CHASM: Security Evaluation of Cache Mapping Schemes

Fernando Mosquera, Nagendra Gulur, Krishna Kavi, Gayatri Mehta, and Hua Sun

University of North Texas, Denton, Texas, USA**

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 evaluate the security of the various cache mapping functions. Using an information-theoretic formulation, our framework (denoted *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 L2). Three, tag bits leak even with the use of address space layout randomization ($16 - 33$ bits). Four, the use of huge pages in order to reduce pressure on TLBs increases leakage (5 additional bits on average). Since many of these techniques have opposing impact on performance and security, we use a new *security-delay* ratio metric to jointly evaluate mapping schemes for both performance and security.

1 Introduction

Hardware security attacks have become a significant threat to the confidentiality and integrity of data, including data residing on personal devices [4, 26]. An important sub-class of such security attacks is the cache side-channel attack[1]. Here, the attacker exploits the behavior of cache space across multiple executing programs. By observing the differences in the timings of memory accesses to cache memory locations (cache lines or 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 address bits used as a cache set. 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

** Contact authors at: fernandomosquera@my.unt.edu

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

significant attention from both attack and countermeasure standpoints [6,18,21, 27,31].

Mitigation techniques can broadly be classified as partitioning or randomization based. In partitioning-based techniques (for e.g., see [13, 17, 34]), 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 (e.g., see [34]) and generally require sophisticated governance mechanisms in hardware.

In randomization-based methods, the hardware obfuscates how memory addresses map to cache sets, 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., [20,24]) have proposed to use encryption mechanisms at the last-level cache to map addresses to cache sets. Prior works (for e.g., see [9,15,16]) have also explored the use of non-trivial mapping functions for L1 and L2 caches, primarily from a performance improvement goal: to redistribute memory addresses across cache sets in order to reduce cache conflicts. Some of these schemes may also improve security.

In this work, we propose *CHASM* – an information-theoretic measure 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 for private and shared caches. Using a combination of synthetic and real programs (SPEC, and cryptography), we establish the following:

- 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 L2 cache when using traditional *Modulo* scheme[3] for mapping addresses to cache sets due to its small ($< 1MB$) footprint.
- Even for programs with larger foot prints, the non-uniformity of accesses to program addresses (see [22]) can leak information about the tag bits of programs.
- Address space layout randomization (see [23]) helps but is not sufficiently strong enough in reducing tag bit leakage. Despite using *ASLR*, programs still leak anywhere between 16 to 33 (out of 42) bits.

[2] Partitioning techniques do not prevent certain types of attacks [35]
[3] Mapping schemes are discussed in Section 3 and workloads in Section 5

– Interestingly, the use of huge pages to reduce TLBs and page table sizes, actually results in higher leakage even for programs with large footprints. We observe additional 5 bits of leakage bits (average) with 2MB pages.
– Using our proposed *security-delay* metric, we evaluate several mappings to identify schemes that are both more secure and better performing compared to the conventional baseline. Such metrics can be valuable in making quantitative decisions about performance-security trade-offs.

2 Background

Figure 1 provides an overview of a typical multi-core embedded processor architecture. Each core is provisioned with private L1 caches (L1 is split between data and instruction) followed by a large shared L2 cache. 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)are used, programs can concurrently execute on the same core sharing private data and instruction caches. Programs concurrently executing on different cores share the L2 cache. Thus, simultaneously executing programs affect (and are affected by) the performance of co-running programs: programs suffer additional cache misses due to interference.

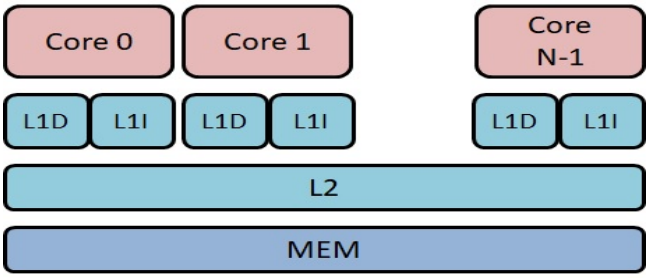

Fig. 1: Overview of Multi-core Embedded Processor

This performance variation caused by sharing of private or shared caches is the basis of cache-based side-channel attacks. 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. For illustration, in the PRIME+PROBE technique [35], 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 accessing its data a second time. Cache misses of spy data (observed by timing

the accesses) will indicate that the victim accessed the same set. This information is sufficient to reveal a portion of the address that the victim accessed.

In addition to PRIME+PROBE, several other attacks exist, such as FLUSH & RELOAD [31], EVICT & TIME [11] and so on. Various works have demonstrated the use of different levels of caches to create covert side channels [18]. 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 uses 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 of implementation.

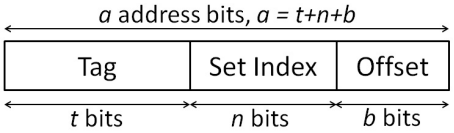

Fig. 2: Cache Set Mapping using the Modulo Mapping Scheme

Researchers have proposed newer set mapping schemes that use some additional address 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 distributing the requests more evenly across cache sets to reduce cache misses), new schemes [20, 24] 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 for 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 proposes an information-theoretic framework for evaluating the security of cache set mapping schemes.

3 Evaluated 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 1.We selected a representative set 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 $([a_8, a_7, a_6, a_{n+5} : a_9])$.

Scheme Name	**Scheme Description**	**Uses Tag Bits?**	**Usage**
Rotate-3 [15]	Rotate-right the set-index address bits by 3 bit positions	No	L1, L2
XOR [15]	XOR the set-index address bits with least significant tag bits of address	Yes	L1, L2
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, L2
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, L2
Odd-Multiplier-7 [15]	Multiply the tag bits by 7, add to set-index address bits	Yes	L1, L2
Intel-Slice [32]	See description in [32]. Two-stage hash of cache slice.	Yes	L2
CEASER	Encryption-based mapping scheme. See [24].	Yes	L2

Table 1: Overview of Cache Set Mapping Schemes

The *XOR* scheme uses n bits of address from the cache tag portion 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[4]. 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.

The *Intel-Slice* scheme is borrowed from Intel's implementation for the last-level cache in Sandy Bridge as outlined in [32]. In this scheme, the L2 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 [24] proposes the use of an encryption cipher implemented as a 4-stage Feistel network to translate the incoming physical address into an L2 cache set index[5].

We note that virtual addresses are used in mapping schemes used in the L1D cache since the majority of L1 caches are virtually-indexed while physical addresses are used in the L2 cache since L2 caches are almost always physically-indexed.

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

[5] In the original work, this scheme was applied to the shared L3 cache.

4 CHASM Formulation

We first describe how our formulation can be used followed by a description of the information leakage estimation technique.

4.1 Use Model

CHASM is not proposing a specific cache side-channel attack or a countermeasure. Instead, it provides a framework for assessing the strength of various cache mapping schemes in mitigating the disclosure of victim address bits. The typical use of *CHASM* is for the design and evaluation of new cache mapping schemes. Hardware architects can explore various mapping schemes (such as those described in Section 3) to identify the most secure schemes. Algorithm and OS developers can explore different software implementation techniques for how secure/vulnerable they are from a cache-based side-channel attack perspective.

CHASM formulation estimates the leakage of the victim's address bits given that the attacker knows which sets are being accessed by the victim. Tools such as the PIN tool for ARM [10] or Intel [19] can be used to collect a trace of a program's memory accesses. These traces are fed to a cache simulator that estimates information leakage using the *CHASM* formulation described next. The cache simulator implements the desired cache mapping schemes and provides statistical estimates of which address bits are leaked and the likelihood that a leaked bit is a 1 or a 0.

4.2 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, if every address that maps to a given set 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 map to a given set, and an attacker can find the value of an address bit if he knows the set used. Additionally, correlations between address bits also reveal information. For instance, if address bit a_j is 1 whenever address bit a_i is 0, then knowing a_i, a_j can be predicted.

While the concept of *mutual information* (see [14, 29]) is widely used to define leakage in the presence of correlations, its computational complexity is high requiring multi-dimensional conditional probabilities to be estimated[6]. We define a simpler metric that uses only pair-wise correlations between address bits. Our metric computes both the individual per-bit leakage as well as correlated leakage in order to estimate the effective leakage.

Individual Leakage: If we denote the probability that a_i has a value of 1 when it maps to set 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:

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

[6] In our case, leakage in bit i would depend on all other 40–50 bits requiring $2^{40} - 2^{50}$ probabilities to be estimated.

When p_i is either close to 0 or 1, then e_i is close to 0 (very small uncertainty). On the other hand, when p_i is close to 0.5, then e_i is close to 1. This individual bit leakage il_i is defined:

$$il_i = 1 - e_i \tag{2}$$

A high value of il_i indicates that by simply knowing the set, bit a_i can be predicted.

Correlated Leakage: In almost all applications, lower-order address bits toggle more often than higher-order bits. Thus we compute correlations of bit a_j with all other previous bits a_{j-1} through a_6[7]. Let $SAME(i,j)$ denote the number of times that a_i and a_j have the same values among addresses mapping to set S. Let $DIFF(i,j)$ denote the number of times that a_i and a_j have different values. The correlation of a_j with a_i $(j > i)$ is defined:

$$c_{j,i} = 1 - \frac{min(SAME(i,j), DIFF(i,j))}{max(SAME(i,j), DIFF(i,j))} \tag{3}$$

If the bits are strongly correlated or anti-correlated, then $c_{j,i}$ is close to 1. Thus we define the correlated leakage $cl_{j,i}$ of j due to i as:

$$cl_{j,i} = el_i \times c_{j,i} \tag{4}$$

where el_i is the *effective leakage* of bit i defined in Equation 5. The correlated leakage of a_j due to a_i is high if a_i has high effective leakage *and* a_j has high correlation with a_i. We now define the effective leakage of a_j as:

$$el_j = max(il_j, cl_{j,j-1}, cl_{j,j-2}, .., cl_{j,6}) \tag{5}$$

Effective leakage of a_j is set to the maximum of its individual leakage, and its correlated leakages with preceding bits a_{j-1} through a_6. If a bit possesses strong (anti-) correlation with a preceding bit or suffers from high individual leakage, then its effective leakage is high.

We express the total leakage as a sum of the effective leakage contributions from all the relevant address bits: set-index bits and tag bits. This is given by:

$$L(S) = \sum_i el_i \tag{6}$$

where the summation is over all set index and tag address bits.

Ideally, the most secure mapping scheme leaks 0 bits of address information, while the weakest scheme leaks all 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 mapping to set S.

$$L_{Cache}^{Avg} = \sum_s L(s)p(s) \tag{7}$$

[7] We stop at a_6 as bits a_0 through a_5 are block offset bits that have no impact on cache mapping. In this paper we assume 64-byte cache blocks.

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 compare various cache set mapping schemes. If a set mapping scheme results in lower average information leakage (L_{cache}^{Avg}), then it is deemed to be more secure scheme.

4.3 Security-Delay Measure

Neither performance nor security alone is a useful measure. High performance under weak security is not desirable, and strong security with significant performance penalties is unacceptable. Therefore, we propose a new cache metric: *security-delay* ratio (denoted SD), defined as:

$$SD_{Cache} = \frac{(t+n)}{L_{Cache}^{avg}} \times \frac{1}{CPI} \tag{8}$$

This is a higher-is-better metric: lower CPI (Cycles Per Instruction) 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_{Cache}^{Avg}. For example, if two schemes S_1 and S_2 have CPIs 2.0 and 1.5 with respective leakages of 20 and 30 bits (out of a maximum of 42 [8]), then their respective SD metrics are 1.05 and 0.933 suggesting that S_2 perhap loses out on security a little too much (even though it is higher in terms of performance).

5 Experimental Methodology

We use a trace-based methodology for evaluating *CHASM*. Memory access traces of various workloads are collected using PIN [19] 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 L2 cache studies (L2 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 [25] - a multi-core, cache hierarchy simulator. The cache simulator is configured to match the Snapdragon Series 8 [7] cache configuration (L1D: 32KB 4-way, L2: 2MB, 8-way, all caches use 64B blocks). The cache simulator is modified to support all the cache mapping schemes listed in Table 1. The simulator is also enhanced to measure information leakage defined in Equations 7 and 8. 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 access consecutive elements of an array in a loop. By varying the size of the array, we observe the

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

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.

6 Results

We first evaluate L1D mapping schemes for their security using synthetic, SPEC and Cryptography benchmarks, followed by an evaluation of L2 mapping schemes.

6.1 Security of L1D Cache Mapping Schemes

Figure 3 plots the average information leakage (L^{Avg}_{Cache}, refer Equation 7) from all address bits observed in synthetic programs across all the evaluated L1 mapping schemes. These results clearly demonstrate that leakage reduces as memory

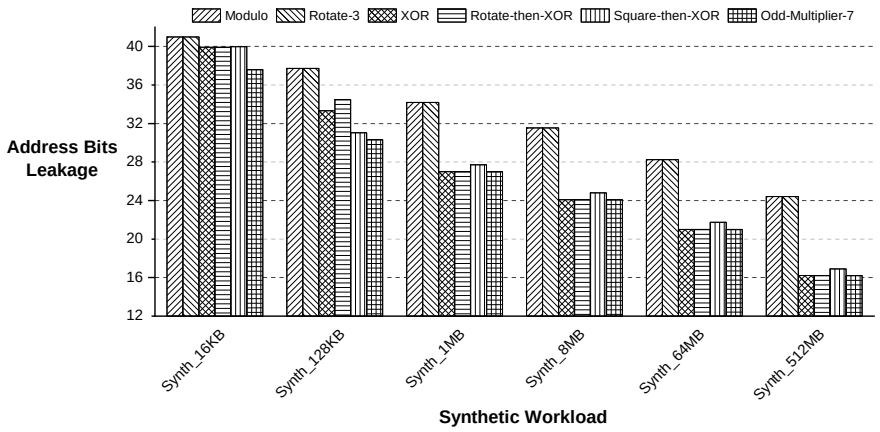

Fig. 3: Leakage of Address bits in L1D on Synthetic Programs

footprint increases. Programs with smaller memory footprints leak many more bits (40 in *synth_16KB*) compared to programs with higher footprints (16 in *synth_512MB*). This is due to less variation in the address bit values in programs with smaller footprints. These results also reveal that every mapping scheme that uses only the set-index portion of address bits to map the 2^n possible address bits to 2^n sets in a 1-1 function is *weak*: it reveals all n bits.

Next, we evaluate the leakage of information for selected SPEC and Cryptography workloads. Figure 4 plots this leakage across all mapping schemes. The first two schemes (that use only the set-index address bits for indexing) leak significantly more bits compared to schemes that leverage tag address bits for indexing. In particular, *AES* and *SHA* encryption programs show high leakage.

This is attributable to their small memory footprints and lack of variation in tag bits, thereby enabling attackers to accurately estimate several address bits. Even if encryption routines are integrated into larger applications, as they are typically invoked in response to encryption requests, knowledgeable attackers can isolate these portions of code execution and attack them.

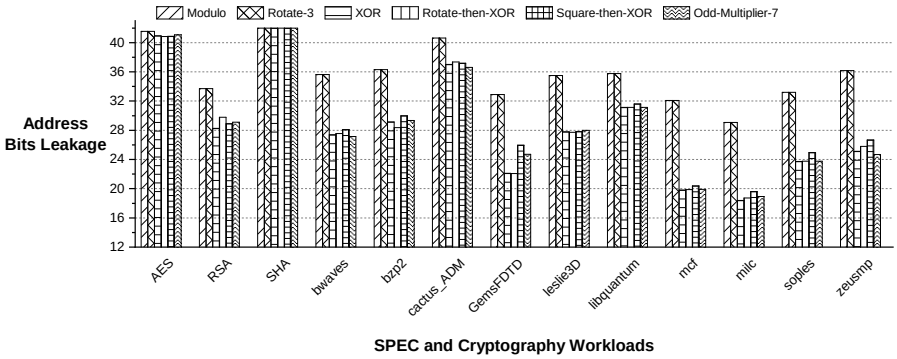

Fig. 4: Leakage of Address bits in L1D on SPEC and Crypto Benchmarks

6.2 Leakage Under ASLR

Address Space Layout Randomization (ASLR, see [23]) 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[9]. 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 5 reports our findings on the L1D cache by plotting the reduction in leakage by ASLR. Observe that this is a higher-is-better graph. The numbers above the bars are the absolute leakage values with ASLR. Overall, ASLR helps reduce the leakage of address bits. In particular, mapping schemes that use tag bits (last four bars for each workload) reduce leakage 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. However, ASLR does not completely prevent leakage. As the

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

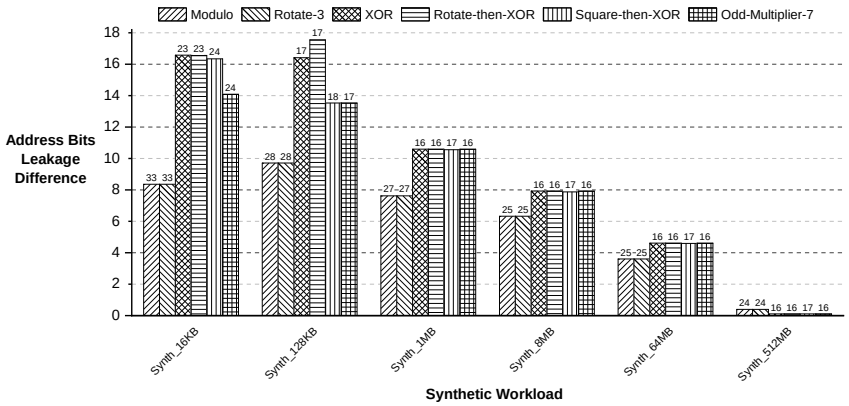

Fig. 5: Reduction in Leakage with ASLR

numbers above the bars indicate, in small workloads (e.g., *synth_16KB*) despite ASLR, $15 - 30$ bits leak.

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.

6.3 Security of L2 Cache Mapping Schemes

Figure 6 plots the leakage of the address bits in SPEC and Cryptography workloads[10]. 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*).

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 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

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

small (4KB) pages. Figure 7 compares leakage of tag address bits in the L2 cache. Across all the schemes, the use of huge pages results in leaking several additional address bits (about 5 on average in our experiments).

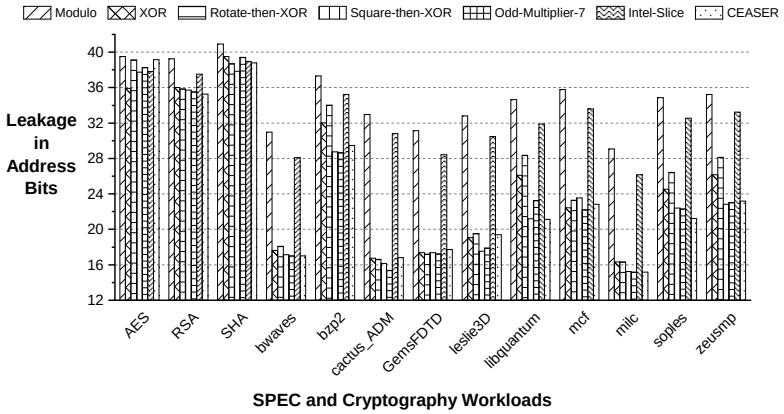

Fig. 6: Leakage of Address bits in L2 on SPEC, Cryptography Workloads

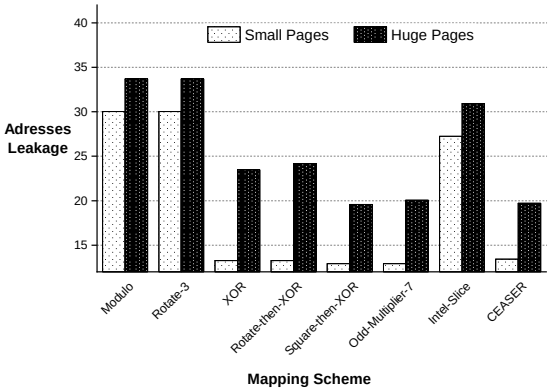

Fig. 7: Impact of Huge Pages

6.4 Security-Delay Measure

We use the metric SD_{Cache} defined in Equation 8 to study the joint effect of each scheme on performance and security. Figure 8 plots the percentage change in SD_{L1D} w.r.t the *modulo* baseline for SPEC and Crypto workloads for four mapping schemes on L1D. In workloads such as GemsFDTD, while *XOR* improves SD_{L1D} by as much as 49%, *Square-then-XOR* improves by only about 26%. On average (geometric mean), the *XOR* scheme performs the best. This evaluation shows that the SD_{Cache} 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 used. Even then 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.

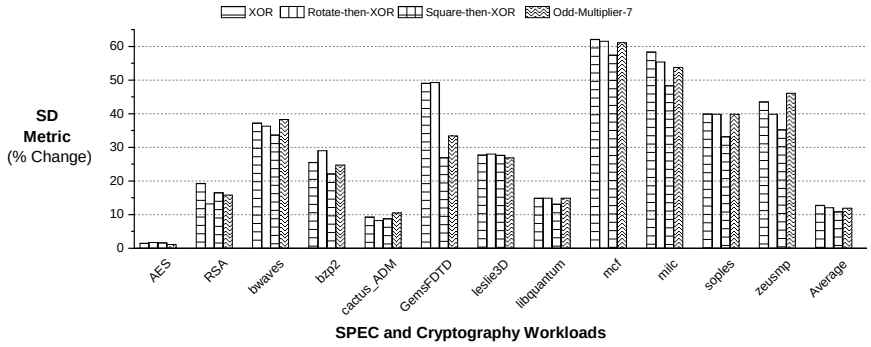

Fig. 8: L1D Security-Delay Ratio Metric of SPEC and Cryptography Workloads

7 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 [15] provides a comprehensive comparison of several mapping schemes used in low-associativity caches. The work in Givargis [9] explores mapping schemes that improve performance by considering correlations among address bits. Kharbutli et al [16] explore the prime-modulo and prime-displacement mapping functions.

The work in [24] 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 [20]. We evaluate these schemes in our work. ScatterCache [30] proposes a per-process keyed mapping scheme. Per-process mappings make inter-process attacks difficult but due to the mapping hardware overhead, they are feasible only at low-level caches. 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 such as memory footprint in determining programs' vulnerability to microarchitectural side-channel attacks.

Address space layout randomization (ASLR) [23] 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 [28].

Orthogonal to *CHASM*, Zankl et al [33] provide a framework for detecting leakage of modular exponentiation software via instruction caches. The work in Doychev et al [8] 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.

8 Conclusions

This work presented *CHASM* – a framework for evaluating strength of the security of cache mapping schemes. We evaluated several schemes at L1 and L2 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 strength of countermeasures against 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. J. A. Ambrose, R. G. Ragel, D. Jayasinghe, T. Li, and S. Parameswaran, "Side channel attacks in embedded systems: A tale of hostilities and deterrence," in *Sixteenth International Symposium on Quality Electronic Design*, March 2015, pp. 452–459.
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. 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.
8. 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

9. T. Givargis, "Improved indexing for cache miss reduction in embedded systems," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03, 2003. [Online]. Available: http://doi.acm.org/10.1145/775832.776052
10. K. M. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the ARM architecture," in *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2006, Seoul, Korea, October 22-25, 2006*, S. Hong, W. H. Wolf, K. Flautner, and T. Kim, Eds., 2006. [Online]. Available: https://doi.org/10.1145/1176760.1176793
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. I. Issa, A. B. Wagner, and S. Kamath, "An operational approach to information leakage," *CoRR*, vol. abs/1807.07878, 2018. [Online]. Available: http://arxiv.org/abs/1807.07878
15. 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
16. 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
17. 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
18. 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
19. 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
20. 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

21. 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.
22. 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.
23. Pax, "Address space layout randomization ASLR," 2003. [Online]. Available: http://pax.grsecuritynet/docs/aslr.txt
24. 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
25. C. Shelor and K. Kavi, "Moola: Multicore cache simulator," in *30th International Conference on Computers and Their Applications (CATA-2015)*, 2015.
26. D. Trilla, C. Hernandez, J. Abella, and F. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," 06 2018, pp. 1–6.
27. 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
28. 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/
29. I. Wagner and D. Eckhoff, "Technical privacy metrics: a systematic survey," *CoRR*, vol. abs/1512.00327, 2015. [Online]. Available: http://arxiv.org/abs/1512.00327
30. M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/werner
31. 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, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671271
32. 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.
33. 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
34. N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. 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.
35. 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