UC Santa Cruz

UC Santa Cruz Previously Published Works

Title

Securing Processors from Time Side Channels

Permalink

https://escholarship.org/uc/item/6j02r9pw

Author

Renau, J

Publication Date

2023-12-11

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Securing Processors from Time Side Channels

Jose Renau
UC Santa Cruz

Abstract—SoCs are required to maintain information private when requested by the Operating System (OS) or the application. From a high level point of view, there are time domains, typically processes or threads, and there should be no time information leak between them. At the same time, cores inside SoCs are supposed to be fast levering many predictors and resource sharing for efficiency. Using predictors like caches and branch predictors can have side effects that leak information across time domains. Specifically, the challenge is that the code executed by one time domain affects the performance of another. This time impact information leak can be exploited as a side channel attack.
The goal of this work is to classify the different side channel **time** information leaks that result from different predictors available in typical high performance cores. The work focuses on side channels that result of changes in execution time, not other side channels like Electro-Magnetic Interference. The proposed classification points that time side effects or leaks can be due to program data, address, program counter, or just execution time. Each of those information leaks can happen during the speculative or the non-speculative execution. This work also goes over all the predictors in current out-of-order cores, and shows mechanisms to avoid the time-based information leak.

Index Terms—Spectre, Information Leak, computer security.

1 Introduction

The Information Leak definition on the Merriam-Webster is "to become known despite efforts at concealment." SoCs should protect the information from the applications to leak outside the application. Nevertheless, there are many side channel attacks that break this protection by leveraging the application side effects. In modern out-of-order processors (OoO), the most common source of information leak is timing side channel.

Timing side channel leaks are a powerful tool for hackers. In the late 90s [14], it became known that many encryption algorithms were susceptible to time side channel attacks. Specifically, it was observed that different data had different branch prediction performance, and this information leak could be observed by other applications or code sections, which compromised the algorithms. Since then, most encryption algorithms are execution latency free to avoid information leak through the branch prediction timing side channel. Other works have shown that similarly potentially dangerous information leak happens with the data caches, prefetchers, and potentially with any prediction in the core. More recently, the Spectre [13] class of attacks leverage time changes in the cache caused by speculatively executed instruction. In all these attacks, the information is leaked because the attacker has a clock or performance counter from the processor, and is able to measure the time impact resulting from executing some code. All these leaks could be avoided if the attacker did not have the capacity to gather any performance information on the code under attack.

A first, but faulty, observation is that all these attacks are a result of having a clock or performance counter. Simply preventing a process of getting good quality clock information is not enough to prevent attacks. For example, the attacker can have a fast loop to create its own local clock. Even better, the attacker can have several threads collaborating to create a global clock. These lower quality clocks are good enough for an attack, at most they just slowdown the information leaked to a lower rate. Even with a low *1ms* accuracy clock which is very easy to build, the attacker just needs to repeat the same attack millions of times to have enough time resolution. Since controlling the clock is not a viable source to solve the problem, this work focusses on the execution time information leak, not on avoiding clocks. This work goes over the details on local vs global clock, and points to the concerns of each one, but the protection is not achieved by avoiding to have a clock.

Since avoiding the clock is not possible, a solution proposed [17] has been to expose to the ISA places where the side channel time leak can happen, and ways to mitigate it or hide it in software. This can be classified as a software solution with hardware support. Examples of such "cooperation" are the patches to prevent Spectre/Meltdown that include retpoline patches to handle branches in the compilers, or KAISER Linux patches to separate address spaces. Although having the capacity to fix, in software, missing timing leaks is important for potential bugs in the hardware isolation, it is a better approach to avoid the time leaks in hardware. Since some of those protections can have overheads, the Hypervisor can decide which hiding techniques should be applied.

The ultimate goal of this work is to avoid time leaks from one code section to another. An attacker application can not infer anything from another application if it does not see any performance counter on the other application, or it does not have any change in execution time as a result of another application running, or it has exactly the same execution program path as a result of another program running. If the attacker application does not have any of the previous changes, we say that there is no time leak. As a result, there can be no side channel time leak attack. This work classifies these types of leaks and ways to avoid them.

One thing to remember is that there are several side channel leaks besides time leaks. Two other main side channels that have shown capacity to expose protection concerns are Electro Migration Interference (EMI) and power management. By measuring EMI [21], it is possible to infer program execution information. This is a difficult side channel to

avoid, but it requires close proximity to the device under attack. Power attacks [27] force out of range DVFS operating conditions to break encryption. This attack also requires precise timing information about the execution time of the code under attack in the secure zone.

There are many side channel attacks. The focus of this work is to avoid information leak in modern processors due to timing side channels when the processor is supposed to conceal the information. Examples of such time leaks are caches or branch predictors.

This work proposes to use the concept of time domain to any application or code section that wants to avoid time leaks to other time domain sections. The time domain concept is independent of the current OS constructs. The SoC could assign a whole VM to a single time domain, or each UNIX-like process can have a different time domain. In fact, the time domain can be assigned even to small code sections like libraries or functions. The smaller the code, the higher the overhead. In a way, current SoC without any protection place all the applications, operating system, and even Hypervisor in the same time domain because there is time leaks across them. The idea is that the Hypervisor could assign time domains and provide protection across time leaks. *This work considers that the there is a leak when the execution time in one time domain can be measured by another time domain*. A typical example is when a predictor performance impact can be seen by other time domain. this work also considers a time leak when the precise execution time can be inferred.

To be more precise, this work creates a classification of time leaks. Briefly, if the time leak has dependence with some data value, the time leak is due to data. Similarly, if the leak is due to some address operation by the core like load/store addresses, the time leak is due to addresses. Sometimes, no address or data or program counter affects the execution time. For example, when one core executes a program or time domain it can use all the memory bandwidth, another core would be affected by but potentially there is no difference between the data, address, or PC. In this case, it is a time leak due to execution time perturbation. Attacks that rely on knowing how long other applications execute also falls in the execution time perturbation leaks.

Ideally, the processor should have zero time information leaks in speculative and committed instructions. In this case, the execution time of all the instructions in a time domain should be the same no matter what other time domains have executed before or concurrently. This is called a SoC with *Safe[D4A4P4E4]* time protection level. DAPE because neither Data (D), Address (A), Program Counter (P), or Execution Time Perturbation (E) information is leaked. 4 is the time level of protection: 0 for no leak protection, 1 for operating system vs application protection, 2 for protection across different cores, 3 for protection within the same core, and 4 for protection all the other levels protected and also inside the same address space. Section 3 provides a classification that it is applicable to all types of processors. Each processor can have a classification time level which indicates which time level of information leak protection provides.

The document starts providing some basic concepts (Section 2) on time leaks. Then, it goes over the time leaks classification (Section 3). Then, it continues with a high level overview of different hardware and software time leak protection techniques (Section 4). Section 5 details an example out-of-order processor that provides several time level protection at boot time using a subset of the techniques. The document finishes by applying several time leak related attacks on the example out-of-order processor to see the protection degree and the point of failure.

2 Basic Requirements

In order to have a working system without time leaks, it is necessary to have time domains (Section 2.1).

There is a subset of attacks that send a request to a time domain, and it waits for a response. These are difficult to avoid time leaks. Even if the code was executed by a totally opaque alien CPU, just by waiting for the request answer, the human computer could infer the time that it took to execute in the alien computer. This request/acknowledge time leak attacks can only be avoided by denying a global clock to the attacker. Section 2.2) goes over clocks and how to provide/deny a local/global clock.

2.1 Time Domains Identifiers (TDI)

A time domain (TD) should not leak information to another time domain. A time domain is different from other constructs like operation system (OS), UNIX processes, or libraries. Any section of code can have associated a time domain. As such, a different time domain can be a dynamic library, an application, the OS, but also a small portion of code section. Some code sections may not have any associated time domain, as such they keep the existing time domain.

Each time domain has a unique identifier (TDI) across the whole SOC. The TDI has to be shared across the whole SoC. It is possible to pass the equivalent of "process id" through the whole system, but it is easier to just extend the physical address space with some TDI. Each core has only one running TDI at a given time [1]. All the physical address memory requests, instruction and data, going out of the core are extended with a TDI field. The TDI field is divided in OS and user level. Table 1 shows the breakdown of the physical address in TDI-OS, TDI-user, real physical address. The real physical address can address the whole memory available in the system. The TDI is just used for time domain management.

The number of bits in the TDI are implementation dependent, but the TDI-OS should have enough to encode as many as available cores. Similarly, to avoid flush overheads, the system should have enough TDI-user. For example, if the physical address space has 44 bits, bits from 44 to 55 could be used as TDI. In such scenario, the hypervisor/OS can assign the TDI. A source of complexity is how to deal when the SoC runs out of TD identifier. If the whole TDI is completely flushed to storage and/or DRAM, there are no side effects left on the system, and the TDI can be re-used.

The TDI is controlled by the program counter running on a given CPU. There are two main ways to change the time domain in a given core: PC switch and system call.

On a PC switch, the core checks the TDI assigned to the new program counter (PC). If the new program counter does not have a TDI associated (TDI==0), there is no time domain switch. If there is a new TDI, the core switches to

1. This work assumes no SMT threads in a single core

TABLE 1: Time Domain Identifier (TDI) field options. nz stands for non-zero value.

TDI-user	TDI-OS	Real	Meaning
0	0	nz	Hypervisor mode no time isolation
nz	0	nz	Hypervisor mode with time isolation
0	nz	nz	OS mode TDI
nz	nz	nz	Normal user level TDI

the new time domain, setting the TDI-user, but keeping the TDI-OS field. This time domain switch can trigger several changes in the SoC.

On a system call, the calling time domain clears the TDI-user bits to zero, and sets back the TDI-user bits at system call return. Inside the OS, there can be changes like in user mode if different pages inside the OS are marked with different TDIs. The advantage of having a TDI-OS is that the OS time domain can be shared across many user level TDIs. Nevertheless, to provide non interference of performance through the OS, it is possible for the OS set the non-zero TDI-user. On a Hypervisor call, the TDI-OS bits are cleared, and restored at Hypervisor return.

It is important to notice that an application switches to another TDI as soon as the memory address of the program counter executed touches another TDI. For example, if an application (TDI-x) has two dynamic libraries with two different TDIs (TDI-y and TDI-z), when a function call enters the first library, the new TDI-y-user is set for the whole core. When the library returns, the TDI-x-user is restored, and when the second library is called, the TDI-z-user is set for the whole core. A similar mechanism happens whenever there are calls and context switches. The application can also have code sections or libraries not mapped with specific TDI or TDI-user set to zero. In such cases, the TDI is not changed.

If there is no hypervisor or OS support to assign TIDs, the system can not provide protection for all the time levels. Without hypervisor/OS support, the SoC can still assign a fix TDI for a each core, and not mark any code section for PC switch. This system can provide at most time leak protection between cores, and between user and OS, but not within core user level applications or even between applications inside the OS.

2.2 Clocks and Performance Counters

Attackers can build not so accurate pseudo-clock programs with typically tight loops, or just leverage existing high precision performance counters available in the core. The idea is to not provide performance counters or high quality clocks to any application unless the operating system (Hypervisor) has granted access. The main reason to provide these counters to the application is for debuggers or benchmarking.

Even if performance counters are granted, the performance counters should be only per time domain. Applications should not sample performance counters across time domains. Otherwise, any time of protection is gone. This totally visible non-protected mode is still possible if the hypervisor sets the hypervisor bit in the address space even for the OS and the applications. Nevertheless, sharing performance counters is a bad idea as it even voids protection across virtual machines.

In this work distinguish between performance counters, global clock, and local clocks.

Performance counters are the most dangerous time leak because provide architectural insights on what happens inside the application. No need to infer branch prediction with time counters if the performance counter tells the outcome directly. As such, we propose to have performance counters only per time domain. Any performance counter across domains voids the time level protection provided.

We group in the performance counter category, any counter that the SoC provides about stats or state. This includes temperate sensors and power consumption sensors. For example, if the SoC has a counter for total power consumption, exposing this counter breaks isolation across cores. Just by reading the temperature or the overall power, one virtual machine (VM) in a core can create power/temperature signatures to know what is running on the other core. The goal of this work is to provide time leak protection, but it is not so beneficial if the SoC provides gratuitous counters for shared resources. The SoC can provide local performance counters like local branch predictor, or local cache miss rate. It even can "estimate" power if it measures only the core used by the time domain. Nevertheless, providing temperature is a bad idea because it has a slow moving average and crosses core boundaries. Global shared counters could be provided only under the following conditions: (1) The counter is not affected by SoC different time domains, like ambient temperature; (2) there is a single OS running on the SoC, then the OS could have access to all the counters; (3) the counter is from a resource used only by one time domain, like cache miss rates per time domain; (4) the counter is a proxy created using performance counters, like estimated power consumption.

The same way that counters cross time domains are not provided, the SoC should not allow to change parameters cross the SoC. For example, if the SoC has a single voltage/frequency domain, one time domain could not change the setting for the other time domains. In such a system the counter change should be ignored if there are several time domains running. If the options are changeable per time domain, it is possible to follow the recommendation, but the parameters should be context switched per time domain, and they should be considered just a “recommendation” never a mandate to follow. If the change request to get out of safety margins for voltage/frequency, the request should be ignored. Overclockers should run the OS and user level applications in Hypervisor mode, then they can change/read everything because it says that there is a single time domain. At boot time, there could be an option for sticky hypervisor mode.

A global clock is different from a local clock. A local clock ticks only when the application has an assigned core for execution. A global clock ticks or advances all the time. The idea is that when there is a system call or context switch, the local clock stops advancing. The global clock continues advancing. The idea is that applications should request to have local and global clocks to the hypervisor/OS. Both

pose different concerns, but a local clock is less problematic than a global clock. Nevertheless, a group time domain with local clocks could cooperate to build a global clock. For example, different time domains can cooperate with a Precision Time Protocol like the IEEE 1588-2008 [11]. This attack could provide sub-microsecond precision with just 10 updates per second.

As mentioned, the attacher can build a clock with a simple loop program. One thing that is possible is to deny a global clock when only one thread is involved. The attacker can create a loop program to create a pseudo-clock, but this loop is executed only when the time domain is active. If the time domain is sleeping, the counter does not increase. If the OS/Hypervisor does not provide a global clock counter to the time domain, it can not have a high precision clock. The only way is to rely on an additional time domain that may be active, and interchange information between the time domains or to request another thread in the same domain.

If the application only talks to other time domains through the operating system, the OS can know what API can leak. For example, writing to the disk does not allow to build a high precision clock. Same for other calls like getpid. Nevertheless, any call to get data from outside time domains either disk or network could be used as a communication channel to create a global clock. Even reading once from outside does not allow to build a clock, it requires a read/write/read or write/read/write loop. This restricts the set of applications that the OS can guarantee lack of global clock, but when guaranteed the time domain can not get information even like in the case of the alien core with the request/acknowledge time leak.

In many constrained applications like javascript browsers, each time domain has a single thread. If they are not able to build a global clock, they can not build histograms or traffic patterns usually needed to perform time leak attacks. If the system provides time level 4, this is not an issue, but if the system provides time level 2 and denies a global clock, it still can guarantee protection. It is important to notice that a time level 2 has some time leaks, but it is still secure if a global clock can not be build by the attacker. Since avoiding a global clock is very restrictive, we think that it is more interesting to avoid the time leak, but this section is about how to deny a global clock to the application.

Avoiding a global clock is also the only way to avoid a request/acknowledge time leak. For these reasons, we propose to deny global clock in a system when possible even when time level 4 is implemented.

2.3 Sharing between Time Domains

The SoC runs multiple time domains and memory protection that typically provides data/address isolation between them. Notice that we say typically because time domains are about avoiding time leaks, not to avoid data leaks across memory protection domains like process in OS. Two time domains could share the same address space which allows to share data without constraints. This data sharing can be used to create a communication channel that allows to create a global clock or to directly pass information about the local performance counters. Time domain isolation is about not allowing to indirectly measure any performance on other time domains. The technique proposed (Section 4.1) guarantees that two time domains running in different cores can not infer any time leak even if they share address space.

Sharing address space complicates the system, but it is a common thing through dynamic libraries. The advantage is that the code is read-only shared, and that the private variables are per linker address space. This means that for this common case there are no read/write shares. Since there are only read shares, the techniques proposed should provide isolation.

If there is a shared writable memory region between time domains, there may be a way to track the communication extending/modifying the coherence protocol, but we leave out of the scope of this work.

This means that the OS/hypervisor can effectively monitor the most common communications between time domains. For an attack to work, it needs to perform the attacking test many times. Being able to detect when attack is attempted is a way to avoid most of the attacks. For this to work, the time domains do not leak precise time information unless active communication happens between the attacking and the target.

2.4 Statistical Source Anonymity

One of the techniques [1], [3], [8], [23], [28] that secure network provide is the capacity to create fake traffic to obfuscate the real network traffic. This area of work is usually called statistical source anonymity, and there are many algorithms used. The idea is that the system should be able to be tuned to have different algorithms.

A basic algorithm is to have a mean, a standard deviation, and to randomly generate periodic traffic. Other papers use more advanced models that mimic fractal characteristics of network communication.

The same techniques to detect network fingerprinting and generate fake network traffic could be used to hide the time information leaked when it is not possible to avoid the time leak. If the system does not provide time level 4, and/or does not provide global clock avoidance, it can leverage statistical source anonymity to obfuscate the pattern. Example of such systems for hardware is the Sanctum [7] that provides statistical source anonymity even between the OS and applications.

2.5 Dynamic Adaptation

To achieve better performance, the SoC is allowed to reconfigure or perform dynamic adaptation periodically. Here the idea is that the time domains have exactly the same performance given the same partitioning. If the SoC does a dynamic adaptation, a new set of guarantees or performance is provided.

If the dynamic adaptations were performed randomly, there would be no time leak. The issue is that they are performed to adapt to the SoC demand. As such they leak information about the overall SoC workload. The goal here is to have a slow and infrequent non-reactive adaptation. As such, it is not possible to leak time information for the reason of the adaptation. The higher the concerns on security the lower the adaptation speed. It is even possible to keep the same time domain with a given adaptation. When a time domain is created the same partitioning can be provided. The idea is that there are many options on how to manage this.

For simplicity, in this work we propose align the dynamic adaptations to some constant intervals. For example, the

SoC can dynamically reconfigure once every 10M cycles. When the 10M cycle is reached, all the pipeline stages affected by the adaptation are flushed, and a new SoC resource allocation can be accepted. The local clocks do not advance during the time to adapt. If a system in the SoC has an adaptation recommendation, the application is delayed until the common cycle to adapt is reached.

The dynamic adaptation can leverage the same idea than statistical source anonymity (Section 2.4). The main difference is that instead of generating fake traffic, it statistically decides the cycle for dynamic adaptation.

3 Time Leak Classification

Not all the SoCs needs or provide the same level of protection. The goal of the proposed classification is to better understand the sources of time leaks, and being able to provide a meaningful way to categorize current systems without having to enumerate list of security bugs affected for each SoC. This section explains the different protection levels (time level), the different information leaks, and the methodology to test the hardware for each time level.

The classification considers four sources of time Leaks for speculative (*Spec*) and non-speculative (*Safe*) executed instructions. A *Safe* instruction is an instruction that commits or retires. A *Spec* instruction is an instruction that the processor speculatively executed and that it never commits or retires. *A time leak happens when an attacker time domain can see the execution time of another attacked time domain*. If there are no performance counters across time domains, the way to see time impact in the attacked domain is to measure the execution time perturbance in the attacking time domain, or to directly/indirectly read prediction table contents. For each speculative or non speculative instruction, there can be a different type of source for the time leak.

- **Data (D)**: We say that there is a Data leak (D) if different data values can potentially trigger different execution time perturbances in another time domain. Most typical leaks come from addresses, but data can directly leak if left in predictors like data value predictor.
- **Address (A)**: An Address (A) leak happens when different load address values can trigger different executions in another time domain. Address leaks are very common in current processors, typical but no exclusive cases are data cache replacements, memory disambiguation predictors, and prefetchers.
- **Program Counter (P)**: A program counter leak (P) happens when different time domains see different time impacts depending on a single instruction being executed. Typical, but not exclusive, cases are branch predictors and instruction caches.
- **Execution Perturbance (E)**: Sometimes the execution of a time domain creates a time perturbance in another time domain. The perturbance does not leak data, or address, or even PC, but it is possible to know that the other time domain is running because there is an execution time perturbance. Ideally, there should be zero change in performance no matter what the rest of the SoC does.

  If there is any concept of global time in the time domain, it would see that there was a time on inactivity. Similarly, for a system call, there is another time domain switch. The application can record the time before the syscall and compare after. The execution time perturbance is against local clock.

The classification also consider the Time Level (0-4) protection provided for each leak source category.

- **No Protection (0):** When the time information leak can cross all the time domains, we say that there is no time domain protection. A typical example with no protection are the caches, in most current systems even applications executed in different cores can have time information leak due to replacements or inclusive caches.
- **Multicore/Process Protection (1):** When the time information can not leak across different cores, we say that the system is secure for multicore (1). If time domain run in different cores, a time level 1 guarantees no change in the local and global clock, performance counters, and execution path.
- **Operation System Protection (2):** Applications do frequent calls to the Operating System (OS). When the execution time in the OS can leak to the application or user level, we say that there is a lack of Operative System protection or that the Time Level 2 is not met. The idea is that time leaks from the operating system should not affect the application level. Similarly, the hypervisor information should not leak to the operating system. The application/user level is allowed to leak to the operating system, and the OS is allowed to leak to the Hypervisor. Time level 2 guarantees no change in the local clock, local performance counters, and execution path. Notice that it does not guarantee global clock because the user level application is inactive during the system call, as a result a global clock could measure this time impact.
- **Same Core Protection (3):** Creating isolation/separation between different cores is easier than within a core. A time level 3 protection guarantees that there are to time leaks between different processes in the same core. Since processes can be context switched when sharing a core, time level 3 protection provides the same guarantees as time level 2 protection.
- **Same Address Space (SAS) Protection (4):** When the SoC can guarantee that the information is the time leaks can not happen threads in the same address space, we say that the SoC provides Same Address Space Protection or Time Level 4. No current processors that we are aware provides this degree of protection, but it is interesting for applications like browser that could execute different clients in different threads while providing time leak protection. Another useful example are verifiable JIT code that the application can pass to the kernel, and the kernel executes it. The verification could check for malicious accesses, but the time leak will protect for side channel leaks. A time level 4 protection guarantees that there are not changes in the local clock, local performance counters, and execution path independent of what the other security domains execute.

To reduce the repetition, it is equivalent to say *Safe2* and *Safe[D2A2P2E2]* When all the accesses have the same protection level, it is not necessary to detail each one. (*Spec3 = Spec[D3A3P3E3]*

To reduce the number of options, we make the time

TABLE 2: All the possible classifications possible based on the combinations of source of information leak and time level protection.

Leak Source		Protection Time Level				
		None (0)	Multicore (1)	OS (2)	Core (3)	SAS (4)
Safe	D	*Safe[D0]*	*Safe[D1]*	*Safe[D2]*	*Safe[D3]*	*Safe[D4]*
	A	*Safe[A0]*	*Safe[A1]*	*Safe[A2]*	*Safe[A3]*	*Safe[A4]*
	P	*Safe[P0]*	*Safe[P1]*	*Safe[P2]*	*Safe[P3]*	*Safe[P4]*
	E	*Safe[E0]*	*Safe[E1]*	*Safe[E2]*	*Safe[E3]*	*Safe[E4]*
Spec	D	*Spec[D0]*	*Spec[D1]*	*Spec[D2]*	*Spec[D3]*	*Spec[D4]*
	A	*Spec[A0]*	*Spec[A1]*	*Spec[A2]*	*Spec[A3]*	*Spec[A4]*
	P	*Spec[P0]*	*Spec[P1]*	*Spec[P2]*	*Spec[P3]*	*Spec[P4]*
	E	*Spec[E0]*	*Spec[E1]*	*Spec[E2]*	*Spec[E3]*	*Spec[E4]*

domain protection levels cumulative. This means that a Time Level 4 implies that the SoC provides protection for SAS, same core, multicore, and Operating System. Similarly, a Time Level 2 provides protection for multicores and OS. It is possible to see a system that provides multicore protection and no Operating System protection. For such cases, we can say that the SoC provides level 2 (OS) but not 1 Time Level protection (multicore). We ordered in this way because time level 1 can provide global clock protection while time level 2 only can provide local clock protection. Based on the classification from Table 2, we can categorize current cores:

- **Intel Skylake, ARM A75, and Apple A11** have a *Safe[D2A0P0E0]* and *Spec0* Most Intel CPUs without Spectre [13] security patches have a very low protection level. This is in-fact the security hole exploited by Spectre. The data in the speculative path can break pass security levels, and leave cache side effects visible across time domains. Non-speculative loads leak addresses but not data in these processors. The reason is that data would not have a side effect across domains unless it is used by a branch predictor or load address. In speculative mode, the attacker can insert false speculative path instructions to leak the data, but this is not possible in non-speculative data.
- **AMD Zen, and ARM A72** have a *Safe[D2A0P0E0]* and *Spec[D3A0P0E0]* The reason is that in these cores when a speculative load has a TLB protection problem, the load does not return data even thought it is in the speculative path. This means that the same application can not leak data values across domains.

Notice that the previous cores have a very low degree of information leak protection. This section shows potential attacks that could be created unless these time side channels are closed.

- **Data**, leaking data is the worst type of leak. Many works propose value prediction like the ones proposed for L2 caches [5] or execution like EOLE [19]. None of those works deal with time leaks, this means that this works have a *Safe[D0]* which is the easiest way to create attacks. Runahead [18] is another technique used by several processors that can also have the same Time Level if implemented as suggested in the literature [2]
- **Address**, leaking addresses is a way to indirectly leak data and to also leak significant information across cores. Leaking addresses across domain can point to code and/or data in another time domain. In fact, this side channel is the typically used to pass information in most attacks. The attacker code creates cache or branch prediction disturbances in one time domain, and it can be seen in another time domain. Most attacks rely in cache allocation with some malicious code, but no attacking code may be necessary. For example, an attacker can learn about the prediction rate for a given address. If the code has new cache misses, it may be because it reached a new set of data.
- **Program Counter** is known to be a security risks, and most recent encryption algorithms are branch-less as a result. If the SoC provides a *Safe[P3]* and *Spec[P3]* protection level, it is possible to use several encryption algorithms safely.
- **Execution Perturbance** is the most challenging side channel to avoid. Examples is when the operating system executes code for a keyboard interrupt. Some works [4], [25] point that by looking at activity rate in the keyboard, it is possible to infer the typed password. This means that an attacker can just monitor performance perturbances that happen periodically due to keyboard interrupts and infer the password.

3.1 Desired Support

The classification helps to understand the different time levels and their respective protection. From a practical point of view, there are three very interesting overall categories: *Spec4 Safe1 Safe2* and *Safe4*

Spec4: An SoC supporting this level means that it behaves like a trivial in-order non pipelined core from a time leak point of view. This means that the developer does not need to worry about speculation for time leaks. Things like speculation fences in ARMv8/x86, retpoline, or KAISER become unnecessary. *Spec4* level means that the processor speculative path does not leak, but it does not say anything about the non-speculative instructions. The *Spec4* protection guarantees the same local and global time, local and global performance counters, and execution path.

Safe1: Provides the *Spec4* advantages, and it also provides performance isolation when the different time domains execute in different cores. A *Safe1* SoC guarantees that time domains across cores can not affect the local or global clock, local or global performance counters, and execution path. It is a very strong isolation as long as the time domains run in different cores. *Safe1* allows to deploy different virtual machines (VMs) in different cores, and guarantees that no performance interference between the cores. This means that servers like Amazon/Google could have different VMs and

2. IBM Power6 and NVIDIA Denver implement runahead, which may make them susceptible to time leaks similar the Meltdown.

guarantee the same performance independent of what the other VM is doing. The only interferene can happen as a result of the hypervisor/OS managing the VMs because a *Safe1* does not provide *Safe2* isolation.

Safe2: Provides the *Spec1* advantages, and provides OS protection. In *Safe2* the user level application can not see any local clock, performance counter, or execution time change as a result of different code executed inside the OS. It is important to remember, that *Safe2* can not provide global clock protection if there is a coordinated attack to build a global clock. In a VM environment, it means that a the VM attacking the other VM needs to access outside the SoC to build a precise clock, and then it can measure the time that the hypervisor/OS takes in hypervisor/OS calls for the calling VM.

Safe4: Provides the *Safe2* guarantees, but allows to share the same core between different VMs and single address space (SAS) time domains and guarantees no performance interference in the local clock, local performance counters, or execution path.

3.2 Time Leaks without Time Impact

The goal of this work is to avoid time leaks across time domains. The reason is to improve security or protection between applications, but it also has a side effect of increasing repeatability because other running applications do not interfere with the executing application.

As previously mentioned, besides time leaks, the system has to deny sharing any performance counter across time domains. Gaining any insight in branch prediction or cache statistics is even more capable of leaking information than a time attack.

Similarly, most structures can not be shared across time domains. Sharing entries in the caches have a clear time leak because of the performance impact. Most structures would have a time impact, but it is possible to leave some information across time domains. For example, a load-link store/conditional pair tend to arm a register. If this register is left across time domain switches, a store conditional would fail/pass differently depending on the previous thread execution potentially leaking time and address information. The testing (Section 3.3) infrastructure should cover all these cases because the same execution path and time should happen in a time domain independently of what happened in another time domain.

3.3 Testing for Time Leaks

Understanding how testing works helps to understand the different classification categories. This section explains some mechanisms to test for time leaks in a hardware development platform. In a way, it is easier to explain the test that each time level should pass than going over all the details on what the time level protection requires.

For each category show in Table 3, we can create a specific set of tests. For all the tests, we have a one or more time domains that are going to be changed (attacked time domains). We also have a single time domain that is going to be the attacker time domain. We introduce different types of changes or perturbations in the attacked time domains, and we should not observe any changes in any of the metrics in the attacker time domain. This means that the attacker time domain has the same local time or local performance counters or execution path. Notice that the testing infrastructure can not cover multithreaded applications because those are not deterministic by nature in current SoCs.

To test for *Spec[D]* we can inject random changes in data values for speculatively executed instructions, and they should have no time effect in any other time domain. In general, data value changes can have effects in the attacked time domain, but not in the attacker time domains. In *Spec* tests, data value changes should not have any time impact neither the attacker or attacked time domain.

The challenge in the *Spec* verification is that random data changes will affect the execution path if the speculative instruction is not discarded. To solve this issue, there could be a two pass simulation to detect all the instructions fetched but that were never committed. In a second run, the testing environment injects fake/random data values to the discarded instructions. There should be no time effect in other time domains. Even better solution based on typical testing infrastructures is to have an emulator in the testing setup. An emulator can be coupled with the fetch, and automatically detect which instructions are fetched that do not correspond to the commit path. Those speculative instructions are target for random data modifications.

To test for time level 1 (multicore), one core runs the attacker time domain, and other cores the attacked time domains. Random changes in the *Spec[D]* path in the attacked time domains should not affect the attacker local or global clock, local performance counters, or execution path.

To test for time level 2 (OS), the testing platform should create attacked applications with system calls. Whatever is executed inside the system call *Spec[D]* path should not affect the user level code. For a time level 2, this means no effect in the local clock, local performance counters, and execution path.

To test for time level 3 (core), the same test as the time level 2 is performed, but all the time domains can freely execute in any core. This means that the test should include cases were the time domains share cores, or migrate, or multiple combinations. Since the attacker time domain can have context switches and thread migrations, it can have a lower overall performance, but it should not have any change in the local clock, local performance counters, and execution path.

To test for time level 4 (SAS), we do the same test as time level 3, but the attacker and attached time domain can share the same address space. The time domains can share data addresses, but they can not use this information as a source of execution path divergence. The reason is that different context switches can slowdown/speedup some time domains. If the attacker time domain can "read" the memory contents, it can see the progress. Time level 4 protection is not against accessing data directly, but not to have any time impact.

In all the tests for time level 3, random code changes in the miss path should have no impact on the execution time.

One thing in all the time level tests is that the *Spec[D]* random changes may result in a exception/missprediction/etc in the microarchitecture. This can not be made visible to the attacker time in any way. Neither local time, local performance counters, nor execution path.

The same technique can be applied for addresses (*Spec[A]*) and program counter (*Spec[P]*) Instead of changing data,

the address or an individual instruction is changed. *Spec[E]* is a bit more challenging because it requires to replace a set of instructions in the speculative path for other random instructions. Those random replaced instructions should not have any effect in time in other time domains.

Testing for *Safe* time leaks is a more straightforward solution as it does not require multiple runs, but it is a bit more challenging to create programs for testing. To test for *Safe[A]* the random program/instruction generator (RIG) can generate programs were data values are different but all the other operations are the same. In this case, all the programs should have exactly the same time effects in other time domains. This technique can be leveraged to detect data (*Safe[D]* address (*Safe[A]* and program counter changes (*Safe[P]* On challenge to remember is that different data/address/PC should have different IPC for a given benchmark because they affect the execution time like branch prediction accuracy, but they should not have different impact on other time domains. This is the thing that the test covers, no impact or time leak in another core.

To handle safe time perturbances (*Safe[E]* the RIG can also insert code sections. The code sections should not affect the execution time in other time domains. The attacker time domain should have the same local clock, local performance counters, and execution path.

Both *Safe* and *Spec* need to handle context switches and system calls. This means that creating random system calls or context switches should not affect the time perturbance in other time domains as long as the same amount of time is spent in the operating system or other domain. The test considers the local clock, local performance counters, and execution path.

It is possible to extend the testing infrastructure to include multithreaded applications in the attacked time domains. This is OK because we are only measure the attacker. Adding a multithreaded attacker is more challenging. The main reason is that multithreaded applications are not deterministic when context switched. Nevertheless, the same perturbations can be applied if one set of cores runs several multithreaded attacked time domains, and a different set of cores run the also multithreaded attacker time domain. Notice that this last test does not allow for Time Level 3 testing with multithreaded attacker.

A time level 3 with a multithreaded attacker requires the OS to context switch all the threads in a time domain when one of the threads is context switched. If one of the threads goes to sleep, it is not necessary to context switch the whole time domain. This is a feature not supported by current OS and that requires being able to synchronize multiple cores for context switch.

If a platform supports time level 3 testing, it can automatically support a time level 4 testing. The only constraint is that the attacker time domain is not allowed to read from the same address space that the attacked time domain writes. The attacker time domain is allowed to write in the other time domains, but the test does not allow to read.

Dynamic adaptation (Section 2.5) also complicates testing. The same dynamic adaptation sequence should be applied for the different tests checking the same time domain performance. This fine when clear boundaries for adaptations are available. When the same test is ran, we change the attacked time domains. As long as we maintain the same attacked domain dynamic adaptations or resources, we should guarantee no changes in performance independent of the dynamic adaptations or resources in the attacked time domains.

4 Protection Techniques

4.1 Protection Hardware Techniques

To have a successful time side channel attack, there has to be 2 main components: The attacker must have a clock or event counters sensitive to other time domains or a change in execution path; the time domain under attack (attacked) may have a difference performance or leave different impact on the SoC depending on the data/address/... used.

Breaking any of the previous breaks a time side channel attack. We know that the local clock can not be avoided, and that the global clock is difficult to avoid if there is a coordinated attack from multiple security domains.

It is possible to measure the performance of other time domain directly, or by observing all the information left as a result of predictors in the processor. If a time domain leaves cache/predictors information populated to be used by other time domains, they can infer the work done. To avoid all this time dependent information leak, the SoC could do flushing (Section 4.1.1), off-loading (Section 4.1.2), and partitioning (Section 4.1.3).

While the previous techniques make sure that information resources are not shared across domains, isolation (Section 4.1.4) ensures that other resources like bandwidth utilization are separated between time domains.

Being able to build a global clock allows to do attacks that use an request/acknowledge protocol or even context switches. This can happen even with isolation, partitioning, flushing, and off-loading. This time leak can be mitigated with regularization (Section 4.2.1) that works by hiding the starting time for time domain tasks. A more strict technique is homogenization (Section 4.2.2) that enforces that code sections have repeatable execution times when executed multiple times. A complementary technique that helps regularization, and homogenization is to have a non-value dependent operation when possible (Section 4.1.5).

If the goal is just to protect from speculative time leaks, it is possible to avoid speculative updates (Section 4.1.6) and/or fix them (Section 4.1.7), avoid speculative data (Section 4.1.8). All these overheads can be reduced by leveraging the point of no return (Section 4.1.9) from modern OoO cores.

4.1.1 Flushing

Flushing guarantees no information leak between time domains by flushing its context each switch between two different time domains.

For large structures, it may be interesting to do partial flushes because it takes time to flush and/or too much history may be lost. To reduce the overhead, is possible to do a partial flush. A partial flush deallocates a chunk of the resource at time domain switch. This partial flush has two implications to the hashing function and to the decide how much is flushed each time.

From a time point of view, a partial flush allows the secure domain to detect the percentage of structure flushed. This is not a problem if the percentage flushed in independent on the length or execution time of the time domain. This means that if we were to flush 25% of a table like BTB each system

call, and allow the OS only to use that 25% of the BTB table, there would be no time leak. This is because there would be always a constant overhead, but it would be the same overhead independent of the work done inside the OS.

The partition also can be done dynamically (Section 2.5). In the rest of the document we refer to these options as **Full Flushing**, **Static Flushing**, and **Dynamic Flushing**.

One additional mechanism that should help flushing is a background state machine that tries to writeback dirty lines. The idea is that an opportunistic FSM can look for idle cycles, and increase the amount of clean lines. The advantage is that flushing is very cheap for clean lines, but costly for dirty lines. The same FSM could have a threshold to guarantee that a percentage of the cache ways are always clean. We call this FSM the **Dirty Cleaner FSM**. There can be a dirty cleaner FSM for each cache with a different policy and thresholds.

One thing to consider is that flushing could take a variable amount of time. This may be a concern with dirty lines. If the core delays the time context switch because there were many dirty lines, it is effectively leaking through the global clock that there are many dirty lines. There are several solutions to this problem: worst case, reserve channel, moving average, and/or "in cresendo".

A first approach, the worst case flushing is to not allow over a % of the cache lines to be dirty, and then always charge the amount of time needed to write back the whole % even if only a subset is dirty.

A second approach, we reserve a channel or BW in the SoC for dirty lines displacement. This is difficult to manage unless time domain switch is coordinated. It is not clear how to handle it.

A third approach is to have a slow moving dynamic adaptation of the maximum percentage of lines that can be dirty.

A fourth approach is to give displace BW "in cresendo" (or increase) to the new time domain as it switches. This allows to reserve a BW for displacements. This is very reasonable because the time domain should not have dirty lines after just starting.

Unless otherwise stated, the more reasonable is to have a dynamic % adaptation, and to reserve BW for displacements during the beginning of the time domain context switch. Having a Dirty Cleaner FSM would help to reduce those percentages.

4.1.2 Off-Loading

Off-loading is a variation over flushing. Like flushing it can be full, static, or dynamic. The difference is that instead of just invalidating, the entries are read and checkpointed. When the same time domain is re-executed again, the checkpoint is recovered.

To avoid blocking, the checkpointing and recovery can be done lazily. The entries are marked invalid, and the checkpointer state machine creates a backup. If the entry needs to be used by a new security context before checkpointing, when the entry is read, the value can be checkpointed. If it is not possible, the entry is just not checkpointed and marked invalid at recovery. Effectively the same as flushing for that entry.

Combining off-loading with flushing can have a significant advantage. For a time domain switch, a small percentage of the structure can be statically flushed. Meanwhile, the rest of the structure is off-loaded. Typically, it is not needed to off-load the whole resource. In systems like caches, it is reasonable to off-load just the MRU set, and to flush the rest of the sets.

As long as the off-load and flushing have a constant time independent of the data/addresses/...flushed/off-loaded, there is no time leak. We call the approach of always off-loading a fix percentage like the MRU set a **Static off-flushing**. The **Dynamic off-flushing** version is possible but the complexity makes it less interesting.

4.1.3 Partitioning

For some resources, it is not easy to flush or off-load. This is the case for most shared resources like last level caches or directories. For such shared resources, the best approach is to partition. Again, the partitioning can be static or dynamic. SecDCP [29] shows a mechanism to dynamically partition a last level cache.

For resources with different utilization ratio, it should have a better performance to have dynamic partitioning. The best example is the last level cache. For some resources with more homogeneous workload, it should be simpler to have a static partitioning. For example, a size directory is mostly a function of the caches in the system. This means that it should be efficiently partition for different cores.

The decisions to handle dynamic partitioning are resourced based. In this work, we call these techniques either **Dynamic Partitioning** or **Static Partitioning**.

4.1.4 Isolation

Partitioning guarantees that entries in resources from one time domain do not affect other time domain. This avoids aliasing and interference. Isolation provides separation or QoS for resources that handle transactions like memory bandwidth.

Even for high performance systems without side channel time leak information worries, resources have some degree of isolation to guarantee that a core does not hog all the resources. For example, a quad core may allow a single core to go between 0 and 50% of the total memory bandwidth. To provide isolation, the constraint is a bit more stringent. The core has a fix percentage of the bandwidth, not a range. A naive implementation would give 25% of the bandwidth to each core in a quad core system. This works but it is not very efficient. The solution is similar to the dynamic flushing/off-loading/partitioning. It is OK to adjust periodically the BW allocated as long as it is not based on some time leak that did not exit before.

A solution is to allocate the reserved BW based on the number of active cores. This means that more cores will result in a lower bandwidth, but once they are executing there would be no leak beyond the fact that other cores are running, this is not consider a fine grain time perturbation. To further optimize the BW utilization, it is possible to dynamically adjust the BW assigned to each core based on average executions. In the rest of the work, we call these **Static Isolation** or **Dynamic Isolation**.

4.1.5 Non-Value Dependent Operations

Branches, memory operations, and ALUs operate over different data. When the value of the data has a different execution time, we expose the time domain to have a time

leak side channel. A time leak can also be exposed when the resource is pin-down for multiple cycles and it can not be killed. For example, if a long latency square root operation can not be killed, it can delay the branch prediction flush effectively leaking that there was a long latency square root.

If ALU operations have a variable latency due to data, it is possible to infer the value contents. Since it is not easy to hide the execution time of memory operations the best way to deal with this type of attack is to have a fix latency for all the values. Some cores and papers propose telescopic units [12] to early exit for simple operations like divide by 1 or 2. This may improve the overall performance, but it is a source of concern for time leaks.

Non-value dependent operation is mostly a floating point and vector issue. In those cases, it means that the operation should executed with the same latency no matter the value. In floating point, this means no telescopic units, and that dealing with denormals should have also the same latency.

Non-value dependent time operations for branches and loads is a high bar that introduces significant overheads. A more efficient solution that still provides fine grain execution perturbation protection is homogenization (Section 4.2.2) or to do code transformations usually done in encryption to guarantee the same execution time.

4.1.6 Avoid Speculative Updates

If a SoC performs flushing, off-loading, partitioning, and isolation, there is no need to avoid speculative updates. The reason is that the speculative updates can only be as a result of the current time domain, and the other protection techniques guarantee that there are no time leaks or performance events between time domains.

Nevertheless, enforcing all these techniques can have a higher performance overhead. For that reason it is interesting to avoid speculative updates. The idea is that the update is delayed until the instruction becomes non speculative. In many cases like the branch predictor or prefetcher, it has a very small overhead to avoid speculative updates.

4.1.7 Fix Speculative Updates

Sometimes speculative updates are not possible to avoid or to delay. This is the case of the global history register (GHR) for a branch predictor, or the return address stack (RAS) updates for function call and returns. In cases were the speculative update is needed, the hardware must guarantee that the update does not change loose any information stored before, and that if the branch is flushed, the information can be recovered.

A solution in those systems is to have two structures. One updated speculatively, and another updated when the instructions become non-speculative. If there is any flush in the pipeline, the speculative structure is restored with the non-speculative one.

4.1.8 Avoid Speculative Data/Address

To provide even more protection during the speculative path, it is possible to not allow to use speculative data in the speculative path. For example, a stride prefetcher could use the load address, but it is equally OK to just use the retired load address and increase the stride delta. In the simple cases when the speculative data can be avoided, it is a way to make sure that the speculative path does not time leak sensitive information.

4.1.9 Point of No Return (PNR)

In this document we propose to classify SoCs time based on the degree of protection of speculative (*Spec*) and non-speculative or safe (*Safe*) side channel time information leak. If a processor does not have any speculation, it automatically has the highest level (*Spec4*). The problem is that modern OoO can not achieve high performance without speculation. A partial solution that mitigates the problem is to leverage the fact that a fraction of instructions are non-speculative.

Point of no return or PNR [16] was proposed as a point in the reorder buffer (ROB) were older instructions than PNR are guarantee to eventually commit. No instruction between the PNR and the retirement point can trigger a branch misspredictions or an exception. This guarantee means that any load/store instruction has full address resolution and knows about TLB permissions, any instruction that can trigger an exception is cleared, and that branches are resolved.

Using ESESC for SPEC2006, an modern A72-like processor has around 25% of the instructions in the ROB beyond the PNR. This means that for security reasons 25% of the in-flight instructions can be considered non-speculative. These non-speculative instructions can update the prediction tables without risking to pollute information.

In the same way that instructions retire in-order, instructions pass the PNR in-order. In this work, we use the word commit for the point when instructions pass the PNR and retire for the point that instructions are removed from the ROB. Both happen in-order, but on average there are 25% of the instructions that have committed but they still can not retire.

Notice that this is different from most definitions were commit and retire have the same meaning. To be more specific, a load or store goes through 4 phases. Commit, Retire, and performed. A load always does commit first, then performs, and then it can retire from the ROB. Perform is the moment that the load is visible to other cores. For loads this tends to be the point that the value is binded to the register file. Typically, a store is performed, when the cache line gets an exclusive (M or E) state and the store contents is copied to the cache. As a result, a store is a bit more complicated in a release consistency mode. Stores always commit first, but the perform and retire can happen in any order. Typically, a more conservative Total Store Order (TSO) processor may restrict to have the load sequence, commit first, perform second, and retire third.

4.2 Mitigation Hardware Techniques

The following techniques do not avoid time leaks, but they help to mitigate the existing time leak by trying to scramble it and make it useless.

4.2.1 Regularization

Isolation is a high protection bar but that may be required for Operating System code. Nevertheless, even static isolation and partitioning do not provide a *Safe3* protection. The example previously explained is the keyboard interrupt, but the same is true for most interrupts like the network. If the system is fully loaded, one application must be context switched when the interrupt happens. An inefficient but simple solution is to always reserve one core for the operating system (*Safe1* suffices in this case). When a core

is available for interrupts and IO, there may be no need to do regularization if the different interrupts are executed by the same time domain. This is the typical case in the Linux kernel were there is no isolation inside the kernel.

For such attack, there must be a global clock, but as we know it can be created with a coordinated attack. To provide some protection even with a global clock, we can do regularization.

Regularization solution is to create fix application interference periods even when no interrupt is generated. Notice that regularization does not avoid a time leak, but it hides the leak to make it useless. The idea is to copy the idea from statistical source anonymity and create fake workload. Regularization creates spurious starting execution times for time domains. Namely, a keyboard interrupt may be adjusted to happen only in multiplies of 5ms, and there can be several spurious fake interrupts after the interrupt for a period of time. This would not affect the keyboard performance, but it will obscure the keyboard rate pattern. The same is true for other interrupts.

The idea is that periodic events can be adjusted to happen at some more regularized time periods, and the SoC should generate fake events even after the activity is finished to further hide the impact. The result is that an attacker can know that there was a coarse period of activity but no fine grain execution time perturbation. When the regularization is applied to all the interrupts and IO traffic, the attacker can not know the exact starting time for these events. Periodic traffic is a very secure method for source anonymity, but it is considered to have a high overhead in networks. The overhead in hardware is not as high for those infrequent events.

For each interrupt type and/or IO handling there should be a discrete interval to regularize. In addition, there should be also the capacity to generate some fake workload. A simple case would be to generate a variable number of fake interrupts afterwards at similar time intervals. More complicated traffic mimicking using Statistical Source Anonymity (Section 2.4) models.

The longer the regularization, the coarser the time execution perturbation that we can protect. The cost is a lower efficiency on the system. There is a point when it is just more efficient to allocate a core for all the interrupts. The idea is that when the traffic mimicking and regularization overhead is too high, the SoC should switch and reserve one extra core to that time domain. Typically, it would reserve one or more core for the Linux as the OS has higher performance demands.

4.2.2 Homogenization

Homogenization is a proposal to hide the variable latency when executing code blocks. The idea is that when a code section is executed, it should have the same IPC no matter what.

Performance modeling using simulation points [9] or program phases [15] know that repeating code phases tend to have the same performance or IPC. There is variation on the branch prediction/misses, but on average there is a repeatable performance. The idea here is that the SoC can track the performance per program phase. By tracking the average and standard deviation per phase, it is possible to enforce that the code section has some consistent IPC.

Homegenization could also be adjusted. For example, a very strict solution is to always execute the code section with the worst possible execution time found up to the moment. This is different from assuming that everything mispredicts. The larger the code section to homogenize, the higher the chance of having lower performance impact.

To implement this approach something like the program phase [10] can be used to track program phases and the associated average and standard deviation IPC. Once the expected IPC is known, enforcing extra delays is trivial.

A hardware only solution can have significant overheads, an interesting extension is to leverage the software to annotate the code. The software could also annotate the code section to protect with homogenization. In that case, it can indicate that the time should always be the same. For example, an encryption algorithm can use branches and indicate the code section for homogenization.

While in the homogenization section, the OS should not try to context switch. Otherwise, it may require to remember the amount of code executed in the homogenized section.

The same homogenization hardware can also be used to detect potential attacks. If a code section with a very tight standard deviation starts to have very different results, it is possible than an attacker is trying to test the system to gain insights.

Because of the software/hardware interaction and the open opportunities/challenges, this document does not further explore this option, but the overall idea and implementation is presented here so that the work remains open.

Having a "worst case" is still not a perfect solution. For example, if there is a password checker that checks very slowly the "1234" password, and very fast otherwise. An attacking system that could see the response time can infer with just one measurement that the password was "1234".

4.2.3 Sanctum

Sanctum [7] provides extensions in the memory hierarchy to provide memory page coloring and randomize the address space. The idea here is that if the physical address spaces are randomized there may be a time impact, but the attacker can not know the source. The same idea works with oram [26].

We consider this memory scrambling not a time leak protection but another security level to handle the global time clock leak.

4.3 Protection Software Techniques

4.3.1 OS/Hypervisor Shuffling

The OS/Hypervisor could try to execute different time domains that do not communicate in different cores when the load is low. There reason is that techniques like flushing, off-loading, and partitioning have an overhead each time that there is a switch between time domains.

Although it creates more overheads and flushing. It is better for security to place in the same core time domains that communicate with a synchronous interface. Otherwise, another "attacking" time domain can detect the time to complete the synchronous communication (See Section 6 to show a potential attack). The reason is that a coordinated attack to build a global clock requires to run in different cores or to communicate outside the SoC.

The challenge here is that time domains do not provide data isolation. To handle the OS has to create a new OS/process for each time domain.

5 Out-of-order Implementation Example

While the previous section goes over the different techniques to secure the time leak side channel, this section goes over the main structures and predictors in a modern OoO, and shows an example of how to use some of the previous techniques. The described SoC always provides *Spec4*, and at boot time it can be configured to provide *Safe1*, *Safe2* or *Safe4* at additional performance overhead. In most ISAs like ARMv8 or RISCV, there is not supported to indicate a switch between security domains in the same address space (level 4). In this work we assume that the upper bits on the physical address space indicate the time domain.

5.1 Software/ISA Changes

Before starting with the hardware, there is a list of features that must be enabled in the software to provide a secure system when something more than *Spec4* is configured at boot time.

- The TDI assignment with 12 bits is used, with 4 bits for TDI-OS, and 8 bits for TDI-user (Section 2.1)
- The Hypervisor should assign time domains. The OS can request time domain ID changes to the Hypervisor, but only within the assigned pool. (Section 2.1)
- There is a local and a global cycle count counter. All the other performance counters are local per core. The OS has a local and a global cycle count but not global performance counters. (Section 2.2)
- Any control register that may cause a slowdown or measure directly or indirectly the performance in another core is only visible to the Hypervisor. (Section 2.2)
- Any cache management or control register only works over the partition assigned to the time domain. Only the Hypervisor can perform operations that apply to all the SoC.
- If there is a single OS running, the OS has access to all the resources behaving like a Hypervisor from a time domain point of view. The only exception is that it can not assign TDIs besides the assigned by the Hypervisor. In the single-OS case, the Hypervisor bit in the physical address space is set even in the OS, but it is managed by the Hypervisor.
- There are new control register to assign TDI-user/TDI-OS to super pages. This control register is restricted to te Hypervisor.
- The hypervisor can disable dynamic partitioning per core, and assign fix static partitioning per core. Not even the hypervisor can assign a resource partitioning per TID.
- There is new control registers for homogenization (Section 4.2.2). The application marks the beginning and end of the code section to homogenize. The hardware enforces that the execution time for the code section is always the maximum of the value provided by the code instrumentation or the worst recorded execution time. There is a table managed by the Hypervisor that points to all the homogenization code sections and IDs. When the OS or application want a new code section, they must request the new ID to the Hypervisor.

5.2 Hardware Changes

5.2.1 Basic Features

- The system calls can not be performed until they reach a PNR and all the older memory operations are locally performed.
- The dynamic adaptations on the system are delayed randomly to be between 1ms and 2ms intervals. All the SoC pauses to perform whatever adaptation is necessary as explained in Section 2.5. If the adaptation is for a resource that only affects a core like a branch predictor, other cores do not need to pause to perform the adaptation.
- All the instructions by the memory operations and branches have a fix latency to execute. This means that the core implements the Non-Value Dependent Operations (Section 4.1.5). The core implements homogenization for crypto-like engines that require also to enforce branch and memory latencies for algorithms (Section 4.2.2).

5.2.2 Data Caches

The caches are large performance predictors. As such, we should have several techniques to handle them. The system tries to reduce the amount of sharing caches, and avoids inclusive caches. We propose that there should be a private instruction and data cache (DL1, DL2, IL1, IL2) per core. The private L2 can be exclusive or free running, but not inclusive. The shared L3 or last level cache should not be inclusive or exclusive, just free running. Ideally, the shared L3 is just a small cache next to the memory controller and directory. We propose to avoid large inclusive last level caches, and instead use cooperative caching approaches [6] without shared last level caches.

Another difference is that we propose to have a Dirty Clean FSM (Section 4.1.1) to reduce the amount of dirty lines when possible.

Non-speculative load/stores are loads or stores performed when they are beyond the PNR. One significant difference from previous works is the cache miss allocation. Since the caches are not inclusive, on a L1 cache miss, the line is allocated only to the DL1, not the L2 or L3. If the line was already in the L2 or L3, the value can be left there or not, but cold misses go to L1 only.

Speculative loads have a special behavior and different to support *Spec4*.

While in *Spec4*, a speculative load cache miss allocates in the DL1 only if it can displace a clean line with the lowest LRU priority [3]. The allocated speculative line is still kept with the lowest LRU priority. Notice that the lowest LRU priority could potentially displace still speculative line which can be a performance issue. To avoid this case, when the lowest LRU line[s] are still speculative, we propose to allocate the line in the SCB. If the SCB is also full, the load waits until it becomes non speculative (PNR). When loads become non speculative, the associated SCB entry is displaced to the DL1 or the associated DL1 speculative line is marked as non speculative and promoted to MRU or at least increased the priority to avoid being LRU. On context switch or system call return, all the lowest clean LRU lines

3. The solution works if instead of just the lowest LRU, we pick one of the two lowest priority lines, if we always flush the two lowest LRU clean lines.

from the cache are flushed independent of being speculative or not.

This makes speculative loads not to impact the displaced lines on the system. The lower level caches are also not affected because caches are not inclusive.

While in *Safe2* and *Safe4*, speculative loads could behave like non-speculative because the caches are flushed at boundaries. Nevertheless, for verification simplicity we propose to keep the same *Spec4* method enable for speculative loads.

Speculative Stores can trigger prefetches but unlike stores, they can not trigger invalidations in the coherence network, but they can not go to the SCB until they pass the PNR, as a result they are not speculative side effects. The prefech information goes to the SCB line in the overflow case of speculative loads. If the store is flushed, the associated speculative clean entry from the SCB is cleared. The previous mechanism is the same in *Spec4*, *Safe2* and *Safe4*.

System call. The SoC has a different behavior when in *Spec4* or *Safe2/Safe4* mode. In *Spec4* and *Safe1* mode the LRU speculative entries in the cache are invalidated at system call.

To support non interference *Safe2/Safe4*, a partition of the cache is given to the OS at system call. The lines do not change line state on syscall but at system call return. By default, a static policy flushes 25% percent of the LRU DL1 cache unless a different percentage is given to the OS/Hypervisor or a hardware learning mechanism. By default, the OS does not allocate on the DL2 cache. At OS/Hypervisor call, the dirty lines that are going to be flushed are being written back. After system call return, half of the write back BW is given to clear the dirty L1 lines for a given amount of time as indicated in the "in crescendo" technique.

In *Safe2/Safe4* mode, the allocated lines to the OS are flushed on a system call return. The lines flushed are the LRU from each set. For example, a 25% flush in an 8 way means that 2 cache lines per set are flushed at system call return. Flushed means writeback and invalidate.

Per core, there is also a state machine (FSM) to detect frequent system calls. The FSM tracks the amount of instructions in user mode vs system mode. It partitions the caches according to the ratio. The caches switch to partition instead of flush. A fix partitioning is given to all the system calls, and the OS cache lines are not flushed at system call return. The OS portion of the cache lines can not be displaced or even used by the user level application. If there is an OS allocated cache line hit while in user mode, the SoC triggers a fake cache miss and populates the user mode of the cache and invalidates the OS mode. This is because the OS level can not have on the user level, but the user level is allowed to have a time impact on the OS level.

The OS partition mode gets the 25%, 50%, 75%, or 100% of the 64KB 8 way DL1 cache, but it allows to allocate more ways on the DL2 cache. The partitioning decision could be done independently for the DL1 and DL2, but for simplicity a 25% of the DL1 is fixed for OS. The DL2 is a 16 way cache. The OS can have 6.25%, 12.5%, 25%, 50%, 75%, or 100% of the DL2 cache depending on the percentage of instructions executed in the OS vs user level.

Hypervisor calls are supposed to be more infrequent. For Hypervisor, we propose to allocate only 1 way in the DL1 cache and 0 ways in the DL2, the DL1 way is flushed at Hypervisor call return.

Context switch does not have effects when in *Spec4*, it just needs to flush the LRU speculative lines like in the system call return case.

On a *Safe1/Safe2/Safe4* mode, all the clean lines are invalidated, the dirty lines are written back and invalidated, and the user level DL1 MRU lines are off-loaded [4]. The new context switch may have some off-loaded lines (MRU-only), these lines are repopulated (prefetch-like) to the DL1. Both off-load and flush should not have a variable time visible to other time domains.

To avoid the time leak and worst case delay for the flush, we have the Dirty Clean FSM active to guarantee that only a small percentage of the lines are dirty. We also propose to give have of the displace BW for a given period of time after time context switch as indicated in the "in crescendo" technique. The off-load has always a worst case penalty because there are always MRU lines unless there is a too frequent context switch in which case we should be in partition mode and no flushing is necessary.

If there are frequent context switches between a small set of security domains, the SoC can decide to partition the DL2 and avoid DL2 flushes. Since the DL2 can populate fast the DL1, we propose not to do static partitioning on the DL1. A state machine monitors the context switch ratio. If it is high, it switches to partition mode. This is not a reactive but a slow moving average adaptation to avoid time leaks. Similar to the system call case, there is a FSM that monitors the amount of time domains executed in the last 10M cycles. It a partitions the DL2 cache based on the amount of time domains giving a last 12.5% of the DL2. If a time domain executes less than 12.5% of the instructions, does not get any DL2 partitioning.

5.2.3 Instruction Caches

Instruction cache are speculatively updated, but typically they do not use any speculative data/address/pc because the instruction cache is driven by the branch predictor. In this work, the branch predictor is always not speculative.

In *Spec4* mode, the instruction cache behaves slightly different from a traditional instruction cache because there is a corner case when a speculative branch triggers a branch missprediction, but it was incorrectly speculated. In this case the speculative branch triggers a speculative update to the instruction cache. We performed simulations in ESESC, and waiting until the branch becomes not speculative has under 2% performance impact for SPEC. A more aggressive option is to allow to go to the instruction cache, but not to trigger instruction cache misses or LRU updates until the last branch redirection becomes non-speculative (passes the PNR). This solution has even a lower performance impact.

System call does nothing in *Spec4* or *Safe1*.

In *Safe2/Safe4* mode, we propose to mimic the DL1 behavior. We give 25% of the instruction cache to the operating system. If there is no partitioning, the IL1 ways allocated to the OS are invalidated at system call return. The IL1 is a 32KB way 4 way instruction cache. No ways in the IL2 are given to the OS by default.

The same FSM used for the data cache also controls the instruction cache (IL1) and IL2 partitioning. The IL2 is a 128KB

4. If the off-load buffer is too small, a percentage of the lines are off-loaded, the others are flushed. The off-loaded should be selected with a stride pattern.

8 way cache. The IL2 can be dynamically partitioned like the DL2. The same percentages allocated to the DL1/DL2 are given to the IL1/IL2.

Context switch does nothing in *Spec4*.

In *Safe1/Safe2/Safe4* mode, we can not leave any side effect to other applications. In this case, we again propose a similar mechanism to the data cache. The only difference is that the IL1 does not off-load like the DL1. If there are frequent context switches, the IL2 is partitioned like the DL2.

5.2.4 TLBs

The TLBs are different from the caches in the sense that the L2 TLB is inclusive with respect to the L1 TLB. If a line is in the L2 TLB, it can be brought to the L1 TLB speculatively and it is not marked speculative. The data L1 TLB is a 256 entry 2 way associative TLB. The data L2 TLB has 2048 entries 8 way associative inclusive TLB.

Speculative update: In all the modes, we propose to reserve 2 data L1 TLB entries for speculative updates. Once the associated speculative load/store passes the PNR, the entry is promoted as non-speculative and inserted in the L1 TLB and L2 TLB. Similarly, if the load/store is flushed the associated speculative entry is also flushed. If more than 2 entries are required for speculative updates, the associated load/store waits until there are free speculative entries. If the load/store is no longer speculative, it allocates entries directly to the L1 and L2 TLB.

We could allocate lines speculatively without reserving entries in *Safe2/Safe4* modes because the L1 TLB is invalidated at context switch and system call return, but the propose to keep using the same 2 L1 TLB speculative entries to simplify verification.

One thing that must be addresses is the page walker. Traditional page walkers allow the L1 and/or L2 caches to cache recently used entries. The reason is that this allows to improve page walk time significantly. In this work, the page walker can also allocate entries in the cache, but only once the page walk has finished, and it is beyond the PNR.

System call does nothing in *Spec4* and *Safe1* mode.

In *Safe2/Safe4* mode, we propose to in invalidate the whole L1 TLB at system call return, and keep the L2 TLBs contents. This is an overkill because the OS/Hypervisor is unlikely to require all the L1 TLBs. Nevertheless, populating the L1 TLB from the L2 TLB is just a few cycles (2-6) for the first page touch. This is a very low overhead.

The data L1 TLB is fairly large with up to 256 entries, the L2 TLB has by dynamic partition by default. The dynamic partition is explained in the context switch case. Similar to the data cache, there is a FSM that tracks the percentage of instructions in OS vs user. The L1 TLB is never partitioned, but the L2 TLB can.

Context switch does nothing in *Spec4*.

In *Safe1/Safe2/Safe4* mode, we invalidate the whole L1 TLB at each context switch like in the system call returns.

The L2 TLB has a dynamic partition by default. A FSM tracks the last 64 time domains executed per core. It only marks a time domain as executed if it has a L2 TLB miss. Based on this, the dynamic allocation randomly assign more or less sets to a given time domain. The FSM also tracks the ratio in L2 TLB misses of OS vs user. The minimum that can be allocated to OS mode or application is 2 set or 16 entries.

To manage the allocation, for each time domain there is a 128 bit vector indicating the sets allocated. The hash function can only create values for these entries. At dynamic adaptation boundaries, the time domain that run since the last adaptation randomly steals entries from another time domains but between themselves. If there are no other time domains to steal from, they can steal between themselves randomly based on the L2 TLB misses ratio but at a lower rate. Each 1M instructions executed, there can be 2 sets stolen by default, and 2 sets every 2M if no other sets are available. The stolen sets can be dedicated to user level or OS depending on the FSM estimated load. If a time domain has no L2 TLB entries left, it is removed from the last 64 active time domains.

If a new time domain starts to execute, it picks to replace the time domain that has the least entries in the allocation. By default, it starts with 4 sets, which can be stolen randomly from other time domains but not time domains executed since the last adaptation. If there are no other sets available, it can randomly steal from anyone.

5.2.5 Data Cache Prefetcher

There are several types of data prefetching, and each has a different potential solution. As usual with prefetch requests in several cores, prefetches can not trigger an invalidate in the coherence network or force a displacement of dirty cache lines. If any such events were to happen in any cache the prefetch is dropped. This section analyzes 3 typical prefetchers (Stride, Region based, Next Line). In the proposed design, the stide prefetcher has 256 entries. The region based prefetcher has 8 learning entries, and 128 learned entries.

5.2.5.1 Software Prefetching: is processed in *Spec4* mode only once the associated instruction passes the PNR.

Having software prefetching complicates the *Safe1/Safe2/Safe4* modes. As a result, we propose to drop all the software prefetches in those modes.

5.2.5.2 PC Stride Prefetchers (Stride): can be updated at retirement and it does not degrade the prefetcher quality. Simulation based results show a noise level impact between update at execute and update at retirement. Some processors update predictors at retirement to avoid the pollution of the wrong path and to guarantee an in-order update. Other processors handle the updates at execute to avoid remembering the full address until retirement. Both solutions are reasonable for performance. For security, it is better to update at retirement to avoid speculative updates (Section 4.1.6)

The prefetch information should be sent when the associated load/store reaches execution. To avoid leaking the speculative address, the prefetcher can use the last retired address and increase the stride delta by the number of same PC loads between the executing load and retirement. Extra tables can track this distance making the speculative prefetch request not to use any speculative data, and therefore safe for *Spec4* information leak.

System call does nothing special for *Spec4* or *Safe1* mode.

In *Safe2/Safe4* mode, the stride prefetcher tables learned at the OS should not affect the user level. To do so, we propose to reserve 1/8 of the stride prefetcher for OS. This avoids the need to flush. The reason is that the OS can significantly benefit from stride prefetcher due to memcopy/clear behaviour. Clearing entries at system call return would affect

the learning. As expected, the user level prefetch entries are not trigeered while in OS mode, and viceversa. Stride prefetchers do not need many entries and going from 256 to 224 has barely any performance difference.

Context switch does nothing special for *Spec4*.

In *Safe1/Safe2/Safe4* mode, we propose to off-load the last 16 stride prefetcher used entries at context switch. If the new user level application share the same OS, the OS portion of the prefetcher tables are not flushed. The non re-loaded entries are cleared to zero.

5.2.5.3 Region Based Prefetcher (SMS): are more aggressive prefetchers, and they tend not to prefetch to the L1, just the L2 or L3. The SMS [24] is a type of spatial prefetcher. Those prefetchers monitor just L1 misses to learn that a region of code has a frequent update pattern. When the region is significantly used, it prefetches the rest of the region with a given learned pattern. Those predictors have two phases: Learning and Prefetching. The learning can be done at retirement without any overhead. The prefetching is better to be done at execute, but this will leak speculative path address.

To support *Spec4*, we propose to learn at retirement, and to use the load/store address once it has passed the PNR (Section 4.1.9).

System call does nothing special for *Spec4* or *Safe1* mode.

In *Safe2/Safe4* mode, the prefetcher can be triggered at execute as long as it does not displace dirty data because the cache policy would be cleared at context switch and system call return.

Like in the stride prefetcher, 1/8 of the stride prefetcher is reserved for OS.

Context switch does nothing special for *Spec4*.

In *Safe1/Safe2/Safe4* mode, the Region based prefetcher behaves similar to the stride prefetcher. The last 8 used entries from the prefetch table are off-loaded. These entries can be OS or non OS. The other entries are invalided.

5.2.5.4 Next Line Prefetcher (NLP): has several similarities to the region based prefetcher but it does not have learning phase. The plan is to delay the next line prefetch until the associated load reaches the PNR.

The NLP does not have learning tables. As a result, there is no state to clear/off-load at system call or context switch. In *Safe1/Safe2/Safe4* mode, the plan is to allow the prefetcher to go in speculative mode as long as it does not displace dirty data in the cache.

5.2.6 Instruction Cache Prefetcher

If the instruction prefetcher uses data from the retired or safe instructions, it is safe to prefetch in *Spec4* mode. For example, the TLB entries can be used to prefetch in the instruction cache because they are updated at retirement. It is not OK to use the front-end RAS, but OK to use the retirement RAS for instruction prefetching. It is also OK to use a next line prefetching in *Spec4* mode because the addresses are as a result of non-speculative addresses.

In *Safe1/Safe2/Safe4* mode, it is OK to use any speculative or non-speculative information to prefetch in the instruction cache because the instruction cache contents would be flushed before switching to another security domain. For verification simplicity, we propose not to speculative update the instruction cache even in those safe modes.

5.2.7 TLB Prefetcher

We plan to have the same operating all the modes for simplicity. We propose to perform a page walk to populate the TLB for speculative load/stores. When the associated load/store that triggered the TLB entry reaches the PNR, the entry is promoted to non-speculative and it is pushed to the L2 TLB if it was not there. If the load/store is flushed the speculative entry and associated page walk is killed. This is the same mechanism as used by speculative load/stores in the processor. The system has two page walks, the speculative page walk can be killed if needed.

To avoid page walk traffic, the prefetcher can allocate the entry and the consecutive prefetch entries on each page walk miss. In this case, the consecutive entries may be allocated to the L2 TLB. If such support is implemented, the L2 TLB allocation would happen only after the associated load/store passes the PNR.

5.2.8 Coherence and Directory

The L1 and L2 are not inclusive so the coherence is not an issue with other cores. The directory has static partitioning like LLC papers do (Section 4.1.3). The idea is that 1/n of the directory belongs to a given core. Then, cores do not pollute each other. When the same security domain is in different cores, the directory shares entries between the these cores.

If a invalidation comes from one security domain to another security domain with request to modify, the directory should raise a protection warning. If two security domains share write pages, the isolation is compromised. The directory will invalidate and work on the other domain over those shared addresses as a traditional directory but it would not provide time leak protection. If the request for sharing across domains is a read-only, it will replicate the directory entries, and *Safe4* isolation would be guaranteed.

One counter intuitive behavior of the partition design is that there can be sharing between two time domains. Even if this happens, the directory would double allocate the entries, and trigger the same performance as a directory miss even though it request the other line to share. This is possible because the directory miss (memory access) is much slower than a coherence management in the directory.

5.2.9 NoC Bandwidth

In this work, the network on chip (NoC) is the interconnection network between the caches and the memory controller. Cores can affect each other performance by creating a high NoC traffic demand. The proposal is to have NoC with quality of service implementing a Fair Weighted Queue system per router with a static partition were each active core has the same BW allocated. This QoS has show a great isolation between network requests.

If cores are idle, only 1/8 of their bandwidth is reserved. The dynamic adaptation changes the bandwidth assigned at course boundaries. This would mean that a time domain would get $\frac{1}{8*n}$ of the system bandwidth were n is the number of cores from start until the first adaptation. To compensate and boost starting cores, we give $\frac{15}{8*n}$ during the second adaptation, and switch back to $\frac{1}{n}$ afterwards.

NoC with strong QoS guarantees waste bandwidth, but we think that this is not an issue. The NoC BW is much higher than the memory BW. If most of the cores share data actively, they should have the same time domain. As such,

they do not waste bandwdith and get a larger chunk of the NoC system. If the apps have different TIDs without data sharing, they are mostly talking to the memory, as such the NoC BW is over provisioned. The result is that strong NoC QoS guarantees should not have significant performance impact as long as they just affect BW and not latency.

5.2.10 Memory Controller

The memory controller has a small cache associated. This LLC-like cache is statically partitioned by the number of cores.

Besides the small cache associated to the memory controller, the memory controller also manages open/close pages. A FSM tracks the number of page misses per time domain, and it assigns open page rights to each time domain based on the miss rate request. The dynamic adaptation is not very frequent, but not avoid rushes, each dynamic adaptation only 2 more open page can be assigned to a time domain at most. At least, there is one page right per core.

If a core sends a new time domain request, and it does not have any open page right, the memory controller can always flush invalidate all the entries in that core and assign those to the new time domain.

One thing that also must be done is to not leverage other time domain open pages. If a core time domain has an open page, and another tries to access the same physical page, it should trigger a fake open page request or charge an equivalent delay to the request. As in the case of caches, there can be prefetching between the application time domain, but not across time domains. The prefetcher across time domains are dropped.

One difficult source of time isolation is the DRAM Rowhammer time leak [20]. Rowhammer means that adjacent DRAM pages can result in data corruption. To avoid it, systems can slowdown an memory request if two adjacent pages access are too frequent. Here the challenge is that one time domain may

> overload a Rowhammer page which can affect the performance of another page used by another time domain. As a high level explanation, in Rowhammer a page and the adjacent page can be accessed "n" times per second at most.
>
> The proposal is simple, just allow at most $\frac{1}{2*n}$ request per time domain. This can slowdown the worst case for a given application, but since there are 2 adjacent pages it guarantees not overload even when both time domains are working on close by pages.

5.2.11 Memory Dependence Predictors

> The Memory Dependence Predictor is a structure that typically benefits from frequent flushes to avoid learning too much. The MPD entries are only allocated at retirement. As such it does not need changes for *Spec4* mode. In our MPD, we track the last 64 loads that triggered a NUKE in the system. In *Safe1*/*Safe2*/*Safe4* mode, we propose to reserve 4 of the entries for the OS. We propose to always flush all the MPD at context switches.

5.2.12 Taken/Not-Taken Branch Predictors

The plan is to implement a TAGE branch predictor.

In *Spec4* mode: The Global History Register or path history used by the global predictors must be updated speculatively. This GHR is corrected on branch miss predictions in current processors leaving no side effects. Current cores either checkpoint or create a shift register for the GHR. As a result, current processors already have a perfect fix for speculative updates. Current processors implemented some Fetch Target Queue (FTQ) that insert entries at fetch, and deallocate at execute. The overhead of fixing at retirement is that the entry must be kept from fetch to retire instead of execute. Based on ESESC simulations from a typical 4-way OoO core, the FTQ must increase from 48 entries for an ARM A72-like core to something like 64 entries.

Most processors update the prediction tables at execute because it guarantees that only as many updates as branch ALUs ports are needed. Updating at retirement does not degrade prediction accuracy, but it requires to handle the case that multiple retiring instructions could be branches. If the retire width is the same as the fetch width, there is a guarantee of only one update at retirement. As such, we propose to update the branch prediction taken/not-taken tables at retirement.

System call copies and restores the GHR at system call. The OS and user level mode do not share a GHR. Besides this, system calls do nothing special in *Spec4* and *Safe1*.

In *Safe2*/*Safe4* mode, we update at retirement as required by *Spec4* mode, but we also dynamically partition the branch predictor for the OS/Hypervisor. As in the case of the caches, we propose a dynamic partitioning that learns per system call and has a slow moving average to adapt. We plan to use the same FSM as in the data cache, but to have 4 partition options for OS.

If there are no frequent system calls, the OS gets 6.25% of the PHT entries, and those entries are invalidated at system call return. If the cache is partitioned, the OS still gets the same 6.25%. If a new OS time domain is called, the OS is invalidated.

If there are frequent system calls, the data cache FSM can find the percentage of OS. The OS dynamic partition can have 6.25%, 50%, or 100% of the PHT entries of the TAGE and bimodal. In each case, the OS/Hypervisor has a hash function that restricts the usage on the PHT tables to only a subset of the PHT entries. During partition mode, a system call return does not flush the OS entries but it does not allow the user level application to access to entries.

Contet switch the GHR is reset to the PC starting the context. It does nothing else special in *Spec4*.

In *Safe1*/*Safe2*/*Safe4* mode, we propose to have a partitioning mechanism were the hash function allows to divide the PHT entries. Unlike the TLB that tracks 64 partitions, we propose to have 1 or 2 partitions. Since there i a 6.25% reserved for OS all the time, this means that in one partition mode, the user level gets 93.75%. In two partition mode,

it gets 46.875%.
By default, there is a single partition. In this case, whenever there is a context switch, the PHTs switch to a 2 partition mode. 46.875% of the entries are invalidated, and the new context executes in the new partition. If there is no context switch back to the original time domain, the whole PHT is allocated to the new time domain after the dynamic adaptation.
If there are frequent context switches, the PHT switches to 2 partitions, keeping ithe 46.875% of the PHT to each time domain. If a new context is executed while 2 partitions exist, the oldest time domain is invalidated.
To avoid flushing and relearning when partitioning, we propose to keep the same hash function in all the partition modes. The only difference is that the least significant bit may be force to 0/1 with 2 partitions, or allowed to be selected by the hash function. As a result, the non partitioned entries can be kept when switching in/out of partitioning.

5.2.13 BTB

BTBs are large like the Taken/Not-Taken branch predictor tables. As such similar update policies apply to them. Namely, the BTB should be updated at commit when the in-order as control flow instructions (CFI) pass the point of no return (PNR). Current processors tend to update the BTB at execute. The advantage is that there is a guarantee of only as many updates as branch execution units, and that the BTB port is free. A straightforward implementation updating the BTB at commit and require one read port for prediction and one write port for commit. This solution works without degradation of performance, but requires extra ports in the BTB. The T/NT Branch Predictor tables had a similar problem, and we suggested to have banking. The difference is that a 512 entry BTB needs updates less than 3% of the cycles.
The same update at commit point policy is recommended for BTBs. Current cores have 2 or more levels of BTBs. The smaller but faster BTBs are speculatively updated with the larger more correct BTBs. This BTBs can be exclusive, inclusive, or free running. Although modern cores update at decode, the prediction accuracy does not degrade for updating at retirement. The cost is that the BTB update information must be preserved until retirement, this means that the FTQ must also keep BTB information. Since the PCs are already in the FTQ, the additional information is small.
In *Safe1/Safe2/Safe4* mode, we propose to follow the same breakdowns and mechanism as in the PHT predictor.
A simpler but less performing solution is to speculatively update the BTBs but flush the whole BTB at system call returns or context switches. This is an interesting approach for small BTBs.

5.2.14 RAS

In *Spec4* mode, we propose to have a RAS at front-end and another safe at retirement.
The return address stack (RAS) must be updated at fetch. Unlike other predictors the update can not be delayed to retirement. Nevertheless, a solution to to fix the RAS recovery is to have 2 RAS: One a fetch, and another at retirement.
If there is a branch miss prediction, the speculative path can incorrectly update the RAS. The pipeline flush will copy the retirement RAS to the fetch RAS. In this way, speculative updates have no side effects.
Although this approach has some overheads, some SPARC processors implemented this because it improves the RAS accuracy avoiding to have overflows for speculatively executed instructions.
To handle the branch misspredictions, there can be a retirement RAS and a front-end RAS. Each fetch boundary, the front-end RAS pointer is checkpointed in the FTQ. When a branch is recovered, the checkpointed RAS pointer is recovered, and the contents from the retirement RAS are copied to the front-end RAS. Notice that in some cases the recered branch may still not be at retirement. In these cases, not the whole RAS is copied. The entries between the retirement RAS pointer and the recovered RAS checkpoint are not copied. The reason is that the retirement RAS still does not have those updated values. Alternatively, a state machine could traverse starting from the retirement RAS and update the front-end RAS. If the front-end tries to use a RAS entry before the checkpoint is created, we can either stall fetch. Stall should be better than using a bad prediction because it is very likely to be a miss predict.
Not related to timing attack is the ROP [22] attach. The idea is that the function call return is changes to address some sections of code and allow to execute malicious code. Having a non-speculative RAS can help to reduce the ROP attacks even in non-speculative cases. If there is a return, and the RAS and code in the stack does not match, the application may be suffering a ROP attack.
System call does nothing special for *Spec4* and *Safe1*.
In *Safe2/Safe4* mode, system calls reserve 8 entries in the RAS. If the OS had a perfect behavior, we could allow the RAS to perform as usual, and invalidate the OS used RAS entries at function returns. Instead, we propose that at system call 8 entries from the RAS (bottom) are allocated to the OS. The OS wraps around only in this 8 entries. At system call return, the 8 entries are invalidated.
Context switch does nothing for *Spec4* mode
For all the other modes, we propose to invalidate the RAS at context switch. If the context switch are frequent, we could off-load the top 2-4 entries in the RAS.

5.2.15 SMT

Symmetric Multithreading (SMT) is the technique used by several cores to share a single core between two different processes. This is a special challenge for side channel time leaks. Cores share resources like predictors but also other resources like issue

logic, physical register file. As a result, isolating two threads in the same core is a very difficult task unless a fix partitioning of core resources is assigned to each thread. The partitioning can change at run-time, but it should not be the result of specific instructions but trends.
Solving the SMT problem is even more challenging. Partitioning should work but it would have a potentially significant performance impact. Even more complicated may be the bandwidth and resource constraints outside the core must be tagged per core thread. This is an area that needs more research to find acceptable solutions.
The current state-of-the-art solution is that there are soo many issues that the SMT contexts should belong to the same security domain because there is no way to avoid time leaks without a significant remodeling and most of the SMT resources are statically partitioned or dynamically partitioned with a slow re-partitioning which is not necessarily the most efficient way to handle SMT.

5.2.16 Runahead Mechanism

Runahead [18] introduces challenges to the speculative execution. Now, the core executes several incorrect instructions. Applying Spectre ideas to a SoC with runahead, the attacker can force the core to execute bad code and to have some side channel side effect like the caches.
If we just do not update on speculative run-ahead mode, we have none of the benefits of run-ahead. If the processor wants to enable a run-ahead, the simplest solution is to have also enabled the *Safe2* and *Safe4* mode. Then, speculative updates have no side effect.

5.2.17 Data Value Predictors

If the core has data value predictors, the simplest way is not allow to pass this information across security domains. Value prediction can work in *Spec4* mode if the value prediction tables are flushed at system call return and/or context switch and updated at retirement. The load/stores/branches can still use value prediction speculative data and being managed by the *Spec4* mechanisms.
If the SoC is in *Safe4* speculative accesses can be allowed too.

6 Checking Attacks

SPectre/Meltdown: Both Spectre and Meltdown get protected by a *Spec4* protection layer. The reason is that speculative execution avoids time leaks. *Spec4* does not flush or invalidate resources at context switch which is a good idea to avoid interference. If only *Spec4* is implemented, the logical thing would be to use a process id in all the branch predictor tables tags in BTB and TAGE. Nevertheless, this is not needed. If a *Safe2* is provided, all those resource sharing are avoided and further security is achieved.
Floating Point: Floating point variable execution time can be exposed and leveraged to do some attacks [2]. This attack measures the time to perform an SVG filter in a browser, and as a result it can infer the pixel values in another website. As a result of this type of attack, browsers like Firefox have switched to use fix point operations instead of floating point ones. Nevertheless, a similar type of attack is possible. For our classification, this is a "data" dependent time leak because the data value affects the execution time of the floating point operation. *Spec4* would not protect against this attack. Since this attack is performed by another thread, only *Safe4* would provide protection. But to be specific about this attack, the non-value time dependence (Section 4.1.5) implementation is enough to protect against this specific attack.

6.1 Coordinated Attack on Bad Crypto

To understand the limits, this shows a potential attack that even *Safe4* can not protect from. Imagine a bad crypto engine that checks the "1234" password very slowly, and all the other passwords very fast. This code runs in a server or security domain that performs this check and nothing else.
One thing is that the attacker should not be able to see the time that it takes to call the password check unless it performs the attack. Since the fast/slow interaction would not be detected by the attacker time domain, there should be no way to leak this. The only potential attack is for the attacker to perform this attack would be if the password check is connected with some IO operation. In this work, we have not extended to protect the IO channels with QoS. If the IO is not included, and the attacker can detect the interference, it may be able to infer the encryption key. If the password/checker can be used by the network, even a QoS may not be able to protect as the attacker may be in a man in the middle router.
From the techniques section, the only effective way to protect against this attack is to have the code instrumented with homogenization support (Section 4.2.2) and the system to implement a *Safe4* protection level. Then, the hardware would enforce that the fast check would be slow too.

References

[1] B. Alomair, A. Clark, J. Cuellar, and R. Poovendran, "Toward a statistical framework for source anonymity in sensor networks," *IEEE Transactions on Mobile Computing*, vol. 12, no. 2, pp. 248–260, 2013.

[2] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 623–639.

[3] O. Berthold and H. Langos, "Dummy traffic against long term intersection attacks," in *International Workshop on Privacy Enhancing Technologies*. Springer, 2002, pp. 110–128.

[4] S. Bleha, C. Slivinsky, and B. Hussien, "Computer-access security systems using keystroke dynamics," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 12, no. 12, pp. 1217–1222, 1990.

[5] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau, "Cava: Using checkpoint-assisted value prediction to hide l2 misses," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 3, no. 2, pp. 182–208, 2006.

[6] J. Chang and G. S. Sohi, *Cooperative caching for chip multiprocessors*. ACM, 2006, vol. 34, no. 2.

[7] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation." in *USENIX Security Symposium*, 2016, pp. 857–874.

[8] J. Deng, R. Han, and S. Mishra, "Countermeasures against traffic analysis attacks in wireless sensor networks," in *Security and Privacy for Emerging Areas in Communications Networks, 2005. SecureComm 2005. First International Conference on*. IEEE, 2005, pp. 113–126.

[9] G. Hamerly, E. Perelman, and B. Calder, "How to Use SimPoint to Pick Simulation Points," in *ACM SIGMETRICS Performance Evaluation Review*, 2004.

[10] M. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to EnergyReduction," in *International Symposium on Computer Architecture*, San Diego, California, Jun. 2003, pp. 157–168.

[11] IEEE Standards Board, "Draft Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," Institute of Electrical and Electronics Engineers, Technical Report IEEE 1588 - 2008, 2008.

[12] E. Kim, D.-I. Lee, H. Saito, H. Nakamura, J.-G. Lee, and T. Nanya, "Performance optimization of synchronous control units for datapaths with variable delay arithmetic units," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. ACM, 2003, pp. 816–819.

[13] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[14] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[15] W. Liu and M. Huang, "EXPERT: Expedited Simulation Exploiting Program BehaviorRepetition," in *International Conference on Supercomputing*, St. Malo, France, Jun.–Jul. 2004, pp. 126–135.

[16] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," in *International Symposium on Microarchitecture*, Nov 2002.

[17] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *International Conference on Information Security and Cryptology*. Springer, 2005, pp. 156–168.

[18] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windowsfor Out-of-order Processors," in *International Symposium on High-Performance Computer Architecture*, Anaheim, California, Feb. 2003, pp. 129–140.

[19] A. Perais and A. Seznec, "Eole: Paving the way for an effective implementation of value prediction," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 481–492.

[20] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks." in *USENIX Security Symposium*, 2016, pp. 565–581.

[21] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–11.

[22] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[23] M. Shao, Y. Yang, S. Zhu, and G. Cao, "Towards statistically strong source anonymity for sensor networks," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008, pp. 51–55.

[24] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 252–263.

[25] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh." in *USENIX Security Symposium*, vol. 2001, 2001.

[26] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 299–310.

[27] A. Tang, S. Sethumadhavan, and S. Stolfo, "Clkscrew: exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium*, 2017.

[28] B. Timmerman, "Secure dynamic adaptive traffic masking," in *Proceedings of the 1999 workshop on New security paradigms*. ACM, 1999, pp. 13–24.

[29] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.