Under the dome: preventing hardware timing information leakage

Mathieu Escouteloup[1], Ronan Lashermes[1], Jacques Fournier[2], and Jean-Louis Lanet[1]

[1] Inria, Univ Rennes, CNRS, IRISA
[2] Univ. Grenoble Alpes, CEA Leti, DSYS/LSOSP

Abstract. Numerous timing side-channels attacks have been proposed in the recent years, showing that all shared states inside the microarchitecture are potential threats. Previous works have dealt with this problem by considering those "shared states" separately and not by looking at the system as a whole.
In this paper, instead of reconsidering the problematic shared resources one by one, we lay out generic guidelines to design complete cores immune to microarchitectural timing information leakage. Two implementations are described using the RISC-V ISA with a simple extension. The cores are evaluated with respect to performances, area and security, with a new open-source benchmark assessing timing leakages.
We show that with this "generic" approach, designing secure cores even with complex features such as simultaneous multithreading is possible. We discuss about the trade-offs that need to be done in that respect regarding the microarchitecture design.

1 Introduction

Since Spectre [18] and Meltdown [20] attacks were published in 2018, the microarchitecture security is under scrutiny. Numerous attacks have now been demonstrated [4, 10, 22, 24, 35] targeting the whole microarchitecture to extract information from timing variations. These weaknesses in the design allow extracting information across different security domains: a userland application can read in kernel memory, a virtual machine (VM) can gain information on another VM, *etc.* Unfortunately, on the software side, efficient countermeasures are lacking, and radical solutions have been forcefully implemented. For example, in 2018, the OpenBSD operating system (OS) decided [15] to disable Intel Hyper-Threading (Intels' simultaneous multithreading (SMT) technology) to avoid information leakage between hardware threads (also called harts), an expensive approach that cannot be reproduced for all hardware mechanisms: disabling Intel HT leads to performance losses of up to 20% [19].

Motivation Solutions have been proposed in the literature [7, 12, 16, 17, 25, 32, 34], but they focus on some microarchitectural components in isolation. In this paper, we outline new generic design rules based on first principles to prevent timing information leakage, and build whole cores immune to them. In particular, we explore the instruction set architecture (ISA) modifications that can help build secure designs for all cores, from simple microcontrollers to complex microprocessors.

Contributions In this paper we propose and implement a process to build cores without microarchitectural timing leakage. After analysing the attacks in the literature (section 2), we extract the design rules that must be followed for leakage-free implementations (section 3). We propose an ISA modification to enable circumventing timing information leakage (section 4). We implement two cores with cache memories, branch prediction or SMT, free of timing leakage (section 5). We propose a security benchmark suite, *timesecbench*, that evaluate the timing leakage with respect to several microarchitectural components (section 6). Our security and performance evaluations highlight the trade-offs required to design leakage-free cores (section 6).

2 The need to redefine the microarchitecture for security

Sharing is one of the basic principles used in modern cores for achieving high performances, *e.g.* cache memories shared between cores or branch prediction information between programs. But sharing leaks timing information between users of the same resources, leaks which can be exploited by attackers.

2.1 Threats

Threat model In this paper, we consider the covert channels scenario where shared resources are used to exchange information. The attacker controls both the trojan application, sending information through timing dependencies and the spy application, reading the information. The applications are supposed to be located in different security domains. A security domain is delimited by a unique security policy: different policies define corresponding domains. Resisting to this threat implies that the system can thwart a side-channel scenario, where the attacker only controls the spy and where the trojan only leaks information unwillingly. We are interested in microarchitectural timing leakage, therefore the timing information read by the spy must only be coming from the microarchitectural state of the core when the spy is executing. Thus, all trojans that functionally leak information, by writing to memory or with a time-dependent function, are out of scope in our paper.

Our threat model is the following: the attacker wins if she is able to transmit information from the trojan to the spy. But she cannot use any architectural means of communication (architectural features are the ones exposed by the ISA). The spy cannot measure the trojan execution time (time measurements are architectural functionalities).

Shared resources attacks Different kinds of sharing have been shown to be sources of timing information leakages in modern processor architectures. Timing variations due to cache memories [6] have been known for many years, with multiple variants in numerous implementations [13]. Similar results have been achieved on different resources like branch prediction tables or translation lookaside buffer (TLB).

The resource usage itself represents an interesting information that can be recovered by measuring resource contention. Different works [29, 3, 5, 4] have shown the possibility to recover information from processors with SMT support. The same kind of observation is also possible with mechanisms like cache controllers shared between cores.

In 2018, timing leakage attacks have reached a new level of complexity with transient attacks, adding the use of hardware techniques like speculation or out-of-order speculation. In Spectre [18] or Meltdown [20] and their variants [10, 28, 9] shared resources are used to leak information. It was an important lesson for designers: even if ignored during many years, timing leakages are still present in all modern systems.

2.2 Related work

Since the publication of the first shared resource exploitation, new countermeasures are regularly proposed.

Hardware solutions modify the microarchitecture to ensure the security. Simply removing the problematic mechanisms is not realistic from a performance point of view [8, 19]. Then, another approach is to design shared resources differently. Some solutions [11, 17] try to partition cache memories among the users: each of them has now only access to its own data. It can be spatial partitioning, where the cache is split between the different users, and/or temporal partitioning by flushing the data at the end of a user's execution. Another approach is to remove the deterministic behaviour by introducing some randomization [26, 21]. In this case, if timing leakages still exist, they do not depend on confidential information. Both approaches have been known for many years [30], and can also be combined to enforce the isolation [12]. Finally, proposals also concern other mechanisms like speculation [16, 34] or port contention [25].

Software solutions are also studied, where the application has to directly consider the microarchitecture. Retpoline [27] tries to protect against branch target injection used by Spectre [18] by influencing and redirecting speculation when it is needed. Existing primitives for microarchitecture management can also be used in some cases. `lfence` instruction exists in some x86 implementations [1] to block branch prediction. Other primitives like `clflush` also exist to manage cache structures.

If both pure hardware or software approaches have interesting properties, they also suffer from significant disadvantages. With pure hardware solutions, the software does not have to consider security issues on the hardware side. However, no flexibility on the applied constraints is possible, harming the performances. Conversely with pure software solutions, the application must perfectly know the microarchitecture to protect itself from attacks on the hardware side, harming its portability. More importantly, it also needs a way to manage all the different mechanisms with dedicated primitives. It therefore leads to study the role of the ISA.

The **ISA** is the interface between the hardware and the software. It creates an abstraction of the hardware for the software. Here again, two strategies have been explored to modify the architecture for security purposes. Regarding the previous software solutions, a first one is to break this abstraction role to allow a better microarchitecture management from the software. This functional approach [14, 33] focuses on designing a complete *augmented* ISA where hardware shared features must be directly manageable with software. In our opinion, these works all suffer from the same conceptual weakness: they consider the timing problem as a microarchitectural design issue. Instead, the problem lies in the limited ISA semantics regarding security notions: the issue cannot be solved only by flushing microarchitectural elements, at the risk of forbidding multithreaded or multicore processors, which require spatial sharing. Other works have shown the efficiency of a more abstract approach, by allowing an ISA interface to guide the resource management by the hardware. MI6 [7] adds a new `purge` instruction to flush microarchitectural state independently of the implementation. The DAWG [17] proposal offers new registers to the software to parametrize security domains in cache-like structures. In ConTExT [23], a dedicated bit is added to each page entry to indicate if transient execution is possible or not.

However, these solutions are still considering only some specific shared resources and not the problem as a whole. By focusing on microarchitectural elements in isolation, they are still missing the bigger picture: we do not want to add numerous mechanisms to finely control the cache or the speculation behaviour. This path leads to stacking of countermeasures, to complex systems, to poor portability (how to use ConTExT [23] without virtual memory ?) and will severely limit the possibility one day of having formal security guarantees for the software running on such processors. Instead of fine-tuning the microarchitecture, we prefer a formal contract between software and hardware. This leads to our contribution where the fully abstract approach allows a clear organization. The ISA must allow the software to communicate its security properties to the hardware.

3 Design guidelines

Shared resources must be designed by considering security constraints. Secure design guidelines can be crafted to avoid timing leakages by considering the attack models based on known attacks.

3.1 Definitions and goals

We call shared resources all states or elements which can be assigned to different users. A resource is temporally shared when multiple users can request it at different times. A resource is spatially shared when multiple users can request it simultaneously. Spatial and temporal sharing are not exclusive: some resources, particularly caches, can use both. For the rest of this paper, we define a user as a security domain which must be isolated from the other ones.

In any implementation, shared resources are limited in number: this is one of the reasons they are shared. A system with multiple security domains implies that at least one piece of information will inevitably be leaked between them: the availability of the resource. If a resource is used by a security domain, it becomes unavailable for another domain. By construction, this cannot be avoided. Yet it is possible to overcome this difficulty by distinguishing between static and dynamic availability. The dynamic availability is the possibility for a resource to be used at any point in time as long as it is not already requested. Static availability is the possibility for a security domain to lock a resource for a potential future usage. When the security domain locks the resource, we say that it is allocated, in which case it is no longer available but not necessarily "used".To allow correct execution of a security domain, resource allocation must be done during its creation and kept during its whole lifetime.

While dynamic availability leaks information with precise execution timings that can be exploited with port contention attacks [4], static availability does not permit this kind of leakage. In our case, we only allow static availability as information leakage, giving the following security property:

Shared resource security property *The only information that a security domain may extract from a shared resource is the domain's own data or the resource's static availability.*

Then, the different shared resources must be modified to prevent other information leakages. These modifications needed to safely support security domains can be summarized in three main strategies: lock, flush and split.

3.2 Resource availability: lock

Design guideline 1: static allocation *The different minimal resources needed by a security domain must be allocated during the domain creation and locked until its deletion.*

Each shared resource can only support a limited number of security domains, which can be one (only temporal sharing) or more (temporal and spatial sharing).

Static allocation allows having the exclusivity of a resource in order to use it without execution timing leakages. Obviously, it is necessary only in systems where multiple security domains can simultaneously be executed, leading to potential spatial leakages. Allocation is simplified when only one security domain can exist at any time in the whole system: it can simply use all the different resources.

3.3 Temporal resource sharing: flush

The static allocation cannot last forever and the resource must be released eventually to make a place for another security domain. The resource design must ensure that there is no leakage between the security domains, which leads to the following guideline:

Design guideline 2: release *When a security domain ends, all its associated resources must be released only when all persistent states have also been erased.*

We call "persistent states" all information stored in registers or memories whether data, metadata, finite-state machine (FSM) states *etc.* All of them are associated with a security domain for which the associated data must be removed before allowing allocation from another security domain. Different works [33, 7] have shown that flushing resources is efficient to make a temporal isolation barrier. Then, all temporally shared resources must support it.

3.4 Spatial resource sharing: split

In some cases, lock and flush strategies are enough: *e.g.* if there is only temporal sharing. But fully locking a resource in an exclusive way during each execution can be limiting. Some resources need to handle requests from different users simultaneously. In this case, the correct strategy is partitioning, also called split. Such a resource must be able to isolate all users from each other.

Design guideline 3: partitioning *A resource able to handle requests from multiple security domains simultaneously must be able to partition each domain state in its own isolated compartment. States and data cannot be shared.*

In other words, any spatially shared resource must be split between the security domains. It can be seen as resources with multiple lock slot: multiple security domains can simultaneously lock a part of this resource, but without any interaction between them.

Because split is only a form of sharing, it also has both temporal and spatial variants. In a temporal split, the resource is successively available for each user and only seems simultaneously available at a global scale. It is simply a way of transforming a partial simultaneous sharing in a local temporal sharing where lock and flush strategies are applied. With spatial split, the resource is truly simultaneously available for each user at any time. It leads to our last design guideline:

Design guideline 4: availability split *A spatially shared resource must ensure that, at any given time, its availability for any security domain is independent from the domains being served.*

Partitioning can take several forms depending on the targeted resources [11, 12, 17, 30, 25]. To efficiently apply all these strategies, the hardware must finally be informed about the security domain switching.

3.5 Exclusive allocation and heterogeneity

As mentioned previously, static allocation only prevents the detection of a resource usage, not its availability. In the case of heterogeneous systems, this information can be exploited to build a covert channel. We call a system heterogeneous when all the users do not have exactly the same resources. It is a common organization in modern microarchitectures: all the threads or cores are not necessarily equivalent, *e.g.* to satisfy different performances or power constraints. Then, if the trojan allocates some resources and not others, a message can be sent to the spy: the latter can deduce the trojan allocation from measuring its own available resources. In a completely secure system without even covert channels, we can deduce the following guideline:

Design guideline 5: homogeneity *During their execution, all users must be treated equally, by allocating the same resources in types and numbers.*

Obviously, strictly applying this rule can be very restrictive: no flexibility is allowed in the resource allocation. If the natural solution would be to have strictly duplicated cores with their own resources, we will present in the next section a manner to prevent this covert channel while preserving some flexibility.

4 Domes

The security domains can only be defined at the software level through the applications themselves. To enforce the shared resource security property we defined above, the hardware has to be aware of the security domains. Therefore to communicate their boundaries to the hardware, the ISA must be modified. In this section, we present our proposal to modify the RISC-V ISA.

4.1 Fine-grained security domains

In current systems, we can find several implementations of security domains. They are the result of a historical evolution of the security needs due to the evolution of the threats. The mostly used and classic security domains are the privilege levels, notably separating the kernel from the userland.

However, in the case of sharing, these domains are too coarse-grained. An application may want to isolate tasks (e.g. a web server isolating several clients, a web browser sandboxing its tabs) while having only one address space. It justifies the works such as ConTExT [23] where security domains are proposed at page granularity, Time-Secure Cache [26] at process granularity or a completely new security domain notion managed by the software in DAWG[17]. This domain notion must now be used by the hardware to manage all the shared resources.

4.2 Fence or context

Boundaries of the security domains have to be communicated from the software to the hardware. In classical systems, privilege levels are changed with a dedicated mechanism, often with specialized instructions. Similarly, we must define the mechanism that allows switching between fine-grained security domains. Before the precise ISA modifications, we must choose between two possible semantics.

The first possibility is to use stateless switches between domains called **fences**, similarly to the timing fences from Wistoff *et al.* [33]. In this case the boundary between domains is specified by a dedicated `fence.t` instruction that separates the security domains before and after the instruction. Typically, the execution of this fence must ensure that all states associated with the current security domain are flushed out of the microarchitecture. Finally, fences are particularly efficient for creating temporal security domains: each one is delimited by the previous and the next fences. But this approach does not consider spatial sharing: for example the hardware has no information to decide whether two harts are in the same security domain.

The second possibility is the use of **contexts**, a stateful switch. Each microarchitectural resource, state or data is at any time explicitly or implicitly associated with a security domain which constitutes the context. With this information, the resource can be adapted to the execution and may share states (same domains) or isolate them (different domains), both temporally and spatially. The context semantics gives more power to the microarchitecture than fences, but increases the system complexity.

Since we want a global solution able to consider all the different shared resources in the microarchitecture, we choose the context semantics. We call our specific implementation a **dome**. A dome is an execution context that corresponds to one security domain. At any given time, each hart is assigned to a unique dome but several harts can share the same dome. At the microarchitecture level, it defines which resources can be used by each hart: all instructions and microarchitectural states are implicitly or explicitly assigned to the corresponding dome.

4.3 ISA changes for dome support

Adding dome support in a core requires to augment the ISA with new instructions, new registers and the corresponding hardware modifications. This proposal can be seen as an extension over the base RV32I [31]. Our goal here is to analyse the ISA with context support: their role, what is needed and the consequences. The contextualization can be implemented in different ways and only one is described in the rest of this paper. Because our proposal does not use specific features of the RISC-V ISA, the same principles can be exported to other ones like x86, ARM *etc.*

Dome identifier Each dome is represented by a unique number, the dome identifier, stored in a dedicated register `domeid`, one per hart. This register is read-only, since a dome cannot dynamically change its own configuration. `domenextid` is the register that indicates the identifier of the next dome when a context switch occurs. The current dome can write into this register. These registers are considered as new Machine-level control and status registers (CSRs): they are manageable by the same instructions described in the RISC-V ISA [31].

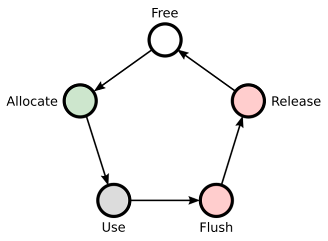

Fig. 1: Resource lifecycle with static allocation.

When our next dome configuration is ready, we need to switch to the new domain with the dedicated instruction `dome.switch`. Each resource has a lifecycle described in Figure 1. Allocated resources have to be flushed and then released, before those needed by the new dome are allocated. At the same time, `domeid` must be updated: it receives the information present in `domenextid`. If we want to free all resources from a security domain, for example before turning the machine off with write-back caches, it is enough to switch to a new domain.

Dome capability Sometimes, there are not enough resources in number to satisfy the needs of each hart. For example, we may think of a system with only one cryptographic accelerator, one floating-point unit, or as in our case one multiply and divide execution unit (MULDIV) execution unit. To deal with this case, we add new registers to store the dome capabilities, specifying if the dome needs access to these few resources: `domecap` and `domenextcap`. Bits are set in these registers if the dome has or need access to some predetermined features (such as RISCV M extension) that map to hardware resources.

Upon a switch, the system will try to lock the resources corresponding to the capabilities of the next dome. Therefore `dome.switch rd` can now fail if the resources asked are not available; in case of success `rd` is set to 0.

4.4 Software implications

In addition to having an impact on the hardware, the ISA also changes the way software must be designed.

Compiler The RISC-V ISA naturally suggests linking a capability bit for each supported extension: because each instruction is already associated to an extension, the compiler knows when a piece of code requires a capability. As a consequence, the compiler can automatically insert the proper instructions for a dome switch (capability and all), apart from the next dome identifier. Indeed, identifying the security domains is part of the application logic.

Dome management In our implementation, domes are managed by the higher level of privilege, the Machine-level. It is responsible for selecting the correct IDs and capabilities for the different domes where are executed the applications. It must also perform the different switches needed. Domes are only tools to allow isolation of the software and, as any tool, they can be used improperly. Software developers have to be aware that these guarantees are offered at dome granularity. Monolithic systems are not going to take full advantages of the dome switching guarantees, while too many dome switches can make static resource allocations similar to dynamic ones. Also, since capabilities are in contradiction with the Design guideline 5, it is the responsibility of this higher privilege-level to ensure that multiple domes are not trying to communicate with resource allocation, *e.g.* abusing `dome.switch`. This can be detected with a failing `dome.switch`.

Spatial sharing in the single hart case The cost of dome switching can be high in some scenarios. For example, in the case of an exception, all the shared resources must be flushed twice. It can be interesting to allow some spatial sharing, even in a single hart case, between an active dome currently being executed and a background dome that will eventually be returned to.

5 Implementation

To demonstrate and validate our design rules, we build several cores with a modular architecture that are evaluated in section 6. We choose the Chisel language to allow a better modularity and configuration management. It becomes particularly easy to compare designs by only modifying some parameters: dome support can be enabled by switching a boolean variable to `true`. Code for our cores and the evaluations are available online: `https://gitlab.inria.fr/mescoute/hsc-eval`.

5.1 Target description

Global view To evaluate dome support in the case of a simple core but also with spatial sharing, two cores have been implemented. The first core, named Aubrac is based on a 5-stage in-order pipeline. The second core, named Salers, is a more complex dual-hart 6-stage in-order pipeline as illustrated in Figure 2. In Salers, the two harts are running simultaneously and can be switched off using custom CSRs: one hart working alone takes all the resources and a classic superscalar execution is achieved.

These two cores are implementations of the open-source RISC-V RV32IM ISA [31], with CSR and `fence.i` support. Both cores have separate first-level write-through cache memories for instructions (L1I) and data (L1D) with branch prediction and basic speculation mechanisms through a branch history table (BHT) and a branch target buffer (BTB). The different modifications, described later in this section, are represented with a dedicated dome unit and with existing modified components bordered with red dotted lines.

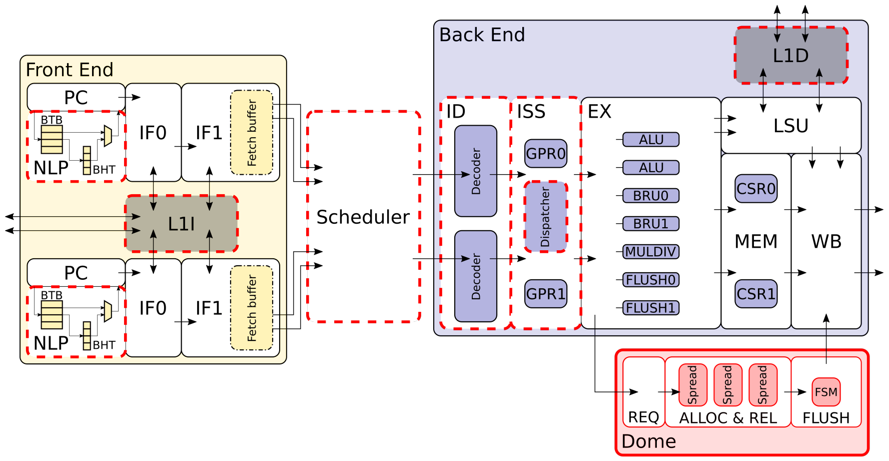

Fig. 2: Global view of the Salers core microarchitecture.

Shared resources These cores are designed to model multiple resource sharing, allowing reproducing a representative sample of attacks from the literature. Several temporally shared resources have to be considered in both cores. The most obvious ones are cache lines in L1I and L1D or the prediction tables. But this also applies to the pipeline, cache controllers or replacement policy registers.

Spatially shared resources are only present in the Salers core, including cache memories and execution units. Since the latter are shared between harts, port contention might occur. Particularly, our MULDIV represents a worst case: operations take many cycles (8 to 32 cycles), timing variations are possible depending on the operation (division or multiplication) and more importantly, there is only one unit for two potential users. When one hart is using this unit, if the other one needs it too, it must wait until the unit is released. This kind of problem is not exclusive to execution units, and is valid for each resource not spatially shared and present in fewer instances than the potential users. For example, port contention could be possible, in our design, with cache memories. They are spatially shared and can securely handle transactions with the pipeline, but contention is possible with the next memory level without the special care described below.

5.2 Aubrac core

In the case of the Aubrac core, only one hart is running at a time: there is no need to support the simultaneous execution of multiple domes. Modifications are simply needed to have dome support (dedicated instruction and CSRs) and to ensure that there is no persistent traces after a dome switch. For that purpose, a dedicated execution unit implements a simplified version of the FSM described in Figure 1. Since only temporal sharing exists here, free and allocate steps are merged: all the resources are always allocated by a dome. The release only occurs when all the resources are empty after the flush cycles.

5.3 Salers core

In the case of the Salers core in the Figure 2, two harts are running simultaneously. We need to ensure dome security properties even with spatial sharing. In addition to the flush strategy, resource allocation with split and lock strategies must also be implemented. We find the different modified components which now also support partitioning and a more complex dome unit, responsible for the resource allocation and release in addition to flush.

Allocation and release During a dome switch, allocation and release must be performed in the case of spatial sharing. To manage each kind of spatially shared resource, a mechanism called spread resource unit has been implemented. It is responsible for associating each resource with a dome depending on the received allocation and release requests. For example, we have one unit dedicated to the arithmetic and logic units (ALUs), another for the MULDIVs, *etc.* They are all implemented in the dome unit in the Figure 2 and are accessed before performing a `dome.switch`.

After the allocation, each resource is tagged with its corresponding dome and has a port number inside this context. Only free resources can be allocated and, to respect as much as possible the Design guideline 5, this allocation is always fixed. Then, independently of the resources available, the same number will always be allocated: only the types can change depending on the capabilities. Finally, when the execution of a dome is ended, it sends a release request to flush and free resources.

The number of implemented spread units depends on the number of spatially shared resources. During a switch, requests to release and allocate resources are sent to the spread units to respect the resource lifecycle. The final result of a `dome.switch` depends on the results of the requests to all spread units.

Spatial sharing Spatial partitioning has been applied for cache memories. It is a well-known mechanism to allow execution of multiple security domains in the memory hierarchy, with multiple variants. In our case, we decided to use soft-partitioning at the way-level. Then, each way is viewed by the corresponding spread unit as a different allocable resource. When a memory request is received by the cache, the dome tag of the request and the one of each way are compared to know if the data can be accessed. It is interesting to note that this is a locking strategy applied locally on each way, leading to a splitting strategy at the scale of the whole cache.

Cache controllers and memory bus to the main memory are other interesting cases in our design, because they cannot be fully duplicated nor fully locked since they are required for all executed domes. A hybrid approach between spatial and temporal sharing is used in the form of fine-grained multithreading. The controllers can be requested only during a fixed cyclic period by each dome, which has an impact on the cache miss operations. The memory bus has also been modified to support dome id transmission: the master controller is responsible for making bus contention transparent.

Based on the previously defined strategies, multiple implementations are possible for the same design: only some possible choices are described in this paper. This is the designer's role to decide where and when which mechanisms are more interesting depending on her constraints: execution units can also be time partitioned, prediction mechanisms partially shared if not fully duplicated *etc.*

6 Evaluation

6.1 Security evaluation

Timesecbench In order to validate the security properties of our approach, we propose Timesecbench: a security benchmark suite that measures timing leakages in various scenarios. It is inspired by the Embench[2] performance benchmark and is fully available online: `https://gitlab.inria.fr/rlasherm/timesecbench`. Obviously, even if the benchmarks can be customized independently of the processor, the microarchitectural mechanisms under test must be implemented.This benchmark suite can be expanded to test other mechanisms or different cores.

Six different attacks are currently available in our security benchmark targeting cache memories, branch prediction mechanisms and execution units. They have been designed with the same following covert channels scenario: a trojan tries to send information to a spy by exploiting timing information leakage due to shared resources. Inspired by the work of Ge *et al.* [14], for each attack we measure a timing associated with the trojan sending a value i (column index) and the spy reading a value j (row index).

We then apply a discrimination criterion (here minimal timing for the spy reading a given value) giving a probability for the spy to read a value j when the value sent by the trojan is i. The benchmarks are executed without any OS, cancelling most noises between tests, allowing a better control of the system and thus reinforcing the power of the attacker. From this joint probability matrix, we can compute the mutual information MI, that gives the amount of information that can be sent through the channel for a uniform distribution at input. The normalized values for our benchmarks are presented in Table 1: it gives the proportion of the trojan information that can be recovered by the spy. For example, in the L1I case, the trojan sends a 3-bit symbol through the channel (choose one set among eight), but the spy can only recover 46% of it (or 1.37 bits per symbol). The channel is closed, *i.e.* our design is secure if the mutual information is zero.

For our security analysis, the benchmarks have been executed on both unprotected and protected versions of Aubrac and Salers cores. In the case of protected designs, trojan and spy are placed in two different domes.

Benchmark results All the benchmark results are shown on Figure 3. The first two benchmarks evaluate timing leakage for both cache memories L1D and L1I on Aubrac. The trojan encodes its value i by accessing the corresponding address, either by loading a value (for L1D) or by executing an instruction (for L1I).

Since a `dome.switch` is performed after the trojan encoding and before the spy decoding, it is able to prevent the timing leakage as illustrated on Figure 3. In the unprotected L1I case, the attack is not perfect as in the L1D case, due to the presence of the benchmark own instructions in the L1I cache. Two benchmarks target the branch prediction mechanisms BTB (for direct jumps) and BHT (for branches) on Aubrac. Here the trojan trains the branch predictor to ensure that only the i-th branch is accelerated by the branch predictor. Dome support is able to remove this timing leakage. The results obtained on the unprotected versions are polluted by the execution of the benchmarks themselves, that do include branches and direct jumps. One benchmark attempts to transmit information across harts on Salers through the L1D cache timing. This is similar to the previous L1D benchmark but trojan and spy are executed on two different harts. Interestingly, the timing depends on the value j read by the spy in this case. This is an overhead due to the fine-grained multithreading technique used by the memory controller. The last benchmark demonstrates that the MULDIV port contention can also be used to encode information. In this case, if we enable dome support, we cannot run this benchmark: the spy hart cannot lock the MULDIV unit, as intended. The unsecured application cannot be run.

Table 1: Normalized mutual information for the 6 benchmarks in Timesecbench with 0 for no measured leakage.

Normalized MI	L1D	L1I	BHT	BTB	cross-L1D	port contention
Unprotected	1.0	0.46	0.38	0.31	0.46	1.0
Protected	0.0	0.0	0.0	0.0	0.0	X

In the scenarios that have been tested, our solution has removed all leakages: timing information leakage cannot occur across security domain boundaries.

6.2 Performances/cost analysis

Dome support involves modifications in the whole microarchitecture. After evaluating its security efficiency, we need to analyse the impact on both performances and area. The different measurements were carried out after performing synthesis and implementation with Vivado 2019.2 (default parameters with *phys_opt_design* enabled), targeting the Xilinx ZCU104 FPGA. Sixteen configurations are compared: we vary the cache size (1 kB or 4 kB), the next-Line Predictor (NLP) support for branch prediction and the dome support, both for Salers and Aubrac. Our goal is to compare both protected and unprotected version of the same cores since performances and area overhead highly depend on the implemented shared resources. Direct comparisons with other works are not relevant: their implemented shared resources are different.

Performances overhead We start by evaluating the overhead in terms of clock cycles to execute the Embench [2] benchmark suite. Considering that we do not

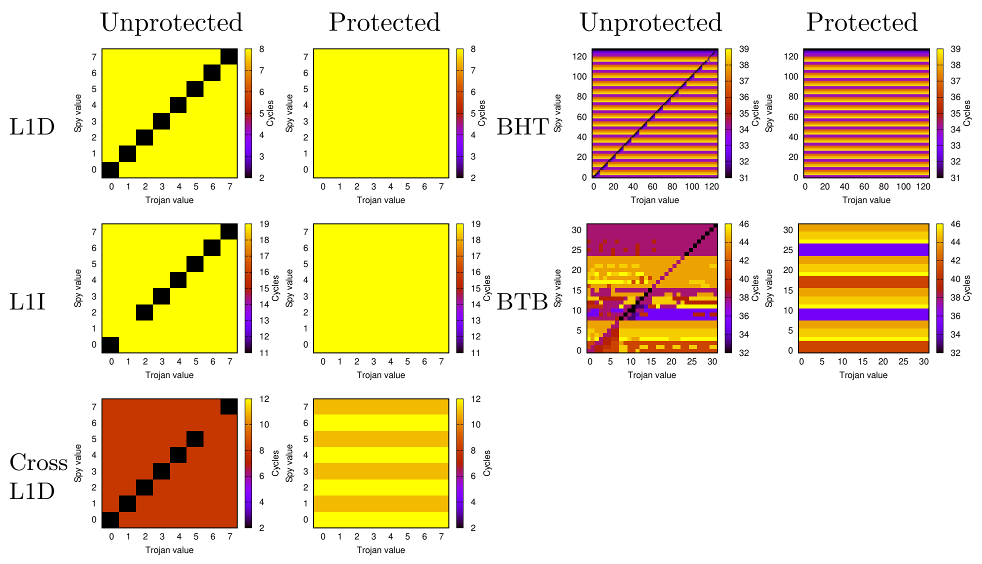

Fig. 3: Timesecbench timing matrices: horizontal variability denotes a timing leakage.

modify the critical path of our design, it is important to note that the clock frequency is not impacted in our designs. The geometric means for the different configurations are shown in Table 2. When comparing Aubrac and Salers, all the Embench benchmarks but two are taken into account: aha-mont64 gives an erroneous output and nbody is too slow and hits the simulation timeout (it involves floating point arithmetic). The single versus dual hart comparison is performed on Salers by taking into account three benchmarks (nettle-sha256, nsichneu and slre) adapted to a multithreaded core.

Table 2: Embench normalized timing geometric means, lower is better. Normalized with respect to the Aubrac-1kB implementation.

Cache size	1 kB	4 kB	1 kB	4 kB
	Aubrac		Salers (1 hart)	
	1.00	0.86	0.95	0.90
Dome	1.00	0.86	1.16	0.92
NLP	0.92	0.78	0.95	0.81
NLP Dome	0.92	0.78	1.07	0.82

Cache size	1 kB	4 kB	1 kB	4 kB
Salers	Single hart		Dual hart	
	1.05	0.91	0.56	0.51
Dome	1.07	0.88	0.76	0.62
NLP	1.03	0.88	0.55	0.51
NLP Dome	1.05	0.86	0.75	0.61

From Table 2, we can see that dome support has no timing overhead for Aubrac, as expected since the benchmarks run in the same security context. Yet in the Salers case, dome support can really slow down the computation due to cache partitioning as the cache size is in effect divided by two. In the worst case, we can observe a loss of 20% on the runtime when L1 caches are split without speculation. The cache influence is confirmed as multiplying the cache size by four reduces the overhead by a factor of 2 (10%).

Table 3: Total timings (cycles) for the security benchmarks

	Unprotected	Protected	Overhead%	overhead (cycles) per `dome.switch`
L1D	27256	35880	+32%	67.4
L1I	57416	64264	+12%	53.5
BHT	1756202	1796774	+2%	19.8
BTB	445544	463443	+4%	35.0
Cross-L1D	188026	134395	−29%	X
Port contention	59250	X	X	X

But our protection is meant to be used, we must therefore evaluate the overhead due to switching domes. Obviously, since `dome. switch` performs a microarchitectural flush, performances are impacted at the beginning of the new dome execution: no data is stored in cache memories and we have cache and prediction misses all the time. This can be seen on Figure 3, where for example in the L1D case we measure in the protected case a timing of 8 cycles, corresponding to a miss, in all cases. Therefore, the cost of `dome.switch` is composed of both the time for the flush and misses due to this operation. As shown in Table 3, in these heavily domain switching scenarios, the average timing overhead is 68 clock cycles per switch for L1D and 54 for L1I.

We see that the total execution of the benchmark is significantly reduced (−29%) with dome support with respect to the unprotected case for the Cross-L1D benchmark. Upon investigation, this performance improvement is due to the better isolation between caches: one hart having a cache operation does not slow down the other hart. The performance cost of having an effective cache size divided by two is low in this case due to the extremely small program executed for this evaluation. In our designs, flushing can be done in few cycles. The cost of dome switching is mostly due to the penalty of increased misses in the microarchitectural buffers. But this is not a universal rule: for example if write-through caches allow efficient flushes, the story is different for write-back caches. For these latter caches, upon a flush the data must be written back to the upper memory level, which cannot be done rapidly.

Hence, the only parameter to modify the switch duration is the microarchitectural flush methodology. This criterion is highly dependent on the other implementation choices.

Area overhead Area results for each core are presented in Table 4. Lookup tables (LUTs) are necessary for the combinatorial logic whereas flip-flops (FFs) are memory elements for storing states in the microarchitecture.

Table 4: FPGA resource utilization.

Cache size	1 kB	4 kB	1 kB	4 kB
	Aubrac		Salers	
	$9,370$ LUTs	$\times 2.62$	$21,000$ LUTs	$\times 2.24$
	$4,408$ FFs	$\times 1.90$	$8,270$ FFs	$\times 1.65$
Dome	$\times 1.03$	$\times 2.62$	$\times 1.09$	$\times 2.14$
	$\times 1.07$	$\times 1.97$	$\times 1.32$	$\times 1.99$
NLP	$\times 1.30$	$\times 2.93$	$\times 1.27$	$\times 2.58$
	$\times 1.19$	$\times 2.10$	$\times 1.21$	$\times 1.88$
NLP Dome	$\times 1.30$	$\times 2.88$	$\times 1.34$	$\times 2.67$
	$\times 1.26$	$\times 2.17$	$\times 1.52$	$\times 2.20$

Finally, the area cost to mitigate security issues due to temporal sharing is only a few percents (between +3% and +0% of LUTs, in the same ballpark as `fence.t`'s +1%[33]). The FFs increase in Aubrac core is more important (up to 7%) but must be qualified: it is mainly due to the addition of several CSRs in a small core. Moreover, considering that a `switch` with only temporal sharing simply performs a flush, these additional registers are not essential in this case. On the other hand, the impact is much more important when spatial sharing has to be considered because of its complexity. For the Salers core, we have a significant impact of up to 32% in the number of flip-flops. It is mainly due to new CSRs for both harts, a more complex dome unit with associated states for each resource and cache controllers with temporal split. In this case, our results are difficult to compare with other works on secure SMT [25]: we modify all the shared resources and not only the multithreaded execution units. Moreover, this overhead must be put into perspective, as SMT is mainly used in much more complex cores with expensive features like out-of-order execution.

7 Discussion and conclusion

For many years now, timing leakages due to resource sharing have been identified as a major threat to the security of processors. Nevertheless vulnerabilities related to such timing leakages are still being found at an alarming rate. This is mainly because, so far, the proposed mitigation look at specific mechanisms of a processor architecture in isolation and not at the processor as a whole. This paper is a step in this direction.

In our approach, we first describe how the ISA has a crucial role to play in making the software communicate to the hardware the applications' security constraints. Two possible semantics, fences and contexts, are discussed. Fences are simple but limited since they cannot handle spatial sharing. Contexts, on the other hand, allow designing secure systems with a lot of liberty on the core features, at the price of more complexity.

We then introduce generic principles for designing shared resources securely whether it is temporal or spatial sharing and with different granularities : we discuss such things like shared memories (like caches), but also more subtle components such as finite state machines (e.g. cache controllers) and buses (shared between several subsystems).

We demonstrate the application of our new approach by implementing two different processors, including one with simultaneous multithreading. We analyse the impact of this new security dogma on the design of such exemplar processors. We also foresee that taking into account security will profoundly modify the canon of processor design: as an example, write-through caches are much faster to flush than write-back and should supersede the latter one in secure designs.

To evaluate the efficiency of our security approach, we propose a new benchmark that shows that the implemented features circumvent timing leakages. This benchmark tests and detects known vulnerabilities. It will be regularly updated so that it can be used to help designers validate the security of their processors at design time. But it cannot be used to guarantee that no timing leakage is present at all. In our opinion, the formalization of the hardware seems the only approach for future works to allow real exhaustiveness, a feat that can only be achieved with a clear ISA semantics to delimit security domains.

Our research shows that, if our principles can be implemented with the adequate ISA, securely implementing resource sharing within processors is possible. A future work will consist in studying the trade-offs between this security dogma, performances and design complexity (size and power). Processors with simultaneous multithreading will be a relevant use case for this, with deep resources sharing that can lead to important leakages. Particularly, an analysis of the trade-offs must be done to compare with multicore processors while maintaining a high level of security. An exploration of domes impact on out-of-order cores and many core systems must also be considered.

References

1. Managing-Speculation-on-AMD-Processors. Tech. rep., Advanced Micro Devices (2018)
2. Embench: A modern embedded benchmark suite (2020), https://embench.org/
3. Aciicmez, O., Seifert, J.P.: Cheap Hardware Parallelism Implies Cheap Security. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007). pp. 80–91. IEEE, Vienna, Austria (Sep 2007)
4. Aldaya, A.C., Brumley, B.B., ul Hassan, S., Pereida Garcia, C., Tuveri, N.: Port Contention for Fun and Profit. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 870–887. IEEE, San Francisco, CA, USA (May 2019)
5. Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H.: On Subnormal Floating Point and Abnormal Timing. In: IEEE Symposium on Security and Privacy (SP). pp. 623–639. IEEE, San Jose, CA, USA (May 2015)

6. Bernstein, D.J.: Cache-timing attacks on AES. p. 37 (2005)
7. Bourgeat, T., Lebedev, I., Wright, A., Zhang, S., Devadas, S.: Mi6: Secure enclaves in a speculative out-of-order processor. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 42–56 (2019)
8. Bulpin, J.R., Pratt, I.A.: Multiprogramming Performance of the Pentium 4 with Hyper-Threading. In: Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD). p. 10 (2004)
9. Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., Yarom, Y.: Fallout: Leaking data on meltdown-resistant cpus. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 769–784. CCS '19, Association for Computing Machinery, New York, NY, USA (2019)
10. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses. In: 28th USENIX Security Symposium (USENIX Security 19) (Nov 2019)
11. Costan, V., Lebedev, I., Devadas, S.: Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 857–874. USENIX Association, Austin, TX, USA (Aug 2016)
12. Dessouky, G., Frassetto, T., Sadeghi, A.R.: HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association (Sep 2020)
13. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering **8**(1), 1–27 (2018)
14. Ge, Q., Yarom, Y., Heiser, G.: No Security Without Time Protection: We Need a New Hardware-Software Contract. In: Proceedings of the 9th Asia-Pacific Workshop on Systems - APSys '18. pp. 1–9. ACM Press, Jeju Island, Republic of Korea (2018)
15. Kettenis, M.: (Jun 2018), `https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html`
16. Khasawneh, K.N., Koruyeh, E.M., Song, C., Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In: Proceedings of the 56th Annual Design Automation Conference 2019 (DAC16). pp. 1–6. ACM Press, Las Vegas, NV, USA (Jun 2019)
17. Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., Emer, J.: DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 974–987. IEEE, Fukuoka (Oct 2018)
18. Kocher, P., Horn, J., Fogh, A., Genkin, a.D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: 40th IEEE Symposium on Security and Privacy (S&P'19). IEEE Computer Society, Los Alamitos, CA, USA (May 2019)
19. Larabel, M.: Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS - Phoronix (Jun 2018), `https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4`
20. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 973–990. USENIX Association, Baltimore, MD, USA (Aug 2018)

21. Qureshi, M.K.: CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 775–787. Fukuoka (2018)
22. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-Flight Data Load. In: 40th IEEE Symposium on Security and Privacy (S&P'19). p. 18. San Francisco, CA, USA (May 2019)
23. Schwarz, M., Lipp, M., Canella, C., Schilling, R., Kargl, F., Gruss, D.: Context: A generic approach for mitigating spectre. In: Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS20). Internet Society, Reston, VA (2020)
24. Schwarz, M., Lipp, M., Moghimi, D., Bulck, J.V., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 15 (May 2019)
25. Townley, D., Ponomarev, D.: Smt-cop: Defeating side-channel attacks on execution units in smt processors. In: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 43–54 (2019)
26. Trilla, D., Hernandez, C., Abella, J., Cazorla, F.J.: Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In: Proceedings of the 55th Annual Design Automation Conference. pp. 1–6. ACM, San Francisco, CA, USA (Jun 2018)
27. Turner, P.: Retpoline: a software construct for preventing branch-target-injection (Jan 2018), `https://support.google.com/faqs/answer/7625886`
28. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: Proceedings of the 27th USENIX Security Symposium (USENIX Security 18). pp. 991–1008. USENIX Association, Baltimore, MD, USA (Aug 2018)
29. Wang, Z., Lee, R.: Covert and Side Channels Due to Processor Architecture. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC'06). pp. 473–482. IEEE, Miami Beach, FL, USA (Dec 2006)
30. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07. p. 494. ACM Press, San Diego, CA, USA (2007)
31. Waterman, A., Asanovic, K.: The RISC-V Instruction Set Manual, Volume I: User-Level ISA (Dec 2019)
32. Werner, M., Unterluggauer, T., Giner, L., Schwarz, M., Gruss, D., Mangard, S.: SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 675–692. USENIX Association, Santa Clara, CA (2019)
33. Wistoff, N., Schneider, M., Gürkaynak, F.K., Benini, L., Heiser, G.: Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core. CoRR **abs/2005.02193** (2020)
34. Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C., Torrellas, J.: InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 428–441. IEEE, Fukuoka (Oct 2018)
35. Yarom, Y., Falkner, K.: FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 719–732. USENIX Association, San Diego, CA, USA (2014)