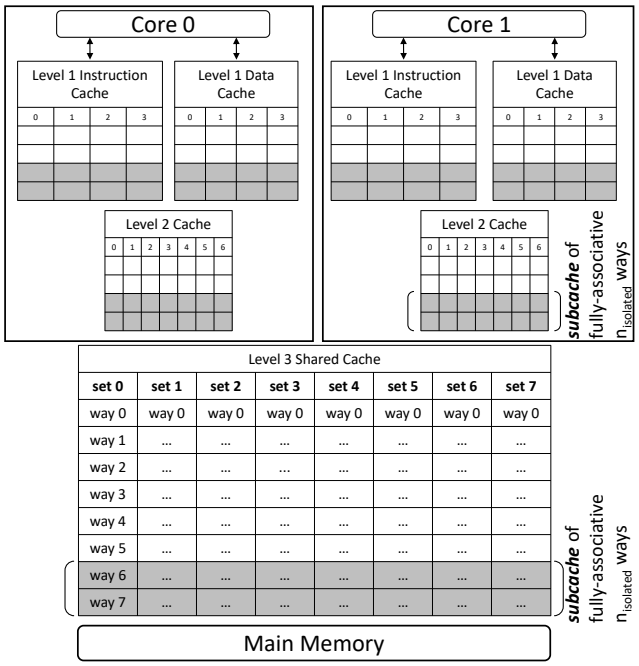


FIGURE 2: HYBCACHE hierarchy and organization

The $n_{isolated}$ ways that form the *subcache* are configured (hardwired) at design-time and cannot change at run-time, because these ways are members of both the primary cache as

well as the *subcache* as shown in Figure 3. It is not feasible to make $n_{isolated}$ run-time configurable, as this would require that *all* the ways are unreasonably wired in both a fully-associative and set-associative organization. Thus, only a small subset of $n_{isolated}$ ways (dark gray ways in Figure 3) is selected to form the *subcache*. Each of the *subcache* ways is augmented with `IsolationDomainID` (`IDID`) configuration bits to identify the isolation domain that placed an occupying cache line in the pertinent way. To provide any cache isolation at the microarchitectural level, a mechanism to bind owners/tags to cache lines is required, thus `IDIDs` are needed. We chose to configure 4 bits for the `IDID`, thus supporting 16 concurrent isolation domains, where an all-zero indicates the NI-Domain. The number of bits allocated in HYBCACHE for `IDID` is a hardware design decision. Increasing the number of designated bits would increase the number of maximum concurrent isolation domains that HYBCACHE can support. However, other metrics such as area overhead and power consumption come into play in this design trade-off.

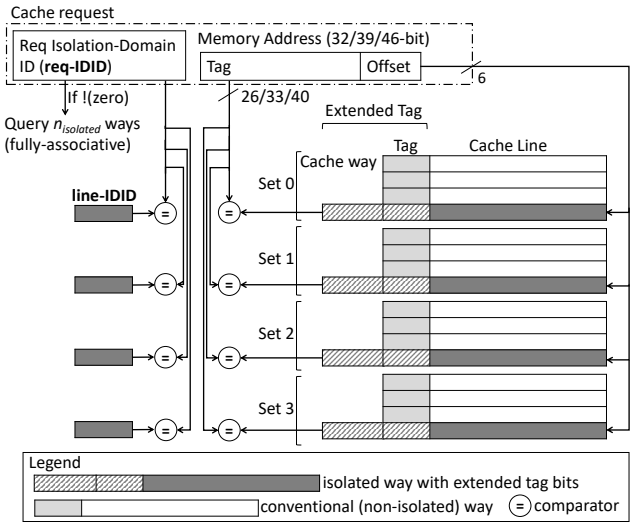


FIGURE 3: HYBCACHE hardware microarchitecture

The *subcache* ways are augmented with an extended tag bits storage (dashed dark gray tag bits of the dark gray ways in Figure 3). When queried fully-associatively (for the I-Domains), all bits, except the offset bits (6 bits for byte-addressable 64B cache line), of the requested address are compared with the extended tag bits of the *subcache* ways to locate a matching cache line. For the NI-Domain, the *subcache* ways are queried set-associatively with the rest of the cache (conventionally), where the request tag bits are compared only with the non-extended tag bits of the *subcache* ways within the located cache set.

4.5 Software Configuration

Abstraction and Transparency. The hardware modifications required for HYBCACHE are transparent to the software and abstracted from it. The trusted software (or hardware) component of the incorporating platform is only required to interface with the HYBCACHE controller to communicate the isolation domain of each incoming cache request. However, HYBCACHE does not stipulate or restrict how these isolation domains are defined and communicated, thus leaving it to the discretion of the system designer to identify how HYBCACHE can be integrated with the comprising architecture.

Isolated Execution. HYBCACHE enables the dynamic isolation of the cache utilization of different isolation domains by using the `IDID` of the process that issues the cache request being serviced. The means by which the isolation domains are defined, generated, and communicated is dependent on how the trusted execution and isolation is deployed. We design HYBCACHE such that it is seamlessly compliant with any trusted execution environment (TEE) where isolation domains (across different processes, cores, containers, or virtual machines (VMs)) are either software-defined by a trusted OS (thus requiring kernel support) or hardware/firmware-defined in case the OS is not trusted (such as in SGX). Different isolation domains can be defined across different isolated address space ranges such as in SGX enclaves, across processes such as in TrustZone normal/secure worlds or by standard inter-process isolation, or even across different groups of processes or different virtual machines.

HYBCACHE is agnostic to the means of defining the `IDIDs` of different isolation domains, and complements any form of isolated execution environment in place to provide it with cache side-channel resilience. If the kernel is trusted, kernel support is required to assign an `IDID` (or an all-zero `IDID` for a non-isolated process) to each process according to its isolation domain. The `IDID` bits can be added as an additional process attribute in each process's process control block (PCB). Otherwise, the trusted hardware or firmware would assign the isolation domains. HYBCACHE assumes that some mechanism of isolation is already enforced for security-critical code that it can leverage to provide the cache-level isolation. We argue why this is reasonable in Assumption A1. Nevertheless, if this is not the case, then isolation domains need to be explicitly defined by the developer if s/he wishes to protect particular code against cache-based side-channel attacks. While HYBCACHE is focused on protecting user code, in principle, kernel code can also be protected by allocating it to an isolation domain.

Backward Compatibility. Similar to processor supplementary capabilities such as Page Attribute Tables (PATs) and Memory Type Range Register (MTRR) for x86, HYBCACHE supports providing side-channel-resilience on-demand while

retaining backward compatibility. HYBCACHE only effectively provides side-channel resilience for the cache utilization of execution when processes are assigned different IDIDs that are communicated with each cache request. Otherwise, from a software perspective, HYBCACHE is identical to a conventional cache architecture. If no isolation domains are assigned to the different processes by the trusted kernel or trusted hardware, HYBCACHE is designed to assign an all-zero IDID by default to incoming cache requests and all execution is treated as non-isolated (see Figure 1) with cache-based side-channel resilience disabled. Only when kernel support is provided (or trusted hardware or firmware in case of SGX) does HYBCACHE behave differently for different isolation domains and provides its side-channel resilience capability.

Shared Memory Support. HYBCACHE supports, by design, that different isolation domains can share read-only memory, usually in the form of shared code libraries, without sharing the corresponding cache lines. This results in having multiple copies of the shared memory kept in cache (multiple cache entries), enforcing that cache entries are not shared between mutually distrusting code. Data coherence is also not a problem, in this case, since this is read-only memory. We elaborate in Section 5 how this effectively mitigates access-based side-channel attacks.

Conventional access to shared writable memory, on the other hand, between different isolation domains is disallowed by design in HYBCACHE, as this makes the victim process vulnerable to access-based attacks and would undermine cache coherence. In order to provide input and output functionality to isolated code, HYBCACHE provides special *I/O move instructions*. These allow code in an I-Domain to transfer data between a CPU register and a memory region (assigned an all-zero IDID when cached) that is designated exclusively for shared memory between processes belonging to different I-Domains. These special instructions are meant to be used to transfer data between domains only through this designated memory. In practice, we expect them to be used only in frameworks like the SGX SDK or a trusted kernel. If code in an I-Domain incorrectly accesses this memory region using regular instructions, or accesses its own memory using these special instructions, this could be disallowed, i.e., detected and blocked by the hardware or microcode, e.g., the MMU. This prevents inserting duplicated writable cache entries which can disrupt cache coherency, while ensuring that HYBCACHE's security guarantees still apply to any access performed using regular instructions.

5 Security Analysis

In the following, we evaluate the effectiveness of HYBCACHE with respect to the security requirements we outlined in Section 4.1. We show that HYBCACHE achieves these security guarantees by mitigating the following leakages:

S1 Malicious software running in an I-Domain or NI-Domain cannot flush or perform a cache hit on a cache line belonging to a different I-Domain.

S2 Malicious software running in an I-Domain or NI-Domain cannot pre-compute and construct an eviction set that selectively evicts a non-trivial subset of the cache lines belonging to a different I-Domain. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim.

S3 Cache hits generated by software in an I-Domain cannot be observed by software running in a different I-Domain or NI-Domain. Cache misses generated by software in an I-Domain can still be indirectly observed by malicious software running in a different I-Domain or NI-Domain, but the malicious software learns no information (e.g., memory address) about the access besides whether a cache miss has occurred.

5.1 S1: Absence of Direct Access to Cache Lines

Access-based attacks, like Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], Flush + Prefetch [25], and Evict + Reload [27], require the attacker to have direct access to the victim's cache lines, normally as a result of shared memory between processes (e.g., shared libraries). As an example, Flush + Reload works by flushing shared cache lines and monitoring which lines the victim accesses and brings back into the cache. HYBCACHE mitigates this class of attacks by preventing shared cache lines between the attacker and victim, as we explain in the following.

Shared Read-Only Memory. Read-only memory is shared between different processes in case of shared code libraries. HYBCACHE provides support for shared read-only memory (Section 4.5), while fundamentally disallowing that any cache line is shared across different I-Domains. Execution within one domain can only access cache lines brought into the cache by the same domain. Separate (potentially duplicate) cache lines are maintained for each domain; flushing and reloading cache lines only impacts those owned by the attacker's domain and cannot influence any other I-Domain or leak any information on its cache lines. Having duplicate cache lines for read-only memory pages does not disturb cache coherency because it is read-only.

Shared Writable Memory. Shared writable memory between mutually distrusting domains is disallowed by design with HYBCACHE. Code in an I-Domain can only exchange data with another isolation domain through the special I/O

move instructions, which transfer data between the CPU registers and memory in the NI-Domain that is designated for shared communication (see Section 4.5). Incorrect usage of those instructions or incorrect access to this designated memory region could be detected and blocked by the MMU to prevent potential cache coherency disruption due to duplicate writable cache entries. However, HYBCACHE still enforces that every cache line only belongs to one domain. Since cache lines always belong to one specific I-Domain or the NI-Domain, code in a domain cannot flush or perform a cache hit on a different domain's cache lines (S1), and attacks that rely on those capabilities are thus impossible.

5.2 S2: Impossibility of Pre-Computed Eviction Set Construction

Without direct access to the victim's cache lines, attackers resort to contention-based attacks, like Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], and Evict + Time [23, 61]. In these attacks, the attacker pre-computes and constructs an eviction set which ensures eviction of a specific subset of the victim's cache lines, e.g., lines that belong to a specific set in a set-associative cache. The attacker process first accesses the whole eviction set, thus ensuring the victim's cache lines are evicted. After a waiting interval, it then checks if its whole eviction set is still in cache by timing its own memory accesses to this set, thus detecting if the victim accessed any of the cache lines of interest. For a conventional set-associative cache, this is possible because of a fixed set-indexing, which can be directly determined from the target address of interest.

HYBCACHE protects I-Domains from such attacks by disabling the set-associativity of the reserved *subcache* entries when they are used by isolated execution: when a memory address is accessed by the isolated victim process, the cache line will be stored in any entry chosen randomly from the whole *subcache* and not from a specific set. The random replacement policy for isolated execution ensures that any of the *subcache* entries is chosen using a discrete uniform distribution, i.e., with an equal and independent probability every time, so the attacker has no means of identifying deterministically and reproducibly which cache set (or entry) will be used to cache a particular memory access of the victim. In order to ensure that a specific cache line of the victim is evicted, the attacker can only evict all lines in the *subcache*, but s/he cannot selectively evict a non-trivial subset of the victim's cache lines. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim (S2). As a consequence, attacks that rely on these capabilities are no longer possible. This holds whether the attacker process is running in an I-Domain or NI-Domain, as long as the victim process is in an I-Domain (Requirements R1 and R2).

5.3 S3: Observable Cache Events

Software running in an I-Domain can only hit on cache lines belonging to the same I-Domain. These cache hits generate no changes to the cache state, thus, they are unobservable by an attacker in a different I-Domain or in the NI-Domain.

Cache misses generated by software in an I-Domain evict a random cache line, which may belong to a different I-Domain or the NI-Domain. Malicious attacker code can then periodically observe how many of its lines are evicted and infer the number of cache misses the victim process is experiencing. The attacker can further use this information to infer the size of the victim's working set, i.e., the number of cache lines in the *subcache* currently belonging to the victim.

This cache occupancy channel is the only side-channel leakage that is not mitigated by the HYBCACHE construction, which is inherently available in any cache architecture where the attacker and the victim processes compete for entries in shared cache resources. It can only be effectively blocked by strict cache partitioning, which we deliberately do not provide in the HYBCACHE construction. This allows different isolation domains to still compete for cache entries, thus preserving maximum and dynamic cache utilization and unaffected performance for non-isolated execution, as our performance evaluation shows in Section 6.1. Note that, due to S2, the information inferred by the attacker from observing this remaining leakage, is effectively reduced to only knowing the working set size at any point in time.

Leveraging this side channel to infer further information and mount an attack in typical settings is not trivial. The victim may evict its own lines when it experiences cache misses due to the random replacement policy. This would not effect a difference in the cache state for the attacker, which complicates the attacker's bookkeeping. Moreover, observations are severely hindered when any other software is concurrently running besides the attacker and the victim processes. Finally, standard software hardening techniques can be applied to mitigate attacks to code implementations that are particularly sensitive to this attack. Furthermore, exploiting this side channel to leak data has not been shown in practice. A recent attack [67] leverages the cache occupancy side channel to infer which website is open in a different browser tab (under the strong assumption that no other tabs are open); however, it does not leak any user data. Cache activity masking is suggested as one of the countermeasures to the attack. Implementing cache activity masking for HYBCACHE is feasible and independent of our cache architecture.

Since the attacker aims to maximize its information and cannot observe cache hits, s/he can attempt to evict all *subcache* entries in order to maximize the number of misses experienced by the victim. As we discuss later, evicting the whole *subcache* takes time for an attacker in either the NI-Domain or in a I-Domain. An unprivileged attacker is unable to pause the victim's execution; thus, the attacker can only measure the

cache usage with limited granularity. However, a privileged adversary, like a malicious OS in the case of an SGX enclave, can stop and restart the victim arbitrarily and leverage tools like SGX-Step [12] to observe the victim's cache usage with fine granularity. HYBCACHE does not mitigate such an attack by construction. However, mitigating it is only possible by strict cache partitioning and the resulting performance costs. We emphasize that we make an intentional design decision in HYBCACHE to allow isolation domains to dynamically compete for cache entries for maximum cache utilization and unaffected performance for non-isolated execution. A HYBCACHE construction that dynamically allocates a dedicated *subcache* for each isolation domain would block this leakage and mitigate attacks that rely on it.

Non-isolated Attacker Process. If the attacker process is in the NI-Domain, in order to guarantee eviction of the whole *subcache* it must fill up all ways in every cache set, including the *subcache* ways. Therefore, the attacker process must construct an eviction set that is as large as the entire cache capacity. A typical data L1 cache holds 512 cache entries. In our experiments, probing (accessing and measuring access latencies) of 512 cache lines takes approximately 30 000 CPU cycles, i.e., a little over 8 µs.[2] For larger caches, such as the LLC, it is not even feasible to mount Prime+Probe attacks by probing the entire cache. The adversary is required to pinpoint a few cache sets that correspond to the relevant security-critical accesses made by the victim and monitor these only [54].

Isolated Attacker Process. If the adversary is in a different I-Domain than the victim process, it still cannot control cache eviction of particular target addresses specifically. Both attacker and victim processes are isolated and can only use the *subcache* ways. Thus, an adversary aiming to perform controlled eviction can only try to evict the entire *subcache*. Because the *subcache* is fully-associative with random replacement, evicting the entire *subcache* requires an eviction set much larger than the *subcache* capacity. We argue below that this is not easier than probing the entire L1 cache (in case the attacker is non-isolated), for instance, even though the *subcache* is significantly smaller. Moreover, it can be only guaranteed up to a certain level of probabilistic confidence. This can be represented statistically by the coupon collector's problem, where coupons are represented by entries in the *subcache*. Let $N_{accesses}$ be the total number of accesses needed to evict all the *subcache* entries n and n_i be the number of accesses needed to evict the i-th way after i-1 ways have been evicted. Both $N_{accesses}$ and n_i are discrete random variables. The probability of evicting a new way becomes $\frac{(n-(i-1))}{n}$. The expected value and variance of $N_{accesses}$ are

$$\mathbb{E}(N_{accesses}) = n \cdot \mathrm{H}_n \qquad \mathbb{V}(N_{accesses}) \approx \frac{\pi^2}{6} \cdot n^2$$

H_n denotes the n^{th} harmonic number. For $n = 128$ *subcache* entries, an average of 695 memory accesses (each mapping to a different 64B cache line) is needed to evict the *subcache* with a variance of $\approx 26\,951$. This is comparably more than the 512 accesses required to probe the entire typical L1 cache if the attacker process is not isolated (see above). Moreover, with such a large variance, significant variations in the number of $N_{accesses}$ required are expected from the mean $\mathbb{E}(N_{accesses})$ every time this eviction process is repeated.

[2] We ran this experiment on an Intel i7-4790 CPU clocked at 3.60 GHz.

6 Evaluation

Cache	Size	Associativity	Sets
L1	64 KB	8-way associative	128
L2	256 KB	8-way associative	512
L3	4 MB	16-way associative	4096

TABLE 1: Cache hierarchy used in our evaluation

Mix	Components
`pov+mcf`	povray, mcf
`lib+sje`	libquantum, sjeng
`gob+mcf`	gobmk, mcf
`ast+pov`	astar, povray
`h26+gob`	h264ref, gobmk
`bzi+sje`	bzip2, sjeng
`h26+per`	h264ref, perlbench
`cal+gob`	calculix, gobmk
`pov+mcf+h26+gob`	povray, mcf, h264ref, gobmk
`lib+sje+gob+mcf`	libquantum, sjeng, gobmk, mcf

TABLE 2: Benchmark mixes used in our evaluation

HYBCACHE is architecture-agnostic and applicable to x86, ARM or RISC-V. We performed our performance evaluation of HYBCACHE on a gem5-based [9] x86 emulator. We evaluated the hardware overhead for an RTL implementation that we implemented to extend an open-source RISC-V processor Ariane [62]. For our prototyping, we applied HYBCACHE to L1, L2, and LLC. We describe our evaluation results next.

6.1 Performance Evaluation

To evaluate HYBCACHE, we chose eight mixes of programs from the SPEC CPU2006 benchmark suite, which are used in the literature[3] [36, 76], shown in the upper part of Table 2.

[3] [76] also uses a ninth mix, `dea+pov`, which fails to run on gem5.

Two-Process Mixes. In order to evaluate the impact of isolating one process in the context of an SMT processor, we configure gem5 to simulate two processors connected to a single three-level cache hierarchy, whose parameters are shown in Table 1. The caches have the latencies used in [76].

For each mix, we first isolate one process, then the other, and we compare the performance of those processes to a third run in which neither process is isolated. We make either 2 or 3 of ways per set usable by the isolated execution processes. The replacement policy for non-isolated processes is LRU. Like in [76], we let gem5 simulate the first 10 billion instructions of each process in order to let the process initialize, then we measure the performance of one additional billion instructions. We measure the performance overhead as the relative change in the instructions-per-cycle (IPC), i.e., the ratio between instructions executed and CPU cycles required. A *positive* overhead represents a *decrease* in performance.

Figure 4 reports the IPC overhead of each program when running in isolation mode, while the other member of the mix runs in normal mode, for 2 or 3 isolated ways. The geometric mean of the positive overheads is 4.95% with 2 isolated ways and 3.47% with 3 isolated ways, with maximum overheads of 16% and 14% respectively for the `cal+gob` mix. For this mix, the overhead is due to a significantly increased L3 cache miss rate: the data miss rate jumps from 0.6% to 17.6%, while the instruction miss rate increases from 2.1% to 9.0%. The working set of `calculix` normally fits in L3 [36] but it does not in the *subcache*, hence the higher overhead. Since HYBCACHE is meant to protect only sensitive applications, which can be expected to be short-lived and only constitute a minority of the workload of a system, we consider those overheads easily tolerable. Figure 5 reports the IPC overhead for the member of the mix that is not isolated. In all cases the IPC overhead is not positive, i.e., the IPC is equal or better than the baseline, thus showing that HYBCACHE does not degrade the performance of non-isolated processes.

Four-Process Mixes. To demonstrate scalability, we also ran four-process mixes, shown in the bottom part of Table 2. We configured gem5 with four cores; two cores share an L1 and L2 cache, the other two cores share one additional L1 and L2, while L3 is shared by all cores. Isolated execution can use two ways per set. We isolated each member of the two mixes (the first eight bars in Figure 6), while the other three processes were running normally. Each isolated process has an overhead similar to that reported in the two-process mix experiments in Figure 4. Moreover, we also isolated two processes in each mix (last two columns in Figure 6). In this case, we measured increased overheads by up to 2 additional percentage points due to the additional competition for the *subcache*. However, those overheads are still easily tolerable given the security benefits and that they are only incurred by the isolated execution.

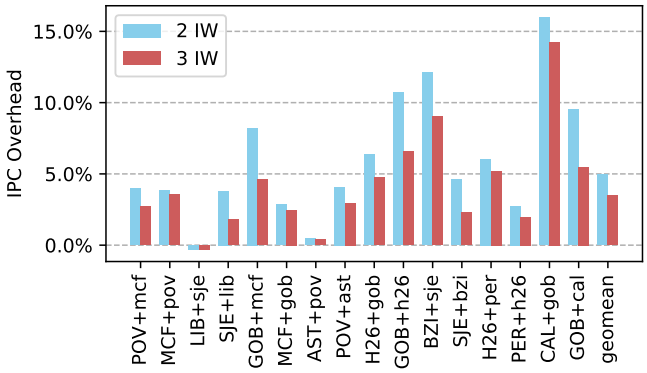


FIGURE 4: IPC overhead of each isolated process when 2 or 3 ways are available to isolated execution. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

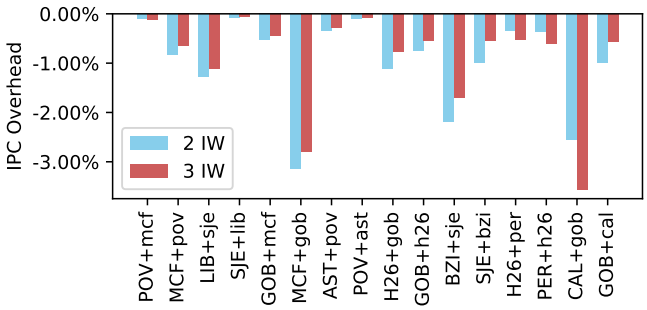


FIGURE 5: IPC overhead of each process when the other member of the mix is isolated. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

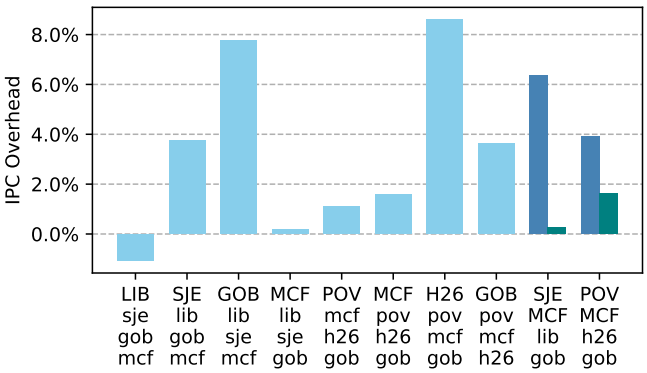


FIGURE 6: IPC overhead of isolated processes for 4-process mixes. The uppercase benchmarks are isolated and the others are not. The last two columns have two bars each since two process are isolated.

$n_{isolated}$	NAND2X1 Gates	Memory Overhead (Kb)
32	6114	0.34
64	12219	0.68
128	24563	1.3
256	48796	2.75
512	97830	5.5
1024	201792	11
2048	458300	22

TABLE 3: Logic and memory overhead estimates for fully-associative lookup of 46-bit addresses for different numbers of isolated cache ways (in any cache level).

6.2 Hardware and Memory Overhead

HYBCACHE requires additional hardware and memory for the fully-associative lookup of the *subcache* entries. We implemented the RTL for HYBCACHE and evaluated it for the hardware overhead for different number of isolated cache ways as shown in Table 3, irrespective of which cache levels this is applied to. While the overhead of the additional hardware is non-negligible, it is reasonable for a fully-associative cache lookup. Nevertheless, it diminishes in perspective with an 8-core Xeon Nehalem [1] of 2,300,000,000 transistors, for example. The logic overhead of HYBCACHE for 2048 fully-associative ways lookup is estimated at 1,833,200 transistors (NAND2X1 count $\times$ 4) which is 0.07% overhead to the Xeon Nehalem. For an 8-way 128-set cache, the memory overhead in our PoC for fully-associative mapping is 7 additional tag bits + 4 `IDID` bits per cache way. With respect to access latencies, the exact timing latency of lookups will eventually depend on the circuit routing but, in principle, for a parallel content-addressable memory lookup (as in our hardware PoC), accesses are performed in 2 clock cycles.

7 Discussion

Design and Implementation Aspects. HYBCACHE relies on a random-replacement cache policy combined with full-associativity to provide its dynamic isolation guarantees. The implementation of the random replacement policy is delegated to the hardware designer and considered an orthogonal problem. Cryptographically-secure pseudo-random number generators (CSPRNG) or even true hardware random number generators can be used and the seed can be changed as often as required. The output of the CSPRNG cannot be predicted if it is seeded with secret randomness at the start of every process. When the seed is changed, re-keying management tasks such as cache flushing and invalidation for the re-mapping are not required, unlike in recent architectures [63, 74]. This is because in HYBCACHE the randomness is only used for selection of the victim cache line, and not for locating existing cache lines in the *subcache*. Furthermore, we emphasize that CSPRNG design and implementations are an orthogonal problem to our work.

The "soft" cache partitioning of HYBCACHE is a generic concept and can be applied, in principle, to any set-associative structure. In this work, we apply it to the L1, L2, and L3 (LLC) caches, but it can also be applied selectively to only some of these cache levels or to the TLB as well, or to only some cache levels in only one or more cores in a multi-core architecture that become dedicated for allocating isolated execution. The choice of which cache structures to apply this to and how many ways to isolate in the *subcache* is delegated to the hardware designer, given that it is a more complex design decision with other metrics and trade-offs that come into play such as the size of the structure, power consumption, and logic overhead. The power consumption and timing overheads associated with building and routing a fully-associative cache lookup in VLSI are significant, but can be alleviated by leveraging emerging hybrid memory technologies such as DRAM-based caches [48] and STT-MRAM caches [30, 31]. In practice, applying HYBCACHE to the LLC or larger caches in general would be more expensive (in terms of hardware) than L1 and L2 caches, and strict partitioning might be applied instead for the LLC. Nevertheless, HYBCACHE can be, in principle, applied to sliced Intel LLCs. In each slice, a number of cache ways (*subcache*) is reserved for isolated execution. Any mapping from the `IDID` to the LLC slices can be used, such that lines from a particular `IDID` are allocated to a specific slice. Fully-associative lookups are thus only be performed on the *subcache* portion of a single slice, thus reducing the performance overheads and allowing scaling to high-core-count processors. The slice-mapping would be based only on the `IDID`, and thus it would not leak any information about the data address or value.

Other design decisions in HYBCACHE include the number of bits designated for `IDID` and thus the maximum number of concurrent isolation domains supported (see Section 4.4). To support more isolation domains (not concurrently) than the hardwired maximum, the cache lines of one domain can be flushed by the kernel or microcode at context switching while the next domain is switched in and is re-assigned the available `IDID`. Nevertheless, supporting too many isolation domains will result in increased cache utilization, and the overall performance will suffer. This is in line with conventional cache behavior, but is aggravated in HYBCACHE because isolated execution is only allowed to utilize the *subcache* portion. However, this violates our working assumption A2 that only the minority of the workload requires cache-level isolation.

We emphasize that cache-based side-channel leakage directly results from the design of the cache microarchitecture and, thus, it is reasonable to investigate the fundamental microarchitectural designs of caches for upcoming processor designs. While this does not address the problem for legacy systems, it provides an exploratory ground of ideas for upcoming processor designs. HYBCACHE is architecture-agnostic

and can be integrated with any processor architecture (we simulated it for x86 and implemented it for RISC-V). It is also compliant with any set-associative cache architecture independent of its hierarchy and organization, and whether it is virtually or physically indexed since no indexing is involved.

Intra-Process Isolation Support. HYBCACHE can also be extended, in principle, to provide *fine-grained* run-time configuration of the isolation domain *within* a process, e.g., between different threads within the same process. Besides kernel support, this requires an instruction extension to enable isolation of particular code regions or threads to different `IDIDs` or disable isolation altogether at run-time (reset its run-time `IDID` to all-zero). However, this requires the developer to identify and annotate security-sensitive code regions. Nevertheless, this is useful in practice since a process might not require cache-based side-channel resilience for its entirety but only for sensitive code such as cryptographic computations. This is a more generalizable approach that is easier and more directly applicable than implementing leakage-resilient variants for security/privacy-sensitive computations.

Deployment Assumptions. HYBCACHE assumes any TEE or trusted computing environment that is leveraged in compliance with their original design intent, i.e., that the much larger portion of the execution workload is not security-critical and only a smaller portion is security-critical and isolated in an I-Domain (A2). Otherwise, if the workload is equally balanced, the isolated execution subset would be restricted to a smaller partition of the cache and would incur a more than tolerable performance degradation especially if it is cache-sensitive. For HYBCACHE to be optimally advantageous, the workload distribution and allocation must be performed by the administrator such that the right balance of overall security and performance is achieved, as shown by the performance results in Section 6.1.

8 Related Work

We describe next the state of the art in existing defenses and their shortcomings that HYBCACHE overcomes.

8.1 Partitioning

Cache partitioning allocates to each process or security domain a separate partition of the cache, hence guaranteeing strict non-interference. Both software-based [20, 40, 51, 82] and hardware-based [24, 41, 72, 73] partitioning schemes have been proposed in recent years, where partitioning is either process-based or region-based.

Process-based partitioning. Godfrey [20] implements process-based cache partitioning using page coloring on Xen, which incurs a prohibitive performance overhead with increasing number of processes. SecDCP [72] is a way-partitioning scheme where each application is assigned a security class and cache partitioning between the security classes is dynamically managed according to the cache demand of non-secure applications. SecDCP is not scalable; selective cache flushing and repartitioning is required if the number of security classes exceeds that of allocated partitions and it may perform worse than static partitioning. Furthermore, both schemes do not support the use of shared libraries. CacheBar [82] periodically configures the maximum number of ways allocated to each process which unfairly impacts performance and cache utilization, and does not scale well with the number of security domains. DAWG [41] partitions the caches where different processes are assigned to different protection domains isolating cache hits and misses. The aforementioned schemes incur the performance overhead for the entire code, whereas HYBCACHE only enables side-channel resilience and the resulting performance overhead only for the isolated execution.

Sanctum [14] protects TEEs by flushing private caches whenever the processor switches between enclave mode and normal mode and partitioning of the LLC and assigning to each enclave a static number of sets. Sets allocated to an enclave can be used exclusively by the enclave and cannot be utilized by the OS. On the contrary, HYBCACHE allows for a flexible and dynamic sharing of cache resources between processes (thus improving performance), while preserving cache side-channel resilience for isolated execution.

Many cache partitioning and allocation schemes [37, 55, 64, 65, 75] have been proposed that focus on cache allocation mechanisms aiming to improve performance for multi-core caches. However, such schemes do not provide security guarantees. HYBCACHE addresses the security/performance trade-off by providing a configurable means to enable the side-channel resilience only for isolated execution while providing non-isolated execution with unaltered performance.

Region-based partitioning. These approaches split the cache into a secure partition reserved for security/privacy-critical memory pages and a non-secure partition for the remaining memory pages. STEALTHMEM [40] uses page coloring where several pages are colored and reserved for security-sensitive data and they remain locked in cache. CATalyst [51] leverages Intel's CAT (Cache Allocation Technology) [3] to divide the cache into secure and non-secure partitions and uses page coloring within the secure partition to isolate different processes' cache accesses to these pages. PLcache [73] locks cache lines and allocates them exclusively to particular processes such that the cache line can only be evicted by its process. However, overall performance and fairness of cache utilization are strongly impacted as the protected memory size increases in relevance to the total cache capacity. Moreover, with PLcache an attacker process may still infer the victim's memory accesses by observing that it is unable to access or evict cache lines (locked by a victim process) from a particular cache set.

Cloak [24] uses hardware transactional memory, such as Intel TSX [2], to protect sensitive computations by pre-loading the security-critical code and data into the cache at the begin-

ning of the transaction and any cache line evictions are detected by the transaction aborting. Cloak incurs prohibitively high performance overhead for memory-intense computations and requires the developer's strong involvement to identify and instrument security-sensitive code and split it into several transactions. Recent works have also explored the LLC inclusion property for defense schemes such as RIC [39] and SHARP [76]. However, both are architecture-specific, RIC requires coherence protocol modifications and cache flushing on thread migration, while SHARP requires modifications to the *clflush* instruction. HYBCACHE, however, is architecture-agnostic, and does not require cache flushing or modifications to coherence protocols or the *clflush* instruction.

8.2 Randomization

Introducing randomization involves introducing noise or deliberate slowdown to the system clock to hinder the accuracy of timing measurements as in FuzzyTime [32] and TimeWarp [57]. These techniques can only defeat attacks which rely on measuring access latency, but cannot prevent other attacks such as alias-driven attacks [28]. They compromise the precision of the clock for the remaining workload, thus affecting functionality requirements.

RPCache [73] randomizes the mapping of all memory lines of a protected application at a per-set granularity from their actual cache set to a randomly mapped cache set, by using a permutation table. NewCache [53] randomizes the mapping at a per-line granularity using a Random Mapping Table. Both RPCache and NewCache schemes do not scale well with the number of lines in the cache (not applicable for larger LLCs) and the number of protected domains. Random Fill Cache [52] mitigates only reuse-based cache collision attacks by replacing deterministic fetching with randomly filling the cache within a configurable neighborhood window whose size impacts the performance degradation incurred. It does not scale well with an increasing TEE size.

Time-Secure Cache [69] uses a set-associative cache indexed with a keyed function using the cache line address and Process ID as its input. However, a weak low-entropy indexing function is used, thus re-keying is frequently required followed by cache flushing which requires complex management and impacts performance. CEASER [63] also uses a keyed indexing function but without the Process ID, thus also requiring frequent re-keying of its index derivation function and re-mapping to limit the time interval for an attack. A concurrent work, ScatterCache [74], uses keyed cryptographic indexing that depends on the security domain, where cache set indexing is different and pseudo-random for every domain but consistent for any given key. Thus, re-keying may still be required at time intervals to hinder the profiling and exploitation efforts of an adversary attempting to construct and use an eviction set to collide with the victim access of interest. HYBCACHE, on the other hand, leverages randomization by disabling set-associativity altogether and using random replacement for isolated execution. Every given memory address can be cached in any of the available *subcache* ways and placement is random and unpredictable; it varies randomly every time the same memory line is brought in cache.

9 Conclusion

In this paper, we proposed a generic mechanism for flexible and "soft" partitioning of set-associative memory structures and applied it to multi-core caches, which we call HYBCACHE. HYBCACHE effectively thwarts contention-based and access-based cache attacks by selectively applying side-channel-resilient cache behavior only for code in isolated execution domains (e.g., TEEs). Meanwhile, non-isolated execution continues to utilize unaltered and conventional cache behavior, capacity and performance. This addresses the persistent performance/security trade-off with caches by providing the additional side-channel resilience guarantee, and the resulting performance degradation, only for the security-critical execution subset of the workload (usually isolated in a TEE) by eliminating the fundamental causes of these attacks. We evaluated HYBCACHE with the SPEC CPU2006 benchmark and show a performance overhead of up to 5% for isolated execution and no overhead for the non-isolated execution.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. We also acknowledge the relevant work of Tassneem Helal during her bachelor's thesis. This work was supported by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the German Research Foundation (DFG) through CRC 1119 CROSSING P3, and the German Federal Ministry of Education and Research through CRISP.

References

[1] INTEL. Intel Xeon Processors. https://www.intel.com/content/www/us/en/products/processors/xeon.html, 2009.

[2] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2016.

[3] INTEL. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, 2016.

[4] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/

01/reading-privileged-memory-with-side.html, 2018.

[5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. *ACM Symposium on Information, computer and communications security*, pages 312–320, 2007.

[6] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. *Cryptographers' Track at the RSA Conference*, pages 225–242, 2007.

[7] ARM Limited. ARM Security Technology – Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[8] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.

[10] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2006.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2017.

[12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution - SysTEX'17*. ACM Press, 2017.

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. https://eprint.iacr.org/2016/086.pdf.

[14] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-free High-precision L3 Cache Attack Using Intel TSX. In *USENIX Security Symposium*, 2017.

[16] Goran Doychev and Boris Köpf. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*. ACM, 2013.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[19] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[20] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[21] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

[22] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[23] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[24] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In *USENIX Security Symposium*. USENIX Association, 2017.

[25] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer-Verlag, 2016.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[28] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2016.

[29] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2011.

[30] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2010.

[31] F. Hameed, A. A. Khan, and J. Castrillon. Performance and Energy-Efficient Design of STT-RAM Last-Level Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[32] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[33] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[35] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2016.

[36] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008.

[38] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-resolution Side-channel Attack on Last-level Cache. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[39] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for mitigating LLC side-channel attacks. In *IEEE/ACM Design Automation Conference (DAC)*, 2017.

[40] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*. USENIX Association, 2012.

[41] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[42] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[43] Helge Klein. Modern multi-process browser architecture. https://helgeklein.com/blog/2019/01/modern-multi-process-browser-architecture/, 2019.

[44] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018.

[45] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-channels. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2012.

[46] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Security Symposium*, 2018.

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *USENIX Security Symposium*, pages 16–18, 2017.

[48] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A Fully Associative, Tagless DRAM Cache. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2015.

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[51] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[52] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[53] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[54] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[55] Wanli Liu and Donald Yeung. Using Aggressor Thread Information to Improve Shared Cache Management for CMPs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[56] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[57] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2012.

[58] Matt Miller. Mitigating arbitrary native code execution in microsoft edge. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-/, Jun 2018.

[59] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. *Cryptographers' Track at the RSA Conference*, pages 21–44, 2018. 10.1007/978-3-319-76953-0_2.

[60] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. Technical report, arXiv:1703.06986 [cs.CR], 2017. https://arxiv.org/abs/1703.06986.

[61] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[62] Pulp-Platform. Ariane RISC-V CPU. https://github.com/pulp-platform/ariane.

[63] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[64] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006.

[65] Daniel Sanchez and Christos Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[66] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[67] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. *CoRR*, abs/1811.07153, 2018.

[68] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy managemen. In *USENIX Security Symposium*, 2017.

[69] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2018.

[70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[71] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.

[72] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[73] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2007.

[74] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[75] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[76] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2017.

[77] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. To appear in the *Proceedings of the IEEE Symposium on Security & Privacy (IEEE S&P)*, May 2019.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[79] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. volume 7, pages 99–112. Springer, 2017.

[80] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive, Report 2016/980, 2016. https://eprint.iacr.org/2016/980.

[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[82] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.