Work in Progress: Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections

Nicolas Gaudin[†], Jean-Loup Hatchikian-Houdot[*],
Frédéric Besson[*], Pascal Cotret[‡], Guy Gogniat[†], Guillaume Hiet[◇], Vianney Lapotre[†], Pierre Wilke[◇]
◇: CentraleSupélec, Inria, CNRS, IRISA, Univ. Rennes, France
*: Inria, Univ. Rennes, CNRS, IRISA, France
†: UMR 6285, Lab-STICC, Univ. Bretagne-Sud, Lorient, France
‡: UMR 6285, Lab-STICC, ENSTA Bretagne, Brest, France

Abstract—**Timing side-channels are an identified threat for security critical software. Existing countermeasures have a cost either on the hardware requirements or execution time. We focus on low-cost microcontrollers that have a very low computational capacity. Although these processors do not feature out-of-order execution or speculation, they remain vulnerable to timing attacks exploiting the varying latencies of ALU operations or memory accesses.**

We propose to augment the RISC-V ISA with security primitives that have a guaranteed timing behavior. These primitives allow constant time ALU operations and memory accesses that do not alter the state of the cache. Our approach has a low overhead in terms of hardware cost, binary code size, and execution time both for the constant time secure program and other programs running concurrently on the same hardware.

Index Terms—**Cache-based Side-Channel, Internet of Things, Security**

1. Introduction

Side-channel attacks exploit power consumption, execution time, or any other physical effect caused by an implementation, in order to deduce information about secret values, such as cryptographic keys. In this paper, we mainly focus on cache-based side-channel attacks (i.e. timing attacks exploiting shared cache memories [8]), e.g. FLUSH+RELOAD [15] or PRIME+PROBE [9].

The principle behind these attacks is the following. By measuring the time needed to perform a memory access, an attacker can deduce whether the requested memory address is cached (*cache hit*, fast) or not cached (*cache miss*, slow). From this piece of information and by carefully orchestrating the contention, an attacker can trace memory accesses of any co-running processes. More generally, any micro-architectural shared resource may induce a timing channel that can be exploited by an attacker [5].

In the literature, several approaches have been explored to thwart cache-based side-channel attacks. On the software side, the programming discipline known as *constant-time programming* [3], [14] forbids memory accesses at addresses that depend on a secret, and conditionals branches with secret conditions. This countermeasure indeed defends against cache side-channel attacks, but requires a strict discipline from the programmer and careful use of the compiler. Goldreich et al. [6] propose a program transformation in order to make the memory access patterns of different executions with different secrets indistinguishable from one another. Yet, this has a prohibitive cost [9].

On the hardware side, there are two main approaches to thwart cache-based side-channel attacks: cache randomization and cache partitioning. RPCACHE [12], SCATTERCACHE [13] and CEASER [11] propose cache designs based on randomization. The randomness is provided by permutation tables or encryption algorithms. Purnal et al. [10] evaluate these solutions and introduce PRIME+PRUNE+PROBE which allows an attacker to find eviction sets in randomized caches. In order to mitigate this attack, the cache mapping needs to be periodically updated (i.e. by generating a new permutation table or encryption key) after a few hundred accesses. The induced remapping cost limits the adoption of randomized cache.

The other approach is based on resource partitioning. NOMO-CACHE [4] partitions the cache by allocating a set of ways to sensitive applications. It implies a very low hardware overhead but leads to a performance drop. Cache partitioning techniques are effective at countering cache side-channel attacks because of process isolation, however, this strong security is at the expense of performance. Wang et al. [12] proposes PLCACHE, a lightweight mechanism allowing the lock of process cache lines, with the aim of protecting against a restricted class of cache-based side-channel attacks. They introduce two instructions to LOCK and UNLOCK a cache line. A locked cache line cannot be evicted by any other process. However, in accordance with the replacement policy, a locked cache line can be evicted by the process which locked it. Moreover, memory access can bypass the cache hierarchy when necessary (e.g. if the replacement policy points to a locked cache line owned by another process). Thus, this approach does not guarantee constant time access to locked cache lines. Yu et al. [16] propose a data oblivious Instruction Set Architecture (ISA) extension. The goal is to expose security guarantees independently of the micro-architecture and not to preclude modern hardware performance techniques (except when it leaks).

Our proposal. We introduce a flexible solution against timing side-channel attacks. More precisely, our design has the following features:

- We augment the RISC-V ISA with two instructions, `lock` and `unlock`, which allow to lock and unlock a cache line inside the cache. This guarantees that memory accesses at locked addresses are constant-time, thus preventing cache side-channel attacks.
- We implement an instruction to switch between constant-time mode and normal mode. This prevents timing attacks exploiting operand-dependent latencies.
- We develop a RISC-V simulator which models the aforementioned addition to the ISA. The simulator generates an abstract leakage trace that is independent of the micro-architecture. Our claim is that this abstraction is sufficient to ensure the absence of timing leakage at the cycle level.
- We implement the proposed solution by extending the CV32E40P RISC-V core. It includes the support of both `lock` and `unlock` instructions and a constant time mode. Furthermore, we develop a data L1 cache hardware design that supports the proposed locking mechanism. We synthesize this core and the associate memory hierarchy on a Xilinx Kintex-7 FPGA.

The rest of the paper is organized as follows. Section 2 describes our threat model and the hypotheses we make about our execution platform. Section 3 gives a high-level overview of our approach, using a few motivating examples to illustrate the need for our new cache locking mechanism.

Then, Section 4 describes in more detail the design of our software simulator, and Section 5 presents the hardware implementation of our solution. Section 6 discusses the limitations of our approach, gives perspectives for further work, and concludes.

2. Threat Model and hypotheses

We consider a platform on which two programs are running concurrently on a single-thread in-order core without speculation. The memory hierarchy is physically addressed and there is one level of data-cache.

One of the programs is the victim program, which manipulates secrets. Another program is a malicious program, under the control of an attacker, and attempts to discover the victim's secrets. We also suppose that the binary of the victim program is known to the attacker and read-only (i.e., we do not consider self-modifying code).

The attacker and the victim programs share the same memory hierarchy, but their address spaces are disjoint. The attacker can perform memory accesses; infer the cache state before and after the victim execution and learn how many cache lines are evicted for each cache set (i.e. cache lines used by the victim). The attacker can measure time in a cycle-accurate manner to determine whether its cache access is a cache hit or a cache miss. The attacker can also measure the time of the victim execution to infer information about the victim execution (e.g. control flow, multi-cycle instruction, memory access). Interrupts can be triggered at any time by the attacker in order to modify the state of hardware shared resources (e.g. cache state) or start/stop cycle accurate timers.

3. Motivating Examples and Approach

Our solution improves the security guarantees of PLcache [12]. We first recall their solution, show their limitations and how we propose to fix them.

PLcache introduces a mechanism to lock and unlock a memory block in the cache. A memory block locked into the cache by the victim program cannot be evicted by the attacker program, but can be evicted by the victim program itself while performing another lock.

Lock and unlock instructions are restricted by the same memory protection as load and store instructions. I.e. a program cannot lock or unlock an address out of its memory space.

In the event of a cache miss, the cache line to evict is chosen regardless of whether it is locked. If the cache line chosen for eviction happens to be locked by another process, it is not evicted, and the metadata associated to this cache line for the replacement policy (in this case LRU (*Least Recently Used*)) is updated as if it had been used, and the memory block that was fetched from memory does not overwrite the locked cache line. Symmetrically, a store would be applied directly on RAM, i.e. the cache is bypassed.

3.1. Forcing the victim to self-evict

A first attack against PLcache takes advantage of the fact that the victim can accidentally evict its own locked data from the cache when performing new locks. The attack is illustrated in Figure 1.

```
lock A1

lock A3
```

(a) Victim code.

```

load A2

```

(b) Attacker code.

Figure 1: First attack: forcing self-eviction. A1, A2, and A3 are mapped to the same cache set.

The left-hand side of this figure shows the victim code, which locks two memory blocks at addresses A1 and A3, while the right-hand side shows the attacker code, that runs concurrently with the victim program, and loads the memory block at address A2 between the two locks. The three addresses A1, A2 and A3 need to be such that they all map to the same cache set. For the sake of brevity, these example programs make the assumption that each cache set consists of only 2 cache ways.

Initially, the cache set in which all three addresses A1, A2 and A3 will be mapped is empty. First, the victim locks memory address A1. Then, the attacker loads memory address A2 into the cache. At this point, the cache set is full. According to the LRU cache line replacement policy, the next line to evict is the one containing memory block A1. The `lock A3` instruction will therefore evict memory block A1.

In the rest of the victim program, memory accesses that should have been protected by the lock instruction are now visible to the attacker, and the usual cache-based timing attacks become possible again.

Our solution. In order to prevent this type of attack, we make the following modifications. Alike PLcache, The lock instruction fetches data from memory and locks it into the cache. Unlike PLcache, locked data stays in the cache until it is released with the unlock instruction, i.e. a process cannot accidentally evict its own locked data. This means that in our proposal, a memory access to a locked piece of data will always result in a cache hit, and therefore be constant-time. We also make the choice to always keep at least one way available in each cache set, so that we never need to bypass the cache. Because of our design choices, the lock instruction may now fail. When a lock instruction would lock the last available way in a set, an exception is raised that the operating system can then handle.

3.2. Leakage through LRU

This second attack shows how information can leak through the metadata associated with the cache line replacement policy.

```
1  lock A1, B1
2
3  load s ? A1 : B1
4  unlock A1, B1
5
6
```

(a) Victim.

```
1
2  load A2
3
4
5  load A3
6  load A2
```

(b) Attacker.

Figure 2: Second attack: leaking information through LRU. A1, A2 and A3 are mapped to the same cache set. B1 is mapped to a different cache set.

Figure 2 shows the code for the victim and attacker programs. In this example, we also consider that cache sets consist of 2 ways, that addresses A1, A2 and A3 map to the same cache set and B maps to a different cache set.

Line 1, the victim locks memory blocks at addresses A1 and B1. Line 2, the attacker fills the cache set containing A1 such that the other way in this cache set is now filled with A2. At this point, the LRU state for this cache set indicates that the least recently used way is the one containing memory block A1.

Line 3, the victim accesses memory block at address A1 or B1, depending on a secret value s. Because both memory blocks are locked in cache, the access is guaranteed to be a cache hit, and the attacker should not be able to learn which address was accessed. However, the LRU state of the cache set containing A1 and A2 now depends on the secret. If s is true, then the least recently used way is the one containing A2. Otherwise, it is the one containing A1.

Line 4, the memory blocks at addresses A1 and B1 are unlocked. Line 5, memory block A3 is loaded into memory instead of the least recently used block. This is guaranteed to be a cache miss, and is thus constant-time. Now the cache contains A1 and A3 if s is true, and A2 and A3 otherwise. The next access at line 6, for memory block at address A2 will now be either a cache hit or miss, depending on the secret. By timing this last access, the attacker can learn the value of s.

Our solution. In order to prevent this attack, accesses on locked lines in our proposal will not update the metadata associated with these lines related to eviction policies (e.g. LRU).

3.3. Non constant-time arithmetic operations

The division and modulo instructions are the costliest integer arithmetic operations performed by our processor. In addition to being costly, their execution time (expressed in number of cycles) depends on the value of the divisor.

Solution. We use an execution mode in which all instructions execute in constant time, similar to the Data Operand Independent Timing mode of Intel [7].

This mode can be enabled and disabled through the use of a dedicated instruction. More details are given in Section 5, which describes the implementation of this so-called "constant-time" mode.

4. Leakage Simulator

Our design ensures that locked data are immune to cache side-channel attacks. However, locking is only a security mechanism. Our high-level security objective is to ensure that no secret information can be obtained through the timing channel if this mechanism is correctly used. As this is a non-trivial task, we provide a RISC-V leakage simulator which allows checking on actual executions that the observations of an attacker do not depend on secret information.

4.1. Abstract Leakage

Our leakage model is inspired by Barthe et al. [2] which model the cryptographic constant-time property and prove how it can be preserved by compiler passes [1]. In our case, the leakage model is meant to ensure that attackers scheduled on the same processors cannot extract more secret information by observing the timing of the micro-architecture than by observing the abstract leakage trace.

In addition to updating the micro-architectural state, each RISC-V instruction emits an event $e \in \mathit{Event}$:

$$e ::= \bullet \mid \mathit{jmp}\ pc \mid \mathit{lock}\ a \mid \mathit{unlock}\ a \mid \mathit{rw}\ a \mid \mathit{div}\ n$$

where pc is a program counter, a is an address and n a 32-bit integer.

We assume that the attacker can always know the current program counter. The rationale is that this information can be inferred because an instruction present in the instruction cache would be executed faster and incur a timing channel. Note that this happens even in the absence of branch prediction. As a result, each branching instruction emits $\mathit{jmp}\ pc$ where pc is the target of the branch. Every constant-time instruction (i.e. which executes in a fixed number of cycles whatever their arguments) emits the event $\bullet$. In that case, the attacker only learns that an instruction elapsed. The event does not distinguish between ALU instructions because this information can be obtained from the program counter that is already leaked by jump instructions. For our micro-architecture, division and modulo are the only non-constant time ALU operation. By looking at the implementation, the timing behavior only depends on the range of the divisor. As a result, an unprotected division emits the event $div\ n$ where n is the value of the divisor. Yet, if the constant-time mode is enabled, the hardware enforces the worst-case number of cycles of the division, and therefore we emit the $\bullet$ event.

To protect against cache side-channel attacks, the address of unprotected memory accesses are also leaked. Yet, we make a distinction between `lock`, `unlock` and other load and store instructions. Every unprotected load or store instruction emits the event $rw\ a$ where a is the address that is accessed. As the program counter is assumed to be known to the attacker, the emitted event does not distinguish between a read or a write access. Because a lock instruction is similar to a load, an attacker could mount a cache side-channel attack to obtain information about the range of locked addresses. Therefore, the lock instruction makes the worst case assumption and emits a $lock\ a$ event where a is the locked address. For the same reason, the unlock instruction also emits a $unlock\ a$. For load and store instructions over addresses that are locked in cache, we only emit the $\bullet$ event. The reason is that the locking mechanism ensures that the data is necessarily in cache and a cache hit is constant-time. Moreover, the access does not update the cache state e.g. the eviction policy, and therefore no information may leak even after an address is unlocked.

4.2. Attacker Trace

Our abstract traces are meant to over-approximate what an attacker could learn by observing the concrete timing behavior of an execution. To validate this assumption, we also have lower-level traces which are only relevant for an explicit attacker. In our model, an attacker is a pair made of some arbitrary code running in a distinct memory space and an attacker controlled scheduling policy. The so-called hardware events $e \in HEvent$ are given below:

$$e ::= \bullet \mid jmp\ pc \mid rw\ b \mid div\ n$$

where b is a boolean and n is a number of cycles.

Similarly to the abstract trace, the current program point is still leaked, and jump instructions emit a $jmp\ pc$ event. Constant-time operations still leak the $\bullet$ event. For the division, we leak the exact number of cycles. For memory load and store, the attacker cannot directly learn the accessed address. However, through timing channels, it may learn whether this is a cache hit or a cache miss. As a result, each lock, unlock, load and store emits a $rw\ true$ if this is a cache hit and $rw\ false$ if it is a cache miss.

4.3. Constant-time Claim

The constant-time property is expressed as a non-interference property, based on the following definition of timing leakage.

Definition 1 (Clock cycle leakage)**.** *The clock cycle leakage of executing a program p on a secret input s, together with an attacker program a, according to a scheduling policy sp, is expressed as the sequence of integers representing the number of clock cycles between each context switch.*

Definition 2 (Cycle-accurate constant-time)**.** *A program p is cycle-accurate constant-time for two distinct secrets s_1 and s_2, if for any attacker program a and any scheduling policy sp, the clock cycle leakage of executing p with secret s_1 and the clock cycle leakage of executing p with secret s_2 is the same.*

Our claim is that the equality of abstract leakage for any possible inputs is a sufficient condition to ensure cycle-accurate constant-time.

Claim 1. *Suppose that executing a program p with two distinct secrets s_1 and s_2 generates the same abstract leakage trace. We have that p is cycle-accurate constant-time for s_1 and s_2.*

The arguments to back this claim is that abstract leakage traces contain more information that any attacker trace, which provides indirect information about memory cache state. Indeed, whereas abstract traces leak the exact addresses that are accessed unprotected, an attacker only observes a cache hit or cache miss. For lock protected addresses, our implementation of the lock instruction ensures that the subsequent memory accesses are necessarily a cache hit i.e. event $rw\ true$. Moreover, as the eviction policy i.e. the LRU tag, does not depend on locked addresses, the attacker cannot learn information about the past accesses even after the address is unlocked.

There is still a gap between the attacker trace and the cycle-accurate timing of the execution. However, for our micro-architecture, we are confident that the attacker trace leaks enough information to reconstruct the exact timing behavior. A subtle point are the hazards of the pipeline that may introduce timing delays. However, as the traces leak the program counter, the hazards of the pipeline can be inferred because they only depend on the sequence of executed instructions.

4.4. Security Evaluation

This section reports preliminary security evaluations of our leakage simulator. The simulator takes as input a RISC-V ELF-32 file and command line inputs; runs the program and outputs abstract leakage traces. It can also be parameterized with an attacker and a scheduling policy to check that the equality of abstract traces indeed implies the equality of concrete traces.

4.4.1. Security evaluation of LRU eviction policies. In Section 3, we have shown that the locking mechanism is vulnerable to a cache side-channel attack if the metadata needed by the eviction policy is updated. To validate our leakage simulator, we have replayed the attack for a vulnerable lock implementation, which updates the LRU tag. In that case, we observe that the concrete attacker traces vary depending on whether the accessed address is either A_1 or B_1. As the abstract traces are not identical, this vulnerable implementation of the lock violates Claim 1. For the fixed lock implementation which freezes the eviction policy for locked addresses, the traces are the

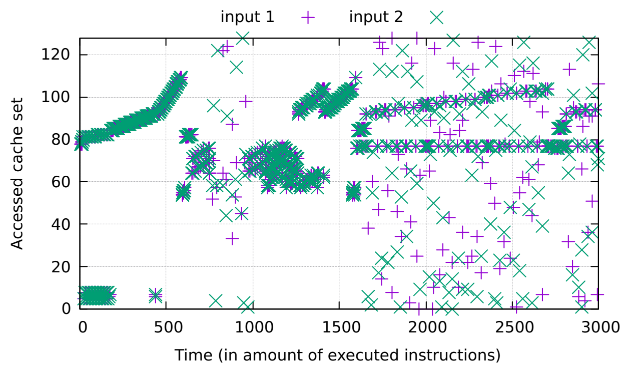

Figure 3: Memory accesses of unprotected Camellia.

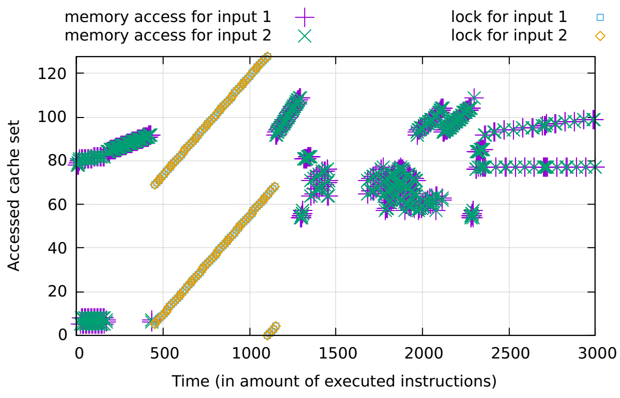

Figure 4: Memory accesses of protected Camellia.

same; the vulnerability vanishes and Claim 1 holds for this example.

4.4.2. Lock protected S-boxes. Unprotected implementations of symmetric block cipher encryption algorithms such as AES or Camellia are vulnerable to cache side-channel attacks. The reason is that the S-boxes are accessed using indexes that are key-dependent.

In Figure 3, we show a graphical representation of the abstract leakage trace of two runs of Camellia[1] for two different keys. We display only the memory accesses performed during the first 3000 executed instructions. Memory accesses done during the execution with the first secret input are shown with a $+$ and those done during the execution with the second secret input are shown with a $\times$.

We can categorize memory accesses in two groups: those where the cache set accessed is the same for both inputs (where $\times$ and $+$ are superposed), and those where the accessed cache sets change depending on the input (disjoint $\times$ and $+$). This second category causes the vulnerability to cache side-channel attacks.

We added our lock to the S-box of Camellia and run it again twice with the same two inputs, as displayed in Figure 4. Memory accesses performed on locked addresses do not appear anymore, since they have no observable effects. There is now only superposed $\times$ and $+$ because secret-dependent accesses are all done on the S-box once it is fully locked. Locks are always done in the same order regardless of the input, since they are also visible to a potential attacker.

We have identical abstract traces for both inputs once the S-box is locked. We can deduce that, with a correct implementation of the lock (i.e. no LRU alteration when accessing locked lines), an attacker observing one of those executions would be unable to distinguish which secret input was used. For this example, this validates that the addresses of the S-boxes are correctly locked. Moreover, under Claim 1, the protected implementation would not be vulnerable to timing attacks.

1. Experimentation done with Camellia implementation of https://github.com/jkivilin/camellia-simd-aesni/tree/master/camellia-BSD-1.2.0

5. Hardware implementation

The proposed implementation is built around a RISC-V processor. We choose the 4-stage in-order 32-bit CV32E40P core that implements the RV32I ISA extended with the standards M (multiplication, division) and C (compressed). The memory is physically addressed on 22 bits, allowing to access 4 MiB of memory (program and data).

5.1. Constant-time operators

In order to make constant-time the instructions whose execution time depends on operand values, we extend hardware operators to support a constant-time mode. When enabled, this mode leads to the execution of extra dummy states to force the worst execution time. In the considered CV32E40P core, both division and modulo operations latency are data-dependent, and require between 3 and 35 cycles. Thus, when the constant-time mode is enabled, the operation latency is forced to 35 cycles. In a concern of performance, the proposed constant-time mode can be software-driven through a Control & Status Register (CSR). In the proposed implementation, the CSR address `0xD00` is used.

5.2. Lock and Unlock

We implement an 8 KiB, 4-way set-associative, L1 data cache as shown in Figure 5. Each cache line is 16-byte long. This cache memory architecture is extended to implement both locking and unlocking mechanisms, as described in Section 3. Since $N-1$ ways can be locked in each set, the proposed cache memory configuration allows to lock a total of 6,144 bytes (3 ways on 128 sets with cache lines of 16 bytes).

Figure 6 presents the Lock handling procedure for our solution. First, LRU eviction policy assigns a way if a cache miss occurs. When a lock is requested, we first determine if the cache memory state permits to lock the selected way. For that purpose, we check whether the selected way is not the last non-locked way of the set. If so, the selected way is locked (i.e. removed from the LRU candidates for eviction) and the LRU updates the remaining non-locked ways. Otherwise, an exception is raised, since we do not allow an entire set to be locked.

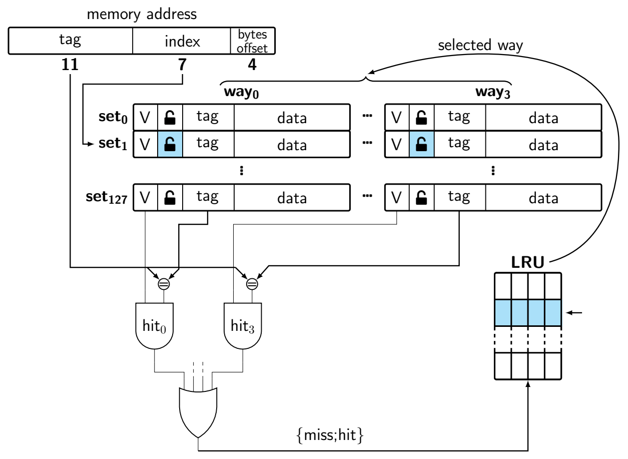

Figure 5: Proposed 4-way set associative cache architecture.

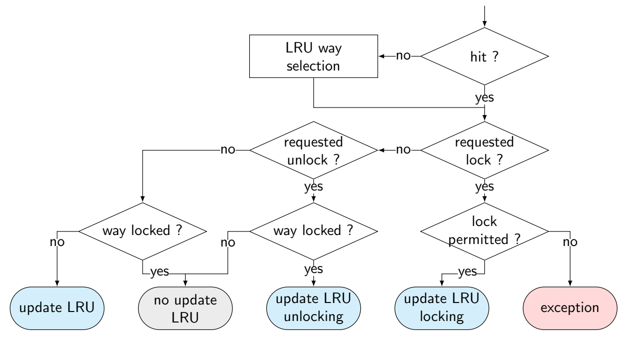

Figure 6: Lock handling procedure.

When an unlock is requested on a locked way, the way is reintegrated within the LRU candidates for eviction. Thus, the way can be selected by the LRU policy. If an unlock occurs on a non-locked way, the state of the LRU is not updated, since a non-locked way cannot be unlocked. It is worth noting that the LRU is not updated when a cache hit is obtained on locked way. Indeed, a locked way is not a LRU candidate for eviction. Finally, when a cache hit occurs on a non-locked way, the LRU is updated for each non-locked ways in the set.

TABLE 1: Post-synthesis area results on Kintex-7 FPGA

	BRAMs	LUTs	FFs
CV32E40P	-	4950	2142
Core+cache (w/o lock)	8	8660	4322
Core+cache (w/ lock)	8	11999	5207

We implemented the proposed solution targeting a Xilinx Kintex-7 FPGA. Table 1 presents the post-synthesis area results. The proposed implementation leads to an area overhead around 39% in terms of LUT and 20 % in terms of Flip-Flops. It is worth noting that these results are preliminary, since the implementation has not been optimized yet.

6. Conclusion and perspectives

We have proposed an extension of a RISC-V processor with instructions to have a fine-grained control over the data stored in cache, and hence a predictable behavior with respect to the cache. We also introduced a “constant-time” mode where all ALU instructions execute in the same number of cycles, regardless of the operands values. We evaluate our proposal both on an executable simulator and on a hardware implementation. Our results show that we can achieve better security than the state of the art (i.e. Wang et al. [12]).

As future work, we plan to evaluate the performance of our proposal. We hope that we can achieve better performance than typical constant-time implementations. On the hardware side, we need to fine-tune the hardware parameters in order to find the best set-up for the cache (number of sets and ways) for a large spectrum of applications, and reduce the hardware cost of the implementation.

We plan to generalize our lock design to multi-level caches, thus allowing to lock larger ranges of addresses at the cost of a slower access time. This requires the implementation of additional precautions for ensuring the secure interactions between levels of cache.

We also intend to prove the claims we did in Section 4, i.e. that reasoning over the abstract traces allows to establish non-interference properties. We also want to investigate whether the program counter can be hidden from the attacker, by protecting the instruction cache, and reasoning about the possible timing leakage associated with the pipeline.

Acknowledgements

This work is supported by the SCRYPT project (grant ANR-18-CE25-0014) and the SCRATCHS project of the Cominlabs excellence laboratory (grant ANR-10-LABX-07-01) funded by the French National Research Agency.

References

[1] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. In *Proc. ACM Program. Lang.*, volume 4, pages 7:1–7:30, 2020.

[2] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic ”constant-time”. In *CSF*, pages 328–343. IEEE Computer Society, 2018.

[3] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *Journal Comput. Secur.*, 27:137–163, 2019.

[4] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 2012.

[5] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2018.

[6] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal ACM*, 43:431–473, 1996.

[7] Intel. *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance*.

[8] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. Winter is here! a decade of cache-based side-channel attacks, detection & mitigation for RSA. *Information Systems*, 92:101524, 2020.

[9] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proc. The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[10] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *Proc. IEEE Symposium on Security and Privacy (SP)*, 2021.

[11] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proc. International Symposium on Microarchitecture (MICRO)*, 2018.

[12] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.

[13] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *Proc. 28th USENIX Security Symposium (USENIX Security)*, 2019.

[14] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *Proc. IEEEEuropean Symposium on Security and Privacy (EuroS&P)*, 2021.

[15] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *Proc. 23th USENIX Security Symposium (USENIX Security)*, 2014.

[16] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *Proc. Network and Distributed SystemSecurity Symposium (NDSS)*, 2019.