# 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, an architecture abstraction that explicitly removes microarchitecture-level timing channels between groups of software running concurrently on a shared multi-core processor. The capability to eliminate timing channels coupled with conventional software isolation enables strong isolation comparable to running software on separate physical machines. This paper shows that such a strong timing isolation can be achieved by systematically removing timing interference in shared resources in a multi-core, and formally checked using information flow analysis on an RTL implementation. Through this systematic process, 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. Experimental studies show that the performance overheads of strict timing isolation can be surprisingly low for a small number of timing compartments; 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 shared by distrusting entities, which traditionally have been isolated on their own physical machines. For example, in cloud computing, mutually distrusting parties own virtual machines on shared hardware. In mobile platforms, users download third-party apps which might leak secrets through timing. While programs and virtual machines can be isolated using access control mechanisms such as virtual memory, timing channels in microarchitecture allow attackers to subvert these boundaries even when the OS and hypervisor are bug-free. Further, unlike physical side-channel attacks such as power analysis that require physical proximity, timing channels can be exploited in software by remote adversaries.

For example, researchers have shown that secret keys can be extracted from co-resident VMs on production EC2 servers through microarchitectural timing channels [43]. Hardware-level timing channels have been shown in many shared hardware components, including caches [42, 7, 10, 41, 53, 3, 49, 10, 31, 23], branch predictors [1, 2], interconnects [56, 59], pipelines [57], and memory controllers [55, 21]. The wide range of vulnerabilities suggests that strong timing isolation requires protection across a full processor, ideally in a way that can be verified.

In this paper, we propose *timing compartments* (TCs), a hardware-enforced abstraction that isolates distrusting software by eliminating timing channels through hardware in a shared, multicore processor. Timing compartments use hardware mechanisms to remove fine-grained microarchitecure-level timing interference that is difficult to control in software. Timing compartments provably enforce timing-sensitive noninterference and are statically verifiable with information flow analysis. 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 are the first formally-verified architecture to remove timing channels between processes sharing hardware in a multicore processor.

To provide strong isolation while allowing formal verification, timing compartments use static partitioning and time multiplexing to eliminate timing channels through shared hardware. The process of systematically applying this protection approach to a modern multi-core architecture revealed new sources of timing channels that have not been studied before. Notably, we show that cache coherence mechanisms cause timing channels even among processes with no shared data, and propose modifications to remove this vulnerability.

By using simple partitioning and time multiplexing to completely remove timing dependence, timing compartments are formally verifiable with information flow analysis. Information flow analysis proves that a system obeys *noninterference*, which requires that attacker-visible outputs cannot be affected by sensitive inputs [22]. Recently, hardware-level information flow control techniques have been proposed to enforce timing-sensitive noninterference in a hardware design statically at design time [34, 33, 65]. We implemented timing compartments in an RTL prototype of a quad-core processor with caches and a memory (DRAM) controller, and performed an information flow analysis using SecVerilog [65]. This result shows that timing compartments are amenable to formal verification.

Unfortunately, we found that such full-processor timing isolation can lead to significant performance overhead when enforcement mechanisms are implemented naively. To improve efficiency while maintaining a strong security guarantee, we propose a set of performance optimizations. First, we propose coordinated scheduling of time slots for time-multiplexed resources in the shared memory hierarchy. Our study shows that coordinated scheduling reduces the average L2 miss latency by up to 62% compared to an uncoordinated baseline. Second, we propose a novel optimization which increases the available bandwidth through a temporally-partitioned memory controller. As our experiments show that memory bandwidth reduction is the main source of performance overhead, this optimization improves performance considerably. The optimizations reduce the performance overhead of timing compartments by 58%.

Simulation results suggest the performance overhead of timing compartments with the proposed optimizations is quite reasonable especially when a small number of compartments run concurrently. Compared to the baseline with no timing isolation, executing two timing compartments reduces system throughput by less than 7% on average and by less than 2% for compute-bound workloads. In the worst case, memory-intensive workloads incur up to an 18% overhead. The results suggest that timing compartments on a shared multi-core processor are far more efficient compared to removing timing channels by running software on separate machines.

To the best of our knowledge, this paper represents the first to show that complete timing isolation on a multi-core processor is feasible with reasonable overhead and can be formally verified. Prior efforts for verifiable timing-channel protection [50, 51, 65] have verified single-core processors that can run only one program at a time. On the other hand, the multi-core processor in this paper runs multiple programs concurrently, sharing caches, on-chip networks, and memory controllers.

The timing compartment design can be extended to provide timing isolation even when the OS/hypervisor is potentially malicious. This extension requires interfaces that let the untrusted OS/hypervisor manage resource allocations while ensuring security. These extensions also require protection mechanisms to handle new timing channels. In particular, these extensions address attacks through page faults, including the one proposed by Xu et al. [62].

The following summarizes the main contributions.

- The paper introduces a new abstraction that enables software to explicitly remove microarchitecture-level timing interference on a multi-core, and shows that strong protection is viable.
- The paper identifies new timing channels on a multi-core, including one through cache coherence, and presents comprehensive timing protection for a full multi-core processor.
- The paper introduces performance optimizations which significantly reduce overhead and enable practical performance.
- The paper shows that timing compartments can be formally verified using an RTL implementation of a 4-core processor.
- The paper shows how full-processor timing isolation can be performed when an OS/hypervisor cannot be trusted. This enables timing channel protection for secure hardware compartments such as Intel SGX [27].

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.

## 2. TIMING COMPARTMENTS

### 2.1 Objective and Scope

The goal of timing compartments is to provide strong microarchitecture timing isolation among software running concurrently on a multi-core processor, in a verifiable manner. Timing compartments aim to eliminate timing channels that are not present among software modules running on dedicated processors. Therefore, the focus is on removing timing interference among parallel processing cores.

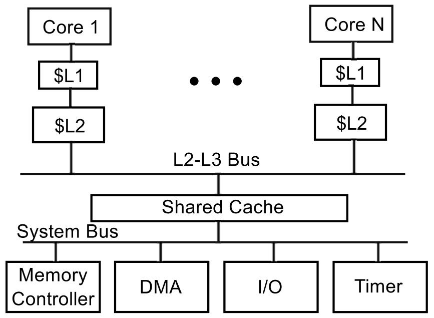

**Figure 1: Baseline multi-core architecture.**

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, 63, 64] 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 shows 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 [48, 46, 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 [43]. 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. PROTECTION MECHANISMS

### 3.1 Approach

Timing channels exist when 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. To achieve a degree of isolation that is comparable to running on separate hardware, timing compartments are designed to remove contention in shared hardware resources.

Timing compartments are designed to be verified with information flow analysis. Existing information flow analysis techniques for hardware enforce noninterference, a strong property which eliminates leakage of any information. Intuitively, an entity $H$ is *noninterfering* with entity $L$ if an observer can learn nothing about data owned by $H$ by observing and controlling only inputs/outputs owned by $L$. Existing information flow tools are *timing-sensitive*, meaning the adversary is assumed to be able to see the time at which $L$ signals change. Timing sensitive noninterference guarantees that there is no leakage via timing channels.

To be compliant with verification tools that enforce timing-sensitive noninterference, we use two approaches that completely eliminate timing channels and are simple to analyze. *Spatial partitioning* removes contention by duplicating or partitioning a resource for each compartment. *Temporal partitioning* removes contention by time multiplexing resources among timing compartments with a fixed schedule. Protection mechanisms that rely on obfuscation [38, 17] cannot be verified with information flow. They mitigate, but do not eliminate timing channels, so they do not enforce noninterference

For most resources, either spatial partitioning or temporal partitioning can be used. For example, to eliminate timing channels in a shared cache, the cache can be temporally partitioned to be used by one compartment at a time or be spatially partitioned by allocating some of the cache ways to each timing compartment. In the rest of the section, we describe how each source of timing channel in a typical multi-core processor can be removed, and discuss the trade-off between spatial and temporal partitioning for each component.

### 3.2 Timing Compartment ID

To track the timing protection boundaries, a management software such as an operating systems assigns a timing compartment ID (TCID) to processes. The same TCID can be assigned to multiple processes that do not require strong timing isolation among them such as ones belong to the same user.

In hardware, each processing core has an active timing compartment register (ATC) which indicates the TCID of the TC that is currently executing on that core. The value of the ATC is appended to each memory request and used by enforcement mechanisms to remove interference between different compartments. The size of the ATC is logarithmic with the number of physical cores as hardware only needs to distinguish active compartments running concurrently. The management software can virtualize TCIDs by maintaining a translation between virtual and physical TCIDs.

### 3.3 Private Resource Protection

To remove timing channels through each core's private resources (such as TLBs, private caches, branch predictors, and pipelines), at most one timing compartment is allocated to a core at a time. Simultaneous multithreading (SMT) is restricted so that only processes with the same TCID can share a core. This approach is already taken in production EC2 servers, which disable SMT to prevent timing channel attacks [67].

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 taken to flush should not depend on each timing compartment's state. For example, cache flushing should not take longer when there are more dirty blocks.

| Component | Timing Channel | Solution |
|---|---|---|
| Shared caches | Replacement | Way partitioning |
| | MSHRs | Duplicate MSHRs |
| | Response ports | Separate queues |
| Memory Controller | DRAM bus | Time multiplexing |
| | Queueing structure | Separate queues |
| | Row buffer | Closed Page Policy |
| | Response ports | Separate queues |
| On-Chip interconnect | Interconnect bus | Time multiplexing |
| | Queueing structure | Separate queues |
| Cache coherence | Coherence bus | Time multiplexing |
| | Cache port | Response by L3 |

**Table 1: Summary of timing channels and protection. Green represents newly identified ones.**

Therefore, we design hardware to support secure flushing that blocks a core for the worst-case writeback time. In our evaluation, we found the performance impact of the flushing is negligible.

## 3.4 Timing Isolation in Memory Hierarchy

Table 1 summarizes the timing channels in the shared memory hierarchy and our approaches to remove them. Newly discovered timing channels are highlighted in green.

### *3.4.1 Cache Contention*

Static cache partitioning [42] eliminates cache interference among timing compartments by allowing a cache block to replace only entries owned by the same timing compartment. While other approaches to cache protection have been proposed [36, 58, 17], timing compartments use way partitioning because it can be formally verified. Way partitioning is a form of spatial partitioning that allocates each cache way to one timing compartment. 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. As with private caches, any partitions owned by a TC must be flushed when it is context switched out.

For shared caches, spatial partitioning provides better performance than temporal partitioning by allowing all active TCs to use a portion of the cache and keep the most heavily used data on-chip. In temporal partitioning, while one TC can use the entire cache, the performance of other TCs will significantly degrade as their memory accesses need to go off-chip.

**MSHR Contention.** In addition to contention for cache arrays, shared ports and MSHRs also require protection. These timing channels are identified through our full-processor implementation and have not yet been described in the literature. Contention for miss status holding registers (MSHRs) in non-blocking caches causes timing channels. The number of outstanding 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 cause a timing channel. To remove MSHR contention, disjoint sets of MSHRs are allocated to each timing compartment. MSHR contention is resolved with spatial partitioning instead of temporal partitioning because MSHRs must be able to serve all active TCs.

**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, shared queues that buffer responses at the ports 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. Temporal partitioning is strictly better than spatial partitioning here. The ports are only interfaced with temporally partitioned networks, so there is no benefit from duplication.

### *3.4.2 On-Chip Interconnect Contention*

For on-chip networks, we adopt a temporal partitioning approach proposed in previous work [59], and extend it with the capability to allow system software to control network bandwidth allocation and scheduling. Each network arbiter 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). Temporal partitioning is preferable to spatial partitioning because bus transactions are short, and duplicating the network would have high area overhead.

### *3.4.3 Main Memory Controller Contention*

The main memory is shared concurrently by multiple cores. As a result, interference among memory accesses from multiple TCs leads to timing channels. We adopt a proposal by Wang et al. [55] to provide memory protection, but propose a new optimization to reduce its overhead (Section 6.2).

This approach uses a set of techniques to remove timing channels in each memory controller component. A shared request queue is replaced with smaller per-compartment queues. To remove timing variation based on row buffer state, the DRAM controller uses a closed page policy. A closed page clears the row buffers by precharging them after each row access. This is effectively flushing at the end of a temporal partitioning turn. Contention for DRAM resources such as the command/data bus, banks, and ranks is removed with temporal partitioning. A period during which 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. A hybrid temporal-spatial partitioning approach can also be taken. In a system with multiple memory channels, a subset of the TCs can be assigned to each channel, and temporal partitioning can be used within each channel.

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

### *3.4.4 Contention in Cache Coherence Protocols*

We found that cache coherence protocols can be a source of timing channels. In our knowledge, this paper is the first to discuss this vulnerability. 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 TCs.

**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 runs two threads on different cores. Both threads run a `for` loop, and write to shared data during each iteration. Before each

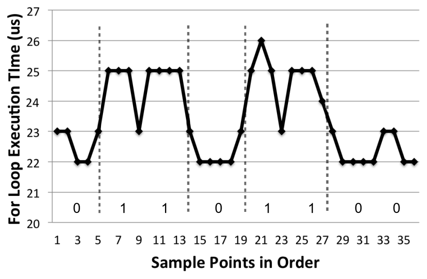

**Figure 2: TC0's timing observation.**

```
1   reg {H} h, {L} l;
2   //Not allowed
3   assign l = h;
4   ...
5       if(h) l = 0;
6       else l = 1;
7   ...
8
9   //TC(0) = TC0. TC(1) = TC1
10  reg {L} tcid;
11  reg {TC(tcid)} data, {TC0} din0, {TC1} din1;
12  ...
13  case(tcid)
14      0: data = din0;
15      1: data = din1;
16  endcase
17  ...
```

**Figure 3: Example SecVerilog code.**

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, we use temporal partitioning to remove interference on them. 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 issues 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 typical MOESI coherence 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.

## 4. VERIFICATION WITH INFORMATION FLOW ANALYSIS

### 4.1 HDL-Level Information Flow

Timing compartments are designed to be verifiable using static information flow analysis. In particular, we use SecVerilog [65], a secure hardware description language (HDL), to verify a prototype multi-core implementation. Secure HDLs use information flow control to formally verify security properties of a hardware design. Information flow control tracks and constrains the propagation of information throughout a system. SecVerilog analyzes and constrains information flow with a type system statically and at design time.

SecVerilog extends Verilog with syntax that allows a programmer to declare security levels, such as $L$ or $H$, for variables. The programmer also supplies an information flow policy describing whether information is allowed to flow in each direction between levels [16]. For example, information can be allowed to flow from $L$ to $H$, but not in the other direction. The policies can also express mutual distrust. That is, information flows can be disallowed in both directions between levels $TC_0$ and $TC_1$.

Figure 3 shows an example of SecVerilog code. Here, the variables `l` and `h` have security levels $L$ and $H$ respectively. Assume that information flow from $H$ to $L$ is disallowed. SecVerilog prevents when a $H$ variable is directly assigned to a $L$ variable as in line 3. The assignments on lines 5 and 6 leak the value of `h` to `l` indirectly through control flow. SecVerilog also prevents these *implicit flows*. SecVerilog supports *dependent types* which describe security levels that depend on the run-time values of variables in the code. In the example, the type of `data` is a dependent type that signifies $TC_0$ or $TC_1$, depending on the value of `tcid`. Dependent types are checked statically. SecVerilog uses static program analysis to constrain the possible run-time values of dependent types.

### 4.2 Prototype Processor Implementation

This paper aims to demonstrate that strict and verifiable timing isolation is indeed feasible for a full multi-core processor using today's tools. For this purpose, we implemented a timing compartment on a multi-core processor prototype written in Verilog. Then, we annotated the design with security types and performed an information flow analysis with SecVerilog [65]. This process formally verified that our prototype implementation enforces strict, timing-sensitive noninterference.

The verified processor has 4 5-stage, in-order MIPS cores with full bypassing, 2-way private 16KB instruction caches (16B blocks), a 2-way shared 32KB data cache, a ring network, and a DRAM controller. The caches are blocking, so MSHRs and branch predictors are unnecessary. Because there is only one data cache and the instruction caches contain read-only data, coherence protocols are also not necessary in our prototype. The rest of the protection features are implemented as described in Section 3. However, the prototype omits the configuration registers which allow cache partition sizes, memory and network turn lengths, and memory and network offsets to be adjusted. These features improve performance, but are not necessary for security. Instead, resources are allocated to each core evenly.

The SecVerilog policy is configured so that each timing compartment has a unique security type $TC_i$ and each timing compartment is mutually distrusting. That is, information is not allowed to flow in any direction among the TCs. With this policy, all timing channels and other flows between timing compartments will cause SecVerilog to report a type error. Generally, spatial partitioning is expressed by statically labeling partitions with the level of the owner, and temporal partitioning is expressed with dependent types that indicate the current owner.

As each core can only run one TC at a time, components within a core all have the dependent type `TC(atc)`. Here `atc` is the TCID in the core's active timing compartment register (ATC) and `TC(atc)` is the corresponding security level. On a context switch, the value in the ATC changes causing the types to change. SecVerilog enforces state to be flushed on such a label change to ensure that there is no information leak.

The active TCID is appended to each memory request, and the address and data signals are given types that depend on the TCID. As a result, the SecVerilog type checks ensure that there is no interference in shared memory hierarchy including caches, on-chip networks, and a memory controller. The spatially partitioned resources such as cache ways and queues are statically labeled according to the owner TC of the partition. The temporally partitioned resources such as on-chip network and memory controller resources are given dependent types that reflect which TC is currently allocated to use the resources. The processor was successfully type-checked indicating that these enforcement mechanisms are sufficient to provide timing-sensitive noninterference between timing compartments.

## 5. HANDLING AN UNTRUSTED OS

Modern operating systems and hypervisors are large and complex, and therefore contain vulnerabilities often exploited by attackers. Secure processors (also called secure hardware compartments) represent a promising approach to protect critical software even under vulnerable low-level software. In this approach, hardware protects the confidentiality and integrity of software running inside a secure compartment while still allowing an untrusted OS-/hypervisor to manage resources. Many secure processor designs have been proposed in research [48, 46, 32, 29, 47, 20, 11, 13, 18]. Intel has recently introduced a hardware compartment technology, named Secure Guarded eXecution (SGX) [27] in commercial processors.

However, existing secure processor architectures are vulnerable to timing channel attacks through shared hardware resources. This section shows how timing compartments can be applied to secure processors to provide timing protection. To do so, we need to solve two additional challenges: 1) the untrusted OS/hypervisor must be allowed to manage the timing protection mechanisms in a way that cannot break timing isolation, and 2) confidential data cannot be leaked to the untrusted OS/hypervisor through timing.

### 5.1 Secure Protection Management

Secure processors distrust the OS/hypervisor, but still grant it the authority to manage the allocation of resources such as CPU cycles and virtual memory. For protection, hardware maintains a compartment ID for each processing core [18, 47, 46, 27] and use it to make access control decisions. The compartment ID is not directly accessible in software, but controlled through special instructions which allows an untrusted OS/hypervisor to make scheduling decisions (such as create, interrupt, resume, or destroy a compartment) while ensuring confidentiality and integrity. The compartment 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) on a compartment switch. Adding timing protection does not require any change to virtual memory management protection of secure processor 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 security. These resources include space in shared caches and bandwidth in on-chip networks and the memory controller. To ensure timing isolation, the software interfaces for resource allocation must satisfy the following properties: 1) each resource can only be 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.

| 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.**

In our design, shared cache is allocated by adjusting the CPCs which associate a TCID with each way. Only one TC can be allocated to each way at a time. Changing the value of the CPCs triggers a flush to the corresponding cache way. Bandwidth for on-chip networks and memory controllers is allocated by changing the NTC and MTC control registers respectively. These registers also allow only one TC can use each time slot. Changes to these registers do not require a flush because these resources are stateless.

### 5.2 New Timing Channels

If the OS/hypervisor is untrusted, information leaks through timing of events that are visible to the OS/hypervisor must be prevented in hardware. In particular, the untrusted system software can observe the timing of events that cause transitions out of the timing compartment. 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 shows how these three new timing channels are handled. First, external interrupts such as I/O events do not depend on data within timing compartments. Similarly, timer interrupts used for process scheduling are set by management software independent of data within the TC.

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 [54] or mitigate [5, 63, 64] such timing channels. To enable mitigation schemes that insert delays, 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 occur during normal execution, and may leak sensitive 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. Xu et al. [62] have recently demonstrated an attack that exploits page faults to leak secrets.

Timing compartments 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 pages before sensitive computations. Locked pages cannot be evicted. Secure processors 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 untrusted software. The compartment aborts on a page lock instruction if the page does not already exist on the protected page list. An attempt by an untrusted OS/hypervisor to remove a locked page from the list returns an error so that the OS can select a different page to remove.

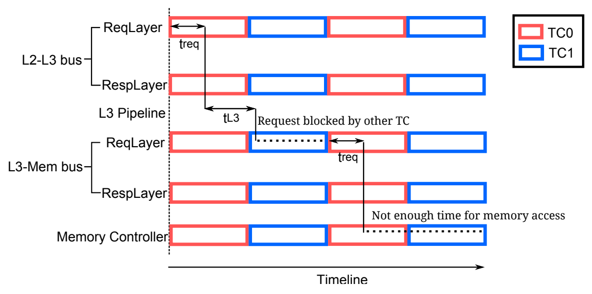

**Figure 4: A bad time multiplexing schedule.**

## 6. PERFORMANCE OPTIMIZATIONS

### 6.1 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 in handling L2 misses, their schedules must be coordinated to achieve high performance. For example, to avoid an unnecessary delay when a request exits the L3-memory bus, the memory controller should be available immediately to handle that request.

Figure 4 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 (Blue) blocks indicate that TC0 (TC1) is scheduled to use the device. In the figure, an access from TC0 that misses in both the L2 and L3 caches 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 TC0 is blocked until TC1 finishes. Then, when it arrives at the memory controller, there is not enough time left to complete a request, so it is blocked again.

The time multiplexing schedule should be coordinated among related resources to avoid this problem. We define a *turn* as the block of time that a TC is scheduled to use a resource, and a *turn length* is the duration of a turn. An *offset* refers to a shift in the start of the turn for a single resource compared to the start of the full schedule. Coordination can be done by controlling the turn lengths and offsets for each time multiplexed resource.

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 delays, the offset should begin each turn when the data is available from the preceding step. Third, the desired schedule should repeat for each timing compartment.

The timing of an L2 miss depends on whether it hits or misses in the L3. After L3 hit, the response is sent back across the L2-L3 bus immediately. After an L3 miss, the request must propagate through the L3-memory bus, the memory controller, and so on. Since the timing differs for these two cases, they produce conflicting timing constraints.

Given the conflicting requirements, we found that deriving the optimal schedule for memory hierarchy in general is a nonlinear optimization problem which is too difficult to be solved analytically. Instead, we derive a schedule heuristically. We built a custom simulator that models the memory hierarchy components involved in an L2 miss. It accepts a schedule (i.e. turn length and offset values) as inputs and calculates the average L2 miss latency assuming the distribution of request arrival times is uniform random. We then used a simulated annealing optimizer to find a schedule that minimizes the L2 miss latency assuming an L3 hit-rate of 90%.

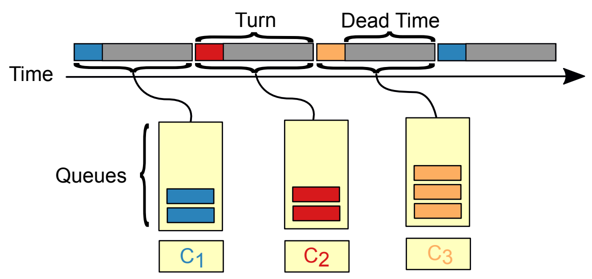

**Figure 5: A temporal partitioning schedule with three security classes.**

The simulation study show that the coordination has a significant impact on memory latency. For L2 misses that hit on an L3, the worst schedule we found just varying offset values with a fixed turn length had an average L2 miss latency that is 2.64X higher than the best schedule. For L2 misses in general (90% L2 hit), the schedule found by the optimizer reduced the average L2 miss latency by 62% compared to the worst schedule found, and by 12% compared to a hand-tuned, best-effort schedule.

### 6.2 Operation-Aware Dead Time

Simulations show that the most significant source of overhead for timing compartments is the reduction in maximum usable memory bandwidth due to temporally partitioning the memory controller. Figure 5 illustrates temporal partitioning [**?**] which removes timing interference in a shared memory controller by issuing requests in a fixed static schedule. Each security domain is allocated to a single time-slice called a turn, and each security domain can only issue transactions during its turn. To prevent transactions issued by one timing compartment from interfering with another timing compartment, the memory controller stops issuing transactions for a period at the end of each turn, called the dead time, in order to ensure that in-flight transactions complete by the end of a turn and do not interfere with the next turn. The dead time is conservatively set to 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 requires that the in-flight transactions from one compartment cannot interfere with transactions issued by another TC in the following turn. However, 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. Transactions that take less time can safely issue later in the turn. We propose an optimization that leverages this observation by coarsely grouping transactions into reads and writes. Then, a different dead time is used for each type of transaction. For example, the following equations show the worst-case times for each memory operation sequence based on DRAM timing parameters.

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

Here, $tCAS$ is the time between a column read command and the placement of data onto the data bus, $tCWD$ is the time between a column write command and the placement of data on the bus, $tBURST$ is the time that the data occupies the data bus, $tRTRS$ is the rank to rank switching time, $tWTR$ is the minimum time between a column write and a column read, $tRRD$ is the minimum time between two row activations, and finally, $tFAW$ is the four-bank activation window, a rolling time period during which no more than for bank activations can occur.

The dead time for reads is set to the worst case time between any read transaction and any other transaction. The dead time for writes is determined similarly. The dead time for each type of transaction determines when that type of transaction can no longer be issued. The dead time for reads is smaller than the dead time for writes, allowing multiple read transactions to be issued in a turn. For example, for the DRAM parameters used in this paper, the dead time for reads is 12 memory cycles whereas the dead time for writes is 18 cycles.

This optimization significantly improves performance for memory-intensive workloads.

## 7. EVALUATION

### 7.1 Methodology

To study the performance overhead of timing compartments a timing-protected multicore processor is simulated using gem5 [8] integrated with DRAMSim2 [44]. The simulations use the ARM ISA. We use simulations instead of our RTL prototype in order to evaluate performance for high-performance processing cores commonly used in cloud computing. 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 single-channel 667MHz 2GB DDR3 memory. The interconnects in the simulator run at 1GHz.

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. The number of cache ways and the network/memory bandwidths are evenly partitioned among timing compartments. Unless otherwise stated, the memory controller protection uses the minimum turn length (23) and the relaxed dead time optimization, which applies different dead times for reads and writes.

Our experiments use multiprogram workloads, and we describe our methodology precisely enough so that it can be repeated [28]. The simulations are fastforwarded until each benchmark has executed at least 1 billion instructions. Benchmarks may reach this threshold at different times, meaning the benchmarks which run faster will be fastforwarded for more instructions. However, detailed simulations begin from the same point for each workload and for all system configurations. After fastforwarding, results are collected with a detailed simulation until each core has executed for at least 100M instructions. Statistics are collected for each benchmark at the 100M instruction mark, but all benchmarks continue to run until the simulation ends, so that there is interference for the entire simulation for the insecure baseline.

### 7.2 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

$$\sum_{i=1}^{n} \frac{IPC_{MP,i}}{IPC_{SP,i}}, \qquad (1)$$

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.

#### 7.2.1 *Overall Performance Overhead*

Figure 6 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). Workload-aware resource allocation uses the amount of cache space and mem-

| | | | |
|---|---|---|---|
| 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.**

| 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.**

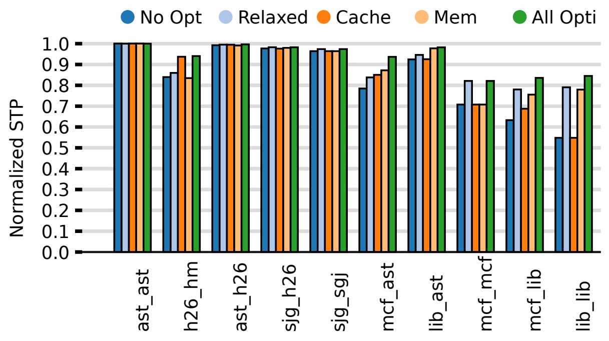

**Figure 6: Performance Overhead of Timing Compartments.**

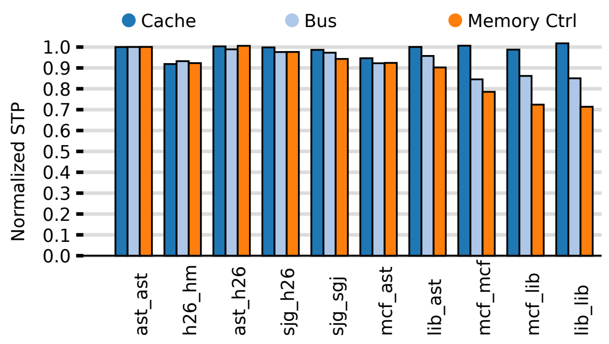

**Figure 7: Performance Breakdown (4 cores).**

ory bandwidth allocated to each application to match the general application demands. The allocation, however, is still fixed for the entire execution and thus only reflect general application characteristics, not data-dependent program behavior. 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.

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.

We note that 2 TCs are enough to support many practical application scenarios. For example, mobile security platforms such as ARM TrustZone [4] isolate applications into two worlds. Each world can be placed in a single TC regardless of the number of cores. As we show later, overheads scale with the number of TCs, not with the number of cores. Similarly, hardware compartments such as Intel SGX are designed to be used as a secure co-processor to run security-critical parts of an application and mostly likely to run one compartment at a time.

### 7.2.2 *Overhead Breakdown*

To better understand the sources of the performance overhead, the overhead of protection mechanisms were evaluated individually. Figure 7 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. 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 [55] 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.

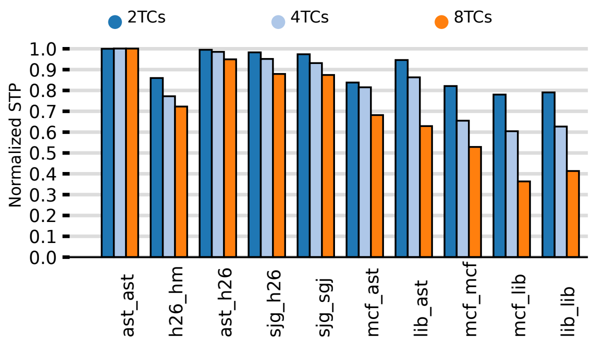

**Figure 8: Norm. STP of TCs as the number of TCs increases.**

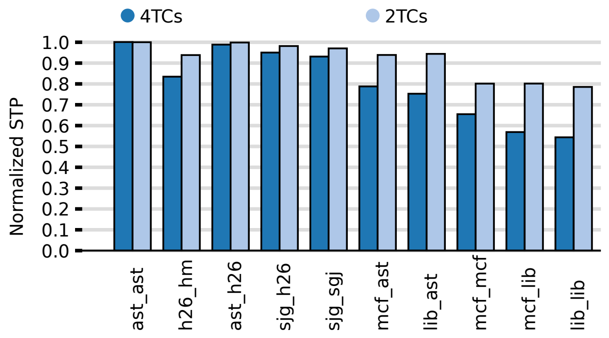

**Figure 9: Benefit of allowing 2 programs to share a TC.**

### 7.2.3 *Scaling the Number of TCs*

Figure 8 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 application characteristics. The performance overhead is low (less than 5%) for compute-intensive benchmarks even with a large number of TCs. Yet, the overhead of memory-intensive workloads increases with the number of TCs, 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. We believe that the overhead for processors with more cores will be still similar to these results with 2-4 cores.

The results suggest that many TCs 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. This is true even without TCs. In cloud computing environments, workloads can be dynamically profiled and more intensive workloads can be allocated to machines with are few memory intensive workloads [15].

### 7.2.4 *Using one TC for Multiple Cores*

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. For multi-threaded programs, one program can also use multiple cores running in the same timing compartment.

Using one timing compartment for multiple cores provides substantial performance improvements because cores within one com-

| Component | No protection ($\mu m^2$) | With protection ($\mu m^2$) | 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.**

partment can share resources as they would in a conventional system without timing protection. Figure 9 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% compared to using 4TCs. 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.

### 7.2.5 Multi-threaded Performance Overhead

The multiprogram workloads do not show the overhead of 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 [60] benchmarks on a 4-core system. For each experiment, we run two copies of a SPLASH-2 benchmark, each with two threads, in two TCs.

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.

### 7.2.6 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. We evaluated the STP of a 4-core system with 4TCs and 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. The overhead is at most 7% (for `h26_hm`) when flushing happens every 10ms.

## 7.3 Area and Frequency Overhead

The Verilog prototype used for security verification is also used to evaluate the area and frequency overhead. The design was synthesized with the Synopsys Design Compiler. Timing channel protection does not affect the clock frequency. Both designs with and without timing channel protection synthesized to 820MHz. The overall area overhead is 2.58%. This figure excludes the cache area which is large compared to the small baseline processor. Therefore, the area overhead for a full processor including caches will be even lower.

## 8. RELATED WORK

Timing compartments represent the first architecture to provably enforce timing-sensitive noninterference among concurrently executing processes on a shared multicore. Tiwari et al. [50, 51] have used information flow to formally guarantee that a process running on a single-core is noninterfering with publicly observable outputs. Execution leases [50] enforce strict upper bounds on the execution times of program subsections. A single-core secure processor has been developed with leases and verified [51] with information flow. However, they do not address timing channels introduced through fine-grained resource-sharing in a multicore processor. We view these problems as orthogonal, and these approaches can be combined.

Information flow tracking is a promising approach for formally verifying security properties of hardware designs. Gate level information flow tracking (GLIFT) [52, 39, 25] allows run-time information flow analysis at the gate level, requiring significant power, performance, and area overheads. Star Logic [40] permits GLIFT to be performed during simulation time, but requires enumeration of the entire state space to attain a static guarantee. Caisson [33] enforces information flow control in a hardware description language type system. Sapper [33] is another HDL type system which reduces the area overhead of Caisson by inserting run-time checks, removing the need for redundant registers. SecVerilog [65] both removes dynamic checks and redundant hardware with a dependent type system that permits hardware to be shared dynamically, but checked statically. Timing compartments are verified using SecVerilog. However, timing compartments are designed to be verifiable with any of the aforementioned information flow tools.

Timing compartments leverage prior proposals which use temporal partitioning to protect the network [59, 56] and memory controller [55]. Shaifee et al. [45] propose rank partitioning and the triple alternation to improve performance of temporally partitioned memory controllers. Both optimizations can be added to timing compartments. However, the previous study [45] showed that the triple alternation has comparable performance with bank partitioning used in this paper. Rank partitioning requires significant restrictions to memory allocation. Ferraiuolo et al. [19] propose optimizations for timing-channel protection for memory controllers when only uni-directional protection is necessary. These optimizations do not apply to timing compartments, which are mutually distrusting.

Timing compartments use way partitioning for cache protection. Liu et al. [35] demonstrated that way partitioning intended for performance isolation in conventional Intel processors can also be used for timing channel protection. Other approaches for cache timing-channel protection have also been proposed, but most cannot be verified with SecVerilog. For example, RPCache [58] and Random Fill cache [36] obfuscate cache timing by randomizing cache replacements and insertions respectively. These cannot be verified with information flow analysis, because they do not provide noninterference. Although the particular blocks chosen for eviction and insertion are randomized, accesses from one software entity still cause blocks owned by another entity to be evicted. This flow represents an actual leak of information — the number of accesses made by co-resident software is not hidden, and this has been used to carry out attacks [31]. NoMoCache [17] partitions some cache ways, but allows interference in other cache ways to reduce timing channel capacity. NoMoCache also cannot be verified, because it does not provide noninterference.

Many microarchitectural timing channels have been identified. This work is the first to point out vulnerabilities in cache coherence protocols and component interfaces. Vulnerabilities have also been found in caches [37, 42, 7, 41, 53, 10, 31, 23], branch predictors [1, 2], processor pipelines [57], networks on chip [56, 59], and memory controllers [55, 21]. Timing compartment removes timing channels between software running concurrently on multiple cores.

Recent studies have proposed detecting attempts to use covert timing channels. Hunger et al. [26] 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 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.

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 processor architectures [66, 18, 20, 47, 30, 32, 11, 13, 27, 9, 4, 29, 48, 50, 24, 61, 14] to provide strong isolation for both explicit and timing channels. Iso-X [18] 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 [47] extends compartments to systems managed by a hypervisor. Other architectures reduce the software TCB rather than eliminating it [30, 11, 32, 13]. Secure processor architectures have also been adopted commercially as well [27, 9]. None of these architectures prevent the timing channels among multiple cores that timing compartments removes. Because these architectures use similar approaches, we believe the timing compartments can be applied to many of them.

## 9. CONCLUSION

This paper presents timing compartments, the first architecture to provably enforce timing-sensitive noninterference among software concurrently running on a multi-core processor. Enforcing noninterference implies that timing compartments are comprehensive, removing timing channels instead of mitigating them. We have shown that timing compartments can be mechanically verified using static information flow analysis and be applied to both traditional systems and secure processor with an untrusted OS/hypervisor. Finally, experimental results show that timing compartments can have surprisingly low performance overhead, especially when a small number of compartments are used concurrently.

## 10. 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] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh. "Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller". In *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[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] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In *IEEE Symposium on Security and Privacy*, 1982.

[23] 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.

[24] 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.

[25] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner. Gate-level information flow tracking for security lattices. *ACM Trans. Des. Autom. Electron. Syst.*, 2014.

[26] 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.

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

[28] A. N. Jacobvitz, A. D. Hilton, and D. J. Sorin. "Multi-Program Benchmark definition". In *International Symposium on Performance Analysis of Systems and Software*, 2015.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 97–112, New York, NY, USA, 2014. ACM.

[34] 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.

[35] F. Liu, Q. Ge, Y. Yarom, C. Mckeen, Frank Rozas, G. Heiser, and R. Lee. "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing". In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, 2016.

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

[37] 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.

[38] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *SIGARCH Comput. Archit. News*, 40(3):118–129, June 2012.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

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

[45] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari. "Avoiding Information Leakage in the Memory Controller with Fixed Service Policies". In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

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

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

[66] 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.

[67] 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.