Security evaluation of Cache Mappings Schemes

***Abstract*—Cache side-channel attacks have become a significant security threat across a variety of hardware architectures. By observing which sets of a cache are accessed by the victim, the attacker gleans critical information about the address bits in the victim's access, thereby revealing portions of secret keys used by encryption algorithms (or other sensitive information). Fundamentally, this ability to deduce information about addresses given the accessed sets depends on knowing (or discovering) how addresses are mapped to cache sets by hardware.**

In this work, we explore and evaluate the security of various cache mapping functions. Using an information-theoretic formulation, *CHASM* estimates the number of address bits that are likely leaked by different mapping schemes. Our analysis leads to several new insights. One, all one-to-one schemes that map n set-index address bits to 2^n set-indices leak all n bits. Two, based on memory footprint, programs often leak several additional (viz., tag) bits (e.g., AES leaks 39 bits out of 42 at L3). This is due to the non-uniform accesses to addresses: some addresses are accessed significantly more often than others and frequently accessed addresses potentially leak more information. Three, tag bits leak even with the use of address space layout randomization ($16-23$ bits). Four, the use of huge pages in order to reduce pressure on TLBs causes increased leakage (5.5 additional bits on average). Since many of these techniques have opposing impact on performance and security, we propose a new performance-security product metric to jointly evaluate mapping schemes for both performance and security.

***Index Terms*—**

I. Introduction

Hardware security attacks have become a significant threat to the confidentiality and integrity of data, including the data residing on cloud-based systems. An important sub-class of such security attacks is the cache side-channel attack[1]. Here, the attacker exploits the hardware provision of sharing cache space across multiple executing programs. By observing the differences in the timings of memory accesses to conflict sets, the attacker draws conclusions about the addresses accessed by the victim. In most current multi-core systems, it is trivially easy to identify the set bits of an address given the conflict set: the conflict set number corresponds to the set index address bits in a simple 1:1 fashion. Knowing the address bits accessed by the victim leads to knowledge about secret keys in cryptography applications [5]. This *address-as-data* side-channel has received significant attention from both attack and countermeasure standpoints [6], [22], [25], [33], [36].

Broadly, mitigation techniques are either partitioning-based or randomization-based. In partitioning-based techniques (for e.g., see [13], [20], [39]), the shared cache is partitioned such that each program uses its own cache partition (typically a subset of cache ways) and accesses to one partition do not result in any state changes to other partitions. While this method is effective at blocking some timing-based attacks[2], it brings a significant performance penalty: programs have varying demand for cache capacity and it is hard to impose efficient static partitioning regimes. Dynamic regimes, on the other hand, can reintroduce timing-based side-channels and generally require sophisticated governance mechanisms in hardware.

In randomization-based methods, the hardware tries to obfuscate cache set mappings. That is, how memory addresses map to cache sets is obfuscated, making it harder for the attacker to translate a timing observation on a cache set to address bits used by the victim. Recent works (for e.g., [24], [31]) have proposed to use encryption mechanisms at the last-level cache to map addresses to cache sets. Prior works (for e.g., see [10], [18], [19], [29]) have also explored the use of non-trivial mapping functions for the L1 and L2 caches, primarily from a performance improvement goal: to redistribute memory addresses across cache sets in order to reduce cache conflicts. While some of these schemes also improve security, no joint performance-with-security studies have been made.

In this work, we propose *CHASM* – an information-theoretic formulation to evaluate the strength of various cache set mapping schemes. *CHASM* estimates the amount of information about memory addresses that a mapping scheme leaks: lower the estimated leakage, the stronger the security defense. Using this formulation, we evaluate several different cache set mapping schemes in private and shared caches. Using a combination of synthetic and real programs (SPEC, and cryptography), we establish the following novel results:

- Schemes that are 1:1 mappings of the n set index address bits to 2^n sets, leak all n address bits.
- Information leakage is higher in programs with smaller memory footprints– higher order bits (tag bits, the bits not used for set index) do not vary significantly. For e.g., the *AES* benchmark leaks 39 bits (out of 42) at L3 cache when using traditional *Modulo* scheme for mapping addresses to cache sets due to its small ($< 1MB$) footprint. At the L1 level, the most secure mapping scheme (*XOR*) leaks an average of 24 bits across several workloads[3].
- Even for programs with larger foot prints, the non-uniformity of accesses to program addresses (see [28]) can leak information about the tag bits of programs.

[1]In this paper we do not directly address attacks based on speculative execution or techniques to mitigate them.

[2]Partitioning techniques do not prevent certain types of attacks [39]

[3]Mapping schemes are discussed in Section III and workloads in Section V

- Address space layout randomization (see [30]) helps but is not sufficiently strong enough in reducing tag bit leakage. Despite using *ASLR*, programs still leak anywhere between 16 to 23 (out of 42) bits.
- Interestingly, the use of huge pages in order to ease pressure on TLBs and page tables, actually results in higher leakage even for programs with large footprints. We observe an additional leakage of 5.5 bits on average when 2MB pages are used.
- Using our proposed performance-security product metric, we evaluate several mappings to identify schemes that are both simultaneously more secure as well as better performing compared to the conventional baseline. Such metrics can be valuable in trading-off performance for increased security against side channel attacks.

Using *CHASM*, hardware designers can evaluate the effectiveness of new mapping schemes. *CHASM* can also be used for operating system and application program analysis to determine how their execution characteristics affect leakage of set and tag bits.

II. Background

Figure 1 provides an overview of a typical multi-core processor architecture. Each core is provisioned with one or two levels of private caches (L1 is split between data and instruction, while L2 is unified) followed by a large shared L3 cache[4]. Depending on hardware scheduling and resource allocation, different programs share one or more caches. Programs that are co-located on the same core share private as well as shared caches. When hardware fine-grained scheduling techniques (such as Simultaneous Multi Threading [34]) are used, programs can concurrently execute while sharing private data and instruction caches. In virtualized systems, guest OSes are allocated on different cores. However, all cores share the last-level (L3) cache. Thus, the performance of programs that are simultaneously executing on different cores affect (and is affected by) the behavior of co-running programs: programs suffer additional cache misses due to programs running on other OS's and cores.

This performance variation caused by sharing of private or shared caches is the basis of cache-based side-channel attacks. The attacker program and the victim program share a cache. Depending on the level of access to the system, the attacker may share a private cache or the last-level cache with the victim. The attacker launches a covert attack on the victim by creating an information leakage channel from the victim to the attacker. Such a channel could be constructed in different ways based on system access, shared pages and observability.

In the PRIME+PROBE technique [40], the attack is deployed in three steps. In the first step, the attacker runs a spy process that fills the shared cache with its own data. In the second step, it lets the victim process execute. The victim brings its data to the shared cache, evicting some of the spy's cache blocks. In the third step, the attacker resumes the spy process which will access its data a second time, this time timing its own accesses looking for cache misses. Cache misses will indicate that the victim accessed the same set. This information is sufficient to reveal a significant portion of the address that the victim accessed.

[4]Some architectures configure L2 as a shared cache

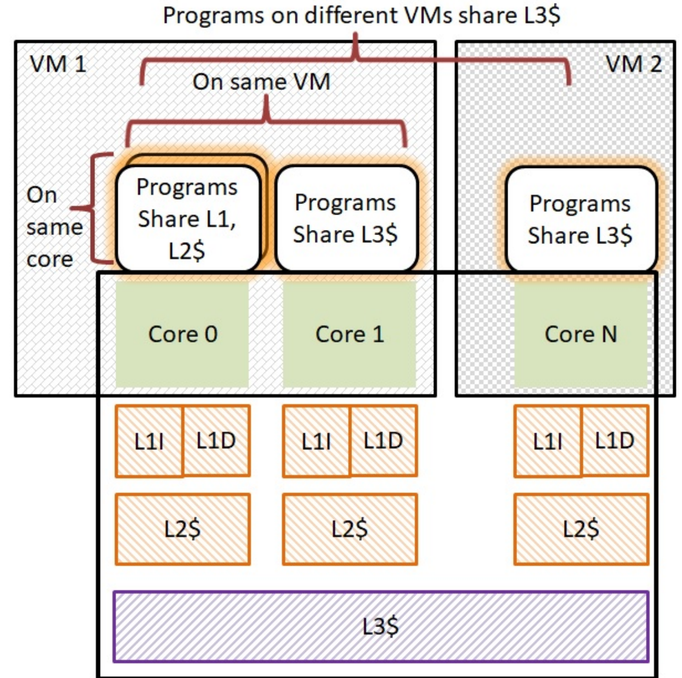


Fig. 1: Cache Sharing by Programs Running on a Multi-core Processor

In addition to PRIME+PROBE, several other attack setups have also been used, such as FLUSH+RELOAD [36], EVICT+TIME [11] and so on. Various works have demonstrated the use of different levels of caches to create covert side channels [21], [22]. At the heart of all of these attacks is the ability for the attacker to determine bits of the address accessed by the victim given the set that was accessed. The vast majority of existing hardware systems use the simple *modulo* mapping scheme to map addresses to cache sets: for a cache with $N = 2^n$ sets (organized as blocks of size $B = 2^b$ bytes), the cache controller uses the n bits $[b : b + n - 1]$ of the address to determine the set index. Figure 2 shows how the cache controller use the a address bits to obtain the block offset, the set index and tag. Under this mapping scheme, if the attacker knows the set index, then (s)he knows the corresponding address bits. Despite this vulnerability, this scheme is the prevalent mapping scheme given its simplicity and low-cost hardware implementation enabling its use in very high speed cache designs. This modulo mapping also preserves spatial localities by mapping addresses linearly to nearby sets, thus improving cache hit rates.

Researchers have also proposed newer set mapping schemes that use some additional address bits along with the set index bits, and applying a simple combinatorial function on these selected bits to compute a new set index. While most prior schemes were designed for higher performance, (by distribut-

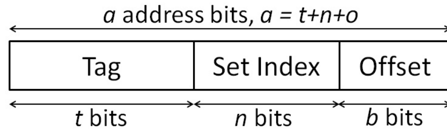


Fig. 2: Cache Set Mapping using the Modulo Mapping Scheme

ing the requests more evenly across cache sets to reduce cache misses), new schemes [24], [31] have been proposed using a cipher to create pseudo-random hash maps from addresses to cache sets. In our evaluations we assume that the same address mapping is used for all processes. In a future study we will explore the use of different mappings to different processes and modifying mappings for a given process periodically.

Thus, given the central role that cache set mapping schemes play in mitigating cache side-channel attacks, our work provides an information-theoretic framework for evaluating the security of cache set mapping schemes.

III. Cache Set Mappings

In this section, we describe the various set mapping schemes that we explored. In addition to the *modulo* mapping scheme covered in the previous section (Figure 2), we evaluate several representative schemes that are summarized in Table I. While the list of schemes is not exhaustive, the selected ones are representative of the classes of mappings used in current systems or proposed by recent research. *For the purpose of describing these techniques we will assume 64-byte cache blocks*.

The *Rotate-3* mapping uses the traditional n set index address bits $a_{(n+5)} : a_6$ and rotates these bits by three bit positions to produce the new set index ($[a_8, a_7, a_6, a_{n+5} : a_9]$).

The *XOR* scheme uses the next n bits of address that constitute the cache tag to XOR them with the set index address bits. The XORed result is used as the set index. The use of the tag bits acts as a *pseudo-randomizer* resulting in obfuscating the mapping of addresses to cache sets. In practice, this technique introduces hardware implementation challenges at L1 as the use of virtual address tag bits for indexing has to be reconciled with coherence messages sent out using physical addresses. The tag bits are not guaranteed to be the same between virtual and physical addresses requiring some additional metadata to be maintained for correctly indexing into the L1 cache. However, this scheme is known to work effectively from a performance point of view as it distributes addresses more evenly across cache sets. The *Rotate-then-XOR* scheme first rotates the set index address bits and then XORs the rotated bits with the tag bits to compute the set index. The *Square-then-XOR* scheme first squares the tag address bits, then extracts the n bits in the middle of the result to XOR them with the set index address bits. While the squaring operation is hardware-expensive, if the number of bits is small (e.g., in the L1 cache) or the operation is invoked infrequently (for e.g., at lower-level caches), the overhead of this operation becomes tolerable[5]. The *Odd-Multiplier-7* scheme multiplies the tag bits by 7, and adds the result to the set index address bits. Modulo 2^n of the result is used as the set-index.

[5]For small bit counts, a table lookup can be used.

The *Intel-Slice* scheme is borrowed from Intel's implementation for the last-level cache in Sandy Bridge as outlined in [37]. In this scheme, the L3 is viewed as a collection of cache slices, and the computation of the slice ID is done using a two-stage function of the input address. CEASER [31] proposes the use of an encryption cipher implemented as a 4-stage Feistel network to translate the incoming physical address into an L3 cache set index. In our experiments, since we did not have access to the exact implementation and keys used, we implemented the DES [27] encryption-based mapping scheme to compute the L3 set index.

We note that virtual addresses are used in mapping schemes used in the L1D cache while physical addresses are used in the L3 cache.

IV. CHASM Formulation

We first describe the attack model followed by a description of the information leakage estimation technique.

A. Attack Model

CHASM is not proposing a specific cache side-channel attack. Instead, it provides a framework for assessing the strength of various cache mapping schemes in mitigating the disclosure of victim address bits. In order to formulate this, we assume that the attacker can execute his[6] spy program on the same hardware as the victim. Depending on the level of cache sharing, the attacker may be able to create a side-channel at L1, L2 or L3 caches (see Figure 1). It is further assumed that the attacker can run synthetic programs and victim programs in order to characterize the programs for the information that they can leak. We assume that that the attacker knows the victim program's implementation (e.g., AES library used in open source implementation [7]) but not the secret keys that it uses. Thus, the assumption that the attacker can execute the victim program with controlled inputs (e.g., plaintext to be encrypted) is reasonable.

The attacker performs an offline characterization of the victim program (synthetic or one of the "real" victim programs) by tracing the execution of the program through a binary instrumentation tool such as Intel Pin [23]. The Pin tool provides detailed access to the execution behavior of a program it is run on, including the memory accesses that the program makes. The attacker creates a pin tool to monitor memory accesses and to perform a cache simulation of these accesses using the same cache configuration as the hardware intended for the attack [7]. By using the Linux pagemap [2] support, the pin tool can also translate the virtual addresses to physical addresses that are needed to study PIPT (Physically Indexed, Physically Tagged) L2 and L3 caches. It may also be noted that there are other ways such as using hardware debug technology to recover address traces.

[6]Without implying any bias, we use 'he' to refer to the attacker.

[7]The Pin tool need not be run on the same hardware, only the cache model in the tool needs to match the attacked hardware cache configuration.

Scheme Name	Scheme Description	Uses Tag Bits?	Usage
Rotate-3 [18]	Rotate-right the set-index address bits by 3 bit positions	No	L1, L3
XOR [18]	XOR the set-index address bits with least significant tag bits of address	Yes	L1, L3
Rotate-then-XOR	Rotate-right the set-index address bits by 1 bit position, and XOR the result with tag bits of address	Yes	L1, L3
Square-then-XOR	Square the tag bits of the address, and XOR the middle n bits of result with the n set-index address bits	Yes	L1, L3
Odd-Multiplier-7 [18]	Multiply the tag bits by 7, add to set-index address bits	Yes	L1, L3
Intel-Slice [37]	See description in [37]. Two-stage hash of cache slice.	Yes	L3
Encrypt	Based on scheme in [31]. We used the DES encryption cipher [27] to compute the cache set index.	Yes	L3

TABLE I: Overview of Cache Set Mapping Schemes

Once the attacker has obtained a map of addresses accessed by the victim and the sets that they map to, the attacker is able to estimate the address bits leaked by the victim when the victim program is run "in the field". In the real attack scenario, even if the victim is run with different inputs (plaintext and/or keys in case of cryptographic programs), the accessed sets give away information about the (unknown) addresses by means of the prior per-set leakage determined by the attacker. It may be emphasized that there are two interwoven aspects to this: the ability to determine set-index address bits given cache sets (hardware characteristic) and the ability to predict tag bits using program characteristics.

This formulation is applicable even when the hardware vendor does not disclose the implemented set mapping scheme. One method is for the attacker to mount a realistic timing analysis using a synthetic program as the "victim". By letting the program "sweep" through a large range of memory addresses and observing accessed sets, the attacker can create conflict sets and use them to recover the set mapping scheme. Such reverse-engineering schemes have been demonstrated (for e.g., see [14], [15], [22], [37]). Thus, our formulation does not require prior knowledge of the mapping scheme. By building conflict sets (addresses that map to the same cache sets) and estimating the leakage, *CHASM* determines which bits are leaked and the likelihood that a leaked bit is a 1 or a 0.

B. Information Leakage Estimation

The attacker's goal is to accurately predict the address bits used by the victim given that the attacker knows which set was accessed. We use an information-theoretic metric to estimate this leakage. An address bit a_i is estimated to be leaked with probability 1 given a set S, if every address that maps to S in the victim program has the same value for bit a_i (either always zero or always one). Intuitively, this is stating the fact that an address bit value remains constant for all addresses that go to a given set. Thus, knowing the set, the attacker knows the address bit. In general, if we denote the probability that a_i has a value of 1 given that the accessed set is S as $p_i = prob(a_i = 1|S)$, then the leakage of information regarding a_i is related to the entropy e_i of the bit, given by:

$$e_i = -p_i log(p_i) - (1 - p_i) log(1 - p_i) \quad (1)$$

When p_i is either close to 0 or 1, then e_i is close to 0 (less uncertainty). On the other hand, when p_i is close to 0.5, then e_i is close to 1. The information leakage l_i is defined:

$$l_i = 1 - e_i \quad (2)$$

When a bit has an entropy of 1, then it leaks no information as it is not possible for the attacker to make a statistical determination whether the bit is a 0 or a 1. When the bit has an entropy of 0, then it tells exactly whether the address bit is a 0 or a 1. For entropy values between 0 and 1, while it is not possible to accurately determine the address bit value, some statistical knowledge of the bit value is revealed.

The leakage of the set index address bits given a set S is given by:

$$L_{SetIndex}(S) = \sum_i l_i \quad (3)$$

where the summation is over all set index address bits. $L_{Tag}(.)$ is defined similarly. The total information leaked about addresses that map to a given set S is simply:

$$L(s) = L_{SetIndex}(s) + L_{Tag}(s) \quad (4)$$

We express the total leakage as a sum of the leakage contributions from set-index bits and tag bits since the two are fundamentally different: leakage of set index address bits is dependent on both the cache set mapping scheme as well as program characteristics, while the leakage of tag bits is dependent only on program and system characteristics (page allocation, ASLR, etc).

Ideally, the most secure mapping scheme leaks 0 bits of address information, while the weakest scheme leaks all $(t+n)$ address bits revealing both tag and set index bits. For a given cache, the total information leaked is estimated as a weighted-average across all the sets, where $p(S)$ is the probability of an address going to set S.

$$L^{Avg}_{Cache_SetIndex} = \sum_s L_{SetIndex} p(s) \quad (5)$$

$$L^{Avg}_{Cache_Tag} = \sum_s L_{Tag} p(s) \quad (6)$$

$$L^{Avg}_{Cache} = \sum_s L(s) p(s) \quad (7)$$

The weighing by $p(S)$ ensures that sets that have received very few accesses do not skew the overall cache-level leakage metric. This also takes program characteristics into account: if a program exhibits a non-uniform use of cache sets, then that is useful information to the attacker and must be included in the metric.

We use this formulation to evaluate and compare various cache set mapping schemes. If a set mapping scheme results in lower average information leakage (L^{Avg}_{cache}), then it is a more secure scheme compared to schemes that leak more information. Using *CHASM*, hardware designers can evaluate the effectiveness of new mapping schemes. *CHASM* can also be used for operating system and application program analysis

to determine how their execution characteristics affect leakage of set and tag bits.

C. Performance-Security Measure

Neither performance nor security alone is a useful measure. High performance under weak security is not desirable, while strong security in a slow system is expensive and inefficient. Therefore, we propose a new cache metric: *performance-security* product (denoted PS), defined as:

$$PS_{Cache} = Cache_hit_rate \times \frac{(t+n)}{L^{avg}_{Cache}} \quad (8)$$

This is a higher-is-better metric. Higher cache hit rate and lower leakage contribute to higher values of the metric. The $(t+n)$ term represents the maximum possible leakage and is used to normalize the security measure L^{Avg}_{Cache}. As a simple example, if two schemes S_1 and S_2 have hit rates 90% and 95% with respective leakages 20 and 30 bits (out of a maximum of 42 [8]), then their respective PS metrics are 1.89 and 1.33 suggesting that S_2 perhaps loses out on security a little too much. For performance, instead of using cache hit rate, a CPU-oriented metric such as IPC (Instructions Per Cycle) can also be used.

D. Using CHASM

Figure 3 outlines how *CHASM* can be used. Hardware designers, and software developers (operating system and application program) can use *CHASM* to characterize the security of cache address mapping techniques and program characteristics.

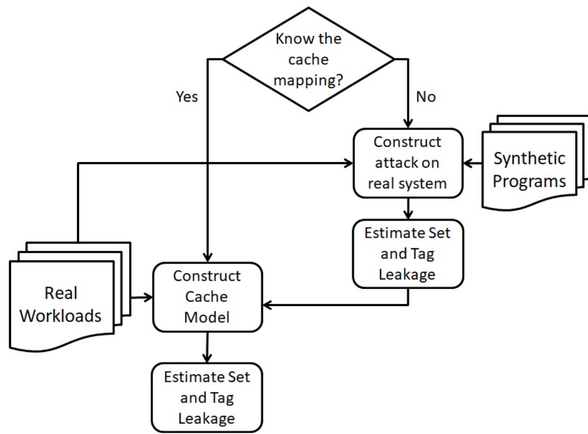


Fig. 3: Outline of using *CHASM*

When the cache mapping scheme is known, a cache model that implements the scheme can be used with real workloads to study the leakage caused by the scheme on these workloads. As will be demonstrated in Section VI, workload memory footprint, and non-uniform access patterns of applications have an impact on observed leakage.

When the cache mapping scheme is unknown, synthetic programs can be used as victim programs on which "attacks" are performed – i.e., their address-to-set mappings are observed via side channel attacks. From these observations, the leakage of set-index bits is established by creating and monitoring conflict sets. Further workload-specific leakage can be determined by similar attacks or via cache models that are reverse-engineered using the observed mappings.

[8]As stated previously, we assume a 64-byte cache line size, and 48-bit addresses.

V. Experimental Methodology

In order to develop and evaluate *CHASM*, we use a trace-based methodology. Memory access traces of various workloads are collected using the Intel PIN [23] tool. These are virtual addresses that are suitable for L1 cache studies (L1 is VIPT - Virtually Indexed Physically Tagged). We also obtain corresponding physical addresses during the PIN tool execution by using a combination of Linux */proc/pagemap* and */proc/kpageflags* utilities [2]. Physical addresses are used for L3 cache studies (L3 is PIPT - Physically Indexed Physically Tagged). As is standard practice, we collect traces after skipping an initial warm-up period.

These traces are injected into Moola [32] - a multi-core, cache hierarchy simulator. The cache simulator is configured to match the Intel Skylake [8] cache configuration (L1D: 32KB 8-way, L2: 256KB, 4-way and L3: 8MB, 16-way, all caches use 64B blocks). The cache simulator is modified to support all the cache mapping schemes listed in Table I. The simulator is also enhanced to measure information leakage defined in Equations 5, 6 and 7. Finally, the simulator reports the relevant statistics – cache performance, and information leakage.

Our workloads comprise synthetic programs, SPEC [12] benchmarks and off-the-shelf cryptography programs. Synthetic programs uniformly access the elements of an array. By varying the size of the array, we observe the impact of memory footprint on security. We explore 6 different sizes and the corresponding workloads are denoted *synth_16KB, synth_128KB, synth_1MB, synth_8MB, synth_64MB* and *synth_512MB*. We use a subset of SPEC benchmarks that are memory-intensive: *bwaves, bzip2, cactusADM, GemsFDTD, leslie3d, libquantum, milc, mcf, soplex* and *zeusmp*. Our cryptography programs include *AES, RSA* and *SHA*. Traces collected from these programs are fed to the cache simulator and simulated for 10 billion memory accesses.

VI. Results

We first evaluate L1D mapping schemes for their security using synthetic, SPEC and Cryptography benchmarks, followed by an evaluation of L3 mapping schemes. We skip the L2 cache as its results are consistent with L1 and L3.

A. Security of L1D Cache Mapping Schemes

In order to understand sources of leakage, we examine the leakage of set bits.

1) Leakage of Set-Index Bits: Figure 4 plots the average information leakage ($L^{Avg}_{L1D_SetIndex}$, refer Equation 5) from set-index address bits observed in synthetic programs across all the evaluated L1 mapping schemes. It may be observed that mapping schemes that use only the set index portion of the address bits (first two bars for each workload) leak consistently more bits than schemes that use additional tag bits in creating set indexes. The *Modulo* and *Rotate-3* schemes possess the

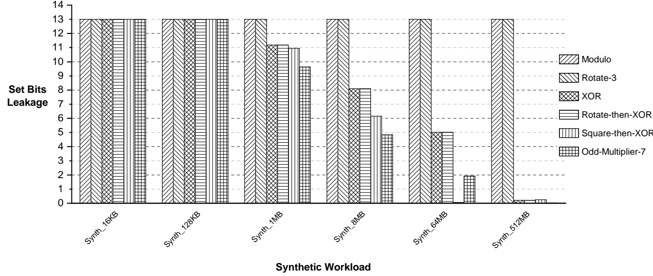


Fig. 4: Leakage of Set-index Address bits in L1D on Synthetic Programs

property that given a set, the corresponding address bits $a_{11} : a_6$ [9]are constant and therefore these bits are predicted exactly. In schemes that also use tag bits for set-index calculation (next four bars for each workload, namely, *XOR*, *Rotate-then-XOR*, *Square-then-XOR*, and *Odd-Multiplier-7*), the set index is not a one-to-one mapping of the n set-index address bits any more and the correlation between an accessed set and the values of the address bits $a_{11} : a_6$ breaks down. Depending on the extent of pseudo-randomization induced by using tag bits, the amount of leakage varies. These schemes perform better on programs with larger footprints (1MB or higher) but perform poorly on smaller footprints. This is because smaller footprints do not cause sufficient variation in the tag bits and therefore the use of tag bits did not induce sufficient randomization.

These results reveal that in the general case, every mapping scheme that uses only the set-index portion of address bits to map the 2^n possible address bit values to 2^n sets in a 1-1 function is *weak*: it reveals all n bits. If the attacker does not know the mapping, all that the attacker has to do is discover the mapping function by constructing conflict sets and observing the addresses that collide. From these addresses, the set-index bits would be revealed as they are constant across all these addresses.

Next, we evaluate the leakage of information for selected SPEC and Cryptography workloads. Figure 5 plots this leakage across all mapping schemes. While the first two schemes that make use of only the set-index address bits continue to leak all 6 bits, the tag-based schemes also leak many of these set bits, with an average of 2.7 bits. In particular, *AES* and *SHA* encryption programs show high leakage. This is attributable to the lack of variation in tag bits, thereby enabling the attacker to accurately estimate most of the set index bits.

2) Leakage of Tag Bits: Next, we study the leakage of tag bits. In the ideal case, knowledge of the conflicting addresses should not reveal any information about the tag portion of the addresses. However, we find that the most significant bits of the virtual address remain constant across all the accesses and therefore easily predictable.

Figure 6 plots the leakage of tag bits ($L^{Avg}_{L1D_Tag}$, refer Equation 6) for synthetic programs. It may be observed that programs with larger footprints exhibit less tag leakage (synth_16KB leaks about 29 tag bits out of 36 bits while synth_512MB only 16 tag bits). This is due to the fact that programs with larger footprint cause a higher rate of higher-order address bit flips as compared to smaller programs, thereby reducing the information leakage about these bits. In this sense, programs with greater memory footprints are more secure against such side-channel attacks.

B. Leakage Under ASLR

Address Space Layout Randomization (ASLR, see [30]) is an OS feature that offers memory protection against attacks by randomizing the base addresses of various program sections such as the stack, heap and text. Across different runs of the same process, the OS places the process sections at different locations thereby making it harder for attackers to reliably determine addresses used by the victim via cache side-channel attacks.

In order to test the effectiveness of ASLR and to identify information leakage under ASLR, we ran our synthetic workloads 50 times each and obtained a merged trace for each workload[10]. By merging the address accesses from different runs of the same workload, we are able to capture the randomness introduced by ASLR and measure the resulting reduction in leakage.

Figure 7 reports our findings on the L1D cache. Two observations may be drawn by comparing these figures to the leakages reported on an individual run of synthetic workloads in Figures 4 and 6. One, ASLR helps reduce the leakage of set-index bits. With mapping schemes that use tag bits (last four bars for each workload), the set leakage drops down to nearly 0, suggesting that a combination of such a mapping scheme operating under ASLR is effective. This is a positive result from a security standpoint – especially for programs with very small footprints. Two, ASLR is not quite effective at randomizing higher-order tag bits. While it helps reduce the leakage by as much as 6.5 bits in *synth_16KB* and by 8 bits in *synth_8MB*, there is still significant tag leakage (23 bits in *synth_16KB* and 16 bits in all others) that can be accurately predicted and leveraged by attackers.

This experiment reveals that a combination of including tag bits in computing set indices, coupled with the use of ASLR can provide a secure mapping scheme preventing leakage of set-index bits. At the same time, ASLR is not strong enough to prevent leakage of several tag bits and a stronger scheme is needed to achieve this.

C. Security of L3 Cache Mapping Schemes

In order to estimate leakage in the L3 cache, we first examine the leakage of set bits. Figure 8 plots the average information leakage ($L^{Avg}_{L3_SetIndex}$, refer Equation 5) from set-index address bits observed in synthetic programs across all the evaluated L3 mapping schemes.

The first two schemes (*Modulo* and *Rotate-3*) do not use any other bits other than set-index bits for cache mapping and leak

[9]L1D cache contains 64 sets.

[10]SPEC and Cryptography workloads exhibit similar trends and for brevity, we omit their details here.

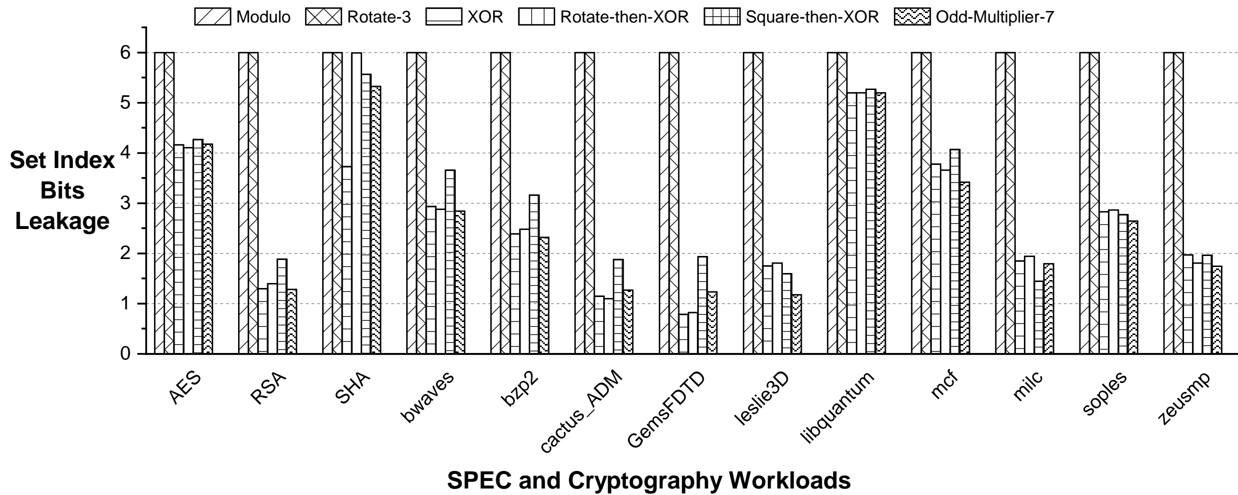


Fig. 5: Leakage of Set-index Address bits in L1D on SPEC and Crypto Benchmarks

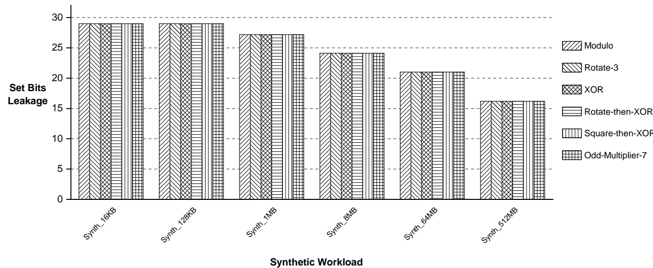


Fig. 6: Leakage of Tag Address bits in L1D on Synthetic Programs

all set-index bits (13 bits in our experiments based on the Intel Skylake Cache configuration). The next six schemes make use of tag bits and achieve reduction in leakage – however, the reduction depends on the memory footprint. As the footprint increases, the variation in the tag bits also goes up making pseudo-randomization more effective. In contrast to L1D, this result shows that it requires much larger memory footprints in victim programs to be able to reduce leakage at the L3 cache level. As L3 cache sizes continue to grow (for e.g., Intel Xeon Platinum series processors have a $39MB$ L3 cache), it requires larger programs to effectively secure the set-index address bits from being leaked. We tested this by executing our workloads on larger L3 caches (upto 64MB) and observing that the leakage grew with cache size[11].

The *Intel-Slice* scheme leaks about 10 bits even with higher memory footprint programs. This mapping scheme modifies only the 3 cache slice bits and does not affect the set indexing within a slice (set indexing uses the conventional modulo mapping). Thus out of the 13 bits used for slice+set-index, only the slice bits are secure, and the set-index bits are easily revealed. Among all the evaluated schemes, the *Encypt* scheme is the best-performing.

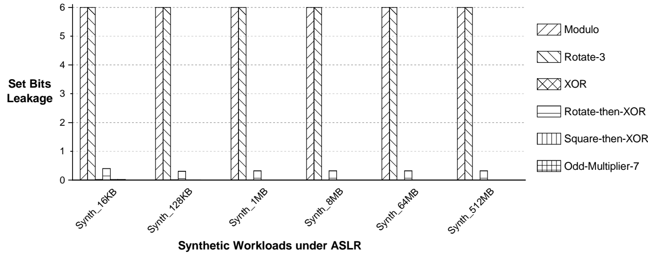
Figure 9 plots the leakage in L3 set-index bits on SPEC and Cryptography workloads[12]. These results are similar to synthetic workloads. Schemes incorporating the use of tag bits generally perform significantly better than schemes that do not. Cryptography workloads – due to their small footprints – leak more bits under all schemes (AES leaks a total of 39 bits – set-index+tag – under *Modulo*). The leakage in tag bits follows a trend similar to L1D (see Figure 6) and is not repeated here.

1) Non-Uniformity and Leakage: It is not uncommon for some cache sets to be accessed hundreds of times more often than less frequently accessed sets. Schemes that tend to distribute the addresses more evenly across all the available cache sets may do better at reducing leakage. In order to confirm this, we measured the number of accesses by set for several workloads on a variety of mapping schemes. Figure 10 shows the distribution in the number of accesses by set in the *libquantum* workload for the *Modulo*, *XOR* and the *Square-then-XOR* schemes. The *Modulo* shows the most non-uniformity and has the highest set leakage (13 bits). The *XOR* scheme shows greater non-uniformity than *Square-then-XOR* and is correlated with its higher set-index leakage (4.89) as compared to the *Square-then-XOR* scheme (0.26). Similar results were observed in other workloads too, suggesting that schemes that distribute more evenly across all the cache sets may be better at reducing per-set leakage.

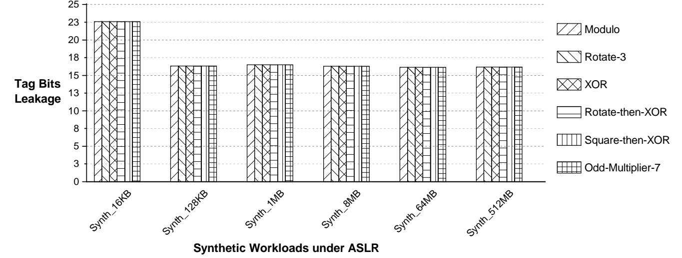
2) Impact of Page Size on Leakage: Large physical pages [1], [3] (also called *superpages*, using page sizes of 2MB through 1GB) have been supported as a way to allocate and manage large contiguous chunks of physical memory efficiently. Large pages reduce TLB (Translation Lookaside Buffer) pressure and page-walk penalty by replacing many individual small page entries in a page table by a single entry. However, use of large pages comes with a security side-effect: all addresses that map to a large physical page have the same higher-order bits as compared to addresses that map

[11] We omit details of this experiment due to lack of space

[12] For lack of space, we omitted the *Rotate-3* scheme as its leakage is exactly the same as that of *Modulo*



(a) Leakage of Set-Index Bits



(b) Leakage of Tag Bits

Fig. 7: Leakage Reduction under ASLR

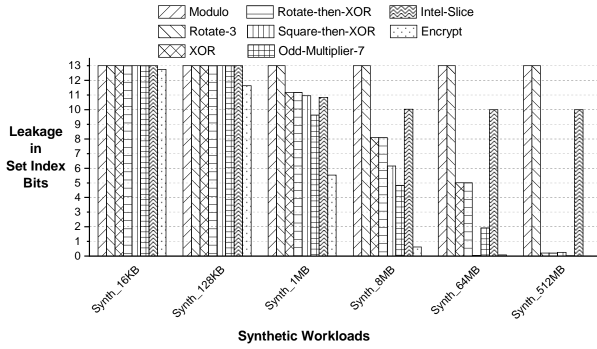


Fig. 8: Leakage of Set-index Address bits in L3 on Synthetic Programs

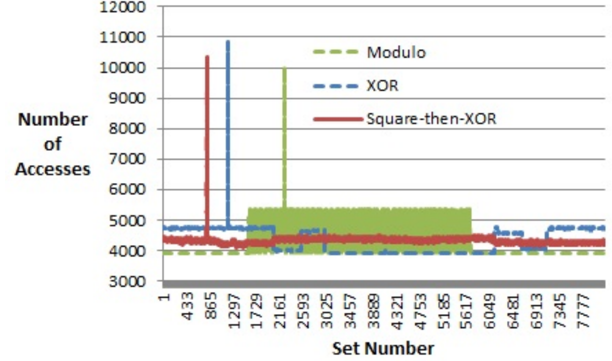


Fig. 10: Variation in Set Accesses by Schemes in *libquantum*

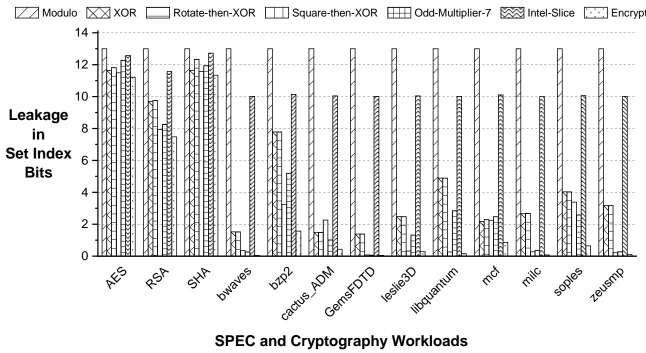


Fig. 9: Leakage of Set-index Address bits in L3 on SPEC, Cryptography Workloads

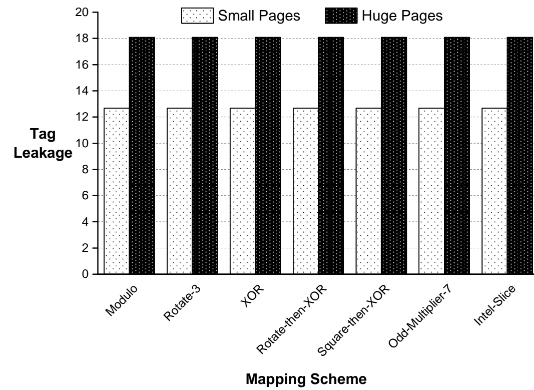


Fig. 11: Impact of Huge Pages

to different non-contiguous small physical pages. Thus, using large pages can result in greater address bit leakage. In order to observe this, we compared two runs of a synthetic program – *huge_pages* allocates and uses huge (2MB in our experiments) pages, while *small_pages* uses small (4KB) pages. Figure 11 compares leakage of tag address bits in the L3 cache. Across all the schemes, the use of huge pages consistently results in leaking several additional address bits (about 5.5 in our experiments).

D. Performance-Security Measure

We use the metric PS_{Cache} defined in Equation 8 to study the joint effect of each scheme on performance and security. Figure 12 plots this metric for SPEC and Crypto workloads for each scheme on L1D. In all benchmarks except *AES*, the tag-based schemes show a better value of this metric as compared to the *Modulo* and *Rotate-3* schemes (note that this is a higher-is-better metric). *AES* is an exception as it leaks several more tag bits under the *XOR* scheme. On average (geometric mean), the *XOR* and *Odd-Multiplier-7* schemes perform the best. This evaluation shows that the PS metric is effective at capturing the joint performance & security impact of cache mapping schemes and can be used as a reliable indicator for future evaluations.

The above results and discussion indicate that hardware-wide static cache mapping schemes are leaky. Set-index bits are almost entirely revealed unless tag-based methods are

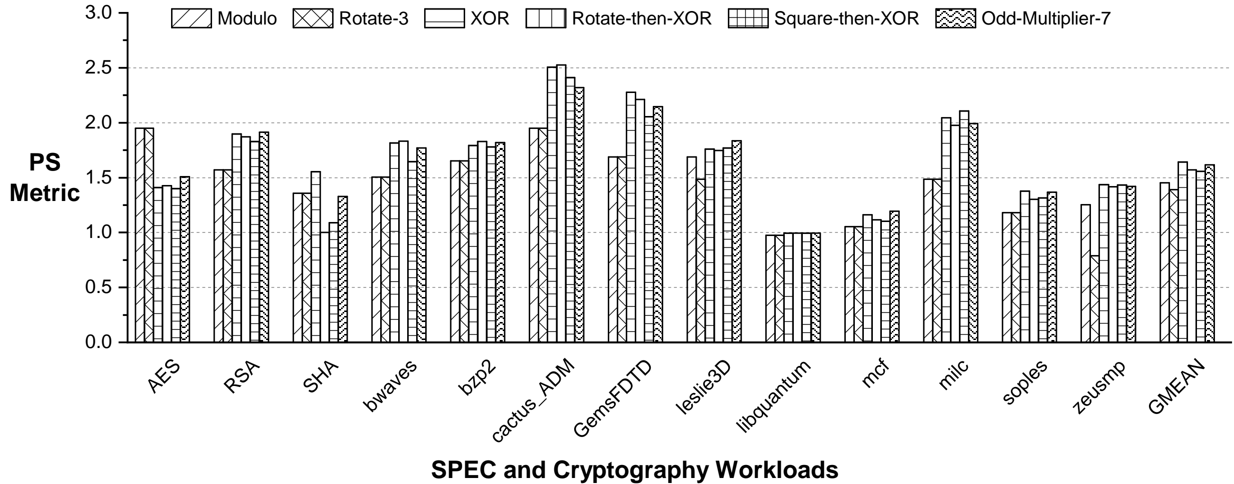


Fig. 12: L1D Performance-Security Product Metric of SPEC and Cryptography Workloads

used. Even with the use of tag bits, leakage still exists in smaller programs. Considerable leakage of tag bits occurs in all schemes even when ASLR is used. Large physical pages tend to leak even more. Large cache sizes also result in higher leakage. Our findings motivate the need for incorporating stronger mapping schemes that are either application-specific, or dynamic or both.

VII. Related Works

Cache set mappings have received attention mainly from a performance perspective. These studies focus solely on performance and do not consider their security implication. The work in Kavi et al [18] provides a comprehensive comparison of several mapping schemes used in low-associativity caches. The work in Givargis [10] explores mapping schemes that improve performance by considering correlations among address bits. Kharbutli et al [19] explore the prime-modulo and prime-displacement mapping functions. Patel et al [29] propose profiling and choosing application-specific address bits to minimize set conflicts. The work in [28] proposes the use of a programmable multiplexer to select the best-performing scheme from among several supported schemes dynamically. In addition, several adaptive or programmable cache mapping schemes have also been proposed.

The work in [31] explores the use of encryption-based mapping of physical addresses to LLC sets with emphasis on addressing security. In this work, the authors not only propose encryption but also re-encrypt and migrate cache blocks periodically to prevent the attacker from learning the encrypted mapping. Intel uses an undocumented hash function of the physical address bits to compute the last-level cache slice index. The details of this hash function were revealed in [24]. We evaluate these schemes in our work.

Address space layout randomization (ASLR) [30] was introduced by Linux as a guard against address-based attacks by randomizing the bases of text, stack, heap and mmap sections. It has however been shown that ASLR does not offer a strong defense [4], [16], [35]. In contrast to these works, our work is not another cache mapping (or address randomization) scheme: rather, it provides a framework to evaluate the security of mapping schemes and highlights the role of program characteristics in the security metric.

Orthogonal to *CHASM*, Zankl et al [38] provide a framework for detecting leakage of modular exponentiation software via instruction caches. The work in Doychev et al [9] proposes *CacheAudit* - a framework for automatic static analysis of cache side channels. *CHASM* differs from this framework in that *CHASM* is driven by address traces factoring in aspects of the run-time system: physical addresses, use of large pages, ASLR and so on. Gleipnir [17] is a Valgrind [26] based cache memory profiling tool that offers cache performance correlated to program source-code structures thereby enabling the exploration of sophisticated cache mapping and other cache performance management functions.

VIII. Conclusions

This work presented *CHASM* – a framework for evaluating strength of the security of cache mapping schemes. We evaluated several schemes at L1 and L3 caches under different system conditions (ASLR, large pages). Our evaluations reveal several insights about the vulnerabilities of cache mappings, including that smaller memory footprints leak more information, that non-uniformity of accesses leaks more information, that ASLR is only marginally effective and that "huge" pages leak more information. These findings indicate that more sophisticated techniques for hiding address mapping to cache sets are needed. Moreover, obfuscating techniques at the OS level as well as application/algorithm level are needed to increase the cost of side-channel attacks.

REFERENCES

[1] “Huge pages - the linux kernel archives.” [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[2] “Pagemap, from the userspace perspective.” [Online]. Available: https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[3] “Transparent hugepage support.” [Online]. Available: https://www.kernel.org/doc/Documentation/vm/transhuge.txt

[4] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, “Cain: Silently breaking aslr in the cloud.” in *WOOT*. USENIX Association, 2015. [Online]. Available: http://dblp.uni-trier.de/db/conf/woot/woot2015.html

[5] D. J. Bernstein, “Cache-timing attacks on aes,” Tech. Rep., 2005.

[6] J. Bonneau and I. Mironov, “Cache-collision timing attacks against aes,” in *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 201–215. [Online]. Available: http://dx.doi.org/10.1007/11894063_16

[7] A. C, R. P. Giri, and B. Menezes, “Highly efficient algorithms for AES key retrieval in cache access attacks,” in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, March 2016, pp. 261–275.

[8] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, “Inside 6th-generation intel core: New microarchitecture code-named skylake,” *IEEE Micro*, vol. 37, no. 2, pp. 52–62, Mar 2017.

[9] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, “Cacheaudit: A tool for the static analysis of cache side channels,” in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 431–446. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534804

[10] T. Givargis, “Improved indexing for cache miss reduction in embedded systems,” in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC ’03. New York, NY, USA: ACM, 2003, pp. 875–880. [Online]. Available: http://doi.acm.org/10.1145/775832.776052

[11] Z. He and R. B. Lee, “How secure is your cache against side-channel attacks?” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 ’17. New York, NY, USA: ACM, 2017, pp. 341–353. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124546

[12] J. L. Henning, “Spec cpu2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[13] Intel, “Introduction to Cache Allocation Technology in the Intel Xeon processor E5 v4 family,” 2016. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[14] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 591–604. [Online]. Available: https://doi.org/10.1109/SP.2015.42

[15] ——, “Systematic reverse engineering of cache slice selection in intel processors,” in *Proceedings of the 2015 Euromicro Conference on Digital System Design*, ser. DSD ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 629–636. [Online]. Available: https://doi.org/10.1109/DSD.2015.56

[16] Y. Jang, S. Lee, and T. Kim, “Breaking kernel address space layout randomization with intel tsx,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: ACM, 2016, pp. 380–392. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978321

[17] T. Janjusic and K. Kavi, “Gleipnir: A memory profiling and tracing tool,” *SIGARCH Comput. Archit. News*, vol. 41, no. 4, pp. 8–12, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2560488.2560491

[18] K. Kavi, I. Nwachukwu, and A. Fawibe, “A comparative analysis of performance improvement schemes for cache memories,” *Comput. Electr. Eng.*, vol. 38, no. 2, pp. 243–257, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.compeleceng.2011.12.008

[19] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, “Using prime numbers for cache indexing to eliminate conflict misses,” in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 288–. [Online]. Available: https://doi.org/10.1109/HPCA.2004.10015

[20] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, “Dawg: A defense against cache timing attacks in speculative execution processors,” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 974–987. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00083

[21] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, “Catalyst: Defeating last-level cache side channel attacks in cloud computing,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 406–418.

[22] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: https://doi.org/10.1109/SP.2015.43

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[24] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon, “Reverse engineering intel last-level cache complex addressing using performance counters,” in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, ser. RAID 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 48–65. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26362-5-3

[25] M. Mushtaq, A. Akram, M. K. Bhatti, R. N. B. Rais, V. Lapotre, and G. Gogniat, “Run-time detection of prime + probe side-channel attack on aes encryption algorithm,” in *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, Oct 2018, pp. 1–5.

[26] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[27] NIST, “Data encryption standard (DES),” 1999. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25

[28] I. Nwachukwu, K. Kavi, F. Ademola, and C. Yan, “Evaluation of techniques to improve cache access uniformities,” in *2011 International Conference on Parallel Processing*, Sep. 2011, pp. 31–40.

[29] K. Patel, E. Macii, L. Benini, and M. Poncino, “Reducing cache misses by application-specific re-configurable indexing,” in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 125–130. [Online]. Available: http://dx.doi.org/10.1109/ICCAD.2004.1382556

[30] Pax, “Address space layout randomization ASLR,” 2003. [Online]. Available: http://pax.grsecuritynet/docs/aslr.txt

[31] M. K. Qureshi, “CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping,” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 775–787. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00068

[32] C. Shelor and K. Kavi, “Moola: Multicore cache simulator,” in *30th International Conference on Computers and Their Applications (CATA-2015)*, 2015.

[33] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks on aes, and countermeasures,” *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00145-009-9049-y

[34] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ser. ISCA ’98. New York, NY, USA: ACM, 1998, pp. 533–544. [Online]. Available: http://doi.acm.org/10.1145/285930.286011

[35] P. Umbelino, “Aslr cache attack defeats address space layout randomization,” 2017. [Online]. Available: https://hackaday.com/2017/02/15/aslrcache-attack-defeats-address-space-layout-randomization/

[36] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671271

[37] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the intel last-level cache," Cryptology ePrint Archive, Report 2015/905, 2015, https://eprint.iacr.org/2015/905.

[38] A. Zankl, J. Heyszl, and G. Sigl, "Automated detection of instruction cache leaks in modular exponentiation software," in *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*, 2016, pp. 228–244. [Online]. Available: https://doi.org/10.1007/978-3-319-54669-8_14

[39] S. K. S. D. L. W. H. Y. Zhang, N, "Truspy: Cache side-channel information leakage from secure world on arm devices," in *IACR Cryptology ePrint ArchiveProceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[40] Y. Zhang, "Cache side channels: State of the art and research opportunities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2617–2619. [Online]. Available: http://doi.acm.org/10.1145/3133956.3136064