Full-Processor Timing Channel Protection with Applications to Secure Hardware Compartments

Andrew Ferraiuolo, Yao Wang*, Rui Xu, Danfeng Zhang, Andrew Myers, and G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
andrew@csl.cornell.edu, yao@csl.cornell.edu, rx37@cornell.edu,
zhangdf@cs.cornell.edu, andru@cs.cornell.edu, suh@csl.cornell.edu

Abstract

This paper presents timing compartments, a hardware architecture abstraction that eliminates microarchitectural timing channels between groups of processes of VMs running on shared hardware. When coupled with conventional access controls, timing compartments provide strong isolation comparable to running software entities on separate machines. Timing compartments use microarchitecture mechanisms to enforce timing sensitive noninterference, which we prove formally through information flow analysis of an RTL implementation. In the process of systematically removing timing interference, we identify and remove new sources of timing channels, including cache coherence mechanisms and module interfaces, and introduce new performance optimizations. We also demonstrate how timing compartments may be extended to support a hardware-only TCB which ensures security even when the system is managed by an untrusted OS or hypervisor. The overheads of timing compartments are low; compared to a comparable insecure baseline, executing two timing compartments reduces system throughput by less than 7% on average and by less than 2% for compute-bound workloads.

1. Introduction

Timing channel attacks have become a major threat as hardware is increasingly consolidated and is shared by distrusting entities. For example, in cloud computing, mutually distrusting parties own virtual machines on shared hardware. Mobile device users download untrusted applications that share hardware with other sensitive software. The access-control based isolation provided by processes and virtual machines limits explicit communication. However, timing channel vulnerabilities in the microarchitecture allow attackers to subvert these boundaries even when the OS and hypervisor are bug-free. Physical side channels such as power consumption are dangerous only when adversaries have physical proximity, but timing channels can be exploited by remote adversaries.

For example, researchers have shown that secret keys can be extracted from co-resident VMs on production EC2 servers by exploiting microarchitectural timing channels [37]. Hardware-level timing channels have also been shown for many shared hardware components, including caches [36, 7, 10, 35, 47, 3, 43, 10, 28, 22], branch predictors [1, 2], interconnects [50, 54], pipelines [52], and memory controllers [49, 21]. The wide range of vulnerable components suggests that a comprehensive method is needed for protecting hardware resources against timing attacks.

To achieve this goal, we propose *timing compartments* (TCs), a hardware-enforced abstraction that isolates distrusting software by eliminating timing channels through shared hardware components. Timing compartments use microarchitecture protection mechanisms to provably enforce timing-sensitive noninterference. When combined with access controls which prevent attacks that do not exploit timing, timing compartments provide isolation that is comparable to running each compartment on a separate processor.

Timing compartments use static spatial and temporal partitioning to completely eliminate, not reduce, timing channels through shared hardware. Sophisticated attacks [24] that succeed despite noise have shown that obfuscation techniques are insufficient to prevent timing channels. Dedicated attackers can make repeated measurements and use statistical analyses to filter out noise.

Systematically applying temporal and spatial partitioning to a modern, multicore-architecture revealed new timing channel vulnerabilities. Notably, we show that cache coherence mechanisms cause timing channels even among processes with no shared data, and propose modifications to remove this vulnerability.

We formally verify that timing compartments enforce timing-sensitive noninterference using an information-flow analysis approach. We implemented a pipelined, multicore processor with the key microararchitectural enforcement mechanisms of timing compartments in SecVerilog. SecVerilog [60] extends Verilog with a type system that controls information flow.

Timing compartments provide timing isolation even when the OS/hypervisor is compromised. Efficiently defending against a malicious OS/hypervisor requires interfaces which allow it to manage resource allocation without leaking information. In particular, we address timing channel attacks through page faults, including the attack proposed by Xu et al. [57].

*The first two authors contributed equally to the work.

Experiments show that naively applying temporal and spatial partitioning has a significant performance overhead. We propose several performance optimizations that improve performance without weakening the security guarantee. First, coordinated scheduling reduces the latency of memory requests which are handled by multiple, temporally partitioned resources. By considering the interaction between these resources when scheduling requests to use them, coordinated scheduling reduces the average L2 miss latency by up to 62% compared to a greedy schedule. Second, we propose a novel optimization which improves the available bandwidth through a temporally partitioned memory controller. Since our experiments show that memory bandwidth reduction is the greatest performance overhead, this optimization improves performance considerably. Finally, we show how resources can be allocated according to the resource demands of the applications without leaking information.

Simulation results suggest that the performance overhead of timing compartments is quite low. Compared to an insecure baseline, executing two timing compartments with the aforementioned optimizations reduces system throughput by less than 7% on average and by less than 2% for compute-bound workloads. The optimizations reduce the average overhead by 58%.

The following summarizes the main contributions

- We propose Timing compartments, a new abstraction that enables software to explicitly remove microarchitecture-level timing interference in a multicore processor, and show that strong protection is viable.
- We identify new timing channels including a vulnerability through cache coherence protocols which may leak information even among processes which do not share data.
- We propose novel optimizations including time-multiplexing coordination, memory controller dead time relaxation, and application-aware resource allocation, which significantly reduce overhead.
- This paper shows how full-processor timing isolation can be enforced when the OS/hypervisor cannot be trusted. This enables timing channel protection for secure hardware compartments such as Intel SGX [25].
- We formally verify an RTL implementation of a timing-compartment-enabled 4-core processor with a shared cache, ring network, and memory controller. Information flow analysis with SecVerilog [60] proves that timing compartments enforce timing-sensitive noninterference.

The rest of the paper is organized as follows. Section 2 introduces timing compartments and presents example applications that can be enabled by strong timing isolation. Section 3 identifies the sources of timing channels in a multi-core processor, and describes protection mechanisms to eliminate them. Section 4 presents the performance optimizations to make timing compartments practical. Section 5 extends the timing compartment for cases with an untrusted OS. Section 6 evaluates the proposed architecture. Section 7 discusses related work, and Section 8 concludes the paper.

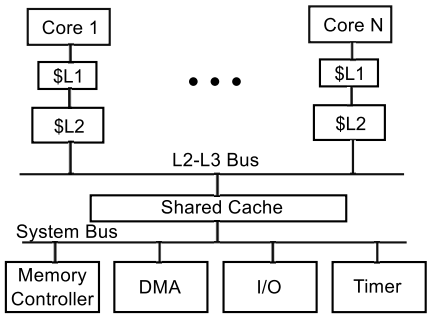

Figure 1: Baseline multi-core architecture.

2. Timing Compartments

2.1. Objective and Scope

The goal of timing compartments is to provide strong microarchitecture timing isolation among multiple software components that share a multi-core processor. Timing compartments aim to eliminate timing channels that are not present among software modules running on dedicated processors.

Timing compartments ensure that the timing of a program in one compartment is independent of program behavior in other compartments. They prevent both intentional (covert-channel) and unintentional (side-channel) information leaks between different timing compartments. However, timing compartments do not remove timing dependence within one compartment. For example, Bernstein's attack [7] showed that an AES key in OpenSSL can be extracted by observing timing variations that depend on cache interference among memory accesses within one program. These timing channel vulnerabilities exist even if a program runs on its own dedicated hardware. Since this is an orthogonal problem, they are not prevented by timing compartments. Language-level techniques have been developed to mitigate [5, 58, 59] these timing channels.

Similarly, since timing compartments aim to eliminate hardware-level timing interference that cannot be eliminated in software, they do not address timing channels introduced at the software implementation level. If necessary, software-level timing channels can be handled separately in software. For example, there has been recent work on preventing OS-level timing channels (e.g., [6]).

Timing compartments allow software to explicitly control hardware-level timing interference among groups of software entities, without enforcing any restrictions within each compartment. Handling timing channels separately from traditional isolation abstractions such as virtual memory allows the overhead of timing channel protection to only be incurred when necessary.

2.2. Architecture Model

Figure 1 a conventional multi-core architecture that is assumed as the baseline in this paper. The architecture has multiple

cores, each with one or more private caches (L1 and L2). The cores are connected to a shared cache (L3) via an on-chip bus. A shared system bus connects the shared cache to a memory controller that manages requests to main memory as well as other system components such as a DMA engine, a timer, and I/O modules. Bus interconnects are used to model on-chip networks. The general approach and findings should apply to other types of interconnect networks as well.

2.3. Threat Model and Assumptions

Our threat model focuses on software attacks from one timing compartment to another. We assume that attackers do not have physical access to the system, and do not consider physical attacks such as ones that tamper with off-chip memory buses or physical side-channel attacks through power consumption or electromagnetic emission. If physical security is required, the proposed timing compartments can be combined with existing off-chip memory protection techniques [42, 39, 20].

We also assume that explicit communications between different timing compartments are prevented using traditional access control mechanisms such as virtual memory. Therefore, timing compartments should have separate address spaces and do not share physical pages except for read-only pages that contain instructions or libraries. There is no point in timing isolation if explicit communication is allowed.

This paper addresses two threat models for privileged software such as an OS or a hypervisor. In traditional systems, the privileged software is trusted and manages protection mechanisms such as virtual memory. In this case, the privileged software is also trusted to manage timing compartments. We use this as the baseline threat model. To apply timing compartments to recent secure processor technologies, we also consider a threat model where the OS or a hypervisor is untrusted. In this case, the timing compartment is extended to allow the untrusted OS/hypervisor to allocate resources while guaranteeing timing isolation and also remove information leaks to the untrusted OS/hypervisor.

2.4. Application Scenarios

The ability to remove timing channels between software modules significantly increases the level of assurance in diverse application domains where distrusting entities share a physical system.

High-Assurance Cloud Computing. In IaaS cloud computing, a tenant may share hardware with competitors or attackers that want to extract sensitive data. Conventional virtualization technologies restrict explicit communication channels among virtual machines, but cannot control timing channels. A practical timing channel attack has been exploited in commercial clouds to extract cryptographic keys [37]. Timing compartments can enable high assurance cloud computing by ensuring that there is no software-level communication channel among virtual machines.

Untrusted Software. The ability to completely eliminate timing channels enables timing compartments to keep information contained even when software is potentially malicious. For example, smartphone users download third party applications that cannot be trusted to manage private or sensitive data. A system may sandbox an untrusted application and restrict its communication channels when it accesses sensitive data. However, access control mechanisms cannot prevent the untrusted application from intentionally leaking information through covert timing channels. These sandboxes can be extended with timing compartments to provide complete isolation that includes timing.

Safety-Critical Systems. Aside from timing channel protection, the capability to control interference in shared hardware can also be used to provide timing guarantees in safety-critical systems. For example, hard real-time systems such as automotive controllers must perform computations within a strict deadline. Unfortunately, multi-core processors cause interference that makes timing guarantees difficult to meet. Timing compartments can be used to ensure that the timing of safety-critical components is not affected by the rest of a system.

3. Full-Processor Timing Isolation

3.1. General Protection Approach

Timing channels exist whenever an adversary in one compartment can correlate the timing of its event to a program behavior in another compartment. As a result, any program-dependent interference in shared hardware resources between timing compartments may lead to timing channels. For this reason, timing compartments remove timing interference rather than mitigating it.

Timing interference between compartments may happen either within each core and its private caches or in the shared memory hierarchy. To remove timing channels through each core's private resources (such as TLBs, private caches, branch predictors, and pipelines) hardware enforcement mechanisms ensure that at most one timing compartment can be allocated to a core at a time. SMT is restricted so that only processes within the same timing compartment can share a core. A similar approach is already taken in production EC2 servers which disable SMT to prevent timing channel attacks [62]. Also, the state of private core resources is flushed on a context switch to a different timing compartment. Section 3.3 discusses details of secure compartment switches.

To eliminate timing channels in the shared memory hierarchy, timing compartments are tracked with each memory request, and enforcement mechanisms remove interference among compartments. Each core has an active timing compartment register (ATC) which indicates the timing compartment ID (TCID) of the TC that is currently executing on that core. The value stored in the ATC is appended to each memory request. The following subsection discusses protection

mechanisms for resources in the shared memory hierarchy.

3.2. Timing Isolation in Memory Hierarchy

To remove timing channels through the shared memory hierarchy, timing compartments rely on static resource partitioning to remove contention. *Temporal partitioning* resolves contention by time multiplexing resources among timing compartments with a fixed schedule. *Spatial partitioning* removes contention by duplicating or partitioning a resource for each process. Both approaches ensure that resource utilization for one compartment is independent of resource demands from other compartments.

In addition to partitioning, *obfuscation* (or *randomization*) is another approach to remove or mitigate the correlation between secrets and contention. For example, cache lines can be evicted [53] or inserted [31] randomly, reducing the correlation between timing and which address was accessed. However, noise may be removed with statistical analysis with a large number of samples. As the goal of this work is to investigate strong timing isolation, we focus on spatial and temporal partitioning.

3.2.1. Shared Caches Static cache partitioning [36] eliminates cache interference among timing compartments by allowing a cache block to replace only entries owned by the same timing compartment. The TC architecture adopts way partitioning [40] in which each cache way is allocated to one timing compartment.

A set of control registers, called the cache partition control registers (CPCs), associate a TCID with each way. On a cache access, only entries in ways owned by the corresponding TC are checked or evicted. By changing these registers, management software can adjust the number of ways allocated to each TC.

While there exist other protection approaches to mitigate cache timing-channel attacks, timing compartments use way partitioning to eliminate timing channels. For example, RP-Cache [53] and Random Fill cache [31] obfuscate cache timing by randomizing cache replacements and insertions respectively, but does not hide the number of cache accesses which can be used for covert-channel attacks. NoMoCache [18], partitions some cache ways but allows interference in other cache ways to reduce timing channel capacity, but does not remove cache timing channels. Since the threat model of timing compartments requires complete elimination of timing channels, none of these three approaches are suitable. PLCache [53] eliminates timing channels by locking cache blocks that may contain secrets in the cache and prevents them from being evicted even through other accesses by the same program. However, this approach targets timing channels caused by interference within a program, which are present even without shared hardware.

In addition to contention for cache arrays, shared ports and MSHRs also require protection. These have not yet been described in the literature.

MSHR Contention. Contention for miss status holding registers (MSHRs) in non-blocking caches can lead to timing channels. The number of outstanding cache misses that the cache can tolerate depends on the number of MSHRs. Once all MSHRs are occupied, the cache will stall on a miss resulting in increased latency for cache accesses. Therefore, shared MSHRs may cause a timing channel. To remove MSHR contention, disjoint sets of MSHRs are allocated to each timing compartment.

Response Port Contention. Cache ports cause another timing channel yet to be discussed in the literature. Conventional caches have CPU-side ports and memory-side ports which are each split into request and response ports. However, each port can only service a single response/request at a time creating timing channels through contention. Similarly, a shared queue that buffers responses at the port may also lead to timing interference. To remove this timing channel, the cache ports are time multiplexed and the shared queue is partitioned into per-compartment queues.

3.2.2. On-Chip Interconnect As in previous work [54], time multiplexing protects the on-chip networks between the private and shared caches and between the shared cache and the memory controller. Here, network protection is extended with the capability to allow system software to control network bandwidth allocation and scheduling. Each interconnect is extended with a ring-buffer of network turn control (NTC) registers and a network turn offset control (NTOC) register. NTCs specify a TCID and turn length. The NTOC allows the start of the bus schedule to be adjusted relative to other time multiplexed resources (namely, other buses and the memory controller).

3.2.3. Main Memory Controller The main memory is shared concurrently among multiple cores. As a result, interference among memory accesses from multiple timing compartments can lead to timing channels. We adopt a proposal by Wang et al. [49], which uses time division multiplexing to remove timing channels in DRAM controllers, but proposes a new optimization to reduce its overhead (Section 4.2).

This approach uses a set of techniques to remove timing channels in each components. A shared request queue is replaced with smaller per-compartment queues. To remove timing variations based on row buffer state, the DRAM controller uses the closed page policy. The contention on DRAM resources such as command/data bus, banks, and ranks are removed using timing division multiplexing with a fixed schedule where only one timing compartment can issue a request at a time. A period where no new requests can be issued, called the *dead time*, is added to each time slice in order to prevent in-flight requests or refreshes from interfering with the next time slice.

Memory controller protection similarly supports fine-grained resource allocation by the OS. The resource allocation control structures are similar to those used for the on-chip interconnects. A ring buffer of memory turn control registers

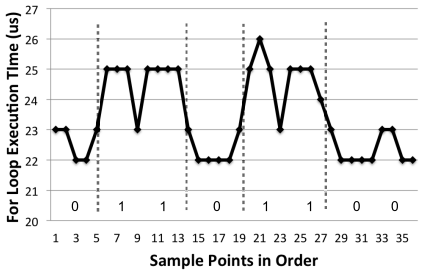

Figure 2: TC0's timing observation.

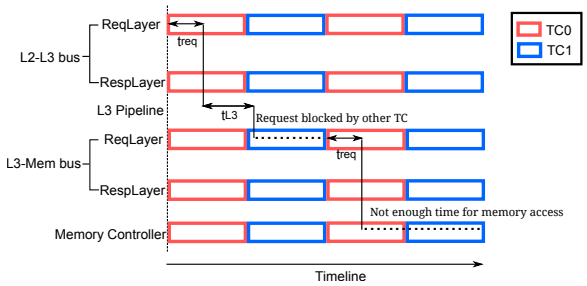

Figure 3: A bad time multiplexing schedule.

(MCTs) controls the owner and length of each turn and the memory turn offset control register (MTOC) controls the turn offset.

3.2.4. Cache Coherence Protocols Multi-core processors use cache coherence protocols. Unfortunately, we found that coherence operations can lead to a new timing channel, which was not discussed previously. Coherence operations can lead to timing interference though coherence bus contention or contention for cache ports. Even when there is no shared data between timing compartments, traffic on the snooping coherence bus can lead to a timing channel because the bus is shared by multiple timing compartments.

Attack Example. Here, we demonstrate a timing covert-channel attack through cache coherence mechanisms using a simulated 4-core system. Each core has private L1 and L2 caches, and the four cores share an L3 cache. The four L2 caches are connected with a snooping coherence bus which uses a MOESI protocol. TC0 runs on core 0 and core 1 while TC1 runs on core 2 and core 3.

TC0 has two threads, each running on a different core. Both threads run a `for` loop, and write to shared data during each iteration. Before each write is performed, one of the L2 caches has to forward the data to the other through the snooping coherence bus and invalidate its own copy. TC0 repeats this process and records the time for each loop. To communicate a secret, TC1 sends a '0' by doing nothing and sends a '1' by spawning multiple threads that write to shared data. Figure 2 shows the execution time of the `for` loop that TC0 observes, which shows clear correlation to the secret '01101100' sent by TC1.

Protection. Cache coherence mechanisms have two sources of timing interference: bus contention and port contention. Similar to the on-chip data bus, timing compartments use temporal partitioning remove interference. However, timing channel protection for the coherence mechanism is different from data bus protection in two ways. While coherence requests are associated with the TCID of the core that issued the request, responses must be tagged with the TCID of the corresponding request, *not the TCID of the core that sends the response*. Also, in the MOESI protocol, a private cache that owns the data may need to send it to another cache. These transactions can contend for cache ports with requests from processing pipelines. To remove this contention, we change the coherence protocol to *serve data from the shared cache instead of from the private caches whenever the data is owned by a different timing compartment*. Because protected pages that are shared between compartments are always read-only, the shared cache or memory always has an up-to-date copy.

3.3. Secure Timing Compartment Switching

The resources that are dedicated to each core such as private caches, TLBs, and branch predictors, are only used by one timing compartment at a time. However, multiple timing compartments can use these resources through time-sharing. Therefore, there exist timing channels if the state is kept across context switches. For example, the branch behavior of one timing compartment may affect the next timing compartment if the branch predictor table is kept across a switch. To eliminate this timing channel, timing compartments flush the per-core state when a core leaves a timing compartment (i.e., the TCID changes).

To prevent information leakage, the time that these flushing operations take cannot depend on the timing compartment's state. For example, cache flushing should not take longer when there are more dirty blocks. Therefore, after flushing, each core is blocked until the worst case writeback time. In our evaluation, we found the impact of flushing and writeback blocking is negligible.

4. Performance Optimizations

4.1. Time Slice Coordination

Timing compartments rely heavily on time multiplexing to protect shared resources including the L2-L3 bus, the L3-memory bus, and the memory controller. Since these resources are all involved to handle L2 misses, their schedules must be coordinated. For example, when a request exits the L3-memory bus, the memory controller should be available to handle that request to avoid an unnecessary delay.

Figure 3 illustrates the problem. It shows when each of two timing compartments are scheduled to use the time multiplexed resources along the L2 miss path. Red blocks indicate that TC0 is scheduled to use the device, and blue blocks indicate that TC1 is scheduled. The block of time that a timing compartment is scheduled to use a device is called the *turn*, and

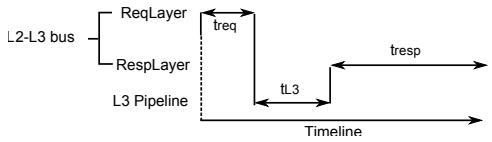

Figure 4: L3 cache hit timing sequence.

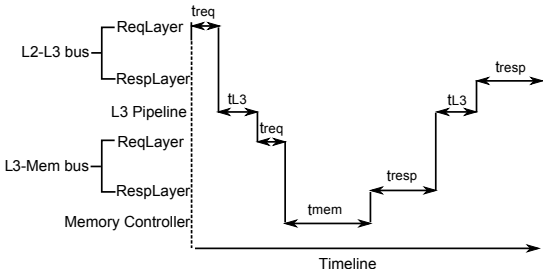

Figure 5: L3 cache miss timing sequence.

the duration of a turn is called the *turn length*. In this schedule, timing compartments are allotted the same turn length for each time multiplexed resource, and the schedule for each device starts at the same time.

In the figure, an access from TC0 that misses in both L2 and L3 is shown. The L2 miss sends a request to the L3 using the L2-L3 bus request layer. When the L3 access is complete, the request must proceed through the L3-memory bus request layer, but at this time TC1 is scheduled to use the L3-memory bus, so it must wait for TC1's turn to finish before the request can proceed to the memory controller. Then, when it arrives at the memory controller, there is not enough time left in the turn to complete a request, so it is blocked again. To reduce the unnecessary stalls, the time slices need to be carefully coordinated.

4.1.1. L2 Miss Timing Paths L2 misses take two different paths and have different timings depending on whether the L2 miss is an L3 hit or miss. This section analyzes L2 miss timings under both cases: an L2 miss followed by an L3 hit, and an L2 miss followed by an L3 miss.

Figure 4 shows the timing for an L3 hit. The arrows indicate the time that the resource in the corresponding column is used. The L2 miss begins by transferring a request over the L2-L3 bus request layer in t_{req} cycles. The request then arrives at the L3 cache, where it takes t_{L3} cycles (i.e. the L3 cache latency). Finally, the data is transferred from the L3 response port back to the L2 over the L2-L3 bus response layer in t_{resp} cycles.

Figure 5 shows the timing for an L3 miss. The request begins by transferring over the L2-L3 bus request layer, and similarly takes t_{L3} cycles to identify that it is a miss. Afterwards, it sends another request to the memory controller over the L3-memory bus request layer in t_{req} cycles. Memory requests take some time to complete, after which the data is returned to the L3 in t_{resp} cycles, written to the L3 in t_{L3} cycles and finally returned to the L2 after another t_{resp} cycles.

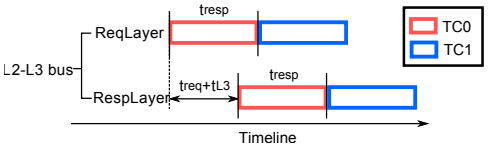

Figure 6: Cache hit timing path schedule.

4.1.2. Devising an Efficient Schedule A good time multiplexing schedule should minimize the average L2 miss latency. This can be achieved by controlling the turn lengths and offsets for each time multiplexed device. Here, an *offset* refers to a shift in the start of the turn for a single device compared to the start of the full schedule.

There are three main criteria for developing an efficient schedule. First, the turn length should be long enough for at least one transaction to complete. Second, to reduce unnecessary waits, the offset should make the turn start when the data is available from the preceding step. Third, the offset should be repeated for each time slice and each timing compartment.

L2 Miss Followed by an L3 Hit. This reasoning can be applied in a straightforward way to derive the schedule that optimizes the L2 miss latency assuming it is followed by an L3 hit. The optimal schedule is shown in Figure 6. From the timing for this case shown in Figure 4, the turn length for the L2-L3 request layer must be greater than t_{req}. The data is available to the L2-L3 response layer after $t_{req}+t_{L3}$ cycles, so this is used as the offset for the response layer. The turn length for the response layer is t_{resp}. Finally, to ensure that the schedule including the offset repeats, we can set the turn lengths for both request and response layers to be the same. The schedule parameters are summarized at the top of Table 1.

L2 Miss Followed by an L3 Miss. Deriving a schedule that is suitable for L2 misses assuming they are followed by L3 misses is similar. The turn lengths for both L3-memory bus layers and the memory controller should be t_{mem}, the minimum memory turn length. The L2-L3 bus layer turn lengths should be the smallest factor of t_{mem} greater than t_{resp}. The L2-L3 bus turn lengths are greater than t_{resp} to improve the schedule's repeatability. Each device schedule is offset so that it starts when the data is available based on the timing of an L2 miss followed by an L3 miss shown in Figure 4. Table 1 summarizes both the turn lengths and offsets in this schedule. Here t_{read} is the worst case memory read time and

$$t^{+}_{req} = Min(\{n | \exists m \in \mathbb{N}. t_{mem} = m \cdot n, n > t_{resp}\}).$$

L2 Miss in General. Finding a good schedule for L2 misses in general is difficult because both L3 hit and miss cases must be considered at the same time. Note that the L2-L3 request and response layer turn lengths and the L2-L3 response layer offsets in the efficient schedules for the L3 miss and L3 hit cases are different. These differences must be balanced for an L2 miss schedule. We were not able to analytically derive an optimal solution for this case. For this

L2 misses followed by L3 hits		
Device Name	Turn Length	Offset
L2-L3 Req	t_{resp}	0
L2-L3 Resp	t_{resp}	$t_{req}+t_{L3}$

L2 misses followed by L3 misses		
Device Name	Turn Length	Offset
L2-L3 Req	t^{+}_{resp}	0
L3-Mem Req	t_{mem}	$t_{req}+t_{L3}$
Mem Ctl	t_{mem}	$2t_{req}+t_{L3}$
L3-Mem Resp	t_{mem}	$2t_{req}+t_{L3}+t_{read}$
L2-L3 Resp	t^{+}_{resp}	$2t_{req}+2t_{L3}+t_{read}+t_{resp}$

Table 1: Efficient L2 miss schedules by case.

general case, we solved the problem using simulated annealing to find a good schedule.

4.2. Operation-Aware Dead Time

Simulations show that the reduced memory bandwidth due to the protection for memory controllers lead to a significant slowdown for memory-intensive workloads. TP [49] reduces the maximum usable memory bandwidth, because it requires a dead time at the boundary between time slices for different timing compartments during which no new transaction can be issued. For security, the dead time is conservatively set as the worst-case time between two transactions. In practice, this is a substantial portion of a time slice. For the parameters used in our simulations, the dead time consumes 22 memory cycles out of each time slice, which range from 23 to 43 cycles.

Security requres only that in-flight transactions issued by one compartment cannot interfere with transactions issued by another compartment in the following turn. The worst-case time between two transactions depends on the type of each transaction — in other words, for some transaction types, the worst-case time is lower, meaning that transactions can safely be issued later in the turn. We propose an optimization that leverages this observation by coarsely grouping transactions into reads and writes. Then, then a separate dead time is used for each type of transaction. For example, the following equations show the dead time for each memory operation sequence.

- Read, Read: $tFAW - 3 * tRRD$
- Write, Write: $tFAW - 3 * tRRD$
- Read, Write: $tCAS + tBURST + tRTRS - tCWD$
- Write, Read: $tCWD + tBURST + tWTR$

The dead time for reads is determined as the worst case time between any read transaction and any other transaction. The dead time for writes is determined similarly. The required dead time for read transactions is much smaller than the one for writes, allowing reads to be issued later in the turn than they would be with a monolithic dead time. The optimization results in significant performance improvements for memory-intensive workloads.

4.3. Application-Aware Resource Allocation

In our baseline configuration, we assume that the resources are equally partitioned among timing compartments. This approach ensures the timing of one compartment is completely independent of other compartments. However, this allocation will rarely match the resource demands of the applications.

This inefficiency can be reduced if the management software can adjust the amount of each statically partitioned resource granted to each TC. In doing so, the amount of resources granted to a TC can match the amount of resources needed by that TC. However, to be secure the allocation decisions cannot depend on confidential data. To solve this problem, we use static information about each program's resource demands, which does not reflect sensitive, input-dependent data used at runtime. These general characteristics can be obtained by profiling the workload with public input sets that do not contain secrets. Alternately, in cloud computing datacenters, the end-user can submit workloads with service level agreements, indicating the resource needs. A previous study on job scheduling in cloud computing infrastructures has shown that the resource needs of a job can be quickly determined and used to find an efficient resource allocation [14]. For example, a workload which requires a large amount of cache space can be allocated to a physical machine where other applications use little cache space.

5. Handling an Untrusted OS

Many security vulnerabilities are caused by bugs in complex software. Significant efforts have been made to reduce the size of the software trusted computing base. One promising approach to achieve this goal uses hardware to protect the confidentiality and integrity of critical software entities without trusting low-level software such as operating systems or hypervisors. Architectures which achieve this goal are often referred to as *secure hardware compartments* or *secure processors*. Many hardware compartment designs have been studied in academia [42, 39, 29, 26, 41, 20, 11, 13, 19]. Intel also announced a hardware compartment technology, called Secure Guarded eXecution (SGX) which will be implemented in upcoming processors [25].

However, existing hardware compartment architectures are vulnerable to timing channel attacks through shared hardware components. This section extends timing compartments so they be applied to conventional compartments to provide isolation from both explicit software access and timing channels even when managed by an untrusted OS and/or hypervisor. In particular, the untrusted OS/hypervisor introduces two additional technical challenges; 1) the untrusted OS/hypervisor must be allowed to manage resource allocation without compromising security, and 2) there can be no information leakage from a timing compartment to the OS/hypervisor through timing.

Timing channel	Protection
System calls	Software protection. Secure HW timer enables SW mitigation techniques.
Page faults	Page lock and unlock instructions
Interrupts	No need. Do not depend on sensitive data.

Table 2: Protection for new timing channels under an untrusted OS/hypervisor.

5.1. Secure Protection Management

Conventional hardware compartments distrust the OS/hypervisor, but still grant it the authority to manage the allocation of resources such as CPU cycles and virtual memory. In many compartment architectures, the software in a compartment is executed (i.e. CPU cycles are allocated to the compartment) by setting a compartment ID register on the core [19, 41, 39, 25]. This ID is used to make access control decisions, so this register is not directly accessible in software. Instead, it is controlled through special instructions which, for example, simultaneously set the program counter and the ID to guarantee that the correct code is executed. This ID can naturally be used as the TCID for timing protection. Instructions which change the ID must be augmented to flush private core state (e.g., caches and branch predictor tables). Adding timing protection does not require virtual memory management protection to differ from previous compartment architectures.

In addition to CPU cycle and memory management, the untrusted management software must be permitted to control the allocation of timing-protected, shared resources without violating secuirty. These resources include space in shared caches and bandwidth in on-chip networks and the memory controller. To ensure timing isolation, the interfaces for resource allocation must satisfy the following properties: 1) each resource is allocated to at most one timing compartment at a time, and 2) the corresponding state is flushed when a resource is deallocated from a timing compartment.

Shared cache space is allocated by adjusting the CPCs which associate an owner TC with each way. Only one TC can be allocated to each way at a time. The CPCs are not controlled in software directly. Instead, ways are allocated through the `TC_ALLOC_WAY` which flushes the contents of a cache way before allocating it to a new TCID. Bandwidth for on-chip networks and memory controllers is allocated by changing a set of control registers. The NTCs and MTCs which allocate turns to TCs permit only one TCID to be allocated to a time slot. These registers may safely be controlled directly in software.

5.2. New Timing Channels

Timing compartments as described thus far remove timing channels through shared hardware resources, and rely on trusted software to manage other timing channels. However, when the OS/hypervisor cannot be trusted, information leaks through events that are visible to the OS/hypervisor must also be prevented in hardware. In particular, the untrusted system software can observe the timing of events that cause transitions out of the timing compartment, which may depend on confidential data. There are three possible ways that a compartment transitions to an untrusted OS/hypervisor: explicit system calls (or hypercalls), exceptions, and external interrupts.

Figure 2 show how these three new timing channels can be handled. First, the timing of external interrupts does not depend on secrets within compartments, and does not leak confidential information. Note that timer interrupts may be determined by a value from a compartment, but this value cannot be a secret as it is already visible to the untrusted OS/hypervisor.

System calls (or hypercalls) are externally visible events that are fully controlled by software within a compartment. Also, the timing channels in externally visible I/O events exist even when a program runs on dedicated hardware with a trusted OS/hypervisor. As a result, there exist countermeasures for information leaks through timing variations within a single program. For example, language-level techniques have been developed to eliminate [48] or mitigate [5, 58, 59] such timing channels. To enable mitigation schemes that insert delays for certain program operations without relying on an untrusted OS/hypervisor, timing compartments provide a new instruction that provides a trustworthy time value even if the OS is compromised.

Most program exceptions come from bugs or errors and should not happen if programs in compartments are well-written. However, page faults are difficult to control in software and their timing may leak confidential information. In modern virtual memory systems, only a finite set of pages are kept in physical memory. When new pages are brought into physical memory, page faults are triggered, revealing which page is being accessed. In fact, Xu et al. [57] recently demonstrated an attack that exploits this vulnerability. We add two new instructions, `TC_LOCK` and `TC_UNLOCK`, to eliminate timing channels through page faults. The instructions allow a program in a timing compartment to preload and lock some pages before sensitive computations so that those pages cannot be replaced. Hardware compartments typically provide instructions to add or remove a page to a compartment, and maintain a protected list of physical pages for each compartment in order to protect them from the untrusted software. The compartment aborts on a page lock instruction if the page does not already exist in the protected memory region. An attempt by an untrusted OS/hypervisor to remove a locked page returns an error so that the OS can select a different page to remove.

6. Evaluation

6.1. Methodology

To study the performance overhead of timing compartments for modern multi-core processors, a timing compartment architecture is simulated using gem5 [8] integrated with DRAM-

Core count			2/4/6/8
gem5 core model			"O3"
CPU Clock			2GHz
Memory		2GB	667MHz
Network Clock			1GHz
L1d / L1i	32kB	2-way	2 cycles
L2	256kB	8-way	7 cycles
L3	2/4/6/9MB	16-way	17 cycles

Table 3: Simulation configuration parameters.

Sim2 [38]. The simulations use the ARM ISA. Table 3 shows the system configuration. The cores use the gem5 "O3" out-of-order core model which runs at 2GHz. Each core has private 32KB L1 instruction and data caches, and a private 256KB L2 cache. The shared L3 cache is varied from 2MB to 9MB depending on the number of cores. The cache configuration parameters are derived from the Intel Xeon E3-1220L and Intel Xeon E7-4820 which are used by Amazon EC2. In DRAMSim2, we simulate a 667MHz 2GB DDR3 memory. The interconnects in the simulator run at 1GHz. Unless specified otherwise, each benchmark is fastforwarded for 1 billion instructions, and run for 100 million instructions.

For most experiments, each core has its own timing compartment. That is, for an n-core system, n timing compartments execute concurrently. We study the impact of having multiple cores in one timing compartment separately. In the baseline configuration, the number of cache ways and the network/memory bandwidths are evenly partitioned among timing compartments, and the minimum turn length (23) is used for memory controllers. Unless otherwise stated, the memory controller protection uses the relaxed dead time optimization, which apply different dead times for reads and writes. For studies on the application-aware resource allocation, the number of cache ways are set based on the cache miss curve (the number of misses as a function of the number of cache ways), and longer turn lengths are allocated to memory-intensive TCs based on the number of L3 misses.

6.2. Security Verification

We formally verified that timing compartments enforce strict, timing-sensitive noninterference. To do so, we implemented a Verilog description of the core microarchitecture enforcement mechanisms of Timing Compartments on a 4-core prototype processor. Then we verified the prototype with SecVerilog, a variant of Verilog which is extended with an information flow type system [60]. SecVerilog allows variables to be declared with a security type. Security types form a lattice [17]. If a SecVerilog implementation typechecks, it is formally guaranteed that it obeys timing-sensitive noninterference with respect to the security policy (i.e. the lattice of security types). Although SecVerilog does checking statically, it provides dependent types which allow security labels to depend on runtime values of variables in the design. Dependent types support fine-grained resource sharing over time.

The verified processor has 4 5-stage in-order MIPS cores with full bypassing, 2-way private 16KB I/D caches (16B blocks), a ring network, and a DRAM controller. The private caches are blocking, so MSHRs and branch predictors are unnecessary. The microarchitecture protection features are implemented as described in Section 3, however the configuration registers which allow the cache partition sizes, memory and network turn lengths, and memory and network offsets to be adjusted are omitted. These features improve performance, but are not necessary for security.

The policy is configured so that each timing compartment has unique security type. The types of each TC are mutually distrusting (i.e. incomparable in the lattice). Since cores are restricted to running only processes within the same TC at a time, on-core components within the same core are given the same security type. Wires carrying memory requests are given dependent types that depend on the TCID of the originating core. Static cache partitions assigned to each TC are labeled accordingly. The prototype was successfully type-checked implying that these enforcement mechanisms are sufficient for removing all timing channels in the off-core resources included in the prototype.

6.3. Impact of Time Slice Coordination

To study the impact of TDM coordinations under a large design space of parameters (namely, the turn lengths and offsets), we built a custom simulator that models the memory hierarchy for an L2 cache miss. The simulator enumerates all possible L2 miss arrival times and calculates the average L2 miss latency assuming the distribution of L2 miss arrival times is uniform random.

Using this simulator, we exhaustively searched for the lowest average L2 miss latency for an L3 hit with all turn length combinations per resource within a wide range and all possible offsets. The experiments confirmed that the intuitive schedule described in the previous subsection achieves the lowest expected L2 miss latency assuming a hit in the L3 cache. Fixing the turn lengths to the optimal values and adjusting only the offsets, the worst case latency is 19.6% higher than the one with the optimal schedule.

The design space for schedules involving all resources from the L2 cache to the main memory is far too large to search exhaustively. Instead we used simulated annealing to search this design space. First, we used the optimizer to study the L2 miss latency assuming the access misses in the L3 as well. The optimizer did not find a schedule better than the one discussed in the previous subsection after 20,000 iterations. Fixing the turn lengths to the best values and sweeping only the space of offsets, the worst schedule we found had an average L2 miss latency that is 2.64X higher than the best schedule, showing the importance of coordinating TDM schedules.

We used the same optimizer to find a schedule that minimizes the L2 miss latency under the assumption that the L3 cache hit-rate is 90%. We found that the memory latency changes by 39% when offsets are varied while the turn lengths are fixed.

Workload	Benchmarks	Memory Intensity
ast_ast	astar astar	low-low
h26_hm	h264ref hmmer	med-med
ast_h26	astar h264ref	low-med
sjg_h26	sjeng h264ref	med-med
sjg_sjg	sjeng sjeng	med-med
mcf_ast	mcf astar	high-low
lib_ast	libquantum astar	high-low
mcf_mcf	mcf mcf	high-high
mcf_lib	mcf libquantum	high-high
lib_lib	libquantum libquantum	high-high

Table 4: Multiprogram workloads.

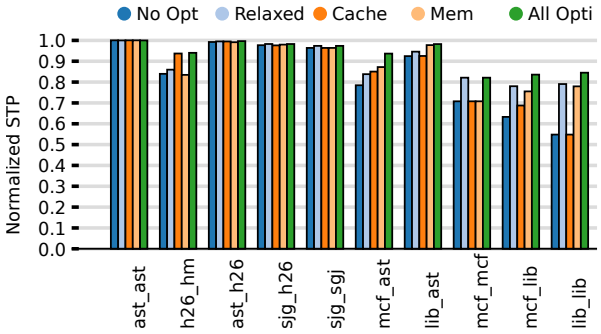

Figure 7: Performance Overhead of Timing Compartments.

6.4. Performance Overhead

This section evaluates the performance overhead of timing isolation by running multiprogram workloads comprised of SPEC2006 benchmarks, and measuring the system throughput (STP). STP is the aggregated normalized IPC of each program relative to the IPC when each program runs by itself. An STP of greater than 1 means that higher throughput is achieved by running the programs in parallel rather than serially. It is computed by

$$\sum_{i=1}^{n} \frac{IPC_{MP,i}}{IPC_{SP,i}}, \tag{1}$$

where $IPC_{MP,i}$ is the IPC of the i^{th} program in the workload when run in parallel with the others, and $IPC_{SP,i}$ is the IPC for the same program when it is run alone on the same system.

The experiments evaluate the performance for the workload mixes shown in Table 4. Workloads were selected to include a diverse set of application mixes that include both memory intensive and compute intensive benchmarks. Applications also vary in cache sensitivity. For experiments with two cores, each workload consists of the two benchmarks in the table. For experiments with more cores, the same labels are used to refer to a workload mix where half the compartments run the first program, and the others run the second half.

6.4.1. Overall Performance Overhead Figure 7 shows the performance overhead of timing compartments in a system with 2 cores both with and without optimizations. The optimizations include the relaxed dead time for the memory controller (relaxed) and static workload-aware resource allocations for the cache (Cache) and memory (Mem). The bar labeled (All Opti) uses all optimizations. The overhead is measured using the STP of the secure system normalized to the STP of the insecure baseline. Performance overhead depends heavily on memory intensity. The overhead is quite low for compute-intensive workloads such as `sjg_sjg`, `ast_h26`, `ast_ast`, and `sjg_h26`, because timing compartments only incur overhead during L2 misses.

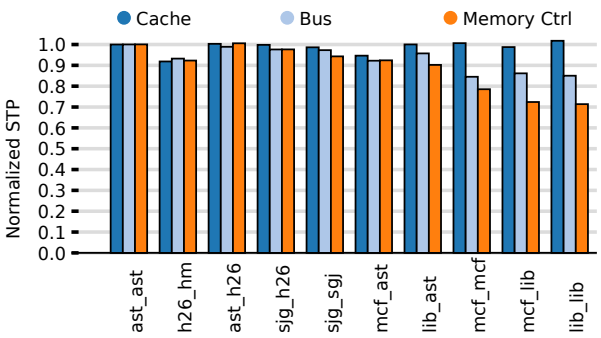

Figure 8: Performance Breakdown (4 cores).

On the other hand, the performance overhead can be quite significant for memory intensive workloads when unoptimized protection techniques are used. For example, `lib_lib` has overhead close to 45%. For these cases, the relaxed operation-aware dead time can significantly reduce overhead. This optimization reduces the worst-case overhead to roughly 20%. The overhead can be further reduced to less than 7% on average and 16% in the worst case if the application-aware resource allocation is also enabled.

6.4.2. Overhead Breakdown To better understand the sources of the performance overhead, the overhead of protection mechanisms were evaluated individually. Figure 8 shows the performance overhead of timing compartments compared to the insecure baseline when only a single protection mechanism is enabled at a time. These protection mechanisms include cache partitioning, time multiplexing for on-chip interconnect, and time multiplexing for a memory controller. The memory controller uses the relaxed dead time optimization, but resources are not allocated based on application characteristics. The results suggest that the memory controller protection is the most substantial source of overhead, and that cache partitioning and bus protection are less costly. This is because memory controller protection requires a dead time [49] to drain in-flight transactions which significantly reduces total memory bandwidth. For example, in our DRAM configuration, the turn length is 23 cycles whereas the dead time is 22 cycles. As a result, only one DRAM request can be issued every 23 cycles, incurring significant overhead for bandwidth-limited applications. While protection for caches and on-chip interconnects introduce inefficiencies, they do not reduce the total cache capacity or the on-chip interconnect bandwidth.

6.4.3. Scaling the Number of TCs Figure 9 shows the performance overhead of the timing compartment as the number of TCs and cores increases from 2 to 8. The relaxed dead time is used, but resource allocations are not optimized based on

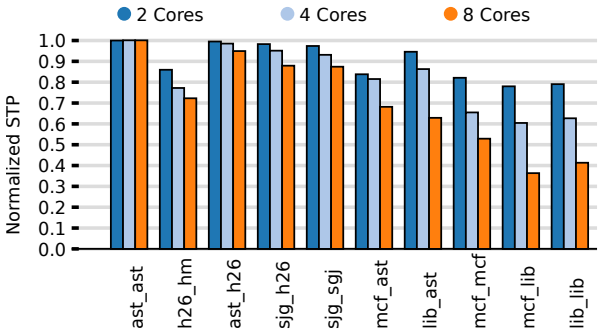

Figure 9: STP of Timing Compartments normalized to insecure as the number of TCs increases.

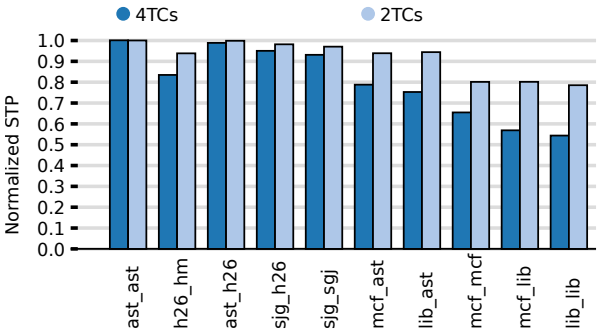

Figure 10: Performance benefit of allowing 2 programs to share a TC.

application characteristics. The performance overhead is low (less than 5%) for compute-intensive benchmarks even with a large number of timing compartments. Yet, the overhead of memory-intensive workloads increases with the number of timing compartments, because more compartments share the same amount of fixed memory bandwidth. While our simulation infrastructure currently only supports one memory controller, we note that commercial systems typically have multiple memory channels, typically one for every 2-4 cores. Therefore, we believe that the overhead for processors with more cores will be similar to these results with fewer cores.

The results suggest that many timing compartments can be supported simultaneously with reasonable overhead for compute-intensive applications. However, many memory intensive workloads should not be allocated to the same machine to keep overheads low, which is true even without timing compartments. In cloud computing environments, workloads can be dynamically profiled and more intensive workloads can be allocated to machines where there are few other memory intensive workloads [16]. Further, in many practical systems, multiple processes can be executed in the same timing compartment.

6.4.4. Using one TC for Multiple Programs Multiple programs or virtual machines can be grouped into the same timing compartment as long as they have the same security needs. For example, cloud users often request several VMs that run on the same physical machine, possibly to avoid network communication latencies. Naturally, VMs owned by the same user can be grouped into the same timing compartment. Also, low-security VMs may not need timing channel protection.

Using one timing compartment for multiple cores provides substantial performance improvements because cores within one compartment can share resources as they would in a conventional system without timing protection. Figure 10 shows the benefit of grouping multiple programs into a single timing compartment by comparing the STP of a system that runs 4 programs in 4 TCs, to the same system running the same 4 programs in 2TCs. The memory controller turn lengths are increased to 30 for the 2 TC system (compared to 23 for the 4TC systems) since TCs running two programs consume more memory bandwidth. The performance improvement is dependent on the characteristics of the applications in the workload. For memory intensive workloads such as `lib_lib` the STP improves by as much as 54%. However, for workloads like `ast_ast` where all applications are compute-bound, the improvement is small. Overall, the performance overhead of running 2 TCs on 4 cores is comparable to running 2 TCs on 2 cores.

6.4.5. Multi-threaded Performance Overhead The multi-program workloads do not show the overhead of timing channel protection for the cache coherence bus, because they do not have shared data. To evaluate the overhead of cache coherence protection, we used SPLASH-2 [55] benchmarks on a 4-core system. For each experiment, we run two copies of a SPLASH-2 benchmark, each with two threads, in two timing compartments.

The overhead of cache coherence protection was evaluated by comparing the normalized execution time of a system with all protection mechanisms to the normalized execution time of the same system with all protection mechanisms except cache coherence protection. The overhead of adding cache coherence protection is quite low; the overhead is at most 1.5% for `ocean_cp`. The overheads for the remaining SPLASH-2 benchmarks is negligible. The overhead is low because coherence protocol transactions are infrequent.

6.4.6. Context Switch Overhead When the TC is context switched out, the remaining state in the private and shared caches, and on-chip resources such as the TLB and branch predictor, need to be flushed and dirty cache lines need to be written back to main memory. To prevent writeback requests from interfering with the incoming process, the core must be stalled until all writebacks are complete. We believe flushing the private and shared caches are the main source of overhead as they are the largest state elements. Figure 11 shows the STP of a 4-core system with 4TCs with private caches that are flushed every 10ms, 50ms, and 100ms normalized to the STP of the system without flushing. The overhead of context switching is quite small. On average the overhead is 2.1% when context switches happen every 10ms and 0.8% when context switches happen every 100ms.

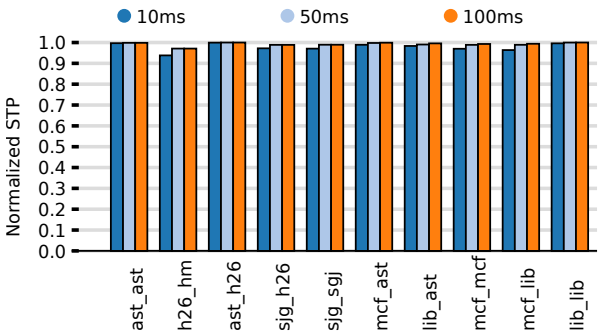

Figure 11: Performance overhead of TC switching.

Component	No protection (μm^2)	With protection (μm^2)	Overhead (%)
Cores	530,524	530,524	0
Inst cache	617,944	617,844	0
Data cache	578,270	577,968	-0.05
Network	621,996	661,309	6.32
DRAM controller	158,757	184,577	16.26
Total	2,507,491	2,572,222	2.58

Table 5: Area breakdown and overhead for the RTL prototype.

6.5. Area and Frequency Overheads

The Verilog prototype used for security verification is also used to evaluate the area and frequency overheads. The design was synthesized with the Synopsys Design Compiler. Timing channel protection does not affect the maximum frequency in this. Both designs with and without timing channel protection synthesized to 820MHz. The overall area overhead is relatively low, at 2.58%, even when the baseline processor is rather small and SRAM areas for caches are not included. The static partitioning in this model was simple. It only supports fixed partitions, and does not lead to any area overhead for caches.

7. Related Work

To our knowledge, this paper is the first study to eliminate microarchitectural timing channels through shared hardware in a full multi-core. Previous studies have identified a number of microarchitectural timing channels in caches [32, 36, 7, 35, 47, 3, 43, 10, 28, 22], branch predictors [1, 2], processor pipelines [52], networks on chip [50, 54], and memory controllers [49, 21]. Researchers have also proposed solutions for timing channels in individual components such as caches [53, 51, 28, 31, 18], memory controllers [49, 21], and on-chip network [50, 54]. While we leverage previous proposals, this study exposed new sources timing channels and new optimizations to reduce overhead.

Execution leases [44] enforce strict upper bounds on the execution times of program subsections, and a secure processor has been developed and verified [45] to provide timing channel protection. Ascend [20] prevents information leaks through off-chip memory accesses patterns by a single program. However, these architectures focus on preventing timing channels within a single program, and do not address interference among multiple cores.

Recent advancements have been made in detecting attempts to use covert timing channels. Hunger et al. [24] formally model timing channels, and demonstrate that *reads* from a covert timing channel are destructive. This facilitates detection since it implies reads can be observed. They also show how attacks can both be performed and detected even through noisy channels. Chen et al. [12] propose CC-Hunter, a framework for timing channel detection that uses hardware support to detect bursts of events that are likely to correspond to attempts to use a timing channel. These detection strategies complement TCs by providing different trade-off points. For high-assurance systems, TCs can provide a strong guarantee that there are no timing channels through shared hardware. For lower-security scenarios, the detection approaches can limit information leaks (but not eliminate them) by enabling protection only after attacks have been detected.

The key enforcement mechanisms of timing compartments are formally verified using SecVerilog [60]. SecVerilog performs timing-sensitive information flow analysis at the hardware description language level. Caisson [30] is another tool which performs information flow analysis at the HDL level. There exist other tools to check information flows in hardware designs. For example, Gate level information flow tracking (GLIFT) [46, 33, 34] allows information flow analysis at the gate level either at run-time or during simulations. These tools can be used to verify the security of timing compartment implementations.

We show how timing compartments can be applied even when the OS/hypervisor is not trusted. This allows timing compartments to be used for secure hardware compartment architectures [61, 19, 20, 41, 27, 29, 11, 13, 25, 9, 4, 26, 42, 44, 23, 56, 15] to provide strong isolation through both explicit and timing channels. Iso-X [19] represents the latest academic compartments architecture. Ascend [20] is a compartment architecture which prevents timing channels through the off-chip memory access patterns of a single program. Hyperwall [41] extends compartments to systems managed by an untrusted hypervisor. Other architectures reduce the software TCB rather than eliminating it [27, 11, 29, 13]. Compartment architectures have been adopted commercially as well [25, 9]. Since the approaches of these architectures are similar, we believe the techniques used for timing compartments can be applied to many of these designs.

8. Conclusion

This paper presents timing compartments, which provide an architecture abstraction to eliminate timing channels. When used in conjunction with access controls, timing compartments can provide strong software-level isolation since other side or covert channel attacks require physical access. To realize timing compartments, we designed a multi-core architecture that eliminates all inter-program timing channels, developed performance optimizations, and also show how they can be applied when the OS cannot be trusted. The simulation studies

show that the overhead is low if only a few compartments need to run in parallel.

References

[1] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. "On the Power of Simple Branch Prediction Analysis". In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, 2007.

[2] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. "Predicting Secret Keys via Branch Prediction". In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2007.

[3] O. Aciiçmez, W. Schindler, and c. K. Koç. "Cache Based Remote Timing Attack on the AES." . In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2007.

[4] ARM Ltd. Trustzone. http://www.arm.com/products/processors/technologies/trustzone.php.

[5] A. Askarov, D. Zhang, and A. C. Myers. "Predictive Black-Box Mitigation of Timing Channels". In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[6] A. Aviram, S. Hu, B. Ford, and R. Gummadi. "Determinating Timing Channels in Compute Clouds". In *The ACM Cloud Computing Security Workshop*, 2010.

[7] D. J. Bernstein. "Cache-Timing Attacks on AES". Technical report, 2005.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. "The Gem5 Simulator". *SIGARCH Computer Architecture News*, 2011.

[9] R. Boivie. "SecureBlue++: CPU Support for Secure Execution", 2012.

[10] J. Bonneau and I. Mironov. "Cache-Collision Timing Attacks Against AES". In *Proceedings of the Cryptographic Hardware and Embedded Systems Lecture Notes in Computer Science*, 2006.

[11] D. Champagne and R. Lee. "Scalable Architectural Support for Trusted Software". In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, 2010.

[12] J. Chen and G. Venkataramani. "CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware". In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[13] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. "SecureME: A Hardware-software Approach to Full System Security". In *Proceedings of the International Conference on Supercomputing*, 2011.

[14] "Christina Delimitrou and Christos Kozyrakis". "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters". In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[15] J. Criswell, N. Dautenhahn, and V. Adve. "Virtual Ghost: Protecting Applications from Hostile Operating Systems". In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[16] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In *Procedings of the 19th international conference on Architectural support for programming languages and operating systems Not.*, 2014.

[17] D. E. Denning. "A Lattice Model of Secure Information Flow". *Communications of the ACM*, 1976.

[18] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. "Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks". *ACM Transactions Architecture and Code Optimization*, 2012.

[19] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution". In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[20] C. W. Fletcher, M. v. Dijk, and S. Devadas. "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs". In *Proceedings of the 7th ACM Workshop on Scalable Trusted Computing*, 2012.

[21] C. W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. "Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-Offs". In *20th IEEE International Symposium on High Performance Computer Architecture*, 2014.

[22] D. Gullasch, E. Bangerter, and S. Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice". In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[23] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. "InkTag: Secure Applications on an Untrusted Operating System". *ACM SIGARCH Computer Architecture News*, 2013.

[24] C. Hunger, M. Kazdagli, A. S. Rawat, A. G. Dimakis, S. Vishwanath, and M. Tiwari. "Understanding Contention-Based Channels and Using Them for Defense". In *21st IEEE International Symposium on High Performance Computer Architecture*, 2015.

[25] Intel Corporation. "Intel Software Guard Extensions Programming Reference", 2014.

[26] S. Jin, J. Ahn, S. Cha, and J. Huh. "Architectural Support for Secure Virtualization Under a Vulnerable Hypervisor". In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[27] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. "NoHype: Virtualized Cloud Infrastructure Without the Virtualization". In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[28] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. "Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks". In *Proceedings of the 2nd ACM Workshop on Computer Security Architectures*, 2008.

[29] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. "Architecture for Protecting Critical Secrets in Microprocessors". In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[30] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. "Caisson: A Hardware Description Language for Secure Information Flow". In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[31] F. Liu and R. B. Lee. "Random Fill Cache Architecture". In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[33] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. "Theoretical Analysis of Gate Level Information Flow Tracking". In *Proceedings of the 47th Design Automation Conference*, 2010.

[34] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. "Information Flow Isolation in I2C and USB". In *Proceedings of the 48th Design Automation Conference*, 2011.

[35] D. A. Osvik, A. Shamir, and E. Tromer. "Cache Attacks and Countermeasures: The Case of AES". In *"Proceedings of the The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[36] C. Percival. "Cache Missing for Fun and Profit". In *BSDCan*, 2005.

[37] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds". In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[38] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. "DRAMSim2: A Cycle Accurate Memory System Simulator". *Computer Architecture Letters*, 2011.

[39] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing". In *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[40] G. E. Suh, L. Rudolph, and S. Devadas. "Dynamic Partitioning of Shared Cache Memory". *The Journal of Supercomputing*, 2004.

[41] J. Szefer and R. B. Lee. "Architectural Support for Hypervisor-Secure Virtualization". In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages*, 2012.

[42] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. "Architectural Support for Copy and Tamper Resistant Software". In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[43] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen. "An Analytical Model for Time-Driven Cache Attacks". In *Proceedings of the 14th International Conference on Fast Software Encryption*, 2007.

[44] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. "Execution Leases: A Hardware-supported Mechanism for Enforcing Strong Non-interference". In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[45] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. "Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security". In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[46] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. "Complete Information Flow Tracking from the Gates Up". In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[47] E. Tromer, D. A. Osvik, and A. Shamir. "Efficient Cache Attacks on AES, and Countermeasures". *Journal of Cryptology*, 2010.

[48] D. Volpano and G. Smith. "Probabilistic Noninterference in a Concurrent Language". *Journal of Computer Security*, 1999.

[49] Y. Wang, A. Ferraiuolo, and E. Suh. "Timing Channel Protection for a Shared Memory Controller". In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[50] Y. Wang and E. Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the 6th ACM/IEEE International Symposium on Networks-on-Chip.*, NOCS, 2012.

[51] Z. Wang and R. Lee. "A Novel Cache Architecture with Enhanced Performance and Security". In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, 2008.

[52] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. ACSAC '06.

[53] Z. Wang and R. B. Lee. "New Cache designs for Thwarting Software Cache-Based Side Channel Attacks". In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[54] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood. "SurfNoC: A Low Latency and Provably Non-Interfering Approach to Secure Networks-on-chip". In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[55] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations". In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[56] Y. Xia, Y. Liu, and H. Chen. "Architecture Support for Guest-transparent VM Protection from Untrusted Hypervisor and Physical Attacks". In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, 2013.

[57] Y. Xu, W. Cui, and M. Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[58] D. Zhang, A. Askarov, and A. C. Myers. "Predictive Mitigation of Timing Channels in Interactive Systems". In *Proceedings of the 18th ACM conference on Computer and communications security*, 2007.

[59] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. 2012.

[60] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. "A Hardware Design Language for Timing-Sensitive Information-Flow Security". 2015.

[61] T. Zhang and R. B. Lee. "CloudMonatt: an architecture for security health monitoring and attestation of virtual machines in cloud computing". In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[62] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis". In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.