Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems

Robert Sison†∗, Scott Buckley∗, Toby Murray†, Gerwin Klein‡∗, and Gernot Heiser∗

† The University of Melbourne, Melbourne, Australia
∗ UNSW, Sydney, Australia
‡ Proofcraft, Sydney, Australia

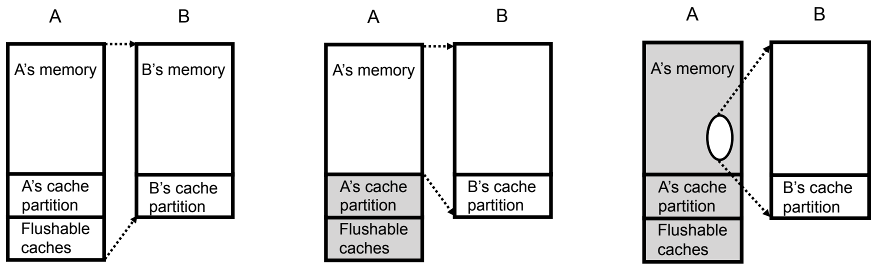
Abstract. Microarchitectural timing channels are a well-known mechanism for information leakage. *Time protection* has recently been demonstrated as an operating-system mechanism able to prevent them. However, established theories of information-flow security are insufficient for verifying time protection, which must distinguish between (legal) overt and (illegal) covert flows. We provide a machine-checked formalisation of time protection via a dynamic, observer-relative, intransitive nonleakage property over a careful model of the state elements that cause timing channels. We instantiate and prove our property over a generic model of OS interaction with its users, demonstrating for the first time the feasibility of proving time protection for OS implementations.

1 Introduction

Microarchitectural timing channels present a major attack vector on information security [11], with the Spectre attacks demonstrating that even seemingly innocuous code can be subverted into a Trojan that leaks secrets via such channels [19]. Ge et al. recently introduced *time protection* mechanisms to prevent microarchitectural channels, experimentally demonstrating their effectiveness [10] on a modified version of the seL4 operating system (OS) microkernel [18].

While seL4 comes with an extensive body of formal proofs, including information-flow enforcement and freedom from storage channels [23], these proofs do not consider properties about timing channels; the same is true for other OS security proofs [2, 7, 20]. Work that does consider timing does not extend to the full OS [4] or assumes mechanisms that are too expensive in practice [3].

Reasoning about timing channels is challenging, as timing is a non-functional property and hardware details that affect timing are intentionally unspecified to enable optimisations. While the correctness of time protection can, in principle,



(a) Channels that would be allowed by the property of Murray et al. [23, 24] at all times that A is running.

(b) *Time protection* excludes channels via microarchitectural state used by A when it is running.

(c) *Policy channels* our property can enforce to be open only when A makes a specific system call.

Fig. 1: Restriction of information flow allowed from domain A to B to occur only via the channel indicated by the dotted lines, i.e. not from any shaded regions.

be reduced to functional properties [14], to date there is not even a precise formulation of the security property it is meant to enforce.

Stating and verifying such a property faces two core challenges. Firstly, the complexity of microarchitectural state requires abstraction to make formal reasoning feasible, while retaining sufficient precision to allow proving a meaningful isolation property. Secondly, time protection is, by its nature, an *asymmetric* property: A (trusted) security domain, e.g. a downgrader, may have the right to communicate with another domain, but this *overt information flow* must happen in the absence of any *covert information flow* (through timing channels). This point is explained later in Section 4.1.

To address the first challenge we formalise (Section 3) an abstraction of the state elements related to temporal flows and their interaction with time, which distinguishes overt memory state from covert microarchitectural state. The abstraction separates that covert state according to the applicable elimination mechanisms (spatial or temporal partitioning), as proposed by Heiser et al. [14, 15] and based on a minimally-augmented *hardware–software contract* [12].

We address the second challenge (Section 4) by formalising a *dynamic* and *observer-relative* intransitive nonleakage property. This property generalises the one used for seL4 by Murray et al. [23, 24] (Figure 1a), to enforce elimination of flows via microarchitectural state (Figure 1b). We find that this requires a form of *policy channel specification*, which allows arbitrary *spatial* precision, i.e. to specify *from where* flows can occur. The natural formalisation of such specifications also supports arbitrary *temporal* precision, i.e. dynamic policy on *when* flows can occur, as depicted by Figure 1c. The observation relation specifying policy channels from a running domain A, as well as the granularity of steps between observation points, are then both relative to the observer domain B.

Finally, we instantiate and prove (Section 5) our property over a generic model of OS interaction with a formalisation of the threat scenario presented in

Section 2. In doing so, we demonstrate the first formal approach to capture a precise, fine-grained, time-protection property, even in the presence of the greater spatial and temporal precision of the policy channel specifications it enables.

All our results are formalised and machine checked using the Isabelle/HOL interactive proof assistant [25] and are provided as supplement material.

2 Threat scenario

We adopt the threat scenario of Heiser et al. [14]: A *spy* in one security domain attempts to obtain information from a *sender* in a different domain in violation of the system's security policy. We assume the defender's worst-case scenario where the sender is a Trojan that deliberately attempts to leak information, i.e. a *covert channel.* If we can prevent this information flow, we implicitly rule out inadvertent leaks (i.e. *side channels*).

Specifically we are looking at leakage through *microarchitectural timing channels* [11]. These result from microarchitectural state, i.e. hardware state hidden by the hardware–software contract (i.e. *instruction-set architecture*) but affected by program execution. This includes caches and other hardware features whose state depends on execution history, such as branch predictors and prefetchers.

We assume that the spy has access to an independent time source. The spy observes the speed of its own progress, looking for variance in execution speed that cannot be explained by any information to which it already has access (i.e. its own state). This includes the latency of memory accesses [13, 26, 27, 31], the latency of system calls operating on deterministic user state [10], or preemption periods resulting from interrupts [10].

Such latency variations can be the result of the sender's manipulation of microarchitectural state, which can happen through accessing memory in specific patterns, executing system calls with specific arguments, or initiating input-output (I/O) operations that result in interrupts at a time chosen by the sender.

Importantly, covert channels must be precluded even where overt channels are permitted. For example, consider an off-the-shelf web browser consisting of hundreds of thousands of lines of code; it handles secret information (e.g. passwords supplied by the user) but cannot be trusted to keep it secret. The web browser runs in security domain H. It communicates with the outside world via an untrusted network interface, running in domain L. The system's security policy requires that H can only communicate with L (and thus the outside world) via a trusted encryption/filter server, acting as a *downgrader*, running in domain D. While there is an overt channel $H \rightsquigarrow D \rightsquigarrow L$, the system must prevent H from using a timing channel that bypasses D. In addition, the system should prevent covert channels between H and D, even when an overt channel is permitted, and likewise between D and L, as otherwise D might unwittingly act as a courier for covert channel information between H and L.[1]

[1] Ensuring that it does not act as a courier requires very sophisticated reasoning about D, e.g. proving that it obeys constant-time programming discipline [1, 5].

We use the term *policy channel* to refer to information flow that is explicitly permitted by the system's security policy and represented by OS *protection state*, such as access control lists, or capabilities [9]; for example, seL4 uses the latter. *Time protection* then becomes the requirement that a confidentiality property is enforced, which prevents any information flow other than through policy channels. We propose such a policy in Section 4.

Reasoning about time protection requires a system model that makes timing channels explicit, by including microarchitectural state. The challenge is to keep this model sufficiently abstract to apply to a wide class of real processors, yet precise enough to allow reasoning about timing channels and their prevention. We present such a model in Section 3. In Section 5 we present an OS security model, parameterised over OS-specific features and processor-specific implementation details, using our model of microarchitectural state to prove enforcement of a time protection property; we hope it will serve as a roadmap for proving the effectiveness of time protection implementations (e.g. Ge et al. [10] for seL4).

3 Modelling channels by elimination strategy

As observed in Section 2, we need a model of microarchitectural state. This must be sufficiently precise to support reasoning about the absence of channels that exploit it, while abstracting away as many implementation details as possible, so it can apply to a large class of real processors. We defer to Section 5 a description of our full state model, depicted in Figure 3b; here, we explain the philosophy behind its microarchitectural and other timing-affecting elements.

Heiser et al. [14] observe that, to implement time protection, the OS only needs to know how microarchitectural state can be partitioned between security domains. Partitioning can either be *spatial*, where the OS can force a domain to only access a specific part of the state, or *temporal*, where state is exclusively owned by one partition at a time, and reset to a defined state when the OS hands ownership to another partition. Implementing time protection is possible if the hardware–software contract ensures that microarchitectural state can be partitioned either spatially or temporally, where the latter comes down to the OS being given a mechanism to reset (flush) that state.

For simplicity, we refer to state that can be spatially partitioned as *partitionable*, while state that can be temporally partitioned we call *flushable*.

To reason about how usermode execution may affect such state, we assume that this happens exclusively through referencing memory (data or instruction) addresses. This matches typical real hardware, for which any state directly accessed by programs is architected and explicitly context-switched by the OS; for such hardware, microarchitectural state can only be accessed indirectly via addresses. Note that programs (including the OS) can only issue *virtual addresses*.

3.1 Flushable microarchitectural state

For our purposes it is not necessary to distinguish between different parts of flushable state (e.g. caches vs. branch-predictor state vs. prefetcher state), even if the hardware provides different mechanisms for flushing different parts of it. We only need to deal with the complete collection of such state, and treat the sum of flushing mechanisms as a single operation.

Furthermore, it does not matter whether issuing distinct addresses affects different parts of microarchitectural state (as with caches) or causes different changes in the same state (as with state machines used in prefetchers). All that matters is capturing *which* state might be affected; for that we can make the worst-case assumption that each address in a domain maps to potentially different state, but some address in a different domain may map to the same state.

Thus we model *flushable state* (flst) as a simple function from address to boolean, representing whether a state referred by a particular address is currently affected – the entire flst is considered observable to the currently executing program. We model OS and user operations to modify flst in an under-defined way, assuming that secrets of the currently running domain can be stored in flst.

In the absence of flushing, our confidentiality property will not hold for this configuration, as secrets are being transmitted through flst. We therefore need to show that the OS performs the flush when switching domains, but also that any changes made to flst during the domain switch (during which the OS must issue addresses) have a deterministic effect on flst: After the flush, the OS must only issue addresses in a sequence that is not affected by user secrets.

3.2 Partitionable microarchitectural state

Partitionable state generally exists outside the processor core (typically caches other than the on-core first-level cache). Such state is accessed by *physical address*, meaning it has undergone address translation by the memory-management unit (MMU). Partitioning may use explicit hardware mechanisms, or may be achieved by the OS restricting the address mapping so that addresses from different partitions access disjoint cache state (this is referred to as *page colouring* [10, 16, 21]). This assumes that the OS understands how collisions may occur, which is realistic for contemporary hardware.

We model *partitionable state* (pst) as a function from address to boolean, similarly to flst. We do not explicitly model how cache collisions occur between addresses, instead we assume that the OS has set up the memory map to prevent collisions between partitions. By modeling both user and OS operations to only access memory visible to the present domain, we can show that no secrets are imprinted on the pst in a way that is visible to another domain.

However, the above assumptions break down if the OS accesses the same memory locations while operating on behalf of different user domains. This is unavoidable, as the OS must access memory while performing a domain switch, meaning that it is impossible to completely partition the OS's memory accesses. We call such non-partitionable memory *shared OS memory*, and its accesses may potentially leak secrets via their corresponding pst impacts.

We model this leak by parameterising our model over the set of all addresses in the shared OS memory in union with all addresses that collide with them. We then allow a user domain to affect the pst of that set, alongside pst corresponding to its own memory. This forces, at the time of domain switch, a flush of the pst for the shared OS addresses and their collisions, to ensure that they convey no secrets from the previously executing domain to the next one. The mechanism used, and cost incurred, for this flush depends on what the architecture offers.

3.3 Interrupts and other directly observed impacts on time

We have thus far encoded the spy's ability to make time-related observations via variations in memory access latency as direct user observations of flst and pst, as these are the primary channels through which execution time can vary. However, some leaks can happen through observation of the real-time clock, which are not directly related to shared microarchitecture.

Interrupts are inherently non-deterministic (their arrival depends on the environment beyond the control of the OS). This can be used as a channel [10]: a domain initiates an I/O operation such that the interrupt indicating completion arrives while another domain is executing. The time the OS takes to handle the interrupt can be observed as a gap in execution by the preempted domain.

Time protection prevents this channel by partitioning interrupts between domains, masking off any interrupts not belonging to the current domain. The preemption-timer interrupt, which causes a domain switch, is not subject to this masking; the rest we call *user interrupts*. We will model hardware masking of user interrupts by asserting that they can only arise during user execution depending on the state of devices belonging to the current domain (Section 5.2).

Our model enforces the following timing properties of interrupt arrivals:

- Timer interrupts arrive at a fixed interval, aside from delays described below.
- User execution will continue until halted either by the timer interrupt arriving at its fixed-interval time, or by a user interrupt arriving before that.
- There is a worst-case execution time (WCET) for handling an interrupt. Handling a user interrupt may then delay the arrival of a timer interrupt by an amount of time up to that WCET after its fixed-interval time.

As interrupt-related leaks are always via accurate observation of the *time*, we now turn to how we model time to be treated as observable by our property.

We model time as a numeric field of the state, tm, which we consider to be observable only by the currently-running domain; others are only able to observe at what point in the schedule the OS resides currently, but not tm directly.

Rather than modelling any automatic progression of time, we bake it in manually as the following assumptions to our under-defined user and OS operations:

- Flushing the flst will take some amount of time that depends only on the original flst state, up to some predefined WCET.
- Partially flushing the pst (for shared OS memory) takes an amount of time that depends only on the part of pst being flushed, up to a predefined WCET.
- Interrupt-handling OS operations obey a predefined WCET (as just noted).
- The OS can perform a *padding* operation that will progress time to a specified value, without changing any other state (by decrementing a register-held counter in a tight loop, or possibly using hardware support [30]).

Our domain-switch operation performs the following tasks:

1. a partial flush of OS shared memory addresses in pst;
2. a full flush of the flst;
3. changing the currently running domain over to the next domain, according to the deterministic schedule (see Section 4);
4. padding to the end of the allocated time.

We know that the padding at the end of this operation will always get us to a predetermined time, as we calculate the end-of-switch time to account for the WCETs for flst and pst flushes, as well as accounting for a potentially late start due to the handling of a user interrupt before domain-switch. At the start of such a domain-switch, as well as at many points in the middle of these operations, the tm field contains secrets from the domain who just executed. However, these secrets are removed when we pad time to a predetermined value.

The actual amount of time allocated to each domain, as well as the sequence of domains scheduled to execute, is completely predetermined; it is not possible to influence how long each timeslice will be, or which domain will execute next. This is implemented via an appropriately adapted *scheduler oracle* [23], whose details we relegate to the Isabelle/HOL supplement material.

4 Formalising Time Protection

What does time protection mean formally, and why are previous security definitions [24], used to state absence of storage channels in OSes, insufficient to express it? In this section we answer these questions by formalising a new dynamic and observer-relative intransitive nonleakage property.

Let *domain* be the set of *security domains* ranged over by u, v, w, etc. Following Murray et al. [23, 24], we distinguish *user domains*, which include one or more user-mode processes, from the *scheduler domain*, which represents the parts of the OS responsible for scheduling the execution of user-mode processes. Therefore let $\mathsf{sched} \in \mathit{domain}$ be the distinguished *scheduling domain.* At any time, a single domain is running, called the *current domain*: execution proceeds in a sequence of *steps* in which sched is interleaved between other domains (to choose the next domain that is to execute after the arrival of a timer interrupt).

Let *state* be the type of system states, s, t, etc. Then $\mathsf{dom}\ s$ denotes the current domain in state s. The part of the system state observable to domain u in state s is defined by an equivalence relation $\overset{u}{\sim}$, such that this part of the state is equal between states s and t iff $s \overset{u}{\sim} t$. The state of the scheduling domain includes which domain is currently running, hence: $s \overset{\mathsf{sched}}{\sim} t \implies \mathsf{dom}\ s = \mathsf{dom}\ t$ [24].

Information-flow security requires that for all domains u, for each step of execution, u only learns information it is supposed to. If time protection holds, what information is domain u allowed to learn? As in prior work [23, 24] we prove deterministic scheduling. Therefore, at all times, all domains are assumed to know which domain is the current domain, i.e. all information given by $\overset{\mathsf{sched}}{\sim}$. Domain u is also allowed to learn everything it can observe (equivalently, already knows), i.e. all information given by $\overset{u}{\sim}$. Finally, it is also allowed to learn certain information communicated to it by the current domain in that execution step.

4.1 State-Dependent Policy Channels

One of our key insights is that according to time protection, what constitutes this "certain information" depends on whether u is the current domain, i.e. time protection is an *asymmetric* property. Specifically, when u is not the current domain, time protection says that at most it is allowed to learn the information communicated by the current domain via *overt* channels (those allowed by the current protection state); but none via *covert* channels (including microarchitectural state). This is strictly *less* than all information observable by the current domain, given by $\overset{\mathsf{dom}\ s}{\sim}$ where s is the state from which the step occurred, because $\overset{\mathsf{dom}\ s}{\sim}$ necessarily includes microarchitectural state visible to the current domain (e.g. its caches influencing its execution speed as captured in our model by tm).

Thus, departing from prior nonleakage properties, for two domains v and u let $\overset{|v \rightsquigarrow u|}{\sim}$ be a state equivalence relation that defines the part of the state whose contents domain v is allowed to send to domain u on an execution step. If v is allowed to send no information to u, then $\overset{|v \rightsquigarrow u|}{\sim}$ is the trivial relation that holds for all states. As we demonstrate later in Section 5, this formulation is sufficiently general to specify dynamic policies, since the part of the state via which domain v is allowed to communicate with u can depend on the state itself. We call $\overset{|v \rightsquigarrow u|}{\sim}$ a *policy channel*, as it defines the allowed channel from v to u.

With these definitions we can define our top-level security property as follows. For now, the argument u to each of $\mathsf{obs\text{-}reachable}\ u\ s$ and $\mathsf{obs\text{-}Step}\ u$ can be ignored (we will explain its meaning directly, in Section 4.2); the former can be read as saying that state s is reachable via a finite number of steps of the latter.

Definition 1 (Observer-relative big-step confidentiality).

$$
\begin{aligned}
&\mathsf{obs\text{-}confidentiality}\ u \triangleq\\
&\forall s\ t.\ \mathsf{obs\text{-}reachable}\ u\ s \wedge \mathsf{obs\text{-}reachable}\ u\ t \longrightarrow\\
&\quad s \overset{\mathsf{sched}}{\sim} t \longrightarrow s \overset{|\mathsf{dom}\ s \rightsquigarrow u|}{\sim} t \longrightarrow s \overset{u}{\sim} t \longrightarrow\\
&\quad (\forall s'\ t'.\ (s, s') \in \mathsf{obs\text{-}Step}\ u \wedge (t, t') \in \mathsf{obs\text{-}Step}\ u \longrightarrow s' \overset{u}{\sim} t')
\end{aligned}
$$

This definition says that an arbitrary observer domain u, on an execution step, is allowed to learn the scheduler state ($\overset{\mathsf{sched}}{\sim}$), the information that the current domain is allowed to send it via the policy channel ($\overset{|\mathsf{dom}\ s \rightsquigarrow u|}{\sim}$), and anything it knew or could have observed already ($\overset{u}{\sim}$), by saying that if two initial, reachable states s and t agree on this information, then u's view of the states s' and t' reached after a single step must be identical: $s' \overset{u}{\sim} t'$.

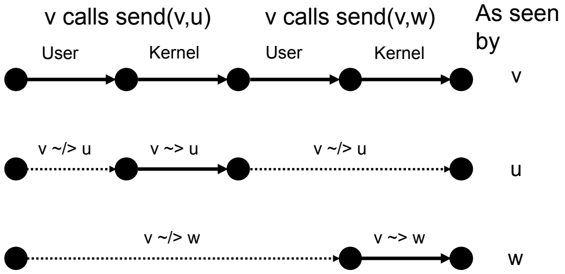


Fig. 2: Observer-relative state transition system model.

4.2 Policy-dependent state observability

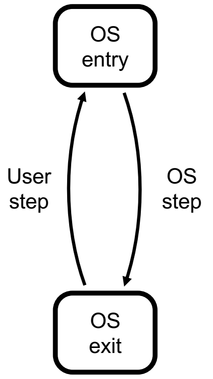
What constitutes an execution step? Our second major observation is that, for time protection, the answer depends on which domain is observing the execution.

Figure 2 provides an illustration of this phenomenon. It depicts (top row) a single timeslice of domain v (the current domain), in which it performs two consecutive system calls. First it makes a system call to communicate with domain u, and then subsequently to communicate with domain w. For each system call, a user-mode step occurs in which v first computes the data it wishes to communicate (e.g. the system call arguments), followed by an OS step in which the OS carries out the system call (i.e. puts into effect the communication). For simplicity, assume the protection state authorises both system calls.

Let us consider what each domain u and w is allowed to observe and when. From u's point of view (middle row of Figure 2), since v communicates with it, it implicitly learns that this communication has occurred. Thus it observes the occurrence of the second step (the first OS step) that v makes. However what v does before that step occurs should remain opaque: e.g. the precise number of user-mode state changes required to compute the system call arguments should not be revealed to u (though it can of course infer what the system call arguments must have been, and so those are observable to it).

From w's point of view (bottom row of Figure 2), all of this activity is opaque: the precise number of execution steps that v performs prior to the OS step that carries out the communication from v to w should remain hidden to w. Indeed from w's point of view, v performs a single execution step from the beginning of its timeslice up to the OS step that performs the v-to-w communication.

Thus execution steps are very much in the eye of the beholder. Our time protection formalisation captures this idea as follows. We require the existence of an underlying *small-step* transition system that defines the system's behaviour. From this we construct for each domain u its *observer-relative*, *big-step* transition system that defines u's view of the system's execution by coalescing together consecutive small-steps between states that are unobservable to u. Which states are observable to u is naturally captured in the policy channel specification $\overset{|v \rightsquigarrow u|}{\sim}$:



(a) Transition system.

record $\mathsf{state}\ \triangleq$

$\mathsf{mem} :: \mathit{mem}$ /* Where $\mathit{mem} \triangleq \mathit{addr} \Rightarrow \mathsf{int}$ */

$\mathsf{flst} :: \mathit{addr} \Rightarrow \mathsf{bool}$ /* Flushable microarch. */

$\mathsf{pst} :: \mathit{addr} \Rightarrow \mathsf{bool}$ /* Partitionable microarch. */

$\mathsf{tm} :: \mathsf{nat}$ /* Time */

$\mathsf{dom} :: \mathit{domain}$ /* Current domain */

$\mathsf{devs} :: \mathit{device}\ \mathsf{set}$ /* Interrupt-generating devices */

$\mathsf{event} :: \{\mathsf{Syscall}, \mathsf{UserInterrupt}, \mathsf{TimerInterrupt}\}$

$\mathsf{args} :: \mathit{args}$ /* System call arguments */

$\mathsf{prot} :: \mathit{prot}$ /* Protection state */

(b) State model.

Fig. 3: Generic OS on which we model enforcement of time protection.

When $s \overset{|\mathsf{dom}\ s \rightsquigarrow u|}{\sim} t$ such that this relation is non-trivial (i.e. the current domain is allowed to communicate with u), then state s is observable to domain u.

This explains why Definition 1 takes the observer domain u as an argument. This definition is defined against a set of big-step transition systems, one for each domain u, that defines u's view of the underlying system's small-step transition system. The state reachability predicate is also parameterised by the observing domain u: $\mathsf{obs\text{-}reachable}\ u\ s$ means that state s is reachable in u's big-step transition system. Similarly for the step relation: $\mathsf{obs\text{-}Step}\ u$ is the set of state transitions in the big-step transition system that represents u's view.

Our confidentiality property (Definition 1) is a generalisation of the one used for the seL4 microkernel [23, 24], which is an intransitive nonleakage property that confines the allowed information flows to those according to a static (possibly intransitive) information flow policy "$\rightsquigarrow$": $v \rightsquigarrow u$ holds iff the policy permits information to flow directly from domain v (while v is the current domain) to domain u. Our property generalises theirs in that (1) fixing our policy channel relation $\overset{|\mathsf{dom}\ s \rightsquigarrow u|}{\sim}$ to be $\overset{\mathsf{dom}\ s}{\sim}$ whenever $\mathsf{dom}\ s \rightsquigarrow u$ and the universal set otherwise, and (2) fixing all big-step transition systems to be the same for all observers u, results in their property ($\mathsf{confidentiality\text{-}u}$, p9 of Murray et al [23]).

5 System model of OS-enforced time protection

We have thus far presented an explanation of how to model covert state (Section 3) and a confidentiality property capable of distinguishing between overt and covert state (Section 4), both as needed to express and model time protection.

Here we apply these principles to demonstrate that, regardless of the set of system calls supported by the OS and the architecture that it runs on, an OS designer now has the means to prove formally that an OS implementation for any given architecture enforces time protection, as long as the designer: (1) can prove that each of its system call handling routines permits only information flow via policy channels that exclude microarchitecture, (2) has proved or can reliably assume the functionality of certain architectural features key to time protection, and (3) has reliably measured and bounded the WCETs of the above.

5.1 Model overview and property

We achieve this level of generality with a model (Figure 3) that abstracts the essential elements of OS-enforced time protection over the following parameters:

1. an OS-specific set of system calls, their implementations, and specifications of their policy channels that can depend on arguments and protection state;
2. architecture-specific implementations of
   (a) an interrupt handling routine and
   (b) a domain switch routine that occurs on timer interrupt;
3. the WCETs of all of the above; and
4. the types of memory addresses *addr*, domain IDs *domain*, IRQ-generating device state *device*, syscall arguments *args*, and protection state *prot*.

Over this model, we instantiate our property so that Definition 1 expresses that the OS enforces time protection, in the sense that information only ever flows to a given user domain from the current domain's overt state elements (like a system call's arguments and relevant memory) as specified by some policy channel, and not ever via any covert state elements (like microarchitecture and user-configured device interrupts) that impact timing. Importantly, time (tm) and microarchitectural state that could influence memory access time (the entire flst, and the relevant pst partition) are always considered observable to the current domain, so our property ensures the absence of timing channels as directly observed after domain switch, and from subsequent variations in memory access latency, respectively. Formally, we then prove that Definition 1 holds at all times as seen by every possible user domain:

Theorem 1 (OS model enforces confidentiality with time protection).

$$\forall u.\ \mathsf{obs\text{-}confidentiality}\ u$$

Our model and its proofs are mechanised in about 7.9K lines of Isabelle/HOL proof script, of which about 2.1K lines are the adaptations described in Section 4 of prior mechanised theory [23, 24, 28] – all are provided as supplement material. This includes an instantiation of our generic model, for a pair of system calls with tightly specified policy channels, to ensure it is nontrivial.

We now describe the requirements our generic model imposes on an OS and its configuration that make it possible to prove that it enforces time protection. In particular, we believe that any OS that implements time protection, such as that of Ge et al. [10] for seL4, can satisfy all these requirements.

5.2 User steps

The user-step model captures requirements on the memory subsystem and device hardware and their adequate configuration by the OS to ensure the partitioning between domains of (i) memory, through address *mappings*; (ii) caches, through *colouring*; and (iii) user-configured interrupts, through *masking*.

It also captures assumptions about the spy and sender: They can choose when to make a syscall and with which argument values, but cannot directly modify the OS protection state; furthermore, they can program their devices to cause interrupts. Thus, we model the reason for OS entry, system call arguments, and device state in the state fields event, args, and devs respectively (see Figure 3b) and specify the user step as free to choose them in a manner dependent on the state accessible to the currently running domain. In contrast, we model protection state with field prot but disallow the user step from modifying it.

Memory and cache partitioning We partition both memory and pst by security *domain*, assuming a mapping addr-domain :: $addr \Rightarrow domain$. We consider the parts of mem that belong to some domain u to be the input addresses where addr-domain $a = u$, and the same for pst.

User steps are restricted to those that do not read from or write to any part of mem that does not belong to the executing domain u. A similar restriction is enforced for pst, except that we do allow user steps to modify parts of the pst outside of u's domain if they are in the shared OS address set. While in reality OS memory will only be affected by system calls, allowing user modification of shared OS memory is a sound over-approximation that simplifies our model.

We implement these restrictions by "quarantining" a transition: we mask off any state that should not be accessed, perform the transition on this modified state, and then return the masked-off data to the output state. A transition that does not read or write outside of its domain will not be modified by quarantining. This process is similar to prior models of OS memory protection [8].

In reality, this kind of memory protection is implemented by the OS correctly configuring the MMU and is covered by typical integrity proofs [29]. Cache partitioning might be implemented via *colouring*. Shared OS memory is, by definition, not partitioned.

Interrupt and device partitioning We partition interrupts by domain, via partitioning *devices* by domain. The state field devs abstracts the states of a set of devices, each assigned to some domain by a parameter device-domain :: $device \Rightarrow domain$. Note it is this assignment of ownership, and not the *device* type we abstract over, that matters for our model; furthermore, we assume that separate devices do not communicate with each other.

We abstract interactions with devices via mostly-arbitrary modifications to the devs field, specifying that user steps (as well as interrupt-handling and syscall steps) can only access or modify the device subset belonging to the currently executing domain. Further, we model the user as being able to choose the event indicating the reason for entry into the OS to be set to UserInterrupt or TimerInterrupt in a manner dependent on that device subset. This model allows us to reason about users interacting with devices outside of the observability points between transitions, and allows for interrupts to be raised at any point during a user's execution, and for their details and timing to be influenceable only by that user.

- If a TimerInterrupt has caused the end of execution, then the time must be at some ideal timer interrupt point, or possibly delayed by up to the interrupt-handling WCET. At this point we will perform a domain-switch.
- If execution was ended by a user interrupt or a syscall, the time must be strictly *before* the ideal timer interrupt point. At this point we will perform the syscall or handle the user interrupt.

In a real-world OS, we expect the interrupt partitioning abstraction to be implemented via the *masking* of interrupts associated with non-running domains, where any two partitions have a disjoint set of unmasked interrupts, with the OS switching the mask when switching partitions. The timing constraints result from the timer inevitably arriving, resulting in a domain switch.

5.3 OS steps

The OS-step model captures requirements on domain switch (triggered by a deterministic timer interrupt), syscall handling, and handling of unmasked user interrupts, each indicated by the event field of the state taking the value TimerInterrupt, Syscall, or UserInterrupt respectively at OS entry.

Domain switch The domain-switch step contains the most concretely defined semantics of any step in our model. Once the appropriate interrupt has arrived, the domain-switch step will pass execution on to another security domain, and will execute some security measures to prevent leaks through microarchitectural state. The specific semantics of this step are as follows.

1. Partially flush the pst: flush all addresses conflicting with shared OS memory; time bounded by w_1.
2. Flush the flst: set the entire flst to a predefined value; time bounded by w_2.
3. Change the domain: update the *dom* field of the state according to the schedule oracle; time bounded by w_3.
4. Pad the execution time to make the overall latency constant.

If T_0 is the ideal time for the timer interrupt to occur, w_0 the WCET of handling another interrupt (i.e. the maximum time by which the timer interrupt may be delayed), and w_3 includes any operations for handling the timer interrupt that are not related to time protection (such as saving processor state), then Step (4) defers further execution until time $T_0 + w_0 + w_1 + w_3 + w_3$.

The result of this padding is that nothing a domain can influence will change the exact time when execution is passed to the next domain. The new domain will also begin execution with an empty flst, and a pst partition that is unchanged apart from the shared OS addresses and its collisions, which have been flushed.

These same operations can be performed in a real-world OS, using mostly hardware-provided primitives to perform flushing and a busy loop for padding if no time padding primitive [30] is provided by the hardware.

Syscall handling and policy channels We model system call handling as consisting of (1) a decode phase that determines whether the requested operation is permitted by the protection state, and if so, (2) a commit of the requested operation. Moreover, policy channels (the allowed information to be transmitted) for system calls are specified via a parameter commit-channels of the form *mem* rel, thereby excluding any covert parts of the state by construction. Whether a system call transmits information to domain u, and what information it may transmit, depend on which system call was made (i.e. the system call arguments args), and whether it is authorised (i.e. the protection state prot). These together form the *policy-determining state fields* that, with commit-channels, are used to define $\overset{|\mathsf{dom}\ s \rightsquigarrow u|}{\sim}$. Specifically $\overset{|\mathsf{dom}\ s \rightsquigarrow u|}{\sim}$ is defined so that (a) the commit-channels information is revealed to u only when args and prot imply that u is the recipient of the system call and the system call is authorised, and (b) args and prot themselves are allowed to be revealed to u only under these same conditions.

Thus when a domain makes a system call, other domains can learn about it only when that system call is authorised, in which case only the recipient of the syscall gets to learn about its occurrence, and all they can learn is the intended information transmitted by the syscall.

To prove this, we impose on the user-supplied parameters the proof obligations: (1) an integrity property enforcing the decode phase should only inspect, not change, any of policy-determining state fields; and (2) a confidentiality property on the commit phase enforcing it obeys the policy induced by these fields, i.e. that any changes to state accessible to the observing domain u flow only via locations specified by $\overset{|\mathsf{dom}\ s \rightsquigarrow u|}{\sim}$.

We provide an example instantiation of the parameters that we prove meets these obligations: a simple model of capability-based access control over a broadcast/subscribe pair of system calls for one-way messaging between domains.

User interrupt handling We underspecify interrupt-handling operations in our model: We model the timer interrupt, but no specific operations in response to user interrupts. We assume that in response to a user interrupt, the OS will perform some action that may modify parts of the user state – this matches microkernels, like seL4, that relegate interrupt handling to user domains.

Consequently, we model the user interrupt very similarly to the user step operation: We specify that only the appropriate parts of mem and pst can be read or modified, limit modifications to devs to those belonging to the currently domain, and do not allow modifications to the prot state.

Bringing all of this together, we define a small-step transition system alternating (Figure 3a) between user steps restricted as described in Section 5.2 and OS steps as described in Section 5.3; this forms the basis for the observer-relative big-step systems upon which our security property is stated (Theorem 1).

6 Related work

Prior proofs of confidentiality for OSes [7, 20, 23] have generally focused only on covert flows via storage channels (memory and registers), ignoring time and microarchitectural state.

Barthe et al. presented formal proofs of the elimination of cache channels by flushing the complete cache hierarchy on a context switch [3], an expensive approach that also does not deal with other microarchitectural state. The same group verified *stealth memory* [17], where the OS reserves some pages and all their cache aliases for cryptographic applications [4]; this protects against side-channel attacks but cannot prevent a Trojan leaking through non-stealth memory. To our knowledge, no prior formalisation of OS security concurrently deals with both partitionable and flushable state, as required by time protection.

Liu et al. [22] prove a property they call *temporal isolation* for an extension of the mCertiKOS kernel with real-time scheduling. Despite its name, their property does not rule out timing channels between domains. Rather it focuses on the scheduler, proving that the behaviour of one domain cannot interfere with the scheduling of another (e.g. by preventing it from missing a deadline). This work might be combined with our approach in future to extend ours beyond the confines of simple deterministic domain scheduling.

Our security property (Definition 1) allows for specifying dynamic policies, in which the information allowed to be released on a step depends on the current state. This is a feature also present in prior information flow proofs for OSes [7, 20, 23] and we expect policies from prior work, including the recent work of Li et al. [20], should be expressible in our framework. Unlike our property, however, none of these prior works allow for defining certain states as being observable to some observers but not others, which we argued in Section 4.2 is crucial for a precise statement of time protection with dynamic policies.

The use of two distinct phases to model system call handling, first establishing preconditions that then guarantee that the call can succeed, was used in EROS [6] and adopted by seL4 [18].

7 Conclusions

We have presented a fully machine-checked formalisation of time protection and its enforcement by an operating system that is generic enough to be adapted to any individual OS implementation on any architecture that provides the necessary hardware support. By proving our time protection property relative to the requirements formalised by our OS model, we provide a roadmap for such future OS verification efforts.

This work demonstrates for the first time the feasibility of formal proof of the elimination of microarchitectural timing channels between users by the operating system they are running on. We hope this work helps to raise the level of assurance and responsibility taken by OSes to protect the confidentiality of their users, while serving to clarify what must be asked of the architectures that will make their implementation possible.

Acknowledgements

We thank our anonymous reviewers, as well as Johannes Åman Pohjola for his feedback on our manuscript. This paper describes research that was co-funded by the Australian Research Council (ARC Project ID DP190103743).

Bibliography

[1] Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX Security Symposium. pp. 53–70 (2016)

[2] Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Formally verifying isolation and availability in an idealized model of virtualization. In: Proceedings of the 17th International Symposium on Formal Methods (FM). Lecture Notes in Computer Science, vol. 6664, pp. 231–245. Springer (2011)

[3] Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Cache-leakage resilient OS isolation in an idealized model of virtualization. In: Proceedings of the 25th IEEE Computer Security Foundations Symposium. pp. 186–197. IEEE (2012)

[4] Barthe, G., Betarte, G., Campo, J.D., Luna, C.D., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014. pp. 1267–1279. ACM (2014), https://doi.org/10.1145/2660267.2660283

[5] Cauligi, S., Disselkoen, C., von Gleissenthall, K., Tullsen, D.M., Stefan, D., Rezk, T., Barthe, G.: Constant-time foundations for the new spectre era. In: Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 913–926. ACM (2020), https://doi.org/10.1145/3385412.3385970

[6] Chen, H., Shapiro, J.S.: Using build-integrated static checking to preserve correctness invariants. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004. pp. 288–297. ACM (2004), https://doi.org/10.1145/1030083.1030122

[7] Costanzo, D., Shao, Z., Gu, R.: End-to-end verification of information-flow security for C and assembly programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 648–664 (2016)

[8] Daum, M., Billing, N., Klein, G.: Concerned with the unprivileged: User programs in kernel refinement. Formal Aspects of Computing **26**(6), 1205–1229 (Oct 2014)

[9] Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Communications of the ACM **9**, 143–155 (1966)

[10] Ge, Q., Yarom, Y., Chothia, T., Heiser, G.: Time protection: the missing OS abstraction. In: EuroSys Conference. ACM, Dresden, Germany (Mar 2019)

[11] Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering **8**, 1–27 (Apr 2018)

[12] Ge, Q., Yarom, Y., Heiser, G.: No security without time protection: We need a new hardware-software contract. In: Asia-Pacific Workshop on Systems (APSys). ACM SIGOPS, Korea (Aug 2018)

[13] Gullasch, D., Bangerter, E., Krenn, S.: Cache games – bringing access-based cache attacks on AES to practice. In: Proceedings of the IEEE Symposium on Security and Privacy. pp. 490–505. IEEE, Oakland, CA, US (May 2011)

[14] Heiser, G., Klein, G., Murray, T.: Can we prove time protection? In: Workshop on Hot Topics in Operating Systems (HotOS). pp. 23–29. ACM, Bertinoro, Italy (May 2019)

[15] Heiser, G., Murray, T., Klein, G.: Towards provable timing-channel prevention. ACM Operating Systems Review **54**, 1–7 (Aug 2020)

[16] Kessler, R.E., Hill, M.D.: Page placement algorithms for large real-indexed caches. ACM Transactions on Computer Systems **10**, 338–359 (1992)

[17] Kim, T., Peinado, M., Mainar-Ruiz, G.: StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: Proceedings of the 21st USENIX Security Symposium. pp. 189–204. USENIX, Bellevue, WA, US (Aug 2012)

[18] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: ACM Symposium on Operating Systems Principles. pp. 207–220. ACM, Big Sky, MT, USA (Oct 2009)

[19] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution[abridged version]. Communications of the ACM **63**, 93–101 (Jun 2020)

[20] Li, S.W., Li, X., Gu, R., Nieh, J., Hui, J.Z.: A secure and formally verified Linux KVM hypervisor. In: IEEE Security and Privacy (2021)

[21] Liedtke, J., Härtig, H., Hohmuth, M.: OS-controlled cache predictability for real-time systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 213–223. IEEE, Montreal, CA (Jun 1997)

[22] Liu, M., Rieg, L., Shao, Z., Gu, R., Costanzo, D., Kim, J.E., Yoon, M.K.: Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. Proceedings of the ACM on Programming Languages **4**(POPL), 1–31 (2019)

[23] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: seL4: from general purpose to a proof of information flow enforcement. In: IEEE Symposium on Security and Privacy. pp. 415–429. IEEE, San Francisco, CA (May 2013)

[24] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for operating system kernels. In: International Conference on Certified Programs and Proofs. pp. 126–142. Springer, Kyoto, Japan (Dec 2012)

[25] Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)

[26] Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Proceedings of the 2006 Crytographers' track at the RSA Conference on Topics in Cryptology. pp. 1–20. Springer, San Jose, CA, US (2006)

[27] Percival, C.: Cache missing for fun and profit. In: BSDCan 2005. Ottawa, CA (2005), http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf

[28] seL4 microkernel code and proofs, https://github.com/seL4/

[29] Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 enforces integrity. In: International Conference on Interactive Theorem Proving. pp. 325–340. Springer, Nijmegen, The Netherlands (Aug 2011)

[30] Wistoff, N., Schneider, M., Gürkaynak, F., Benini, L., Heiser, G.: Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V core. In: Design, Automation and Test in Europe (DATE). IEEE, virtual (Feb 2021)

[31] Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: Proceedings of the 23rd USENIX Security Symposium. pp. 719–732. USENIX, San Diego, CA, US (2014)