# Enabling Hardware Randomization Across the Cache Hierarchy in Linux-Class Processors

Max Doblas[1], Ioannis-Vatistas Kostalabros[1], Miquel Moretó[1] and Carles Hernández[2]

[1]Computer Sciences - Runtime Aware Architecture, Barcelona Supercomputing Center
{max.doblas, vatistas.kostalabros, miquel.moreto}@bsc.es
[2]Department of Computing Engineering, Universitat Politècnica de València
carherlu@upv.es

***Abstract*—The most promising secure-cache design approaches use cache-set randomization to index cache contents thus thwarting cache side-channel attacks. Unfortunately, existing randomization proposals cannot be sucessfully applied to processors' cache hierarchies due to the overhead added when dealing with coherency and virtual memory. In this paper, we solve existing limitations of hardware randomization approaches and propose a cost-effective randomization implementation to the whole cache hierarchy of a Linux-capable RISC-V processor.**

## I. Introduction

Recently reported security vulnerabilities exploit key, high-performance, processor features such as the use of speculative execution [11] in order to get access to classified information of victim processes executed in the CPU. In Spectre-like attacks, speculative execution is used to force the execution of instructions not belonging to the regular program path to obtain private information from the victim's address space. Since the misspeculated paths are not functionally visible, they have to rely on microarchitectural side-channels to obtain the secret information. Cache side-channel attacks have been shown to be a feasible and powerful tool to leak sensitive information from a victim process [13], [16].

The design of secure caches has become a very active area of research. The most promising secure-cache design approaches randomize the mapping from memory line to the cache-set in order to thwart cache side-channel attacks [14], [18], [20], [21]. These cache designs use a parametric, keyed function to index cache contents. The function is fed with a subset of the target address's bits and a key obtained from an entropy source and it dictates the cache set to look for that value. When the key value used in this function is altered, the memory line to cache-set mapping changes. By extend, the cache conflicts observed will be completely different.

The complexity of building an attack exploiting cache conflicts relies on the ability of the attacker to find an *eviction set* [19]. When cache randomization is in place, eviction sets have to be rediscovered every time the key used to index cache contents is modified. The key modification frequency defines the maximum vulnerable time-window of the system against a side-channel attack.

Recent proposals have shown that cache randomization can be successfully applied to Last-Level Caches (LLCs) to prevent cache side-channel attacks at the cost of increasing cache access latency [14], [15], [21]. However, in many computing domains (e.g. safety-critical systems) protecting LLCs is not enough as it does not provide any security guarantees for attacks targeting other upper-level caches [22].

Unfortunately, applying cache randomization to the whole cache hierarchy is more challenging as it requires the randomization approaches to have low implementation costs and support virtual-to-physical memory translation. The majority of current processors rely on Virtually-Indexed Physically-Tagged (VIPT) first level caches for enhanced performance. Such designs allow reducing cache access latency as virtual-to-physical translation is done in parallel with the cache access. Current cache randomization solutions cannot be directly applied to such cache designs without sacrificing performance. Moreover, keeping coherence across the cache hierarchy requires being able to transparently index cache contents both with physical and virtual addresses.

In this paper, we propose the first randomization mechanism that can be applied to the whole cache hierarchy of a processor design with virtual memory support. In particular, this paper makes the following contributions:

① A mechanism to deal with both virtual and physical addresses in randomized cache designs while retaining cache coherency.

② A performance/security balanced solution in which the best randomization strategies are employed at different cache levels to improve performance vs security trade-offs.

③ An FPGA implementation of our proposal in a RISC-V processor that is able to successfully boot the Linux operating system (OS) including randomization in the whole cache hierarchy.

④ An evaluation of the security and performance overheads of the proposed solution showing for the first time the feasibility of applying randomization across the whole cache hierarchy.

## II. Randomized Cache Designs for Security

### A. Cache Side-Channel Attacks

Cache-based side channel attacks have become a serious concern in many computing domains [13], [16]. These attacks are able to bypass existing virtualization and software isolation mechanisms as they infer confidential information from cache conflicts between the victim and the attacker.

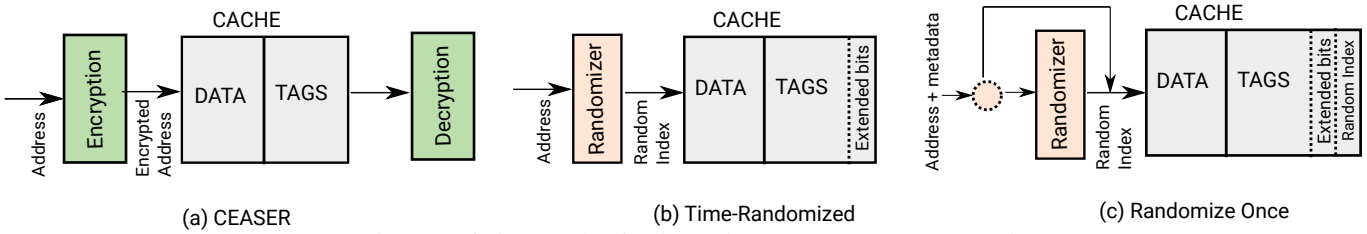


Fig. 1: Existing randomization schemes versus our proposal

Contention-based attacks work in private caches (same-core) or shared-caches (cross-cores) and exploit cache evictions to learn the access patterns of victim applications. Same-core side channel attacks require the victim and the attacker to be co-located on the same core by exploiting hyperthreading [16] or OS scheduling [13]. While cross-core attacks impose less restrictions to the victim's and the attacker's co-location, they encounter more obstacles, as timing measurements suffer from interference noise coming from multiple cores.

### *B. Randomization Countermeasures*

Cache-layout randomization schemes use a parametric function that combines the address with a key-value to randomize the mapping from the cache line tothe cache-set. Proposed functions rely on hashing schemes [12], random permutations of the set index [8], or whole address encryptors [14].

**Single vs multiple security domains.** Currently, two alternative schemes exist, that offer different levels of protection. In the single domain scheme [14], all processes share the same key-value to access cache contents. Protection is provided by ensuring that the value of the key is modified *frequently*. The frequency at which the key is changed, determines the maximum amount of time, that the attackers have in order to build a successful attack (e.g. discover an eviction set). In the second scheme [18], [21], different security domains are defined such that every domain uses a different and independent key-value. With this latter approach and assuming that the victim and the attacker belong to different security domains, cache conflicts cannot be directly associated to a specific address. As a result, an additional profiling process is required to determine the actual relationship between victim and attacker congruent addresses. Moreover, using security domains, provides additional protection against unauthorized control-information-tampering attacks [10].

### *C. Current Limitations*

A typical memory hierarchy of a high-performance processor targeting data-center or embedded systems domains includes several levels of caches and support for virtual memory. In multicore designs, the first cache level is private. The second level of the cache can be shared across different cores or can remain private when a third level of cache is present. Typically, L1 caches are VIPT, whereas upper cache levels are Physically Indexed Physically Tagged (PIPT). The coherence protocol is in charge of invalidating cache lines in the different cache levels using physical and/or virtual addresses.

Private caches are weaker against a powerfull side-channel attack due to their small size. Nevertheless, current randomization approaches have only been shown suitable for LLCs [14], [21] or cache hierarchies with limited virtualization support [9]. Consequently, there is an imminent need for solutions that deal with randomized VIPT cache designs.

## III. Enabling Randomization across the Entire Cache Hierarchy

In this section we propose a novel mechanism that supports virtual memory in randomized caches. We also discuss the complexity and performance of existing schemes.

### *A. Cache Layout Randomization*

First, we analyze the suitability of different randomization schemes to implement cache randomization in the whole cache hierarchy. The CEASER approach [14] (see Figure 1 (a)) uses an encryption/decryption scheme for the addresses accessing/stemming from the cache. The main advantage of this approach is that the cache structure remains unaltered. However, its main drawback is the increased access latency due to the encryption and decryption process. The latter are being handled by a Low-Latency Block Cipher (LLBC) which utilizes a Feistel network [**?**] to produce the necessary encrypted/decrypted bits. As shown by Bodduna et al. [5] this LLBC is vulnerable to key and bits invariance attacks and therefore is deemed futile at thwarting cache side-channel attacks against a powerfull adversary.

An alternative scheme consists of using a randomization function to produce the cache set's index [8], [12], [21] (see Figure 1 (b)). This scheme requires extending cache tags to include all bits in the address, except the offset bits, to avoid colisions between blocks with the same tag in the same set. Moreover, the latency of the randomization process can be partially hidden, as it occurs in parallel with the cache access. Nevertheless, this may have an impact on the timing of the cache provided that the randomization delay does not fit in the available timing slack. Finally, set-index randomization is only applied when cache contents are accessed and therefore no decryption process is required. In this paper, we opt for cache-set randomization as the baseline randomization scheme to avoid the latency overheads of CEASER.

### *B. Dealing with Virtual Memory in Randomized Caches*

Randomized cache designs employ upper address bits to determine the cache set index. Thus, physical and virtual addresses pointing to the same cache line will very likely produce different cache set indexes, thus creating a coherency

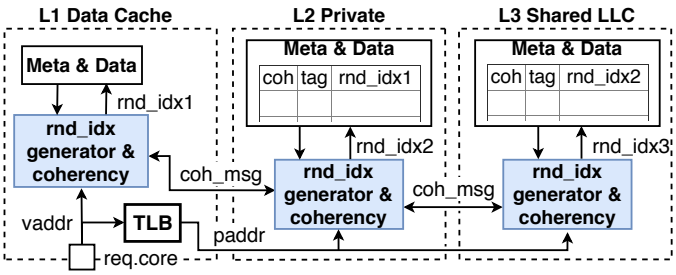


Fig. 2: Overview of a randomized cache hierarchy with three levels of cache.

problem. To overcome this limitation, we propose a new set-index randomization scheme (see Figure 1 (c)).

Initially, we apply randomization when the data petition arrives at the cache from the core. This is the only case that data request is happening with the virtual address. Thus, all core requests go through the randomizer module to determine the cache set. In case of a cache hit, data is served to the CPU. In case of a miss, a request including the randomized index is propagated to the L2 cache. This index is stored in the metadata of the corresponding cache line in the L2 cache. When the L1 miss is served, the incoming requested cache line incorporates the randomized index to access the L1 cache without using the randomizer module, since the randomized index has already been produced.

The same solution can be also applied in the other levels of the cache hierarchy. Incoming physical addresses go through the randomizer module, thus generating a randomized index and accessing the L2 cache. This randomized index will be stored in the metadata of the L3 cache. Therefore, all memory requests to the different cache levels are modified in order to propagate the corresponding randomized index. In the case of the LLC, the randomized index does not need to be propagated to main memory but only to its Miss Status Holding Register (MSHR) in order to serve LLC misses.

Apart from storing the randomized index at the upper cache level, we also need to augment the tags inside the metadata structure with the original (not-randomized) index of the address. The reason for this extension is twofold. First of all, we may have collisions of different blocks having the same tag in the same set due to the randomization. Moreover, these extra bits of information are also needed to generate the block address in case of eviction because the (random) set index and the (original) address index are different.

Finally, all the petitions from the coherence protocol will use the randomized index stored in the metadata of the different cache levels, apart from that of the L1. Note that this modification does not alter the behavior of the regular randomized cache. Figure 2 illustrates how the proposed randomized cache hierarchy works.

The proposed mechanism supports the in-flight change of the mapping function of the randomizer module. When the randomized index for a given address is changed, then a cache miss may happen even if the data is already in the cache (coupled with the previous randomized index). Since the previous randomized index is stored in the metadata of the upper cache level, the old line can be invalidated and the up-to-date version of the data can be recovered. All necessary modifications with respect to the coherence protocol functionality are explained in detail in section IV-C.

Summing up, the proposed mechanism, enables cache randomization across the entire cache hierarchy, not only for virtually indexed cache designs but also for the physically indexed ones.

### C. Cost-effective Randomization Functions

Applying randomization to small caches is challenging as timing and area margins are really tight in these memory structures. Secure randomization functions proposed for LLCs [14], [21] are too complex to be applied without increasing cache access latency. As an alternative, simple randomization functions can be used to index small caches. In that case, it is critical that such functions uniformly distribute addresses to cache sets to preserve security properties [19]. Time-randomized caches offer a solution to this challenge as they implement cache randomization with uniform set distribution at very low implementation cost. The current proposal, which is agnostic to the underlying randomization function, implements a hash function [12] and a random modulo [8] solution that incur low overheads and preserve the uniform distribution of the input. However, since L1/L2 caches are smaller than the LLC, the complexity required to find eviction sets, thus building an attack, is significantly reduced for these caches. To achieve the highest protection in these caches, we also integrate an explicit, per-way randomization function, which significantly increases the robustness of the system against cache side-channel attacks [15], [21].

## IV. Implementation in the lowRISC SoC

We have implemented our proposal on top of lowRISC [3], a RISCV System-on-Chip (SoC) with a memory hierarchy of two cache levels based on the Rocket-chip generator [7]. In the first level, we have private, per-core, and separate instruction and data caches. In the second level, there is a shared L2 cache, which is also the LLC in this SoC.

In order to deal with physical and virtual addresses, we randomize the cache index once per L1 data cache request, storing also the original index at the L1 cache metadata (explained in III-B). As described in the same section, we also propagate the generated random index to the L2 cache which stores it in its metadata in order to maintain coherency in the system.

### A. Modifications to the L1 Data Cache

The Rocket core L1 data cache is composed of four stages and includes a MSHR, a prober and a writeback module. The stages of a successful access are: 1) accessing the metadata and data structures at the same time as the TLB; 2) comparing the N-way tags stored at the corresponding set with the Physical Page Number (PPN) tag from the TLB; 3) if the request is a load, respond to the CPU with the required data; 4) if it is a store, update the respective structures.

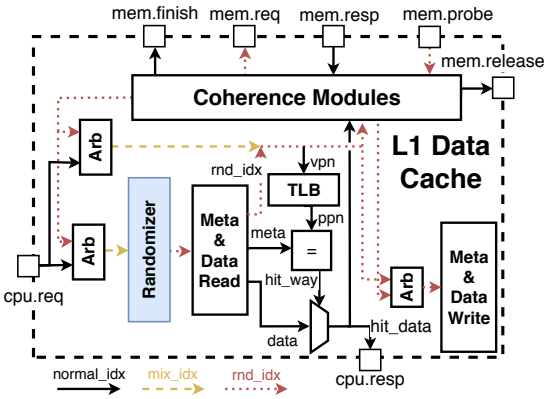


Fig. 3: Simplified diagram showing the different types of indexes on the L1 randomized data cache implementation.

First of all, we need to place the randomizer module between the CPU request port of the cache and the metadata (tags) and data arrays. The randomized index generation occurs only once per request and takes place in the first stage. This module generates the random index used to access the metadata and data structures only if the address introduced lacks a randomized index (i.e is accessed for the first time). If the address has already a randomized index, this module is bypassed. Our proposal supports using different randomization functions to generate the random index. Figure 3 depicts a schematic of the L1 randomized data cache. As shown in the diagram, the randomized index is present at the output of the metadata and data reading. Then, it is propagated with the request to later stages such as the writing stage or the coherence modules.

Another module that has to be slightly modified is the metadata array in the cache. Since two different cache lines with the same tag can be placed in the same set due to the randomization function, we need to deal with that issue. A simple solution consists of extending the tag residing in the metadata with the original index of the address. As we already mentioned (III-B) these extra bits are also used to generate the address in the case of an eviction.

Next, we elaborate on the required changes of the coherence modules. As we explained before, we need to send the randomized index to the upper-level cache, the L2 cache in this case, to be stored in the metadata. To do that, we send this index to the L2 in every data request produced by a L1 miss. The L1 MSHR is in charge of that. Finally, in every coherence petition, the randomized index present in the L2 cache metadata is sent along with the petition to access the L1 cache structures. As a result, it is possible to deal with the virtual and physical addresses seamlessly.

### B. Modifications to the Shared L2 Cache

In the L2 cache, we include a randomizer module to randomize the access to the cache contents. Also, we have modified the directory structure to support interfacing with the VIPT L1 caches. In particular, we add a new field to the directory metadata to store the randomized index of L1 in each valid line. The random index of a cache line is updated in every request affecting this line. Thus, the directory always has the updated reference of the data stored in the lower level cache. Such a mechanism allows the coherence protocol to make correct petitions to the randomized cache.

### C. Modifications to the Coherence Protocol

The Rocket core uses a MESI protocol to maintain data coherence in the whole memory hierarchy. The coherence protocol relies on a directory placed in the L2 cache. With the proposed modifications, there can be specific cases for which the protocol malfunctions (as it will be explained later). The original MESI protocol along with the randomized cache, can result in multiple instances of the same data valid in the same core. This may create ghost copies that could no longer be invalidated.

Ghost copies can appear when there is a modification on the key of the randomizer module or a modification in the page table. If a new CPU request from a data that is already in the cache comes after any event mentioned before, this request will have a new randomized index assigned by the randomize module. Then a miss will be produced. At this point, if the original data in the L1 data cache is in the shared state and the petition comes from a load, the L2 cache sends the data without invalidating the old instance. At this moment, the new randomized index is stored in the L2 cache and replaces the old one. This action makes the other line (the one valid before the key/page table change) remain valid and untracked without the possibility to be invalidated.

To solve this issue, we alter the MESI protocol. Now the cache that makes a petition is bound to receive coherency requests from the upper-level cache in order to remove potential ghost copies. If the state is shared, and the core that is making the petition has already the data, the coherency protocol sends an invalidation message to this cache before sending the data. All other protocol's mechanisms remain unchanged.

### D. Implementation of the Skewed Cache

To increase the security of randomized caches, a different randomization module per way can be used. To implement this functionality, we only need to increase the number of functions inside the randomizer module to get one function per way. The propagation and storage of the randomized index are still necessary. Now we need to choose the correct randomized index out of the N-way indexes. The index selected in the case of a hit is the index of the way that produced the hit, and in the case of a miss, we choose the index of the cache line that will be evicted.

### E. Interface with the Operating System

To allow the OS to handle the randomized-cache-hierarchy keys which affect the random module address hashing (e.g. random index generation), we use some custom Control and Status Registers (CSR). We use a dedicated CSR per cache only accessible in privileged mode (N-way CSR in the case of the skewed cache) to store the active keys used by the

TABLE I: Microarchitectural configuration of the target SoC.

| SoC Configuration & System parameters | |
|---|---|
| **Component** | **Description** |
| Core details | 200 MHz, in-order, 5-stage pipeline, 64-bit RISCV IMA extensions, 32 integer registers |
| Private L1 data cache | 16KB 4-way set assoc, 64B line size, random replacement, 3-cycle latency, VIPT |
| Data TLB | 8 entries |
| Private L1 instr. cache | 16KB 4-way set assoc, 64B line size, random replacement, 2-cycle latency, VIPT |
| Instruction TLB | 8 entries |
| Shared L2 cache | 64KB 8-way set assoc, 64B line size, random replacement, 8-cycle latency |
| MSHR Size | 2 entries |

randomizer module. Using that, the OS can easily change the key for each process and achieve the desired isolation level. Finally, these CSRs are initialized at a random value on every reboot. Using that property, the mapping of the caches changes every time the machine is rebooted, thus preventing an adversary from knowing the initial seed in advance. This feature is also present in other solutions [14].

## V. Evaluation

This section explains the results obtained with the experimental campaign of our proposal. We discuss performance, security properties and area overhead separately.

### A. Experimental Setup

As explained in Section IV, we have implemented all the modules required for the hardware randomization in the cache hierarchy of the Rocket core [7]. The RISC-V core, as well as the rest of the SoC, was synthesized and emulated at a Kintex-7 FPGA development board. The SoC contains all the required peripherals to function (i.e memory controller) and to communicate with a host machine (i.e UART). The testbed used, comprises of the FPGA development board and a host machine which collects and interprets accordingly the data sent over the UART from the FPGA. The configuration parameters of the SoC and core modules are summarized in Table I. We perform the experimental evaluation of our Soc using the non-floating point benchmarks from the EEMBC suite [2].

### B. Functional Verification

To validate the modifications introduced in the processor's RTL, we have executed the RISC-V ISA tests [4], a battery of more than 105 unit tests. In particular, the executed tests have been wrapped with the virtual environment code. With this environment, the tests use a virtual-to-physical address translation, thus forcing the core to utilize all the virtual memory mechanisms it supports. We validated that our modifications do not alter the functional behavior of the system. Also, we proved that the EEMBC tests are producing the correct results using a cyclic redundancy check (CRC) validation.

Additionally, we have been able to successfully load the version 3.14.41 of the Linux kernel on top of our randomized platform. The kernel booted provides the user with the basic shell environment functionality. Note that booting Linux was not possible without including the support to deal with virtual memory in the caches.

TABLE II: Accesses required to find an eviction set.

| Processor | L1 | L1 (skewed) | L2 | L2(skewed) |
|---|---|---|---|---|
| Rocket | 4288 | 68608 | 35456 | 2269184 |
| Neoverse | 17152 | 274432 | 2269184 | 18153472 |
| Skylake | 17728 | 1134592 | 67584 | 1081344 |

### C. Security Assessment

We assess the security properties of our randomized cache hierarchy using the formulation provided in [21]. In particular, we use the number of accesses required to build a profiled eviction set for a *prime-probe* attack [22] as a security vulnerability indicator. Having the ability to build an eviction set (i.e. determine congruent set of addresses required to evict a particular cache set) is a necessary step to many cache side-channel attacks [21]. Assuming that the eviction set consists of addresses colliding with the victim in exactly one way each, the eviction probability ($P_e$) for a skewed cache is determined by $1-(1-\frac{1}{n_{ways}})^{\frac{\overline{A}}{n_{ways}}}$ where $\overline{A}$ represents the number of colliding addresses. Finally, the expected number of accesses ($N_a$) in a cache with $n_{sets}$ number of sets is $\overline{N_a} = n_{ways}^2 * n_{sets} * \overline{A}$.

Table II shows the estimated number of accesses required to build an eviction set in the different cache modules assuming a success probability $P_e$ equal to 99%. We report values for the cache configuration used in our RISC-V prototype and for the cache configurations included in the ARM Neoverse [1] and the Intel Skylake [6]. The Neoverse CPU includes 4-way 64KB L1 caches and 8-way 512KB L2 caches both with 64 bytes cache lines. The skylake has 8-way 32KB L1 caches and 4-way 256KB L2 with 64byte cache lines. Results show that for small caches the number of accesses required to build an eviction set is rather small. Thus, including skewed randomizers becomes fundamental to have meaningful security properties. For the Skylake and Neoverse setups with skewed caches the computed number of accesses shows that randomization is an effective way to protect the system especially for the Skylake due to its higher associativity in the L1. Note that as explained in Section IV our randomization scheme allows re-keying the cache without flush and independently for each way.

We have also analyzed the randomization properties of the randomization functions implemented. Having a randomization function with almost ideal properties avoids the possibility of reducing the time required to build a sucessful side channel attack. Additionally, having ideal randomization properties is very useful to protect the system from software implementation vulnerabilities [10]. We run the NIST randomness test suiteworks [17] using $10^8$ different keys and the same number of random addresses to prove that the randomizers implemented can be used safely in our design. This is an interesting observation since the complexity of such functions is much lower than the complexity of the functions proposed in other works [14], [21].

### D. Performance Evaluation

Each benchmark has been executed 1000 times with 1000 different random keys for each underlying randomization mode in our RISC-V FPGA prototype. Note that for the

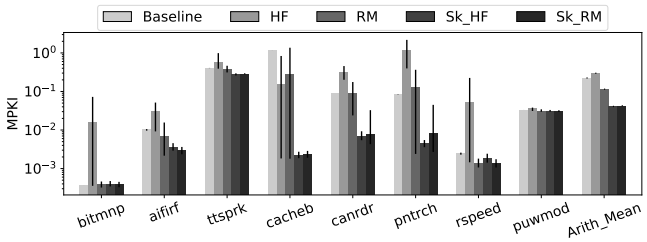


Fig. 4: Mean L1 data cache MPKI for the EEMBC benchmarks with the baseline design, cache randomization based on random modulo (RM) and on hash function (HF), and the combination with skewed cache (Skewed_RM and Skewed_HF).

TABLE III: FPGA resources utilization for the baseline, hash function (HF) and random modulo (RM) configurations.

| | | LUTs | FF | CLAs |
|---|---|---|---|---|
| L1 | Baseline | 3249 | 2514 | 82 |
| | HF | 4587 (+41.2%) | 2598 (+3.3%) | 87 (+6.1%) |
| | RM | 3553 (+6.0%) | 2598 (+3.3%) | 87 (+6.1%) |
| | HF Skewed | 7862 (+142.0%) | 2676 (+6.4%) | 87 (+6.1%) |
| | RM Skewed | 3718 (+14.4%) | 2676 (+6.4%) | 87 (+6.1%) |
| L2 | Baseline | 11047 | 3778 | 85 |
| | Others | 13607 (23.2%) | 3999 (+5.8%) | 93 (+9.4%) |
| Total | Baseline | 15301 | 7636 | 199 |
| | HF | 19199 (+25.5%) | 7941 (+4.0%) | 212 (+6.5%) |
| | RM | 18055 (+18.0%) | 7941 (+4.0%) | 212 (+6.5%) |
| | HF Skewed | 22474 (+46.9%) | 8019 (+5.0%) | 212 (+6.5%) |
| | RM Skewed | 18330 (+25.5%) | 8019 (+5.0%) | 212 (+6.5%) |

tested benchmarks, 1000 runs suffice to produce a distribution of cache misses that is independent and identically distributed (i.i.d). The randomization functions we chose are, as stated before, random modulo [8] and hash-based random placement [12]. For each randomization function we have also implemented its skewed variant. For the baseline setup we have also performed several runs to capture the inherent platform variability which was shown to be lower than 1%.

Figure 4 shows the average as well as the 5% and 95% percentiles for Misses per Kilo Instruction (MPKI) values over 1000 runs with different random keys. Initially we observe that not all of the benchmarks show a significant MPKI value since EEMBC benchmarks do not have a big memory footprint. The ones that do exhibit more misses are (ttsprk, cacheb,canrdr and pntrch). For most of the benchmarks the skewed cache architecture (both with Random Modulo and with Hash Function) configuration is the one providing better results. On the other hand, the benchmarks being executed on top of the hash function randomization module yield the highest value of MPKI since this randomization scheme has worse spatial locality properties than modulo or random modulo. Interestingly, when hash randomizer is deployed in a skewed scheme the behavior of this scheme improves significantly due to the elimination of the spatial locality pathological situations produced by the hash [8].

### *E. Area Utilization*

Table III reports the resource utilization of the FPGA implementation. The impact on resource utilization of RM solution is very low in the L1 caches. This solution needs 6.0% more LUTs and 3.3% more flip-flops (FF) compared with the baseline. In contrast, the HF solution is using 41.2% more LUTs and 3.3% more FFs than the baseline. Both solutions have some impact on the L2 cache design. The majority of the LUTs increase in the L2 cache is produced by the added hash function. As the number of BRAM used in the different configurations is the same, the increase of 5.8% in FF is due to the storage of the randomized index at L1 data cache. As expected, the integration of the skewed caches uses more resources to implement multiple randomization functions. Nevertheless, L2 remains identical because, as mentioned in Section IV-D, we propagate only one randomized index.

## VI. Related Work

The earliest cache randomization approaches aimed at improving the security of caches by relying on hardware mapping tables to find the location of cache lines [20]. This approaches suffer from scalability limitations and require having the ability to classify applications as protected or unprotected. More recent randomization schemes propose an encryption and/or randomization of the address to index cache contents [14], [18]. Authors in [18] suggested the use of a parametric function to randomize the cache line mapping, reusing cache designs already proposed for probabilistic real-time systems [8], [12]. The work in [18] recommended including the process id to the randomization function to create independent cache layouts for different processes. Later, this approach has been generalized in [21] introducing the concept of security domains. The security properties of randomization schemes has been analysed in several recent works [14], [15], [21]. [14] introduced an encryption mechanism coupled with a remapping process for LLCs showing that for contemporary attacks the performance impact caused by remapping can be neglected.However, the simplicity of the LLBC used in [14] creates a major vulnerability against some cryptanalysis attacks. Recent works [15], [19] have shown that the cost of finding eviction sets can be reduced which makes initial randomization schemes vulnerable. To solve this issue the latest randomization schemes have proposed skewed mechanisms implementing independent per-way randomization functions [15], [21]. With skewed randomization the complexity of building side-channel attacks is significantly increased making performance cost of key remapping negligible for LLCs. Unfortunately, existing randomization schemes are only useful for LLCs in which randomization only deals with physical addresses [14], [15], [21] or in simpler processor configurations [18].

## VII. Conclusions

In this paper, we have introduced a novel randomization mechanism for high performance cache designs that makes use of virtual and physical addresses while retaining cache coherency. The proposed approach can be successfully applied to a RISC-V processor capable to boot Linux, significantly improving its security against cache-based side-channel attacks.

## Acknowledgments

This work has been supported by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy

and Competitiveness (contract TIN2015-65316-P), and by Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328). The DRAC project is co-financed by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of total cost eligible. We also thank Red-RISCV for the efforts to promote activities around open hardware.

This work has received funding from the EU Horizon2020 programme under grant agreement no. 871467 (SELENE).

M. Doblas has been partially supported by the Agency for Management of University and Research Grants (AGAUR) of the Government of Catalonia under Beques de Col·laboració d'estudiants en departaments universitaris per al curs 2019-2020. V. Kostalabros has been partially supported by the Agency for Management of University and Research Grants (AGAUR) of the Government of Catalonia under Ajuts per a la contractació de personal investigador novell fellowship number 2019FI_B01274. M. Moretó has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Raón y Cajal fellowship number RYC-2016-21104.

## REFERENCES

[1] Arm neoverse n1 cpu. https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-n1.

[2] EEMBC. embedded microprocessor benchmark consortium. https://www.eembc.org/techlit/.

[3] lowRISC. Open to the core-enabling open source silicon through collaborative engineering. https://www.lowrisc.org/.

[4] RISC-V Tools. RISC-V tools including ISA simulator and tests. https://github.com/riscv/riscv-tools.

[5] R. Bodduna, V. Ganesan, P. Slpsk, C. Rebeiro, and V. Kamakoti. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 2020.

[6] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *26th USENIX Security 17*, pages 51–67, Vancouver, BC, Aug. 2017.

[7] K. A. et. al. The rocket chip generator. Technical Report UCB/EECS-2016-17, University of California, Berkeley, Apr 2016.

[8] C. Hernández, J. Abella, A. Gianarro, J. Andersson, and F. J. Cazorla. Random modulo: a new processor cache design for real-time critical systems. In *DAC*, pages 29:1–29:6, 2016.

[9] C. Hernández and et. al. Design and implementation of a time predictable processor: Evaluation with a space case study. In *ECRTS*, pages 16:1–16:23, 2017.

[10] Jun Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *SRDS*, pages 260–269, 2003.

[11] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SSP*, 2019.

[12] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Efficient cache designs for probabilistically analysable real-time systems. *IEEE Trans. Computers*, 63(12):2998–3011, 2014.

[13] M. Neve and J. Seifert. Advances on access-driven cache attacks on AES. In *SAC*, pages 147–162, 2006.

[14] M. K. Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787, 2018.

[15] M. K. Qureshi. New attacks and defense for encrypted-address cache. In *ISCA*, pages 360–371, 2019.

[16] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212, 2009.

[17] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Booz-allen and hamilton inc mclean va, 2001.

[18] D. Trilla, C. Hernández, J. Abella, and F. J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *DAC*, pages 98:1–98:6, 2018.

[19] P. Vila, B. Köpf, and J. F. Morales. Theory and practice of finding eviction sets. In *IEEE SSP*, pages 39–54, 2019.

[20] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505, 2007.

[21] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[22] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SSP*, 2019.