A Fine-Grained Dynamic Partitioning Against Cache-Based Timing Attacks via Cache Locking

Nicolas Gaudin, Pascal Cotret, Gogniat Guy, Vianney Lapotre

A Fine-Grained Dynamic Partitioning Against Cache-Based Timing Attacks via Cache Locking

Nicolas Gaudin[1], Pascal Cotret[2], Guy Gogniat[1], Vianney Lapôtre[1]
[1]UMR 6285, Lab-STICC, Univ. Bretagne-Sud, Lorient, France
[2]UMR 6285, Lab-STICC, ENSTA Bretagne, Brest, France

***Abstract*—Cache-based timing side-channel attacks are prevalent and correspond to a security threat for both high-end and embedded processors. In this paper, we propose and implement a fine-grained dynamic partitioning countermeasure relying on a hardware-software collaboration. The proposed approach extends the RISC-V Instruction Set Architecture (ISA) with `lock` and `unlock` instructions to allow a program to explicitly lock cache lines in the data cache memory, ensuring constant-time accesses. Experimental results show that the proposed solution defeats contention-based cache side-channel attacks such as PRIME+PROBE and leads to a low area overhead (<3%), a low impact on binary code size (<0.3%) and a low impact on miss rate (<2%).**

***Index Terms*—Cache Architecture, RISC-V, Security, Cache Side-Channel**

I. INTRODUCTION

The observation of physical effects resulting from processor execution can be used to retrieve confidential information [1]. Among them, this work focuses on cache-based timing attacks [2]. These attacks rely on the capacity of a malicious process to deduce secret information about a victim process by observing its use of shared cache memories. They usually rely on both cache contention and cache profiling and exploit a timing side-channel due to memory hierarchy latency.

Numerous countermeasures have been proposed to thwart cache-based timing attacks. At the software level, a code transformation is performed to remove secret-dependent memory accesses [3]. Although effective, this approach is tedious for the programmer and requires an extensive knowledge of the hardware and the associated compiler.

Hardware-wise, cache randomization and cache partitioning are the two main protection techniques. The objective of randomized caches [4]–[6] is to prevent attackers from finding eviction sets by scattering addresses with identical indexes. However, PRIME+PRUNE+PROBE [7] defeats most randomized cache solutions. It allows one to find an eviction set after a few hundred accesses, reducing the security efficiency of randomized cache. Thus, the random mapping needs to be refreshed regularly to maintain the security level. This costly operation limits the adoption of randomized cache.

The alternative hardware mitigation relies on resource partitioning with the aim of preventing an attacker to manipulate victim's data by allocating dedicated cache resources. A class of solutions such as NOMO-CACHE [8] or SECDCP [9] proposes to allocate a set of ways to sensitive applications. However, He and Lee [10] highlight that this class leads to important performance loss due to coarse-grained partitioning. PLCACHE [4] is a lightweight partitioning mechanism that reserves cache lines through `lock` and `unlock` instructions to protect against contention-based cache side-channel attacks. A locked cache line cannot be evicted by any other process. Nevertheless, PLCACHE suffers from security limitations [11], [12]. First, depending on the cache state and the current state of the replacement policy metadata, data stored in locked cache lines can be implicitly evicted by the owner process. Second, memory access can circumvent the cache hierarchy when the replacement policy targets a locked cache line owned by another process. Consequently, an attacker can manipulate the replacement policy metadata in order to take advantage of the two previous statements and force non-constant time memory accesses to victim's data. In [11], authors propose new definitions of the `lock` and `unlock` instructions to guarantee constant-time access to data locked in cache lines. In this approach, only the explicit use of the `unlock` instruction allows the eviction of locked data. However, the proposed solution is limited to the protection of a single process and leads to a high area overhead (+39%). Finally, the security evaluation is limited to an instruction and cache simulator. In this paper, we go beyond previous works by proposing a low-cost, flexible, secure, and efficient solution to mitigate cache-based timing attacks targeting low area system-on-chips. Contributions of this paper can be summarized as follows:

- We introduce a strict cache locking mechanism considering a multitask embedded system.
- We propose a low-overhead hardware implementation of this mechanism, targeting a Field Programmable Gate Array (FPGA).
- We evaluate both timing performance and security using a set of benchmarks and considering the PRIME+PROBE attack.

Section II gives the prerequisites. Section III describes our threat model. Section IV describes our strict cache locking mechanism. Section V details our implementation and synthesis results. Sections VI and VII evaluate security and performance of our solution. Section IX discusses our approach. Finally, Section X concludes the paper.

This work was supported by the Cominlabs excellence laboratory with funding from the French National Research Agency (ANR-10-LABX-07- 01).

II. Background

Main cache-based timing attacks rely on the contention on shared cache memories and the exploitation of timing side-channels caused by the temporal distance of memory accesses induced by the memory hierarchy to monitor victim's activities and deduce sensitive information such as cryptographic keys. To perform such attacks, an attacker should be able to tamper the state of the target cache memory and distinguish between cache hits and cache misses, through time measurements, when accessing its own data. The following three steps are performed in most cache-based timing attacks:

1) The attacker sets the cache (or part of it) in a known state by filling it with its own data or by flushing a set of data.
2) The victim executes and may load or evict some data monitored by the attacker.
3) The attacker times its own memory accesses to determine whether the victim has evicted or accessed the monitored data.

Based on these three steps, numerous attacks variations have been studied to conform to the specifications of the ISA (*e.g.* EVICT+RELOAD [13]) or to circumvent the latest countermeasures (*e.g.* PRIME+PRUNE+PROBE [7]). The PRIME+PROBE [14] technique is the most popular attack exploiting cache contention. In the PRIME step, the attacker program fills different cache sets with its own data. The attacker then goes into an idle state, in which it lets the victim execute. In the PROBE phase, the attacker observes its own filled cache by monitoring the time to load data that has been cached during the PRIME phase. By repeating this process along the victim execution and by analyzing which data has been evicted by the victim, the attacker obtains information on memory addresses accessed by the victim. From this information, the attacker deduces sensitive information that is associated with these memory accesses (*e.g.* AES S-Box access pattern or instructions flow).

III. Threat model

In [15], authors demonstrate that cache-based timing side-channel attacks can be exploited on mono-core CPU to extract sensitive information. In this paper, we assume two processes running on an in-order mono-core processor, including a single level of shared data cache. These processes do not share their memory spaces. The attacker has knowledge of the binary code of the program belonging to the victim (*e.g.* via open-source cryptographic library or through reverse engineering). The attacker can infer the cache state before and after the victim's execution and after each context switch. The attacker can accurately measure time through the processor cycle counter register, thus determining if its memory accesses result in a cache hit or a cache miss. The attacker is able to synchronize with the victim process by triggering its execution.

The entire software stack in charge of process scheduling is part of the Trusted Computing Base. We assume that the control flow of the victim does not leak any sensitive information (*e.g.* using constant-time programming techniques [3], [16]). Thus, only memory accesses can be secret-dependent. As a consequence, the attacker focuses on contention-based attacks such as PRIME+PROBE to manipulate the cache, attempting to infer information about the secret-dependent accesses of the victim.

IV. Dynamic partitioning through cache locking

From the aforementioned threat model and literature, we consider the locking mechanism introduced in [11] to thwart cache-based attacks in our system. It consists of a software-driven fine-grained partitioning that allows data to be locked in cache memories at the cache line granularity. An extension of the ISA allows using the proposed mechanism by implementing two instructions: `lock` and `unlock`. Any access to data stored in a locked cache line leads to a cache hit. A locked cache line cannot be evicted, regardless of which process is manipulating the cache. The only way to unlock a cache line (and then evict it) is that the owner process explicitly uses the `unlock` instruction. Once the cache line is unlocked using the `unlock` instruction, the cache line can be evicted by any process.

A. Cache access procedure

The use of this locking mechanism is motivated by the need to optimize the cache usage by protecting a limited range of the memory space for a limited period of time. For a programmer, the motivation is to lock a set of data which accesses are secret-dependent. Despite that unavailable cache space may result in a drop in performance for other running processes, this on-demand approach allows the protected process to release the locked cache lines as soon as possible and then continue its execution. In order to prevent an access from bypassing the cache because all the ways of the accessed set have been locked, at least one free way is kept available to ensure a minimum performance for running applications. An hardware exception is raised when a lock instruction targets a fully-locked cache set. This exception has to be caught by the software stack to pause or stop the current process.

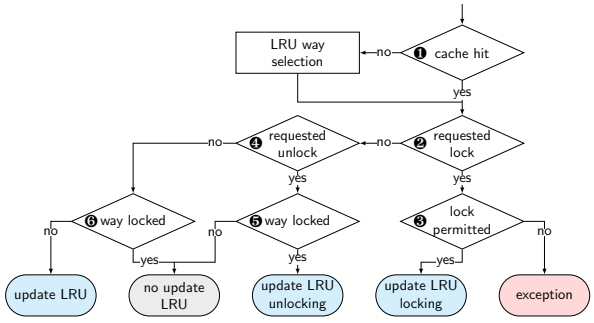


Fig. 1. Cache access procedure with locking mechanism.

Figure 1 details the procedure when accessing the cache with the proposed locking mechanism. First, a test checks whether the accessed data is already cached ❶ (cache hit), if

not, the replacement policy selects the way to cache the data. Then, if a lock is requested ❷, a test determines whether the state of the cache set permits to lock the accessed cache line ❸. If a way is available, the accessed cache line is marked as locked and the Least Recently Used (LRU) replacement policy is updated for candidate ways. Otherwise, an exception is raised. If an unlock request is received ❹, and if the accessed cache line is locked ❺, the unlock is achieved by re-introducing the cache line among the LRU candidates. Otherwise, the request concerns a non-locked cache line and the request is discarded. Conventional `load` or `store` requests (❻) are treated as follows: if a locked cache line is accessed, the LRU is not updated. Otherwise, the LRU is updated for each non-locked way in the set.

B. Extending LRU for locking mechanism

In this paper, we propose to extend the LRU replacement policy metadata and its update process to support the locking mechanism. Considering an N-way set associative cache and the original LRU, the metadata associated to a way can be associated to N states (*e.g.* for a 4-way set associative, states are included from 1 to 4). This state reflects the age of the cache line in the cache set (*e.g.* 1 as most recently used, and 4 as least recently used). Our solution relies on an extra LRU state dedicated to locked cache lines (locked state equals to 0). Thus, $N+1$ states have to be considered. When a memory access is performed, the LRU metadata is updated taking into account the following rules. If the request is associated to a lock instruction and the lock is allowed (see Section IV-A), the selected way state is updated to the lock state and the rest of non-locked ways metadata of the set are updated with state values between $N_{lock}+1$ and N, where $N_{lock} < N$ is the current number of locked cache lines in the set (*e.g.* if N_{lock} equals to 2, two states are 0 and the 2 remaining states are respectively 3 and 4). If the request is associated to an `unlock` instruction, the selected way state is updated to a state value equals to N_{lock}. In this case, the rest of non-locked ways metadata of the set does not need to be updated. Finally, if any other memory access is performed, the LRU metadata of the non-locked ways of the set are updated with state values between N and N_{lock}.

C. Practical use-case

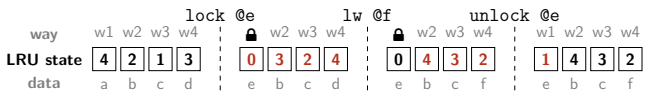


Fig. 2. Use-case : Behavior of LRU states considering our `lock` mechanism with N=4 ways.

Figure 2 presents a pedagogical example highlighting the behavior of LRU states for three memory accesses considering consecutively a `lock`, a conventional access (`lw`) and a `unlock` where the addresses mapped to the same cache set. It presents the replacement policy metadata (*i.e.* LRU states) for each way of a cache set. The cache set is composed of N=4 ways (wX) and is warm, *i.e.* already used. Data `a, b, c,` and `d` are stored in the 4 ways. Before considering the access `lock @e`, we can notice that least recently used way is w1 (LRU state = N). Moreover, there is no locked cache line (*i.e.* N_{lock} = 0 and LRU states $\geq N_{lock}+1$ for all ways). Then, after performing the `lock @e` memory access instruction, data `a` has been evicted and w1 is now considered as locked (LRU state = 0) as well. Also, states of the other non-locked ways have been updated, and now there remains three LRU candidates where LRU state $\geq N_{lock}+1$ with N_{lock} = 1. w1 cannot be selected by the replacement policy as it is locked. After loading a word at address f (`lw @f`), data `d` has been evicted by data `f` and way w4 becomes the most recently used LRU state = $N_{lock}+1$. LRU states for w2 and w3 have been updated while w1 is still locked. There are still three candidates for the replacement policy. Finally, w1 is unlocked by executing instruction `unlock @e`, and has been inserted back as a potential candidate and is now considered as the most recently used.

D. Software utilization

```
1  void fct(int* sensitive_table, int* input){
2    //lock phase
3    for(int i=0; i<sizeof(sensitive_table); i+=16)
4        lock_macro(&sensitive_table, i);
5
6    //algo accesses table depending on secret
7    algo(sensitive_table, input);
8
9    //unlock phase
10   for(int i=0; i<sizeof(sensitive_table); i+=16)
11       unlock_macro(&sensitive_table, i);
12 }
```

Listing 1. Example of use of the cache locking mechanism.

```
1 c.mv  t4,a4 # move &table in t4
2 c.mv  t5,a5 # move i in t5
3 c.add t4,t5
4 lock  x0,0(t4)
```

Listing 2. `lock_macro` in assembly.

In the proposed approach, the lock mechanism has to be explicitly used by the programmer to protect a set of sensitive data. Listing 1 shows a function `fct` that implements the lock mechanism to secure the secret-dependent accesses within the called function `algo`. It illustrates the insertion of `lock` and `unlock` instructions, which are inserted before and after `algo` in line 7. During the lock phase, a `for` loop is used on lines 3-4 to parse the sensitive table with a step size equal to the cache line size B = 16 bytes. In order to lock the whole table, an assembly macro, presented in listing 2, is called for each cache line. Furthermore, it is important to highlight that the number of locked cache lines should not depend on the secret manipulated by `algo`. After the execution of the sensitive algorithm, the sensitive table is unlocked from the cache with a similar `for` loop in lines 10-11. This loop uses

an unlock macro, which is similar to Listing 2, except that instruction in line 4 is replaced by an `unlock` instruction.

Listing 1 shows how a programmer can explicitly lock a set of cache lines. It is worth noting that this task could be automated by coupling programmer's annotations and taint propagation techniques. However, this work is out of scope of this paper and could be part of future works.

V. Proposed Implementation

A. Hardware design

We choose to extend the CV32E40P RISC-V core [17], which has been designed for low-cost embedded systems. This four-stage in-order core implements the RV32IMC instruction set. Figure 3 shows the enhanced CV32E40P architecture. Shaded blocks highlight our modifications of the microarchitecture.

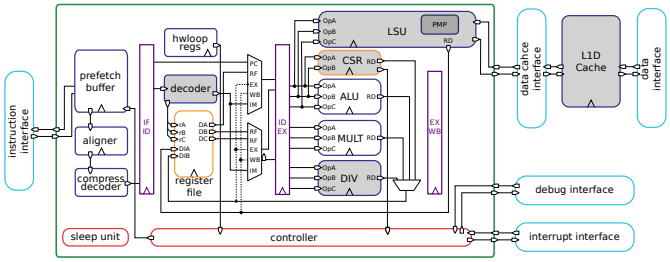


Fig. 3. CV32E40P block diagram with proposed hardware extensions to support a fine-grained locking mechanism.

The main extensions concern the three following features: 1) the *decoder* has been extended to support both `lock` and `unlock` instructions; 2) a first-level data cache (*L1D Cache* in Figure 3) has been associated to the core; 3) the core Load/Store Unit (LSU) has been extended to include a static Physical Memory Protection (PMP) module for process isolation. Furthermore, since the original CV32E40P core proposes non-constant time arithmetic operations, we also sightly modified the execution stage to prevent some timing-attacks exploiting arithmetic operation latency. For that purpose, a dedicated Control and Status Register can be set by a sensitive process to enable a constant-time mode where multi-cycle arithmetic computation such as division is performed with a fixed latency (*i.e.* the worst execution time of the operator). In our core, it only concerns division and modulo instructions.

1) Cache architecture: We couple the CV32E40P core by implementing an 8 kB 4-way set associative L1 data cache as presented in Figure 4. It implements $S = 128$ sets and cache lines of $B = 16$ bytes. Since in each set, a maximum of 3 ways can be locked, a maximum of 6 kB of data can be simultaneously protected. The core accesses the memory hierarchy using addresses of size $A = 22$ bits. The cache manipulates the address as follows: the $log_2(B) = 4$ least significant bits are used as an offset in 16-byte cache lines, the $log_2(S) = 7$ following bits of the address are used to select the cache set among the 128 available sets, and the 11 remaining bits are used as a tag to compare whether the

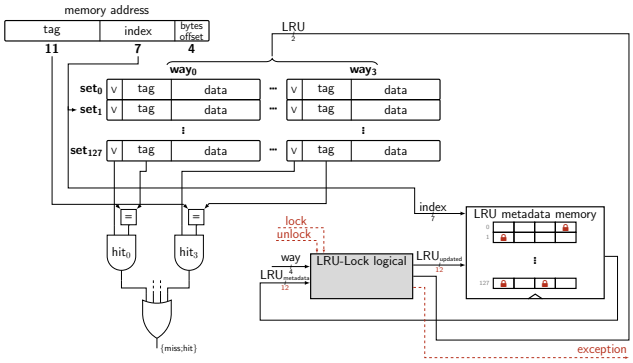


Fig. 4. A 4-way set associative cache architecture extended with the proposed locking mechanism.

accessed address is legitimate. The cache operates cache lines consisting of 16 bytes of data, $A - log_2(B) - log_2(S) = 11$ bits of tag and one validity bit, resulting in a total of 140 bits per cache line. With 128 sets, each way consists of 17,920 bits (128 sets $*140$ bits) and a total of 71.680 bits are stored in Block-RAMs (BRAMs). In order to optimize the cache access latency, 8 BRAMs are implemented (*i.e.* 2 BRAMs per way on Xilinx 7 series FPGAs). Indeed, using four separate BRAMs banks allows one to perform parallel tag comparisons. If a tag comparison matches, meaning a cache hit, the access continues with the corresponding way. Otherwise, a cache miss occurs and the LRU replacement policy selects the way to be evicted. In both cases, metadata of the replacement policy is then updated as mentioned in Figure 1.

2) Locking mechanism: In order to support the `lock` and `unlock` instructions, we both extended the core microarchitecture and the compilation toolchain. In this paper, we focus on the hardware extension. The `lock` and `unlock` instructions are mainly decoded as conventional `load` instructions. However, 2 additional control signals indicating a lock or unlock operation are generated and propagated to the cache interface through the LSU. In order to implement the locking mechanism presented in Section IV, the replacement policy metadata states have been extended (see Section IV-B). In Figure 4, the $LRU_{metadata}$ signal represents the state of the LRU candidates of the selected cache set, while the $LRU_{updated}$ represents the updated state by *LRU-Lock logical* module. Considering 4 ways, each LRU candidate can have 5 different states: state 0 which stands for the locked status and states 1, 2, 3, 4 which stand for LRU status. Thus, 3 bits are used to store the state of each way, resulting in 12 bits per set. In total, 1,536 bits (128 sets $*12$ bits) for the LRU states have to be stored (*e.g.* in half a BRAM considering a Xilinx 7 series FPGAs).

B. Synthesis Results

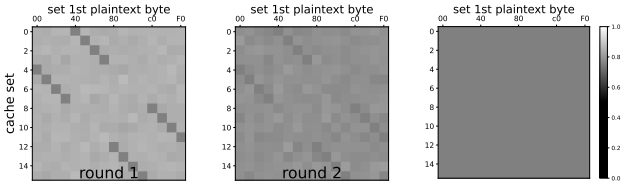
Table I shows the post-implementation area results targeting a Xilinx Kintex-7 chip with Vivado 2022.2. Cache area is extracted from CPU results, as well as the replacement policy area from the cache to present the results.

TABLE I
POST-IMPLEMENTATION AREA RESULTS ON KINTEX-7 FPGA.

		Cache	CPU
Baseline	LUTs	980	5,661
	FFs	1,065	3,465
	BRAMs	8.5	8.5
Protected	LUTs	1,007 (+2.8%)	5,683 (+0.7%)
	FFs	1,077 (+1.1%)	3,481 (+0.3%)
	BRAMs	8.5	8.5

Results show that our solution does not impact the number of required BRAMs. In both baseline and protected designs, 8 BRAMs implement the cache memory and half a BRAM stores the LRU metadata. Contiguous to this post-implementation result, the number of bits stored is hidden in BRAM for a FPGA target. However, this metric is relevant for an Application-Specific Integrated Circuit (ASIC) where bit storage is not hidden in a BRAM number. Regarding our cache configuration, the baseline cache stores 72,704 bits (cache lines + LRU metadata) of which we add 512 bits to implement our lock mechanism: it results in an overhead of 0.7%. Regarding LUTs and Flip-Flops registers considering the Cache column of Table I, our cache locking mechanism leads to 2.8% and 1.1% overheads, respectively. While for the CPU column, of which core and cache are gathered, the lock mechanism leads to an overhead of 0.3% for LUTs and 0.7% for Flip-Flops registers. It is worth noting that the overhead considering a CPU with a low-end processor is low. Regarding the maximum frequency, we obtain 105 MHz without our protection and 104 MHz with our protection. Consequently, our proposed implementation leads to a very low area overhead and a negligible impact on timing performance. The following evaluations consider the Digilent Genesys 2 board running at 100 MHz. In addition to the CPU and the cache, the memory hierarchy is completed with a 36-cycle Double Data Rate Random Access Memory (DDR RAM) of 32-bit data bus.

VI. SECURITY EVALUATION



(a) Unprotected. (b) Unprotected. (c) Entirely locked.

Fig. 5. PRIME+PROBE on AES-128 locking S-Box, key = `0x42`.

In this section, we evaluate the security of our solution. For that purpose, we considered the PRIME+PROBE attack on the AES-128 encryption algorithm. This case study focuses on the first key byte. We considered a known-plaintext attack and an AES S-Box Lookup Table implementation where memory accesses are both plaintext- (P) and key- (K) dependent. Consequently, the PRIME+PROBE attack can be used to determine which index of the substitution table has been accessed, and as a consequence, the attacker infers values for the target key byte by relying on the fact that cache line accesses are made to entries $SBOX[P_i \oplus K_i]$ with $0 \leq i \leq 16$.

Figure 5(a) illustrates a PRIME+PROBE analysis performed after 1 round of an unprotected AES encryption process for the 256 possible values of the first plaintext byte and a set key value (`0x42`). It is worth noting that the AES substitution table fills one way in 16 cache sets. The Y-axis represents the probed cache line, and the X-axis represents the value of the first byte of the plaintext. The other bytes of the plaintext are filled randomly for each experiment. These heat maps represent the average hit rate measured by the attacker over 300 encryptions. In Figure 5, the darker the square is, the more the victim has evicted the attacker data from this cache set after the PRIME phase of the attack. We can distinguish a pattern representing the substitution table accesses pattern induced by the fixed plaintext byte and the secret key value. Figure 5(b) shows that the emphasis of the pattern is reduced when the attacker probes after the second round of AES due to the diffusion property of this encryption algorithm. Thus, in the following security analysis, we consider the worst-case scenario where the attacker is able to perform his analysis after the first round of AES.

In order to evaluate our locking mechanism, Figure 5(c) shows that the attacker cannot infer any information when the victim locks the entire AES substitution table in the cache memory. An attacker can only observe a constant hit rate of 75% in the locked range due to self-eviction. Indeed, the attacker uses a prime set of 4 addresses to fill the cache set, and during the probing phase, the attacker evicts itself one of its primed addresses since one way is locked for the AES substitution table.

VII. PERFORMANCE EVALUATION

This section studies the impact of the proposed cache locking mechanism on performance. We first evaluate the performance overhead and binary code size introduced by the use of `lock` and `unlock` instructions on the encryption algorithms. Subsequently, we analyze the performance overhead induced by locked cache lines on a process that does not rely on the locking mechanism during its execution.

A. Performance of protected process

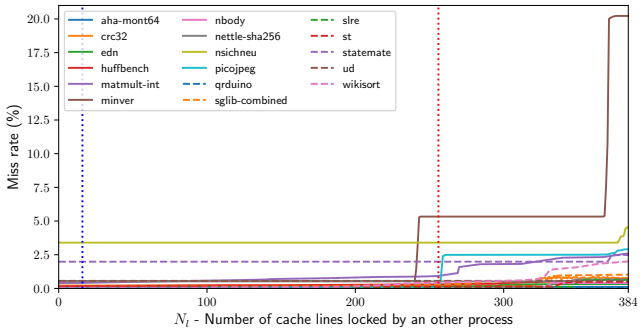
TABLE II
EXECUTION TIME OVERHEAD (%) INTRODUCING CACHE LOCKING MECHANISM TO CAMELLIA AND AES-128.

N_b	1	4	8	16	64	128	512	1024
Camellia	367.7	99.6	54.22	28.88	7.62	3.85	0.97	0.48
AES-128	2.77	0.71	0.35	0.18	0.04	0.02	-	-

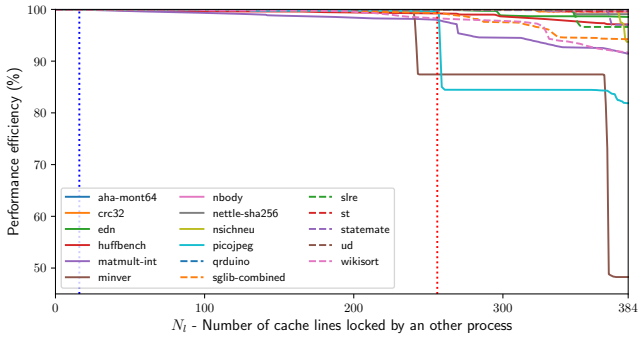
In order to conduct the evaluation, we consider two symmetric block ciphers: AES-128 and Camellia, both encrypting 128-bit plaintext blocks. AES-128 accesses a 256-byte S-Box while Camellia accesses four 1024-byte substitution tables. Table II compares execution times between a conventional

and a protected encryption of N_b plaintext blocks. While the conventional execution consists only in the encryption, the protected execution consists in a three-step process (in accordance with Section IV-D): lock the entire S-Box, encrypt N_b plaintext blocks and finally unlock the entire S-Box. Considering Camellia, when a single block is encrypted (N_b = 1), the execution time overhead increases to 368%. This cost is caused by the execution of `lock` and `unlock` instructions to lock the four 1024-byte S-Boxes. However, the performance overhead drops below 1% for 512 consecutive encryptions. The insertion of `lock` and `unlock` leads to an overhead of 0.23 % in the binary code size. Regarding AES-128, we observe a 2.8% performance overhead when encrypting a single block and drops to 0.7% when N_b = 4. The binary code size overhead is about 0.28% for AES-128. These results show that the impact of the proposed protection on performance can be considered as negligible as soon as the number of blocks to be encrypted is sufficient.

B. Impact on non-protected processes



(a) Miss rate results.



(b) Performance efficiency results.

Fig. 6. Embench-IoT 1.0 results when a concurrent process locks N_l cache lines. Blue dotted vertical line represents the need for an AES-128 process. Red dotted vertical line represents the need for a Camellia process.

In this section, we study how locking cache lines affects processes not relying on the locking mechanism during their execution. Figure 6 shows performance results related to the Embench-IoT 1.0 benchmark suite [18] while a concurrent process locks a given amount of cache lines (N_l). Figure 6(a) focuses on the miss rate metric, while Figure 6(b) focuses on the performance efficiency metric. While most kernels maintain a low and stable miss rate, some show increased miss rates as N_l tends to 384. Indeed, at this point, the cache behaves like a direct-mapped cache. Nevertheless, we can notice a visible degradation for the `minver` kernel when the concurrent process locks more than 230 cache lines. From this threshold, the `minver` kernel stack collides with its own data and causes a self-eviction due to the constraint space available in cache. `picojpeg` exhibits similar behavior when N_l = 256. Regarding the performance efficiency, Figure 6(b) shows the same behavior as performance efficiency is related to the miss rate metric. It is a matter of fact that an increase in miss rate results in a concomitant decrease in performance efficiency due to the greater number of accesses to main memory.

In order to illustrate relevant usage of the lock mechanism, two vertical dotted lines are inserted in both Figure 6(a) and Figure 6(b). The blue line, located at 16 locked caches lines, represents the number of cache lines to be locked to protect the AES-128 S-Box, while the red line, located at 256 locked cache lines, represents the four Camellia S-Boxes. AES-128 does not affect other concurrent applications because of its light load on the cache memory, for both miss rate and performance efficiency metrics. The 256 cache lines locked from Camellia increases the average miss rate by 0.36%. Notably, the `minver` kernel suffers from a significant rise from 0.55% to 5.33%, while other kernels have negligible effects. Regarding performance efficiency, Camellia affects the average performance efficiency by 1.08%. The `minver` kernel is affected by 12.5%, while `matmult-int` and `wikisort` kernels are affected by less than 2.5%.

VIII. Related work

Table III sums up the performance and area overheads for many countermeasures thwarting cache-based timing attacks. Most of the performance overheads have been extracted from Deng *et al.* evaluations [19] relying on high-end systems. The first solution, Winderix *et al.* [3], is a software transformation of the code through the compiler to make it constant time. This solution brings a performance overhead of 19% to 76%, depending on the program mitigated, and requires no hardware modifications, saving on surface area.

Regarding the hardware countermeasures modifying the cache architecture, we first focus on the randomized solutions [4]–[6]. These solutions propose hardware remapping of cache memory accesses. They imply a low performance overhead (<10%) and an area overhead ranging from 5 to 10%. It is important to notice that the performance overhead applying a randomization-based countermeasures does not consider the PRIME+PRUNE+PROBE attack where a frequent rekeying (or remapping) is required to maintain an acceptable level of security but leading to an higher performance overhead.

Then, partitioning-based solutions [8]–[9] propose a partitioning among cache ways, where cache ways are allocated to a process or a group of processes to be protected. This coarse grained partitioning introduces an acceptable performance overhead (<12%), however the evaluation results depend on

TABLE III
COMPARISON WITH CACHE DESIGNS THWARTING CACHE-BASED TIMING ATTACKS FROM THE LITERATURE.

Countermeasure	Mechanism	Type	Constant-time	Performance Overhead [19]	Area Overhead
Winderix *et al.* [3]	Compiler-Assisted Hardening	Software	✓	19-76%	0%
[4]–[6]	Randomization	Hardware	✗	1-10%	5-10%
NOMO-CACHE [8]	Fixed way partitioning	Hardware	✗	5%	-
SECDCP [9]	Dynamic way partitioning	Hardware	✗	12%	-
PLCACHE [4]	Dynamic cache line partitioning	Hybrid	✗	12%	-
Our contribution	Dynamic cache line partitioning	Hybrid	✓	2%	<3%

the application requirements. Although PLCACHE [4] relies on a similar approach, the performance and area overhead would be higher than our contribution because of the need to support cache bypassing and the complexity to handle an access passing through the cache. Nevertheless, none of the proposed solutions guarantee constant-time accesses.

IX. DISCUSSION

We have implemented and evaluated our cache locking mechanism on a low-cost and low area core. However, we believe that the proposed locking mechanism can be integrated into a performance-oriented processor integrating optimizations such as out-of-order execution, speculation or deeper pipeline. Such microarchitectural optimizations would not impact the security level offered by the proposed mechanism since a locked cache line cannot be tampered until a dedicated unlock instruction is executed. However, this claim remains true if and only if the software developer does not introduce software structure in which unlock instructions could be speculatively executed.

Security-wise, we are aware that an attacker is able to deduce the number of locked cache lines. As mentioned in Section VI, since the attacker hit rates depend on the number of locked cache lines in a cache set, he deduces that one cache line is locked if his PROBE phase leads to a hit rate of 75%. The attacker observes a 50% hit rate when 2 cache lines are locked, 25% for 3 cache lines. Consequently, the number of locked cache lines should never be secret-dependent.

The management of the proposed solution at the OS level is out the scope of this paper. However, when the limit of locked cache lines is reached and causes an exception, we believe that the process scheduling could be paused until resources are released (i.e. *unlocked*). Moreover, we believe that the proposed mechanism should be available for a reduced set of critical kernels (e.g. critical OS services) to limit the risk of Denial of Service attacks.

X. CONCLUSION

In this paper, we propose a fine-grained partitioning relying on a cache locking mechanism to thwart cache-based timing attacks. In order to provide an on-demand solution, the proposed cache locking mechanism is software-driven by extending the RISC-V ISA with new `lock` and `unlock` instructions. We detail our hardware implementation on a low area RISC-V processor and show that the proposed implementation leads to a very low area overhead (<3%).We demonstrate the security capacity of our solution using a PRIME+PROBE attack on AES-128. Our performance evaluation shows a low overhead for protected processes when using the proposed protection mechanism. The impact on concurrent processes is lower than 2% on a realistic framework. As a perspective, we aim to explore generalizing the locking mechanism to multi-level cache and implementing the proposed approach on multicore systems. We also plan to study and develop operating system services dedicated to the management of the proposed lock mechanism.

REFERENCES

[1] Q. Ge *et al.*, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, 2018.

[2] Y. Lyu *et al.*, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, 2018.

[3] H. Winderix *et al.*, "Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks," in *Proc. IEEEEuropean Symposium on Security and Privacy (EuroS&P)*, Sep. 2021.

[4] Z. Wang *et al.*, "New cache designs for thwarting software cache-based side channel attacks," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.

[5] F. Liu *et al.*, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, 2016.

[6] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. International Symposium on Microarchitecture (MICRO)*. IEEE Press, 2018.

[7] A. Purnal *et al.*, "Systematic analysis of randomization-based protected cache architectures," in *Proc. IEEE Symposium on Security and Privacy (SP)*, May 2021.

[8] L. Domnitser *et al.*, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, jan 2012.

[9] Y. Wang *et al.*, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *Proc. Design Automation Conference (DAC)*, 2016.

[10] Z. He *et al.*, "How secure is your cache against side-channel attacks?" in *Proc. International Symposium on Microarchitecture (MICRO)*, 2017.

[11] N. Gaudin *et al.*, "Work in progress: Thwarting timing attacks in microcontrollers using fine-grained hardware protections," in *Proc. IEEEEuropean Symposium on Security and Privacy Workshops (EuroS&PW)*, Jul. 2023.

[12] W. Xiong *et al.*, "Leaking information through cache lru states in commercial processors and secure caches," *IEEE Transactions on Computers*, vol. 70, no. 4, 2021.

[13] D. Gruss *et al.*, "Cache template attacks: Automating attacks on inclusive Last-Level caches," in *Proc. 24th USENIX Security Symposium (USENIX Security)*, Aug. 2015.

[14] F. Liu *et al.*, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symposium on Security and Privacy (SP)*, 2015.

[15] N. Zhang *et al.*, "Truspy: Cache side-channel information leakage from the secure world on arm devices," 2016.

[16] S. Blazy *et al.*, "Verifying constant-time implementations by abstract interpretation," *Journal Comput. Secur.*, vol. 27, 2019.

[17] O. Group. (2019) OpenHW Group CORE-V CV32E40P RISC-V IP. OpenHW Group. Https://github.com/openhwgroup/cv32e40p.

[18] Embench. (2020) Embench™: Open Benchmarks for Embedded Platforms. Embench. Https://github.com/embench/embench-iot.

[19] S. Deng *et al.*, "Analysis of secure caches using a three-step model for timing-based attacks," *Journal of Hardware and Systems Security*, vol. 3, dec 2019.