# Cache Side-Channel Attacks and Time-Predictability in High-Performance Critical Real-Time Systems

David Trilla
Universitat Politècnica de Catalunya and BSC

Carles Hernandez, Jaume Abella
Barcelona Supercomputing Center (BSC)

Francisco J. Cazorla
BSC and IIIA-CSIC

## ABSTRACT

Embedded computers control an increasing number of systems directly interacting with humans, while also manage more and more personal or sensitive information. As a result, both safety and security are becoming ubiquitous requirements in embedded computers, and automotive is not an exception to that. In this paper we analyze time-predictability (as an example of safety concern) and side-channel attacks (as an example of security issue) in cache memories. While injecting randomization in cache timing-behavior addresses each of those concerns separately, we show that randomization solutions for time-predictability do not protect against side-channel attacks and vice-versa. We then propose a randomization solution to achieve both safety and security goals.

## CCS CONCEPTS

**•Security and privacy → Embedded systems security; •Computer systems organization → Real-time system;**

## KEYWORDS

Cache, randomization, side-channel attacks, probabilistic analysis

## 1 INTRODUCTION

Increasingly autonomous and connected Automotive Systems (ATS) require on-board computing systems with resilient operation under timing faults and attacks. On the one hand, increased connectivity opens the door to security threats, e.g. side-channel attacks (SCA), an effective security intrusion for obtaining secret keys. In particular, cache timing SCA ( referred to as SCA) allows attackers to fully or partially recover keys, which can later be used to take control over the ATS. On the other hand, ATS increasingly deal with safety (e.g. autonomous driving), which requires a reliable response time of all critical software services. For instance, compliance with ISO-26262 requires software units to be assigned time budgets and supporting evidence of adherence to those budgets.

Until recently, ATS comprised relatively simple Electronic Control Units (ECU) deploying 8- and 16-bits microcontrollers. The execution time of software on such simple devices was mostly jitterless (or suffering very small execution time variability) simplifying the task of deriving worst-case execution time (WCET) estimates and mitigating the risk of SCA. However, the advent of more complex value-added software functionalities, managing increasing amounts of diverse data, has raised the performance needs for the automotive sector, for which ARM expects a 100x increase in performance by 2024 w.r.t. 2016 [6].

The execution time of tasks on complex hardware strongly depends on their input data and processor's state, thus exposing the system to SCA. This is a major concern for ATS to protect sensible information and prevent safety issues as ATS control critical aspects with humans in the loop, e.g. autonomous driving.

Interestingly, randomization has been independently proposed as a solution for WCET estimation and preventing SCA. For WCET estimation, the most extended industrial practice builds on collecting execution time measurements of the software running on the target platform. Obtaining evidence about whether those measurements are representative of the WCET during operation is challenging on complex hardware [1]. These difficulties have been addressed by using probabilistic techniques to WCET analysis, so called Measurement-Based Probabilistic Timing Analysis (MBPTA) techniques [10]. MBPTA benefits from injecting randomization in cache timing to simplify modeling and provide evidence for certification as needed by safety standards. For SCA, randomization solutions  break the dependence between input data (and/or cache state) and execution time so that for the same input data and processor state a sufficiently random execution time is experienced. However, it remains to be proven whether a single solution can tackle both, WCET and SCA issues.

In this context we make the following contributions:
① We make an in-depth analysis of the properties required to enable MBPTA to deal with jittery timing behavior of applications running on complex hardware. We also cover how randomization helps dealing with SCA . We describe the vulnerability of randomization support for MBPTA to specific SCA. Likewise, we assess the time predictability of randomization support for SCA solutions showing that they fail to meet MBPTA principles.
② We propose Time-Predictable Secure Cache (TSCache) that provides increased resilience in front of specific SCA while keeping MBPTA compliance. Hence TSCache reconciles security (robustness against certain SCA) and safety (by adhering to MBPTA principles to derive reliable timing budgets) in cache design.
③ With a simulator modeling a commercial automotive processor, we experimentally show the resilience of our solution against the Bernstein attack [7] while keeping MBPTA compliance.

## 2 MBPTA AND SCA PROTECTION

### 2.1 Random Caches for MBPTA

MBPTA delivers a probabilistic WCET (pWCET) distribution representing the highest probability with which one run of a task


*DAC '18, San Francisco, CA, USA*

DOI: 10.1145/3195970.3196003

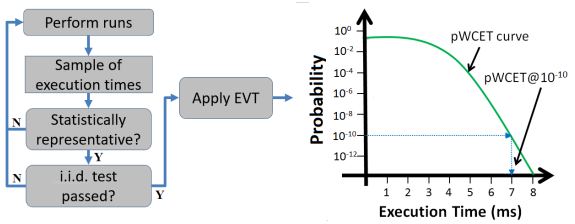


**Figure 1: MBPTA process and example of pWCET curve**

exceeds a time bound. Figure 1 (right) shows an illustrative pWCET distribution for which the probability of the task exceeding 7ms is below $10^{-10}$ per run. MBPTA is well-settled with industrial case studies performed in automotive, avionics, and space [14, 17, 29].

Often, in automotive, system integrators subcontract the development of certain software to different software providers. Across software integrations of the software units contributed by each provider, the objects of a function (i.e. globals, stack and code) can change their addresses resulting in different cache layouts, with arbitrarily different hits/misses and execution time [19]. In general, it is unfeasible for users creating test scenarios during the analysis phase (AP) accounting for the worst memory placement (and cache layouts) that can occur during system's operation phase (OP) [1].

Cache randomization [15, 24] ensures that a new random cache layout is exercised on every program run so that the impact of caches on execution time becomes independent of the actual memory layout. This relieves users from controlling memory/cache layout of objects since (random) cache layouts experienced during AP and OP are probabilistically identical. MBPTA builds upon collecting a sample of execution times of the task under analysis on the target platform and verifying that those samples meet certain statistical properties in order to properly apply statistical tools to enable timing analysis, see Figure 1 (left). In particular, MBPTA applies Extreme Value Theory (EVT) [10] that requires independence and identical distribution of the execution times [10].

Cache randomization involves three main elements: a pseudo-random number generator (PRNG), random placement and random replacement, the latter of which is optional. Several works show the existence of low-overhead PRNG that provide enough quality in the sequences produced to avoid correlations [3]. Regarding random placement [15, 16, 24], it requires to adhere to certain properties for MBPTA compliance (referred to as *mbpta-px*).

**mbpta-p1** *Time composability* across incremental software integration ensures that early phase WCET estimates (ideally at the unit testing level) hold upon integration. This decreases the risk of costly detection of timing violations during late design phases. Time composability, relates cache layouts, i.e. how addresses are mapped to cache sets, during AP and OP. Time composability builds on one of the following properties on random cache placement.

**mbpta-p2** *Full Randomness.* Let us assume two different addresses $A$ and $B$, i.e. they differ at least in one bit (excluding offset bits within the cache line) and a cache with $S$ sets.

(1) $A$ (and $B$) is randomly placed to different sets for different seeds: That is, there exist seeds $seed_i$, $seed_j$ and $seed_k$ so that $S_A^{seed_i} \neq S_A^{seed_j}$, and $S_A^{seed_i} = S_A^{seed_k}$.
(2) The set where $A$ and $B$ are mapped to is not systematically identical. That is, for some seeds, $seed_i$, $S_A^{seed_i} \neq S_B^{seed_i}$, whereas for others, $seed_j$, $S_A^{seed_j} = S_B^{seed_j}$.
(3) It is required to keep the same cache-line alignment during AP and OP so that if $A$ and $B$ belong to different cache lines at OP, they also do in experiments carried out at AP.

Note that, for any seed, it is not needed that $A$ and $B$ have the same probability of being mapped to the same set.

**mbpta-p3** *Partial APOP-fixed Randomness.* In this case, randomization is carried out at memory page boundary.

(1) If $A$ and $B$ are in the same page boundary, the probability to map them to the same set is null for any seed.
(2) If $A$ and $B$ belong to different pages, the same principles than for full-randomization apply.
(3) It is required to keep the same memory-page alignment during AP and OP so that if $A$ and $B$ belong to different pages at OP, they also do in experiments carried out at AP.

## 2.2 Cache Timing Side-Channel Attacks (SCA)

With cache timing-based SCA, or simply SCA, the attacker infers information about the keys based on the execution time variability caused by cache memories [7, 8, 25]. SCA exploit the difference in time that memory access patterns expose; in particular, hit and miss patterns that occur in caches. When these patterns are related to the placement of the data in memory, for instance, attackers can exploit the deterministic behavior of high-performance computing caches to extract cryptographic keys [7]. In particular, SCA are enabled by i) the time difference between accesses: misses on a cache take longer to resolve than hits, hence leaking which data is being used and present in cache, and which data is not there; and ii) the use of table lookups that are input-dependent in cryptographic algorithms (e.g. AES). In this paper we focus on the particular SCA attack referred to as Contention Based attacks [18].

Contention attacks are based on conflicts between cache lines, and include Prime-Probe and Evict-Time attacks. Given that accesses to the lookup tables depend on the input data, an attacker is able to extract cryptographic keys by measuring the time it takes to the victim or to itself to load certain data. Most of this kind of attacks assume that the attacker shares the use of the processor with the victim [23]. However, it is not necessary to have a contender in the same processor in order to perform contention attacks: Bernstein [7] proved that interference inside the victim's own accesses might be enough to reveal full cryptographic keys.

Let $\Omega$ be the universe of input states, $\Gamma$ the universe of execution times that a task can exhibit, $t_\omega^\gamma$ the execution time $\gamma \in \Gamma$ of a cryptographic task given the input $\omega \in \Omega$, and $P(t_\omega^\gamma)$ the probability of observing such execution time. For protection against SCA, **sca-p1** cache designs must ensure that no correlation exists between the input data and the observed execution time. In this way, any single input state can exhibit several execution times with equal probability, thus preventing any attacker from identifying the input state from the execution time: $\forall i \in \Omega, \forall g \in \Gamma \mid P(t_i^g) = \frac{1}{|\Gamma|}$.

## 3 ASSESSING THE TIME-PREDICTABILITY OF SECURE CACHE DESIGNS

The **RPCache** [27] decouples cache interference of the attacker from the victim by randomizing interference whenever a memory access from a process different from the victim's one contends for the same cache line. On the event of a contention event that might leak information, a random set is selected for replacement. So far the MBPTA compliance of this design has not been assessed. In particular we identify two features of this approach that fail to meet MBPTA requirements: First, the timing behavior of the task under analysis depends on the actual addresses accessed. Therefore, WCET estimates do not hold across integration with other software units, which may change the actual addresses of the task and hence, change its timing behavior arbitrarily. This prevents achieving time composability (requirement mbpta-p1). And second, contender tasks produce cache evictions in random sets upon contention with the task under analysis. Hence, since whether contention exists is fully dependent on task's contenders behavior, so is the case for task's cache evictions. Thus, the WCET estimate obtained for the task strongly depends on the contenders behavior, typically unknown in early design phases (when WCET estimates need to be successfully assessed against timing budgets). This is against time composability needed in ATS (mbpta-p1), as explained before.

The **Newcache** [28] builds on the same concept as RPCache and introduces several improvements to reconcile high-performance and security, offering low miss rates and faster accesses. However, the main concept and limitations for MBPTA-compliance behind the placement policy remains the same as for RPCache.

**Aciicmez** [2] proposes a placement policy to secure instruction caches that randomizes the cache set where addresses are placed by XORing the index bits with a random number. Let's assume two addresses $A$ and $B$ and further assume that they have identical index bits. Hence they are placed in the same set with modulo. By XORing their (identical) index bits with a random number, the set obtained is random, but identical for both addresses, hence breaking mbpta-p2 principles. Furthermore, if $A$ and $B$ have different index bits they are placed in different sets with modulo. By XORing their (different) index bits with a random number, the set obtained is also different, so they are placed in different sets. This breaks mbpta-p2 since it is not the case that for different seeds $seed_i$ and $seed_j$: $S_A^{seed_i} \neq S_B^{seed_i}$ and $S_A^{seed_j} = S_B^{seed_j}$.

While [2] performs a random permutation of cache sets based on the random number used, its timing behavior is strictly dependent on the addresses and analogous to that of modulo placement, breaking mbpta-p1. Also, performance (and so WCET) are strictly dependent on the actual addresses accessed, so WCET estimates become invalid upon integration since different memory locations of objects will lead to arbitrarily different cache conflicts.

## 4 ASSESSING THE SCA-ROBUSTNESS OF TIME-PREDICTABLE CACHE DESIGNS

For MBPTA compliance caches implement random placement and (optionally) random replacement. Random placement, determines the cache set where an address is mapped by operating the address (tag and index bits) together with a random number called random seed or just *seed* [15, 16, 24]. The remaining address bits (offset bits) are only used to select the particular word accessed within the cache line. Given an address and a *seed*, random placement delivers a fixed cache set. However, differently to the proposal by Aciicmez [2], addresses are placed randomly and *independently* in cache. For instance, addresses $A$ and $B$ are placed on the same set for some seeds only. Hence, cache conflicts across different seeds are random, making actual addresses irrelevant when determining the cache sets they are mapped to. This relieves the end user from controlling memory mapping. Two different designs implement random placement: hash-based parametric random placement (hashRP) [16] and random modulo (RM) [15, 24], see Figure 2.

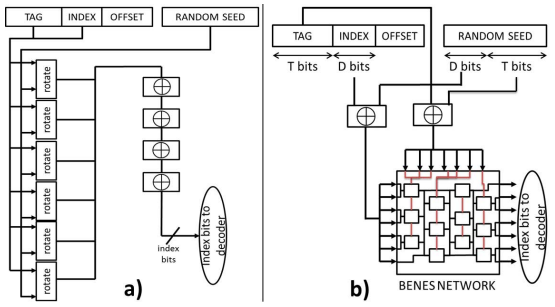

**Figure 2: (a) HashRP and (b) RM Cache Architectures**

**HashRP** [16] operates on tag+index address bits with a *seed* by means of rotator blocks and XOR gates so that conflicts in different sets are made random, see Figure 2 (a). HashRP poses no constraint on whether cache lines need to belong to the same page. Its performance is slightly lower than that of RM and modulo placement, but it is compatible with second level (L2) and third level (L3) caches whose way size may be much larger than the page size. HashRP achieves *Full Randomness* (mbpta-p2).

**RM** [15, 24] takes as input the XORed bits of the *seed* with the index and tag bits of the address, see Figure 2 (b). The XORed index bits are the input to a Benes Network [4] and the XORed tag bits are used to drive the network. The output of this Benes Network is a randomized permutation of the index bits that point to a particular cache set. RM is compatible with caches whose page size is equal or a multiplier of the cache way size. Often first level (L1) caches use a way size equal to the page size. With RM each address is placed in a random set with uniform probability, but addresses in the same page are placed in different cache sets. RM is compatible with MBPTA when contents in each memory page are preserved across integrations (easily achieved by current RTOS), while allowing pages move freely in the memory space. Overall, RM achieves *Partial APOP-fixed Randomness* (mbpta-p3).

**Compliance with SCA protection properties**. MBPTA compliance for caches relies on random placement to exhibit randomized execution times. To achieve SCA robustness, random placement must also decouple cache interference of the attacker from the victim. That is, memory addresses from victim and attacker's processes must not contend systematically in the same cache set. Instead, each memory address from each process must be randomly and *independently* placed in a set, thus randomizing interference. In the following section we detail how to achieve time-predictability and security against SCA in the same design.

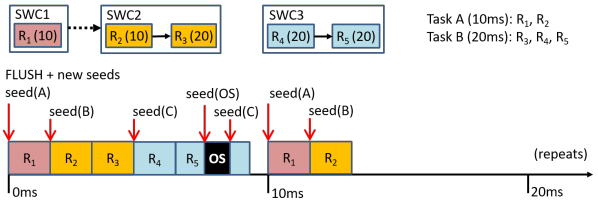


**Figure 3: Example of AUTOSAR app. and seed management.**

## 5 TIME-SECURE CACHES

In order to attain both, MBPTA-compliance and SCA robustness, either MBPTA-randomization solutions are made SCA-aware or sca-randomization solutions are made MBPTA-aware. Without loss of generality, we opt for the former. SCA-aware caches cause variations in timing behavior for which achieving MBPTA-compliance require specific ad-hoc solutions. Studying those solutions is part of our future work.

HashRP and RM preserve the same *seed* during the execution of a task, so that cache contents can be retrieved and kept consistent. When cache contents are private to each task (there is no shared data), then flushing is not needed across different tasks despite using different seeds since consistency is not affected. Whenever a given task instance (job) executes, then either cache contents need to be flushed or the *seed* used in the previous job of the task has to be used again to preserve consistency.

MBPTA-compliance adds light constraints on seed management. Depending on the scope of the application of the WCET estimate, which for instance defines whether exceedance thresholds apply to the task as a whole or to each job independently, the granularity at which the seed has to be changed varies. On one extreme of the spectrum the seed is (randomly) set once before the execution of the first job of a task. On the other extreme of the spectrum the seed is changed right before every job release.

Interestingly MBPTA-compliance sets no constraint on the seeds used for different software units (tasks), which threatens security since two different tasks could have the same *seed* and therefore their behavior can be reproduced.

In the context of SCA, for contention attacks if the attacker task is allowed to use the same *seed* as the task under analysis, then it will have the same (random) placement as the victim. Hence, it will have the ability to learn about the victim as we show later in the evaluation section, using contention-based attacks. Instead, if each task is forced to have a different *seed*, conflicts between attacker's and victim's cache lines are random and independent across runs, thus defeating any contention-based attack, since the attacker loses the ability to create contention for specific victim's data.

**Implementing per-process unique seeds.** In order to prevent contention-based attacks with hashRP and RM, a different seed has to be provided to each process which requires some Operating System (OS) support, see Figure 3. In the context of AUTOSAR, applications are divided into software components (SWC), and each SWC is further divided into runnables (the atomic unit of execution). Each runnable has an associated execution period, see Figure 3, where an application has 2 SWC (SWC1 and SWC2), and another 1 SWC (SWC3), consisting of 1, 2, and 2 runnables respectively. For instance, runnable $R_2$ executes every 10ms and it produces some output read by $R_3$. Also, SWC1 produces some output read by SWC2. Communication across runnables in the same SWC can be done via shared memory, whereas across SWP with message passing. Finally, runnables of different SWC are organized into tasks where each task has a specific execution period. For instance, task $A$ includes all runnables with period 10ms ($R_1$ and $R_2$). Runnables are scheduled within a task preserving application dependencies.

In order to allow the communication between runnables of a given SWC via shared memory, with the TSCache all runnables of a given SWC *must* use the same seed. Preserving the same seed across runnables of the same SWC also reduces the number of cache flushes and hence, overheads. In the case of runnables of different SWC, they may have been developed by different providers (even if they belong to the same application). Hence, they *must not* use the same seed to prevent contention-based attacks across SWCs[1]. This implies that, on a context switch across runnables of different SWC, the OS must store the seed in the *task struct* of the SWC, empty the pipeline, and restore the seed of the incoming SWC. This way SWCs cannot learn from the cache behavior of the other SWCs. This is indicated in Figure 3 with red arrows and the seed that needs to be set. Note that whenever the OS is invoked (e.g. during $R_5$), the OS seed needs to be used for memory consistency and to prevent also contention-based attacks. Finally, whenever the whole hyperperiod elapses (20ms in the example), the OS needs to set new random seeds and flush cache contents. This ensures that execution times across runnables in different hyperperiods are random and independent. Note, however, that if the instances of a given runnable within a hyperperiod use the same seed, their timing is not independent (e.g. the two instances of $R_1$). The only practical implication is that their – arbitrarily low – exceedance probability is not independent and, if one of those instances would ever overrun its pWCET, all other instances of this runnable in the hyperperiod could also do it.

Despite seeds are changed often, cache is not flushed, so the overhead is negligible (emptying the pipeline and updating seed registers). Instead, cache flushing occurs only once per hyperperiod, as already needed for MBPTA compliance.

## 6 CASE STUDY

### 6.1 Methodology

*6.1.1 Berstein Attack and Threat Model.* Due to the current nature of ATS, Bernstein's attack [7] is a realistic scenario, since attacker and victim do not have to share the processor during the contention attack, thus being less restrictive. Additionally the vector of attack for Bernstein's intrusion matches the processor architecture currently used in ATS (i.e. the cache hierarchy).

We emulate two independent processors that execute cryptographic operations independently, the victim and the attacker. Both processors execute 128-bit AES encryption functions. For the attacker the key is known, for the victim, a randomized 128 bits key is generated. We collect then timing measurements from the processes of encryption, and then we perform a statistical correlation on the timing profiles of attacker and victim to find the secret victim's key. In the original Bernstein's experiment, victim's timing

[1]Note also that by enforcing different seeds across SWCs the system is also protected from memory attacks exploiting software vulnerabilities

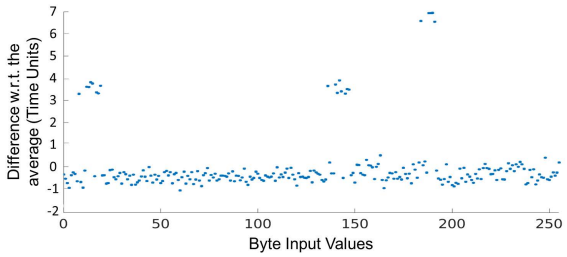


**Figure 4: Time variations with respect to average across all different values of input byte number 4.**

measurements were taken on the victim's machine to reduce interference. Hence, performing the analysis on-line or off-line gives exactly the same result. For this experiment victim and attacker obtain $10^7$ samples of AES encryption operations each.

We try Bernstein's attack on different setups, basically extracting for each 16-byte input value the average computation time per byte and value. In particular, we evaluate the robustness of the different setups by executing the attack and observing how much information (bits from the key) the attack is able to disclose. When computing the correlation between execution times observed and key values, we use for each byte the most stringent correlation factor so that (1) the number of combinations preserved is minimized while (2) keeping the correct value amongst those regarded as feasible. Hence, this is the best case for the attacker.

*6.1.2 Experimental Setup.* We use a cycle accurate simulator based on SocLib [22], modified accordingly to include the RPCache, HashRP and RM caches. The processor setup resembles the ARM920T [5] a single-core automotive microcontroller chip. We model a 5-stage processor; with 16KB, 128 sets, 4-way first level instruction and data caches; and a 256KB, 2048 sets, 4-way L2 cache. We evaluate four different setups: (a) *deterministic*: a baseline vulnerable processor with time-deterministic caches; (b) *RPCache*: a secure processor implementing the RPCache [27]; (c) *MBPTACache*: a processor implementing a random cache for MBPTA compliance; and (d) *TSCache*: our proposal to simultaneously handle timing and SCA. For MBPTACache and TSCache, the L1 caches implement RM while the shared L2 cache HashRP.

## 6.2 Results

*6.2.1 SCA robustness.* In Figure 4 we show how certain values for a given input (byte number 4) take slightly longer to be processed on the baseline *deterministic* setup. Those values with higher execution time allow the attacker to retrieve the value of the particular byte of the key or, at least, reduce the number of potential combinations drastically so that a brute force exploration of the (limited) remaining combinations can break the key.

Figure 5 shows for each cache setup the different bytes of the key in the y-axis and their potential values in the x-axis. White cells correspond to values effectively discarded by the attack, whereas grey cells correspond, to values that could not be discarded. Black cells correspond to the particular value of the key. Hence, the whiter, the more effective the attack. As shown, Bernstein's attack is effective for half of the bytes on the deterministic setup (top left plot): the attack is able to determine 33 bits out of the 128 bits of the key and other combinations are also discarded, the number of

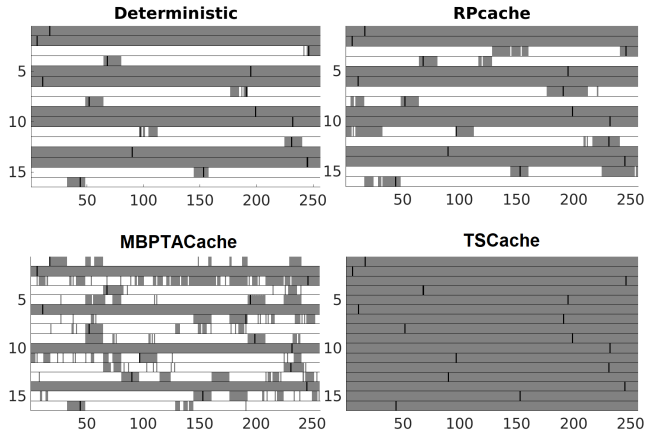


**Figure 5: Effectiveness of the Bernstein's attack.**

remaining combinations in decreases down to $2^{80}$. This number of potential key values is below the $2^{128}$, decreasing the cost of a brute force attack by a factor of $2^{48}$.

For the RPCache, the same bytes as for the baseline setup are vulnerable to the attack. However, the RPCache proves to be stronger in front of this attack by keeping the number of potential keys to explore at $2^{108}$. Still, some information is leaked. When using the MBPTACache (bottom left plot), vulnerability to the attack occurs in different bytes as for the other caches. Overall, the number of potential key values is $2^{104}$, thus close to the case of the RPcache.

Finally, with the proposed TSCache, the best case attack is unable to disregard any value for any byte. Since TSCache makes placement fully random and independent of the actual address accessed across different seeds, Bernstein's attack fails to reveal any information, thus preserving key strength at $2^{128}$. In fact, Bernstein's attack regards some values as more likely to be the key ones for several bytes, but discarding the key value for some of those bytes. Hence, TSCache, rather than preventing the attack from inferring any information by transforming it into noise, fools the attacker by providing wrong information that would not allow a brute force attack to reveal the key if fewer combinations are explored.

**Generalization**. Contention-based attacks, such as Bernstein's one, rely on deterministic eviction of controlled cache lines. Hence, Prime-Probe and Evict-Time Attacks, both contention-based, are thwarted by using secure time-predictable caches since the cache layouts of different processes are completely independent and randomized. As explained in section 4, those attacks rely on the ability to reproduce and infer from timing profiles the inputs used by the victim. By having different seeds for victim and attacker tasks, their input state differs and so the timing profiles also differ. Hence, contention-based attacks cannot relate execution time variations with any other information, thus failing as Bernstein's one.

*6.2.2 MBPTA-compliance.* TSCache achieves Partial APOP-fixed Randomness properties (mbpta-p3), maintaining MBPTA-compliance, see Section 2. We further validated that the observed execution time fulfills the independence and identical distribution properties as required by EVT as used in MBPTA. We use the Ljung-Box independence test [9] to test autocorrelation for 20 different lags simultaneously, a very strong independence test. We have also

applied the Kolmogorov-Smirnov two-sample i.d. test [13]. All our samples have passed both tests for a $\alpha = 0.05$ significance level.

*6.2.3 Overheads.* RM and HashRP area and performance overheads are as follows. **Area**. RM and HashRP caches have already been implemented on a LEON3-based multicore processor causing no operating frequency degradation on an FPGA [15]. In terms of area, while we cannot isolate the cost of cache modifications, making the whole processor MBPTA-compliant (so modifying all caches, bus arbitration and FP units included) and adding an enhanced tracing feature costed less than 1% processor area increase. **Performance**. hashRP and RM have been shown to have no effect on the maximum operating frequency of their FPGA implementation [15]. Also, they have similar cache behaviour to that of standard modulo placement. Specially RM has shown a miss rate 1% far from modulo [15], hence with negligible impact on average performance. The impact in performance due to the seed change is also negligible. Seed changes are produced for security reasons for which restoring the seed of the process to be executed next would only require to wait until all accesses in flight of the previous process have been served, which would take tens of cycles. Also changing the seed for time-predictability reasons this implies flushing the cache. However this is required at coarser granularity, hence the relative cost of flushing is contained.

## 7 RELATED WORK

RM and hashRP caches have been implemented on a commercial design [26] showing negligible area and operating frequency impact are negligible. While performance cost of RM and hashRP has been shows low w.r.t. default deterministic cache designs in industrial case studies [14, 17, 29], RM and hashRP are only needed for time- or security-critical tasks. They can be disabled at will for efficiency reasons for other applications by bypassing RM and hashRP modules, and forwarding index bits from the address directly to cache decoders since RM and hashRP implementation is little intrusive and does not affect the design of the main cache components.

Several works mitigate different cache SCA [12, 23]. In this context, cache partitioning has been proposed to solve both, contention-based SCA [11, 27] and to achieve time predictability [20]. The idea is to disable interference by isolating cache lines across different processes. However, cache partitioning severely limits the effectiveness of shared caches in multicores affecting both, performance and the ability to share data across threads running in different cores [21]. This relates to the difficulties to partition also all cache buffers and queues, as well as to the performance impact of reduced cache associativity per partition.

Randomization mitigates the amount of information leaked [11, 18, 27]. However, as stated previously, the applicability of these solutions to timing analysis is inherently compromised. Overall, to the best of our knowledge this is the first work proposing hardware designs that address both, timing analysis and SCA concerns.

## 8 CONCLUSIONS

Increasing performance needs in ATS require the adoption of high-performance hardware features such as caches, that however, challenge time-predictability and make systems vulnerable to timing-based SCA. While those concerns have been addressed individually, existing solutions have not been proven compatible for both concerns. We analyzed the suitability of the solutions devised for each concern against the requirements of the other, proving that they fail to achieve both goals simultaneously.We propose TSCache which effectively delivers time-predictability for MBPTA and robustness against contention cache-timing SCA

## ACKNOWLEDGEMENTS

This work has been partially funded by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal fellowship number RYC-2013-14717. Authors want to thank Boris Kpf for his technical comments in early versions of this work.

## REFERENCES

[1] J. Abella et al. 2015. WCET Analysis Methods: Pitfalls and Challenges on their Trustworthiness. In *SIES*.
[2] O. Aciicmez et al. 2011. Method and system for securing instruction caches using substantially random instruction mapping scheme. (Nov. 8 2011). http://www.google.tl/patents/US8055848 US Patent 8,055,848.
[3] I. Agirre et al. 2015. IEC-61508 SIL 3 Compliant Pseudo-Random Number Generators for Probabilistic Timing Analysis. In *DSD*.
[4] V. Bene al. 1964. Optimal Rearrangeable Multistage Connecting Networks. In *Bell System Technical Journal*.
[5] ARM. 2001. *ARM920T Technical Reference Manual*.
[6] ARM. 2015. *ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade*. Technical Report. ARM. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php.
[7] D. J. Bernstein. 2005. Cache-timing attacks on AES. In *Tech. Rep.*
[8] J. Bonneau and I. Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *CHES*.
[9] G.E.P. Box and D.A. Pierce. 1970. Distribution of Residual Autocorrelations in Autoregressive-Integrated Moving Average Time Series Models. *J. Amer. Statist. Assoc.* (1970).
[10] L. Cucu-Grosjean et al. 2012. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *ECRTS*.
[11] L. Domnitser et al. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. In *ACM TACO*.
[12] Qian Ge et al. 2017. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In *Journal of Cryptographic Engineering*.
[13] W. Feller. 1996. *An introduction to Probability Theory and Its Applications*.
[14] M. Fernández et al. 2017. Probabilistic timing analysis on time-randomized platforms for the space domain. In *DATE*.
[15] C. Hernandez et al. 2016. Random Modulo: a New Processor Cache Design for Real-Time Critical Systems. In *DAC*.
[16] L. Kosmidis et al. 2013. A Cache Design for Probabilistically Analysable Real-time Systems. In *DATE*.
[17] L. Kosmidis et al. 2016. Measurement-Based Timing Analysis of the AURIX Caches. In *WCET Workshop*.
[18] F. Liu and R.B. Lee. 2014. Random Fill Cache Architecture. In *MICRO*.
[19] E. Mezzetti and T. Vardanega. 2013. A rapid cache-aware procedure positioning optimization to favor incremental development. In *RTAS*.
[20] M. Paolieri et al. 2009. Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In *ISCA*.
[21] M. Slijepcevic et al. 2014. Time-Analysable Non-Partitioned Shared Caches for Real-Time Multicore Systems. In *DAC*.
[22] SoCLib. 2003-2012. -. (2003-2012). http://www.soclib.fr/trac/dev.
[23] J. Szefer. 2016. Survey of Microarchtiectural Side and Covert Channels, Attacks and Defenses. In *IACR Archive*.
[24] D. Trilla et al. 2016. Resilient Random Modulo Cache Memories for Probabilistically-Analyzable Real-Time Systems. In *IOLTS*.
[25] Y. Tsunoo et al. 2003. Cryptoanalysis of DES implemented on computers with cache. In *CHES*.
[26] http://www.gaisler.com/index.php/products/processors/leon3. [n. d.]. *Leon3 Processor*. Areroflex Gaisler.
[27] Z. Wang and R.B. Lee. 2007. New Cache Designs for Thwarting software Cache-based Side Channel Attacks. In *ISCA*.
[28] Z. Wang and R.B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *MICRO*.
[29] F. Wartel et al. 2013. Timing Analysis of an Avionics Case Study on Complex Hardware/Software Platforms. In *SIES*.