A Case Study in Information Flow Refinement for Low Level Systems

Roberto Guanciale[1], Christoph Baumann[2], Pablo Buiras[1], Mads Dam[1], and Hamed Nemati[3]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
{robertog,pablo,mfd}@kth.se
[2] Ericsson Research Security, Kista, Sweden christoph.baumann@ericsson.com
[3] Stanford University, Stanford, United States and CISPA Helmholtz Center for Information Security,Saarbrücken, Germany hnnemati@stanford.edu

Abstract. In this work we employ information-flow-aware refinement to study security properties of a separation kernel. We focus on refinements that support changes in data representation and semantics, including the addition of state variables that may induce new observational power or side channels. We leverage an epistemic approach to ignorance-preserving refinement where an abstract model is used as a specification of a system's permitted information flows that may include the declassification of secret information. The core idea is to require that refinement steps must not induce observer knowledge that is not already available in the abstract model. In particular, we show that a simple key manager may cause information leakage via a refinement that includes cache and timing information. Finally, we show that deploying standard countermeasures against cache-based timing channels regains ignorance preservation.

1 Introduction

The last decade has seen a number of formally verified separation kernels [1, 18, 25], which can provide strong isolation among software components. Nevertheless, their resilience against sophisticated attacks that use low level microarchitectural features [32, 40, 24, 27, 33, 34, 3] is not proven. For realistic microarchitectures a monolithic analysis of kernel's confidentiality properties is not feasible, since this would require to take into account caches, multiple cores, pipelines, buses, GPUs, devices, and so on. To cope with this complexity, a modular approach based on some form of refinement [2] is essential, since it would allow to handle different security threats at different abstraction levels. The main problem is that standard refinements (e.g., trace inclusion) do not support confidentiality properties [36]: the refined model may provide an observer with information of the execution environment such as execution time or power consumption that may introduce *side channels*.

We have recently developed [11] a theory for secure refinement that supports data refinement, i.e., changes in data representation, reducing nondeterminism and underspecification, and adding state variables that may introduce discriminating power. The key idea is to use the abstract model as a specification of the permitted information flow, and then to ensure that this flow is an upper bound of the corresponding flow in the refined model. We achieve this using knowledge in the sense of [20]. If the progression of observer knowledge on all refined computations is the same or weaker than the one on corresponding abstract computations then the refined model does not leak more information than the abstract model, hence we say it is a *Ignorance-Preserving Refinement*, IPR.

Here we demonstrate our theory by analysing a provably secure low-level system. The system consists of a kernel and processes that use different types of communication, provide services to each other, and operate concurrently. Attackers are compromised processes with unknown behaviour that attempt to acquire secret information from the trusted victims.

For our case study, we formalize a machine with a flat memory. Since verifying functional correctness of the kernel is out of our scope, we axiomatize the expected kernel properties on the abstract model. In order to demonstrate the notion of knowledge, we introduce a simple key manager, which is a process that owns a secret key and allows a potentially malicious client (i.e., another process) to obtain the key-dependent Message Authentication Code of input data.

The goal of IPR is to only protect secrets that are already represented at abstract level, hence IPR does not imply noninterference preservation "out of the box": a concrete model may introduce new types of unrelated implementation information that can be freely leaked. We illustrate the usage of IPR via a refined model, where we add caches and timing information. The question we then need to answer is if this refinement step can cause information to be leaked that would not be possible in the abstract model. Unsurprisingly, given the many results on this topic in the literature [40, 21, 44], we show that for the key manager the answer is affirmative, due to timing differentials when caches are involved. We then demonstrate that IPR is regained when a simple countermeasure is deployed.

In general, IPR is not sequentially composable, therefore some procedures are secure only if they are executed once. In this paper we also show how sequential compositionality for IPR can be achieved via a kind of relational Hoare logic [12] lifted to refinements. We further apply our proposed solution to verify that IPR is guaranteed when cache colouring [30] or constant time programming, both standard countermeasures against cache-based timing channels, are deployed.

To make presentation easier to follow, we structure the paper by interleaving the theoretical definitions and results of [11] with their application to the case study. In particular, in Section 2 we introduce an abstract modeling framework and the epistemic notions of knowledge and ignorance, which allow to formalize the abstract information flows. Section 3 presents instantiation of the abstract framework to our case study. In sections 4, 5 and 6 we present our account of refinement and IPR and show the usage of IPR via a refined model. Sections 7 and 8 discuss our solution to make IPR sequentially composable and apply it to some examples countermeasures against cache-based timing attacks. Finally, in Sections 9 and 10 we discuss related work and give our concluding remarks.

2 Models, Knowledge, and Ignorance

The models we consider in this work are extended transition systems equipped with a store and an observation function. A *model* $\mathcal{M} = (S_0, S, \rightarrow, L, PId, \mathcal{O}, Obs)$ is a labeled transition system with a set of states $S = Var \rightarrow Val$ that map variables from Var to values in Val; $S_0 \subseteq S$ the set of initial states; $\rightarrow \subseteq S \times L \times S$ the transition relation; $\mathcal{O}$ a set of observations, PId a set of process, or observer id's, and for each $p \in PId$ an *observation function* $Obs_p : S \rightarrow \mathcal{O}$ that returns the observations in $\mathcal{O}$ an observer can make in a given state.

We let s, s' range over states, α, β range over observations and write $s \text{-}\alpha_p\text{→} s'$, if $Obs_p(s') = \alpha$ and $s \rightarrow s'$. A *final state* is any state s in which no transition starts, i.e., for which no s' exists such that $s \rightarrow s'$. Observation functions Obs_p allow to encode both statically determined sets of variables observed by process p and dynamically varying notions of observability.

In the context of a given transition system, a *run* $\rho \in \mathcal{R}(\mathcal{M})$ is a finite sequence $s_0 \cdots s_n$ such that $s_0 \in S_0$ and $s_{i-1} \rightarrow s_i$ for all $i > 0$ for which s_i is defined. The i'th state of ρ, $\rho(i)$, is s_i, the first (last) element of ρ is $fst(\rho)$ ($lst(\rho)$), $|\rho| \in \mathbb{N}$ is the length of ρ (i.e. number of states in the run), and $\rho(: i)$ is the prefix of ρ having length i. A *complete* run is one that cannot be extended, i.e. there is no s such that $\rho(|\rho| - 1) \rightarrow s$. In that case $lst(\rho) = \rho(|\rho| - 1)$ is final.

The notions of observation trace, observation equivalence, and the epistemic notions of knowledge and its dual, ignorance, are standard. First, two states s_1, s_2 are observationally equivalent as seen by process p, $s_1 \sim_p s_2$, if p has the same observations in the two states, $Obs_p(s_1) = Obs_p(s_2)$. We write $\langle s \rangle_p$ for the equivalence class that contains s. An observation trace for p is the sequence of observation of some run ρ, i.e. $Obs_p(\rho) = Obs_p(\rho(0)) \cdots Obs_p(\rho(|\rho| - 1))$. A complete trace is a trace of a complete run, and the runs ρ_1 and ρ_2 are p-observation equivalent $\rho_1 \sim_p \rho_2$, if p's observations in ρ_1 are the same as p's observations in ρ_2, i.e. $Obs_p(\rho_1) = Obs_p(\rho_2)$. To avoid clutter, we often omit the "p"-subscript when understood from the context.

In the context of a given model $\mathcal{M}$ we view a *property* as a set $\phi \subseteq \mathcal{R}(\mathcal{M})$, namely the set of runs for which the property holds. This allows to define the standard epistemic modality $K_p\phi$ of perfect recall knowledge and its De Morgan dual $I_p\phi$ of "ignorance" on properties ϕ in the following way:

- $\rho \in K_p\phi$, if for all $\rho' \in \mathcal{R}(\mathcal{M})$, if $\rho' \sim_p \rho$ then $\rho' \in \phi$.
- $\rho \in I_p\phi$, if there is $\rho' \in \phi$ such that $\rho' \sim_p \rho$.

In this paper we focus on confidentiality properties, i.e. observer ignorance rather than knowledge.

The set $I_p\phi$ is the set of runs ρ that are "compatible" with some ρ' in ϕ in the sense that p cannot tell ρ' from ρ. Thus, if ϕ holds for ρ', for all p can tell ϕ may hold for ρ as well. Accordingly, we call a set ϕ a *p-ignorance set* if ϕ is closed under $\sim_p$. The *initial ignorance* of a property ϕ is the set of initial states of runs in $I_p\phi$, i.e. $I_p^{\text{init}}\phi = \{\rho'(0) \mid \rho' \in I_p\phi\}$.

If the transition relation is deterministic, for each initial state there is a unique maximal run. We use the notation $\mathcal{R}(s)$ to identify the maximal run starting from state s, and $\mathcal{R}(s, n)$ to identify the run starting from s and taking exactly n steps. These definitions are extended pointwise to sets of states, so if S is a set of states, then $\mathcal{R}(S)$ is the set of maximal runs starting from states in S, and $\mathcal{R}(S, n)$ is the set of runs of length n starting from states in S. Notice that for a deterministic system the standard notion of non-interference can directly expressed in terms of initial ignorance:

Proposition 1. *A system is non-interfering from m to n if for every pair of runs $\rho \sim_p \rho' \in \mathcal{R}(S_0, m)$ it holds that $\mathcal{R}(\rho'(0), n) \subseteq I_p(\mathcal{R}(\rho(0), n))$. A deterministic system is non-interfering from m to n iff for all $I_p(\mathcal{R}(S_0, n)) = \mathcal{R}(I_p^{init}(\mathcal{R}(S_0, m)), n)$.*

That is, after m transitions from the initial state, a system does not leak information for $n - m$ transitions if the ignorance after n transitions is equal to the runs obtained by staring from a state that was in the initial ignorance after m transitions.

3 Case study: Processor Model and Separation Kernel

We introduce the abstract processor model used in the paper, ignoring things like caches and time related to the refined model introduced in Section 6. The example is based on an operating system that allows processes $\{0, \ldots, N\}$ to execute on a sequential processor, where process number 0 represents the kernel and others are unprivileged processes.

A state of the abstract model is a total function $s : \mathit{Var} \to \mathit{Val}$, where $\mathit{Var} = \mathit{regs} \cup \mathit{mem}$, regs is the set of registers (including special purpose registers that control memory protection, program counter, etc.), and mem is the set of memory addresses.

The transition relation $s \xrightarrow{l} s'$ is deterministic and represents the execution of a single machine instruction. We annotate the transition system with label l to capture the list of memory operations performed by the instruction. This list includes all addresses that are involved in the elaboration of instructions such as page tables and instruction memory. Operations (rd, a) and (wt, a) model the reading and writing of address a respectively. We use $R(l)$ and $W(l)$ to extract the set of read/written addresses.

The following notation allows us to abstract from the kernel, exposing common abstractions that depend on special purpose registers and the internal kernel data structures. We use $P(s) \in \mathit{PId}$ to identify the active process in s. The kernel control's memory resources allocation and configures the Memory Management Unit (MMU) accordingly. The sets $W(s, p) \subseteq R(s, p)$ represent the sets of addresses that process p is allowed to write/read. Figure 1 shows an example with three user processes.

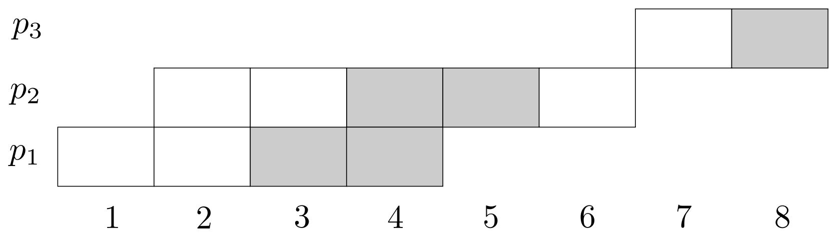

Fig. 1. Access permissions of eight memory pages for three processes. Gray boxes represent addresses that are in W and white boxes represent addresses that are only in R. Processes p_1 and p_2 can directly communicate using addresses in page 4, which can be written by both processes. Page 3 provides a unidirectional channel, since it cannot be modified by p_2. Page 2 is readable by both processes and can be used to store shared libraries. Page tables affect the process behaviour and should not be modifiable by unprivileged processes, therefore they cannot be in $\{3, 4, 5, 8\}$. Process p_3 is isolated and its communication with p_2 must be mediated by the kernel, which may decide to copy data from/to $\{7, 8\}$ or change access permissions.

We make some general assumptions that reflect the above intuition. First, processes cannot violate MMU settings, and they are unable to escalate their access privileges without mediation of the kernel:

Kernel Assumption KA 1 *If $s \xrightarrow{l} s'$ and $P(s) = p \neq 0$ then*

1. *$R(l) \subseteq R(s, p)$ and $W(l) \subseteq W(s, p)$*
2. *$\forall p'.\ R(s, p') = R(s', p')$ and $W(s, p') = W(s', p')$*
3. *$\forall a \in mem \setminus W(l)\ .\ s(a) = s'(a)$*

In other words: An unprivileged process p can read and write only addresses for which it has the necessary permissions (1.1). It cannot affect the read/write permissions of any process (1.2), and if p does not write to a given address, then the content of that address remains unchanged (1.3).

Secondly, the behaviour of the active process depends on the registers, the region of memory that can be accessed, and the content of the memory that is read:

Kernel Assumption KA 2 *If $P(s_1) = P(s_2) = p$, $R(s_1, p) = R(s_2, p)$, $W(s_1, p) = W(s_2, p)$, $s_1 \xrightarrow{l} s_1'$, and for all $a \in regs \cup R(l)$ then $s_1(a) = s_2(a)$, then there is some s_2' such that $s_2 \xrightarrow{l} s_2'$ and for all $a \in regs \cup W(l)\ .\ s_1'(a) = s_2'(a)$*

These properties are enforced by all secure kernels, cf. [1, 18, 25], and they do not restrict the kernel design. The kernel is free to change memory grants (i.e. to allocate, free, and change ownership of memory regions) and to copy data among processes. Processes can communicate via shared memory if the kernel allows it. Moreover, these properties do not constrain the presence of microarchitectural communication channels, e.g., due to the insecure usage of caches. Finally, these rules accommodate collaborative as well as preemptive multi-tasking and do not constrain information flows made available by the kernel scheduler.

To apply the epistemic framework introduced in Section 2 we need to also provide an observation model. Processes are able to observe the CPU registers when they are active. Moreover, in this example we use a so-called trace driven model [41]. An adversarial process is able to capture the state of its accessible memory (and cache if available) while another process is running. This model allows us to take into account scenarios where memory operations have effects on some other components, like caches or memory mapped devices, that could be controlled by an attacker. In these cases, the intermediary states of the memory that is accessible by the attacker provide a sound overapproximation of the information available to the attacker. Accordingly, the observations of process p in state s, $obs_p(s)$, contain:

1. The identity of the active process, $P(s)$.
2. p's memory rights, $R(s, p)$ and $W(s, p)$.
3. The content of the accessible memory,

$$\{(a, s(a)) \mid a \in R(s, p)\} \ .$$

4. The CPU register contents when p is active,

$$\text{if } (P(s) = p) \text{ then } \{s(r) \mid r \in \mathit{regs}\} \text{ else } \emptyset \ .$$

Notice that a process is able to observe resources that can affect its behaviour only indirectly. For example in Figure 1, process p_2 can observe that the first memory page is not in $W(s, p_2)$, since writing in this memory page raises a page fault and activates the kernel.

3.1 A Simple Key Manager

To demonstrate the model of knowledge and prepare the ground for later refinements we introduce an abstract model/specification of a simple key manager. The system executes the assumed kernel and two processes: p_1 is the key manager and p_2 is a potentially malicious client. Memory is statically partitioned like in Figure 1.

The key manager owns a secret key and provides a service that allows other processes to obtain the Message Authentication Code (MAC) of a piece of input data using the secret key. This MAC can be used, for instance, by a client to remotely authenticate the device. Process p_2 cannot directly access the key, which is stored in page 1, and input data and results are communicated via the shared page 4. This system has an intended information leakage: the MAC of the input data with the secret key. In terms of language-based security, the key-manager declassifies the result of this computation, and the goal of the later refinement step is to show that it adds no more information channels than what is already allowed by the abstract model.

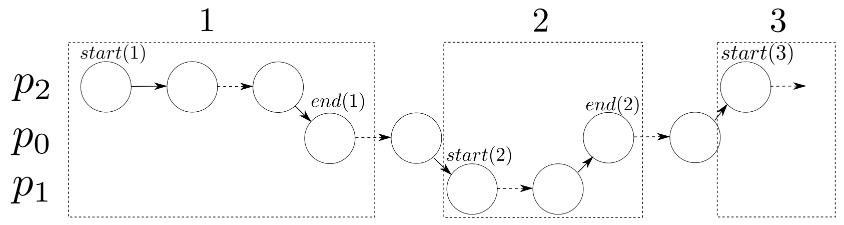

Fig. 2. Three phases and context switches of the example. Dashed arrows represent multiple transitions.

For the purpose of demonstrating our framework it is not important that the key manager uses cryptographically secure primitives. Therefore, a MAC algorithm based on m rounds of a naive Feistel cipher is used: data $d = d_0 d_1$ consists of two bytes, key $k = k_1 \dots k_m$ consists of m round keys, and the i-th round is computed as follows:

$$\begin{aligned} \mathrm{MAC}_{-1}(d,k) &= d_0 \\ \mathrm{MAC}_0(d,k) &= d_1 \\ \mathrm{MAC}_i(d,k) &= \mathrm{MAC}_{i-2}(d,k) \oplus T_i[k_i + \mathrm{MAC}_{i-1}(d,k)] \end{aligned}$$

where $+$ is addition modulo 256, and $\oplus$ is bit-wise xor. The MAC is computed using m tables T_i of 256 entries, which implements publicly known byte permutations.

We assume that process p_2 is active in the initial state s_0, and that the system progresses in three phases of Figure 2:

1. p_2 writes d in page 4, and requests a new service,
2. p_1 computes locally $\mathrm{MAC}_m(d,k)$ and writes the result in page 4,
3. p_2 uses the received MAC.

In between the three phases, the kernel simply context switches between the two processes, without affecting any resource that is observable by the two processes (i.e. it does not change memory permissions and does not modify pages $1 \dots 6$). For simplicity, we also assume that the context switch requires a constant amount of instructions. Phase i is started after $start(i)$ transitions and completed after $end(i)$ transitions. Let s_0 be the initial state:

– $start(1) = 0$, since we regard the system as starting only when boot is complete and the kernel hands over control to the client, p_2.
– $\mathcal{R}(s_0, end(1))$ is the run ending once p_2 has prepared the request and control has been passed to the kernel.
– $\mathcal{R}(s_0, start(2))$ is the run ending when the kernel has completed the context switch and passed control to p_1.

Our analysis focuses on the ignorance of the client p_2.

Initial ignorance. $I_{p_2}(\mathcal{R}(s_0, 0)) = I^{\text{init}}_{p_2}(\mathcal{R}(s_0, 0)) = \langle s_0 \rangle_{p_2}$, which is set of initial states where p_2 is active and pages $\{2 \ldots 6\}$ have the same memory content as s_0, independently of the content of pages $\{1, 7, 8\}$. Thus, initially p_2 has no information about k.

Phase 1 and 3. During the first $n \leq \mathit{end}(1)$ transitions, due to KA 1 and KA 2, for every $s_1 \in \langle s_0 \rangle_{p_2}$ let $\rho_1 = \mathcal{R}(s_1)$, it is the case that $\rho_0(n) \sim_{p_2} \rho_1(n)$, therefore $\mathcal{R}(s_0, n) \sim_{p_2} \mathcal{R}(s_1, n)$ and $I_{p_2}(\mathcal{R}(s_0, \mathit{end}(1))) = \mathcal{R}(I^{\text{init}}_{p_2}(\mathcal{R}(s_0, 0)), n)$. Moreover, $I^{\text{init}}_{p_2}(\mathcal{R}(s_0, \mathit{end}(1)) = I^{\text{init}}_{p_2}(\mathcal{R}(s_0, 0))$: as expected, the operating system prevents the client from gaining information without an explicit communication performed by p_1 or by the kernel itself. For the same reason, during the third phase, i.e., for $n \geq \mathit{start}(3)$, it is the case that $I_{p_2}(\mathcal{R}(s_0, n)) = \mathcal{R}(I^{\text{init}}_{p_2}(\mathcal{R}(s_0, \mathit{start}(3))), n)$.

Context switches. Since we assume that the kernel does not affect any resource that is observable by the process and that the context switch is done in “constant time”, then $I_{p_2}(\mathcal{R}(s_0, \mathit{start}(i+1))) = \mathcal{R}(I^{\text{init}}_{p_2}(\mathcal{R}(s_0, \mathit{end}(i))), \mathit{start}(i+1))$ for $i \in \{1, 2\}$.

Phase 2. The client p_2’s ignorance now decreases, since process p_2 learns the MAC. In fact, a run ρ_1 is in $I_{p_2}(\mathcal{R}(s_0, \mathit{end}(2)))$, if its initial state s_1 is in $I^{\text{init}}_{p_2}(\rho_0)$ (i.e. $s_1 \sim_{p_2} s_0$) and

$$\mathrm{MAC}_m(d, k) = \mathrm{MAC}_m(d, k')$$

where k' is the key in s_1. In terms of information-theoretic security, by learning the MAC, p_2 also learns a correlation between k and k' that reduces the entropy of k by 8 bits.

In the trace based model, the attacker controlling p_2 can observe its memory while process p_1 is executing. Therefore, if p_1 uses the shared page 4 to temporary store the value of $T_1[k_1 + d_1]$, instead of computing locally the first round, then the ignorance of p_2 is further reduced and the attacker directly learns the first byte of the key.

4 Refinement and Ignorance Preservation

We now compare two models, an abstract model $\mathcal{M}_a$ with states $s \in S$, runs $\mathcal{R}_a$, and transition relation $\rightarrow_a$, typically used to predict the desired behavior, and a concrete, or implementation model $\mathcal{M}_c$, with states $t \in T$, runs $\mathcal{R}_c$, and relation $\rightarrow_c$, that is used to describe how the abstract behavior is realized. We write ρ, ϕ for (sets of) abstract runs and σ, ψ for concrete ones. The two models are connected by a *refinement relation* $s \Downarrow t$, or function $\lceil t \rceil = s$, which for each concrete state t produces one or more abstract states s, which t is intended to refine. We refer to refinement relations of the latter form as *functional.* In this section we set out the basic properties we assume of refinements before we turn in Section 4.1 to information flow preservation.

A large section of work in the refinement domain is based on the notion of (forward) simulation, cf. [23, 45, 17], which in the present synchronous setting can be cast as follows.

Definition 1 (Simulation, Observation Preservation).

1. *The refinement relation $\Downarrow$ is a simulation of $\rightarrow_c$ by $\rightarrow_a$, if $s \Downarrow t$ and $t \rightarrow_c t'$ implies $s \rightarrow_a s' \Downarrow t'$ for some s'. Moreover, if $s \Downarrow t$, then s is initial or final if and only if t is.*
2. *The relation $\Downarrow$ preserves p-observations, if whenever $s \Downarrow t$ it holds that $Obs_p(t) = Obs_p(s)$.*

In the functional case the equivalent condition to 1.1 is that $t \rightarrow t'$ implies $\lceil t \rceil \rightarrow \lceil t' \rceil$. The simulation property 1.1 allows abstractions to be point-wise extended to runs by $\rho = s_0 \cdots s_n \Downarrow t_0 \cdots t_n = \sigma$ if $s_i \Downarrow t_i$ for all $i : 0 \leq i \leq |\rho| = |\sigma| = n+1$ and for sets ϕ, ψ, $\phi \Downarrow \psi$, if for all $\rho \in \phi$ there is $\sigma \in \psi$ such that $\rho \Downarrow \sigma$, and vice versa, for all $\sigma \in \psi$ there is $\rho \in \phi$ such that $\rho \Downarrow \sigma$. In the functional case Moreover, we denote by $\Uparrow$ the direct image of $\Downarrow^{-1}$, i.e., $\psi\Uparrow = \{\rho \mid \exists \sigma \in \psi.\ \rho \Downarrow \sigma\}$ and obtain:

Corollary 1. *If $\Downarrow$ is a simulation of the concrete model, then 1. $\phi \Downarrow \psi \Rightarrow \phi \subseteq \psi\Uparrow$ and 2. $(\psi\Uparrow) \Downarrow \psi$.* $\square$

For functional correctness, refinement usually requires both simulation and observation preservation. In this work we rely on the simulation condition as the crucial hook needed to relate computations at abstract and concrete level. Preservation of observations in the sense of 1.2 is used, e.g., in [4] but its necessity appears less clear. For ignorance preservation the key issue is preservation of observable distinctions and not necessarily the observations themselves. Indeed, as we show in this paper it is perfectly possible to conceive of meaningful refinement-like relations that preserve observation distinctions but not the observations themselves.

The key is to shift attention from preservation of observations to preservation of distinctions. In particular we distinguish observational equivalence relation $\sim_p$ on the abstract model from its counterpart (written $\approx_p$) on the concrete model. This motivates the following well-formedness condition:

Definition 2 (Well-formedness). *The refinement relation $\Downarrow$ is well-formed, if $s_1 \Downarrow t_1$ and $t_1 \approx_p t_2$ and $s_2 \Downarrow t_2$ implies $s_1 \sim_p s_2$.*

For functional refinement relations this becomes the condition that $t_1 \approx_p t_2$ implies $\lceil t_1 \rceil \sim_p \lceil t_2 \rceil$.

Well-formedness reflects the expectation that information content of models should generally increase under refinement. Then, if two abstract states are observationally distinct, we should expect this discriminating power to be preserved to concrete level. We obtain:

Proposition 2. *Suppose that the simulation $\Downarrow$ is well-formed. Then:*

1. *If $\rho_1 \Downarrow \sigma_1$ and $\sigma_1 \approx_p \sigma_2$ and $\rho_2 \Downarrow \sigma_2$ then $\rho_1 \sim_p \rho_2$.*

2. Suppose $\phi \Downarrow \psi$, then $I_p\phi \supseteq (I_p\psi)\Uparrow$.

Proof. 1. Follows immediately from Def. 2. 2. If $\rho \in (I_p\psi)\Uparrow$ then we find $\sigma \in I_p\psi$ such that $\rho \Downarrow \sigma$ and a $\sigma' \in \psi$ such that $\sigma \approx_p \sigma'$. By $\phi \Downarrow \psi$ there is $\rho' \in \phi$ with $\rho' \Downarrow \sigma'$ and by well-formedness $\rho \sim_p \rho'$. But then $\rho \in I_p\phi$.

In other words it follows directly from well-formedness and the simulation property that ignorance is preserved from concrete to abstract level. We define:

Definition 3 (Refinement). *The refinement relation $\Downarrow$ is a* refinement*, if $\Downarrow$ is well-formed and a simulation.*

4.1 Ignorance Preservation

One key idea for confidentiality preservation, proposed originally by Morgan [38], is to compare ignorance at abstract level with ignorance at concrete level: If the ignorance at concrete level is "at least as high as" (in [38]: a superset of) the ignorance at abstract level, no more information is learned by executing the protocol at concrete level than what is learned by executing the ideal functionality. While Prop. 2.2 is useful, our interest, however, is in preservation of ignorance in the opposite direction.

However, ignorance at abstract and concrete levels is not readily comparable, as in our setting (as opposed to [38]) the state spaces related by the refinement are different. In general, refinement will reduce nondeterminism and add observational power by implementation choices, e.g., for data representation. Nevertheless, reflecting our view of the abstract model as specifying the desired information flow properties, all information relevant for the analysis of information flow preservation is available already at abstract level. Thus we can use the refinement relation to push epistemic properties between the abstract and concrete levels, as follows:

Definition 4 (Ignorance-Preserving Refinement, IPR). *The refinement $\Downarrow$ is p-ignorance-preserving, if $\Downarrow$ is a well-formed simulation such that $\phi \Downarrow \psi$ implies $I_p\phi \Downarrow I_p\psi$.*

We relativize ignorance preservation to the processes p since this allows to use different abstraction functions for each p, reflecting the potentially different views each process may have of the refinement. It becomes clear that Def. 4 is the desired property if we consider an equivalent formulation:

Proposition 3. *The refinement $\Downarrow$ is p-ignorance-preserving, iff $\phi \Downarrow \psi$ implies $I_p\phi = (I_p\psi)\Uparrow$.*

Proof. By Cor. 1.1, $I_p\phi \Downarrow I_p\psi$ implies $I_p\phi \subseteq (I_p\psi)\Uparrow$ and by well-formedness (Prop. 2.2) $I_p\phi = (I_p\psi)\Uparrow$. The other direction follows directly via $((I_p\psi)\Uparrow) \Downarrow I_p\psi$ by Cor. 1.2.

Thus, IPR means that we have the same ignorance for observer p on both levels, when viewed in terms of the abstract model. In particular, a concrete model observer p cannot distinguish more behaviors than possible on the abstract model, when "re-abstracting" the set of indistinguishable concrete runs. The following is a useful sufficient and necessary condition for ignorance-preserving refinement:

Definition 5 (Paired Refinement). *The refinement $\Downarrow$ is paired, if for all ρ, ρ', σ':*

$$\textit{If } \rho \sim_p \rho' \Downarrow \sigma' \textit{ then there exists } \sigma \textit{ s.t. } \rho \Downarrow \sigma \approx_p \sigma'. \qquad (*)$$

Proposition 4. *The paired refinement condition $(*)$ holds for refinement $\Downarrow$ if, and only if, $\Downarrow$ is p-ignorance-preserving.*

Proof. The implication IPR $\Rightarrow (*)$ follows directly from $I_p\phi \Downarrow I_p\psi$ for $\phi = \{\rho'\}$ and $\psi = \{\sigma'\}$. For direction $(*) \Rightarrow$ IPR, assume that $\phi \Downarrow \psi$ for the refinement $\Downarrow$. For any $\rho \in I_p\phi$ we find $\rho' \in \phi$ such that $\rho \sim_p \rho'$ and a $\sigma' \in \psi$ such that $\rho' \Downarrow \sigma'$. By $(*)$ we find σ s.t. $\rho \Downarrow \sigma$ and $\sigma \approx_p \sigma'$, i.e. $\sigma \in I_p\psi$. Conversely, if $\sigma \in I_p\psi$ then we find $\sigma' \in \psi$ such that $\sigma \approx_p \sigma'$ and then a $\rho' \in \phi$ such that $\rho' \Downarrow \sigma'$. By the simulation property we find ρ such that $\rho \Downarrow \sigma$ and then by well-formedness, $\rho \sim_p \rho'$, i.e. $\rho \in I_p\phi$, as desired.

Intuitively, $(*)$ requires that for each pair of indistinguishable abstract runs, if one of them is implemented, so is the other one and the corresponding concrete runs are indistinguishable as well.

Assume models $\mathcal{M}_i$, $0 \leq i \leq 2$, and assume we have ignorance-preserving refinements $\Downarrow_i$, $i \in \{1, 2\}$ from $\mathcal{M}_{i-1}$ to $\mathcal{M}_i$. Then the relational composition $\Downarrow_1 \circ \Downarrow_2$ from $\mathcal{M}_0$ to $\mathcal{M}_2$ should be ignorance-preserving, too. The simulation property and well-formedness properties are easily checked, it remains to show that a vertically composed refinement is an IPR if its component refinements are:

Proposition 5. *If the refinements $\Downarrow_1$ and $\Downarrow_2$ are ignorance-preserving then so is $\Downarrow = \Downarrow_1 \circ \Downarrow_2$.*

Proof. Let $\phi \Downarrow_1 \psi \Downarrow_2 \xi$. Using $(*)$, assume $\rho_0 \sim \rho_1 \Downarrow \tau_1$. We find σ_1 such that $\rho_1 \Downarrow_1 \sigma_1 \Downarrow_2 \tau_1$. By $(*)$ there is σ_0 with $\rho_0 \Downarrow_1 \sigma_0 \approx \sigma_1$. Applying $(*)$ again for $\sigma_0 \sim \sigma_1 \Downarrow_2 \tau_1$, we obtain τ_0 with $\rho_0 \Downarrow \tau_0 \approx \tau_1$ and conclude via Prop. 4.

Vertical composability enables a common verification strategy to deal with perfect recall attackers in epistemic settings: extend the abstract state with an observable history variable that can be computed from existing observations; prove that the abstraction that disregards the history variable is a CPR, and finally analyse a refined model w.r.t. the extended model.

5 Case Study: Adding a History Variable

In the model processes can observe the active process $P(s)$. Therefore we can add an observable variable H that keeps track of the number of transitions performed by each process (this variable simplifies the formalisation of constant time execution in Section 8). Let $s \xrightarrow{l} s'$, then $u = (s, H) \xrightarrow{l} (s', H') = u'$ and

$$H'(p) = H(p) + \begin{cases} 1 & \text{if } P(s) = p \\ 0 & \text{otherwise} \end{cases}$$

For this extended model, the simulation simply disregards the history variable.

Lemma 1. *For every process p, $s \Downarrow (s, H)$ is a IPR for the model of Section 4.1.*

Proof: The extended model is simulated by the abstract model by construction, similarly well-formedness trivially holds. Therefore it suffices to demonstrate Eq. $*$. Let $t = (s, H)$, $\sigma \in \mathcal{R}(t)$, $\rho \in \mathcal{R}(s)$ such that for every $\sigma(n) = (\rho(n), H_n)$ for some n, $\rho' \in \mathcal{R}(s')$, $\rho' \sim_p \rho$, and $t' = (s', H)$. By construction exist $\sigma' \in \mathcal{R}(t')$ such that for every n exists H'_n such that $\sigma(n) = (\rho(n), H_n)$. Since $\rho(n) \sim_p \rho'(n)$, and since $P(\_)$ is observable we can conclude that $H'_n = H_n$. Therefore $\sigma' \approx_p \sigma$. ■

6 Case Study: Cache Aware Model

To demonstrate IPR we first introduce a refined version of Section 3's processor model. In this model the processes are executed on data-cache enabled hardware and are allowed to measure the time needed to execute their own instructions. The refined state has the form $t = (s, H, c, \tau_0, \ldots, \tau_n)$ where H is the history variable introduced in above, c is a shared cache and τ_i is the clock for the process p_i. In this model, the processes do not have a shared clock, which is reasonable for systems that offer only virtualized time to processes.

To be general we use an abstract model for caches. The cache has $\mathbb{S}$ entries (sets), and c is a total function from $\{0, \ldots, \mathbb{S} - 1\}$ to cache entries. A cache entry $e = (h, d)$ is a pair, where h contains metadata (e.g. validity, tag, state of replacing policy, dirtiness flags, etc.) and d contains the data of the entry. In case of a direct mapped cache this is the complete data stored in the cache line, in multi-way caches the data stored across all the ways. We use the following notation: $idx(a)$ identifies the cache entry corresponding to the address a, $hit(h, a)$ holds if the address a is stored in the entry e, and $get(e, a)$ extracts the content of the address from the entry.

To simplify the notation we introduce the following operators to filter lists of operations accessing the same cache entry i:

$$\varepsilon|_i = \varepsilon \quad \text{and} \quad ((op, a) \circ l)|_i = (op, a) \circ l|_i \text{ if } idx(a) = i \text{ else } l|_i\,,$$

where $\circ$ is the list constructor. To extract metadata of cache entries that collide for a given set of addresses A, we have:

$$c|_A = \{(idx(a), c(idx(a)).h) \mid a \in A\}$$

$\lceil(s, H, c, \tau_0, \ldots, \tau_n)\rceil = (s', H)$ defines the abstraction map for the cache-enabled model, where

$$s'(a) = \begin{cases} get(c(idx(a)), a), & \text{if } hit(c(idx(a)).h, a) \\ s(a), & \text{otherwise} \end{cases}$$

The behaviour of the cache is governed by four model assumptions, similar in spirit to those of Section 3.

First, we assume that the kernel abstractions P, R, and W are overloaded for the refined model and invariant w.r.t. the abstraction function:

Kernel Assumption KA 3 *$P(t) = P(\lceil t\rceil)$ and for every process p, $R(t, p) = R(\lceil t\rceil, p)$ and $W(t, p) = W(\lceil t\rceil, p)$.*

Secondly, the cache is transparent:

Cache Assumption CA 1 *If $t \xrightarrow{l} t'$ then $\lceil t\rceil \xrightarrow{l} \lceil t'\rceil$*

In other words: A transition enabled at refined level remains enabled at abstract level once any state information added in the refinement is abstracted away. Note, that this implies that the simulation condition (Def.1) holds in our model. System software must use special precautions to ensure CA 1 and KA 3, for example by flushing caches and Translation Lookaside Buffer (TLB) when page tables are updated.

Moreover, the metadata of a cache entry does not depend on cache data or accesses to addresses belonging to other entries:

Cache Assumption CA 2 *Let $t_1 \xrightarrow{l_1} t_1'$ and $t_2 \xrightarrow{l_2} t_2'$. For every entry index $i < S$ such that $t_1.c(i).h = t_2.c(i).h$, if $l_1|_i = l_2|_i$ then $t_1'.c(i).h = t_2'.c(i).h$.*

CA 2 expresses that for any two transitions in the cache-aware model, if:

– the metadata associated with a cache entry i is the same in the two prestates, and
– the two transitions read and write the same addresses,

then the cache metadata associated with entry i is the same in the two poststates.

Assumptions CA 1 and CA 2 are general enough to grant a wide scope to our analysis. They accommodate write-through as well as write-back caches, both direct and multi-way associative caches, several types of replacement policies, and do not require inertia (i.e. they allow the eviction of lines even in absence of cache misses for the corresponding entry).

Finally, for the processes' virtualized clocks we make the following assumptions:

Time Assumption TA 1 *If $t_1 \xrightarrow{l} t_1'$ and $p = P(t_1)$ then*

1. *For all $p' \neq p$, $t_1.\tau_{p'} = t_1'.\tau_{p'}$.*
2. *If $P(t_2) = p$, $t_2 \xrightarrow{l} t_2'$, $t_1.\tau_p = t_2.\tau_p$, then for all $a \in regs \cup R(l)$. $\lceil t_1\rceil(a) = \lceil t_2\rceil(a)$, and $t_1.c|_{R(l)\cup W(l)} = t_2.c|_{R(l)\cup W(l)}$, then $t_1'.\tau_p = t_2'.\tau_p$.*

The upshot of TA 1 is that: Only the clock of the active process is incremented (TA 1.1); The execution time of an instruction depends only on the register state, the accessed memory contents, and cache metadata of accessed cache entries (TA 1.2).

For guaranteeing CA 2 and TA 1.2 the cache must provide some sort of *isolation among cache entries*. This is usually the case when the replacement policy does not have state information that is shared among cache entries. An example that violates the requirements is prefetching of adjacent entries in case of cache misses. In this case, the metadata of a cache entry is dependent on the accesses performed in the adjacent entries. Also, for the same type of cache, the prefetching of adjacent cache entries can slow down the execution of a memory access.

In the refined model, a process p can further observe the corresponding private clock and the resources that can indirectly affect it, namely the metadata of the cache elements that can be accessed using the readable memory. Formally, $obs_p(t)$ now contains:

1. The identity of the active process, $P(t)$,
2. p's memory rights, $R(t,p), W(t,p)$
3. The content of accessible memory: $(a, \lceil t \rceil(a)) \mid a \in R(t,p)\}$
4. The CPU registers when p is active: if $(P(s) = p) \wedge a \in regs$ then $s(a)$ else $\perp$.
5. p's local clock, $t.\tau_p$
6. The cache state as seen by p, $t.c|_{R(t,p)}$

Lemma 2. *For the models of Sections 3 and 6, the function $\lceil \cdot \rceil$ is a well-formed refinement.*

Proof: The first part of Definition 1 is obvious because cache and time are transparent on the abstract model and other observations are identity-mapped. For Definition 2, for every $s \sim_p \lceil t_0 \rceil$ let t be a state with the same registers and memory of s, the same timers as t_0, $t.c(i).h = t_0.c(i).h$ for all i, and $t.c(i).d$ such that $get(c(idx(a)), a) = s(a)$ if $hit(t_0.c(idx(a)), a)$. Then t satisfies $s = \lceil t \rceil$ and $t \sim_p t_0$. ■

From KA 3 it follows directly that two states are observation-equivalent if their corresponding abstractions are observation-equivalent, the process clocks are the same, and the metadata of the accessible cache entries are equivalent:

Lemma 3. *If $\lceil t_1 \rceil \sim_p \lceil t_2 \rceil$, $t_1.\tau_p = t_2.\tau_p$, and also $t_1.c|_{R(t_1,p)} = t_2.c|_{R(t_2,p)}$, then $t_1 \approx_p t_2$ holds.*

6.1 Timing Channels in the Refined Model

Caches and timing information pose threats to the key manager. For simplicity, we assume that variables of p_1 and every entry of tables T_i are allocated on different cache entries. Moreover, we assume that in the initial state $\lceil t_0 \rceil = s_0$, clocks are zero, the cache is initially empty, and that process p_2 knows the memory layout of the key manager.

CA 1 guarantees that in the abstract and refined models process p_2 prepares the same inputs and process p_1 provides the same replies. Therefore p_2's knowledge obtained by observing registers, memory, and active process is the same in the two models. However, in the refined model cache and timing effects must be taken into account, as these are not reflected at the abstract level. We assume that p_2 "primes" the cache every time it is executed, filling all entries with data belonging to page 6. For simplicity, we assume that victim p_1 accesses only the addresses needed to implement the key manager, and that the kernel always accesses the same sequence of addresses during context switches.

It is easy to show that IPR holds for the first $start(2)$ transitions. Let ρ_n and σ_n be the runs $\mathcal{R}(s_0, n)$ and $\mathcal{R}(t_0, n)$ respectively.

Initial knowledge. For every run (consisting only of the initial state) $\rho' \in I_{p_2}(\rho_0)$ there is exactly one corresponding run $\sigma' \in I_{p_2}(\sigma_0)$ where the initial state has empty cache, zero clocks, and such that $\lceil \sigma' \rceil = \rho'$. Therefore, $\lceil I_{p_2}(\sigma_0) \rceil = I_{p_2}(\lceil \sigma_0 \rceil)$, as desired.

Phase 1. For the first $n \leq end(1)$ transitions, the attacker clock and the cache metadata depend on the initial cache state, which is empty in the initial state of every $\sigma'_0 \in I_{p_2}(\sigma_0)$. Therefore for every $\sigma'_0 \in I_{p_2}(\sigma_0)$ there is $\sigma'_n \in I_{p_2}(\sigma_n)$ such that $\sigma'_n(0) = \sigma'_0(0)$. This means that $I_{p_2}(\sigma_n) = \mathcal{R}(I^{\text{init}}_{p_2}(\sigma_0), n)$, hence $\lceil I_{p_2}(\sigma_n) \rceil = I_{p_2}(\lceil \sigma_n \rceil)$.

Context Switches. For $end(i) < n \leq start(i+1)$ after phases $i \in \{1, 2\}$, the kernel accesses the same sequence of addresses and it cannot modify the process clock. Therefore $I_{p_2}(t_0, start(i+1)) = \mathcal{R}(I^{\text{init}}_{p_2}(\mathcal{R}(s_0, end(i))), start(i+1))$ and $\lceil I_{p_2}(\sigma_{start(i+1)}) \rceil = I_{p_2}(\lceil \sigma_{start(i+1)} \rceil)$ if $\lceil I_{p_2}(\sigma_{end(i)}) \rceil = I_{p_2}(\lceil \sigma_{end(i)} \rceil)$.

However, Phase 2 is more challenging. Starting from the last state of $\sigma_{start(2)}$ the key manager accesses the input data, the key, and $T_i[k_i + \text{MAC}_{i-1}(d, k)]$ for $i \in \{1 \ldots m\}$. Therefore, after $end(2)$ transitions, the cache entries that are evicted are $idx(T_i + k_i + \text{MAC}_{i-1}(d, k))$, where $T_i + k_i + j$ is the address of the $(k_i + j)$'th element of T_i. On the other hand, if the system had started from the initial state $t'_0 \in I^{\text{init}}_{p_2}(t_0, start(2))$ with key k', then it would have evicted the entries $idx(T_i + k'_i + \text{MAC}_{i-1}(d, k'))$.

In other words, for the last state of $\sigma_{end(2)}$ a dependency of the cache metadata $c(j).h$ on indices $j = idx(T_i + k_i + \text{MAC}_{i-1}(d, k))$ is being introduced, causing $\lceil I_{p_2}(\sigma_{end(2)}) \rceil$ to become (in general) a strict subset of $I_{p_2}(\rho_{end(2)})$, i.e., p_2 in the refined model learns more than in the abstract model.

In practice, this side-channel enables p_2 to discover the key. In fact, for the first round, we cannot guarantee that the cache metadata is the same when starting from an initial state where $idx(T_1 + k'_1 + d_1)$ differs from $idx(T_1 + k_1 + d_1)$. This can result in different clocks for p_2 after the next context switch (i.e. $n > start(3)$). The same reasoning can be done for the other rounds.

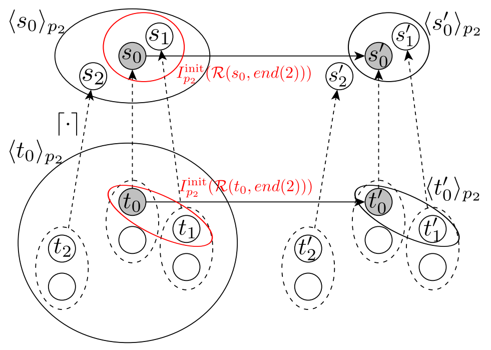
$\langle s_0\rangle_{p_2}$
s_0
s_1
s_2
$\langle s'_0\rangle_{p_2}$
s'_0
s'_1
s'_2
$I^{\text{init}}_{p_2}(\mathcal{R}(s_0, end(2)))$
$\lceil\cdot\rceil$
$\langle t_0\rangle_{p_2}$
t_0
t_1
t_2
$I^{\text{init}}_{p_2}(\mathcal{R}(t_0, end(2)))$
$\langle t'_0\rangle_{p_2}$
t'_0
t'_1
t'_2

Fig. 3. Ignorance-Preserving Refinement for p_2, which can distinguish by design the key in $s_2 = \lceil t_2\rceil$. Refined states t_0, t_1 have different key and random seeds. Still, $\lceil I^{\text{init}}_{p_2}(\mathcal{R}(t_0, end(2)))\rceil = I^{\text{init}}_{p_2}(\mathcal{R}(s_0, end(2))) = I^{\text{init}}_{p_2}(\mathcal{R}(\lceil t_0\rceil, end(2)))$.

6.2 A Naive Countermeasure for the Key Manager

In this section we show the IPR condition in action by verifying a naive countermeasure, which relies on randomization and the fact that the key manager performs only a single access to each permutation box. We assume that p_2 knows the memory layout of p_1 with the exception of the entries of the tables. Each table T_i has been permuted using a random byte r_i by moving the j'th entry to position $j \oplus r_i$. We assume that p_2 does not know r_i. For this refinement we have the same abstraction map, with exception of the entries of T_i which are mapped as follows: $\lceil t\rceil(T_i + j) = t(T_i + (j \oplus r_i))$. Therefore, in the refined model the MAC is computed as $\text{MAC}_i(d, k) = \text{MAC}_{i-2}(d, k) \oplus T_i[(k_i + \text{MAC}_{i-1}(d, k)) \oplus r_i]$.

We use Figure 3 to illustrate the scenario. Each state of the refined model is mapped to a unique abstract state via the abstraction $\lceil\cdot\rceil$. The vice-versa is not true: there are at least 256^m refined states (for different values of $r_1 \dots r_m$) that are mapped to the same abstract state. This arises from the fact that refined states have more information than abstract ones.

The function $\lceil\cdot\rceil$ induces an equivalence relation: Two refined states are *abstraction-equivalent* if their abstraction is the same. In our example, states that are abstraction-equivalent must have the same key, but they can have different randomization of the permutation boxes. The equivalence class $\langle t_0\rangle_{p_2}$ can be partitioned into abstraction-equivalent classes (shown as dashed ellipses in the figure), each corresponding to an abstract state of $\langle s_0\rangle_{p_2}$.

To demonstrate IPR for process p_2 we focus on Phase 2, which is the one that violates IPR due to the cache side channel. Let t_0 be a given state with a fixed key k and σ_n a run of n transitions starting from t_0. The argument is essentially relational. Assume a run $\rho'_n \sim \lceil\sigma_n\rceil$. By (5) we have to find a refined state t' such that $\lceil t'\rceil = \rho'_n(0)$ and $\sigma'_n \sim_{p_2} \sigma_n$ for $\sigma'_n \in \mathcal{R}(t', n)$, i.e. such that $\sigma_n(n') \sim_{p_2} \sigma'_n(n')$ for all $n' \leq n$. By the argument of Section 6 we must show that we can find t' such that

$$\mathcal{R}(t', end(2)) \sim_{p_2} \mathcal{R}(t_0, end(2)) \ . \tag{1}$$

Once this is established, agreement on the local clocks and cache states allows (1) to extend to arbitrary $n > end(2)$.

Assuming that, except for the table lookup, the MAC is computed using registers only, finding t' is tantamount to identifying a state that satisfies, for every round i,

$$r'_i \oplus (k'_i + \text{MAC}_{i-1}(d, k')) = r_i \oplus (k_i + \text{MAC}_{i-1}(d, k))$$

where r_i and r'_i are the random seeds of t_0 and t', respectively, d is the data in both states, and k' is the key in t'.

In fact, starting from any such t_0 and t' the key manager accesses exactly the same memory locations (even if they have different keys) and therefore the same cache lines have been evicted in $\mathcal{R}(t_0, end(2))$ and $\mathcal{R}(t', end(2))$. This, and the fact that t_0 and t_1 produce the same MAC due to $\rho'_n \approx_{p_2} \lceil \sigma_n \rceil$, guarantees that $\mathcal{R}(t_0, end(2))$ and $\mathcal{R}(t', end(2))$ are indistinguishable by p_2. This completes the argument that the naive countermeasure is sufficient for the key manager to satisfy IPR.

States that are abstraction-equivalent to t_0 or t' are dropped from the initial ignorance set $I_{p_2}(\sigma_0)$ during the computation, if they have a different random seed than t_0 or t', i.e., even though they agree on the resulting MAC they are not in $I^{\text{init}}_{p_2}(\sigma_{end(2)})$ because the refined model reveals the different seed. However, this does not violate IPR.

Unfortunately, the above countermeasure is not compositional: i.e., cannot guarantee IPR if the key manager is invoked multiple times. Let the attacker provide the data d in the first step, and in the third step request a second MAC for data d'. Each step of the key manager satisfies IPR when executed in isolation. However, their composition fails to satisfy IPR if the permutation boxes are not re-randomized. In fact, in the abstract model the attacker learns the value of $\text{MAC}_m(k, d)$ and $\text{MAC}_m(k, d')$. However, in the refined model, the attacker can learn more information regarding the key used for the first round. The first execution of the key manager allows p_2 to additionally learn $r_1 \oplus (k_1 + d_1)$, while the second execution allows the same process to learn $r_1 \oplus (k_1 + d'_1)$. Therefore, the combination of the two executions enables p_2 to discover the value of $(k_1 + d_1) \oplus (k_1 + d'_1)$ which leaks additional bits of k_1 depending on the values of d_1 and d'_1.

7 Relational Verification

In order to handle an example of the size and complexity of our key manager a way to sequentially compose refinements is very useful. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ differ only in that the initial states can be different in the two models, and that final states of $\mathcal{M}_1$ are initial states in $\mathcal{M}_2$. That is:

1. The state spaces of $\mathcal{M}_1$ and $\mathcal{M}_2$ are identical.
2. If $s_1 \to s_2$ in $\mathcal{M}_1$ (i.e., $s_1 \to_1 s_2$) then $s_1 \to_2 s_2$.
3. If $s_1 \to_2 s_2$ and $s_1 \to_1 s_2'$ (i.e. s_1 is not final in $\mathcal{M}_1$) then $s_1 \to_1 s_2$.
4. Final states in $\mathcal{M}_1$ are initial states in $\mathcal{M}_2$.
5. The observations in $\mathcal{M}_1$ and $\mathcal{M}_2$ agree, i.e., $Obs_1(s) = Obs_2(s)$ for all state s in $\mathcal{M}_1$ and $\mathcal{M}_2$.

These properties allow to compose the two models sequentially while preserving observability properties. Note in particular that we allow states, not only initial/final ones, to be present in both $\mathcal{M}_1$ and $\mathcal{M}_2$, which is meaningful for unstructured programs. Also, the definition allows models to be sequentially split and recomposed in a very flexible fashion, by simply stopping execution of $\mathcal{M}_1$ at whichever state is convenient for the analysis. In particular we can define the sequential composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ as the model $\mathcal{M}_1;\mathcal{M}_2$ with the initial states of $\mathcal{M}_1$, states and observations of $\mathcal{M}_1$ (or $\mathcal{M}_2$), and transitions of $\mathcal{M}_2$. We obtain:

Proposition 6.

$$\begin{aligned}\mathcal{R}(\mathcal{M}_1;\mathcal{M}_2) &= \mathcal{R}(\mathcal{M}_1);\mathcal{R}(\mathcal{M}_2)\\ &= \{\rho_1;\rho_2 \mid \rho_1 \in \mathcal{R}(\mathcal{M}_1), \rho_2 \in \mathcal{R}(\mathcal{M}_2), lst(\rho_1) = fst(\rho_2)\}\end{aligned}$$

where $(\rho_1 s);(s\rho_2) = \rho_1 s \rho_2$ is the sequential composition of runs.

Under information flow constraints the sequential composition of refinements is generally highly delicate as shown in [43]. Here, we follow the approach of [11] based on ideas from relational Hoare logic [12].

Definition 6 (Relational Refinement). *Let symmetric relations $R_{pre}, R_{post} \subseteq T \times T$ be given such that $R_{pre} \subseteq \approx_p$. The triple $\{R_{pre}\} \Downarrow \{R_{post}\}$ is a* relational refinement, *if $\Downarrow$ is a well-formed refinement from $\mathcal{M}_a$ to $\mathcal{M}_c$ such that:*

1. *If $s_1 \Downarrow t_1$ are initial states, then given any $s_1 \sim_p s_2$, we can find a t_2 such that $s_2 \Downarrow t_2$ and t_1 R_{pre} t_2.*
2. *If $fst(\sigma_1)$ R_{pre} t_2, $\rho_1 \Downarrow \sigma_1$, $\rho_1 \sim_p \rho_2$, $fst(\rho_2) \Downarrow t_2$, then a run σ_2 exists with $fst(\sigma_2) = t_2$, $\rho_2 \Downarrow \sigma_2$, $\sigma_1 \approx_p \sigma_2$, and if σ_1 is complete, so is σ_2 and $lst(\sigma_1)$ R_{post} $lst(\sigma_2)$.*

The triple $\{R_{pre}\} \Downarrow \{R_{post}\}$ expresses that whenever there is a complete run from a concrete state t_1, which is a refinement of an abstract state indistinguishable from s_2, then it is possible to find a complete run from some other concrete state t_2, which is a refinement of s_2, and such that the two runs are indistinguishable, R_{pre} holds on the initial states of the runs, and R_{post} on the final states of the two runs.

By conditioning R_{post} on whether σ_1 and σ_2 are complete, the definition covers both terminating and diverging programs. Clearly, the definition ensures that the IPR condition holds.

Corollary 2. *Any relational refinement is an IPR.* □

Also, we can show that relational refinements provide sequential compositionality. To this end let $\Downarrow$ be a relational refinements from both $\mathcal{M}_{a,1}$ to $\mathcal{M}_{c,1}$ and $\mathcal{M}_{a,2}$ to $\mathcal{M}_{c,2}$ (even if $\Downarrow_1$ and $\Downarrow_2$ are identical as relations they may not both be relational refinements). For clarity we use $\Downarrow_i$ when we want to refer to $\Downarrow$ as a relational refinement from $\mathcal{M}_{a,i}$ to $\mathcal{M}_{c,i}$, and $\Downarrow_1; \Downarrow_2$ when we want to refer to $\Downarrow$ as a relational refinement on $\mathcal{M}_1; \mathcal{M}_2$. Sequential compositionality now follows from the definitions in a straightforward fashion.

Theorem 1 (Sequential Compositionality [11]). *Suppose* $\{R_{pre}\} \Downarrow_1 \{R\}$ *and* $\{R\} \Downarrow_2 \{R_{post}\}$ *are relational refinements. Then* $\{R_{pre}\} \Downarrow_1; \Downarrow_2 \{R_{post}\}$ *is a relational refinement.* □

It follows by Cor. 2 that if $\{R_{pre}\} \Downarrow_1 \{R\}$ and $\{R\} \Downarrow_2 \{R_{post}\}$ are relational refinements then the refinement $\Downarrow_1; \Downarrow_2$ is ignorance preserving.

8 Case Study: Verification of Constant Time Execution and Cache Coloring

In this section we verify security of two widely adopted countermeasures against trace driven cache-based side channels: Constant time execution and cache coloring.

Hereafter we only consider models that have been extended with the history variable H. The following definitions provide a formalization of cache coloring and constant time execution. For each process p we use E_p to identify the indexes of cache entries that process p is allowed to access. The kernel must restrict the process-accessible memory to ensure this property:

Countermeasure C 1 *If* $s \xrightarrow{l} s'$ *and* $P(s) = 0$ *then for all* p*:*

$$idx(R(s,p)) = idx(R(s',p)) = E_p \ .$$

Since processes cannot directly change their access permissions (KA 1), C 1 allows the set of cache indices from the point of view of p to be partitioned into two sets: private entries ($E_p^P = E_p \setminus \cup_{p' \neq p} E_{p'}$) and shared entries ($E_p^S = E_p \cap \cup_{p' \neq p} E_{p'}$). A trusted process p' can perform unrestricted accesses to $E_{p'}^P$ while accesses to $E_{p'}^S$ must satisfy constant time execution. The latter is formalized by the following property, which requires that memory accesses that involve a cache entry accessible by process p depend only on information that is available to p:

Countermeasure C 2 *A system is constant time w.r.t. process p if for every* $s_1 \sim_p s_2$*, if* $s_1 \xrightarrow{l_1} s_1'$*, and* $s_2 \xrightarrow{l_2} s_2'$*, then* $l_1|_{E_p} = l_2|_{E_p}$.

Both C 1 and C 2 can be verified using the abstract model, since they only constrain the list of accessed addresses and addressable indices, and they do not require to explicitly analyse the cache state. C 1 is a kernel invariant to be verified by standard techniques of program analysis. A number of tools exist to check C 2, including relational analysis [7] for binary programs, and abstract interpretation [13] for source code in conjunction with secure compilation [9].

Finally, we can demonstrate that constant time execution and cache coloring (or a mix of the two) prevent side-channels:

Theorem 2. *For a process p, C 1 and C 2 guarantee IPR.*

Proof: In order to prove IPR we show that the two properties ensure a relational refinement for each transition. This allows us to analyze each transition independently and compose the refinements to obtain properties of complete executions. In other words, we look at transition systems that have traces of only one transition per trace and we compose horizontally to obtain the entire transition system. Here the refinement pre- and post-relations are the same: $\approx_p$. Therefore condition (1) holds by definition and (2) holds by well-formedness of $\lceil \_ \rceil$.

By simulation (CA 1), if $t_1 \approx_p t_2$, $\sigma_1 = t_1 \xrightarrow{l_1} t_1'$ then $\rho_1 = \lceil t_1 \rceil \xrightarrow{l_1} \lceil t_1' \rceil$. Since transition systems here are left-total, then exists $\sigma_2 = t_2 \xrightarrow{l_2} t_2'$. Let $\rho_1 \sim_p \rho_2 = \lceil t_2 \rceil \xrightarrow{l_2} s_2'$, by simulation (CA 1) and determinism of the transition system we have that $s_2' = \lceil t_2' \rceil$.

Therefore we must prove that $t_1' \approx_p t_2'$. Hypothesis $t_1 \sim_p t_2$ ensures that $P(t_1) = P(t_2) = p'$. Lemma 3 guarantees that $\approx_p$ is established if $t_1'.\tau_p = t_2'.\tau_p$, and $t_1'.c|_{R(t_1',p)} = t_2'.c|_{R(t_2',p)}$. These equalities are demonstrated for two cases: when p is the active process ($p = p'$) and when it is suspended ($p \neq p'$).

Case $p = p'$ As access permissions are not affected by the cache (KA 3) and cannot be directly changed by the process (KA 1.2) we get for $i \in \{1, 2\}$:

$$R(t_i', p) = R(\lceil t_i' \rceil, p) = R(\lceil t_i \rceil, p) = R(t_i, p) \tag{2}$$

Since the process can only access its own memory (KA 1.1), which is observable, and the same observable access permissions are in place, it performs the same instruction with the same effects on the abstract level (KA 2.1), hence:

$$R(l_1) \cup W(l_1) \subseteq R(\lceil t_1 \rceil, p) = R(t_1, p) \text{ and } l_2 = l_1 \tag{3}$$

Therefore, for cache metadata we have

$$\begin{aligned}
& t_1 \sim_p t_2 & \\
t_1.c|_{R(t_1,p)} &= t_2.c|_{R(t_2,p)} & \text{Def. } \sim_p \\
t_1'.c|_{R(t_1,p)} &= t_2'.c|_{R(t_2,p)} & \text{by (3) and CA 2} \\
t_1'.c|_{R(t_1',p)} &= t_2'.c|_{R(t_2',p)} & \text{by (2)}
\end{aligned}$$

Similarly, for the process clock we have:

$$\begin{array}{rcl} & t_1 \sim_p t_2 & \\ & \forall a \in R(l_1).\lceil t_1 \rceil(a) = \lceil t_2 \rceil(a) & \text{Def. } \sim_p \\ & \text{and } t_1.c|_{R(l_1)\cup W(l_1)} = t_2.c|_{R(l_1)\cup W(l_1)} & \text{and (3)} \\ & t_1'.\tau_p = t_2'.\tau_p & \text{by TA 1.2} \end{array}$$

Case $p \neq p'$ For the non-active process, the equality of τ_p follows directly from TA 1.1. By $t_1 \sim_p t_2$ we get:

$$R(t_1, p) = R(t_2, p) \tag{4}$$

Since standard processes ($p' \neq 0$) cannot change access permissions (KA 1.2) and kernel ($p' = 0$) constraints the observable cache entries of p to E_p (C 1), then

$$\begin{aligned} idx(R(t_1', p)) &= idx(R(t_1, p)) = idx(R(t_2, p)) \\ &= idx(R(t_2', p)) = E_p \end{aligned}$$

Therefore $t_1'.c|_{R(t_1',p)} = t_2'.c|_{R(t_2',p)}$ is equivalent to showing that $t_1'.c(i).h = t_2'.c(i).h$ for every $i \in E_p$:

$$\begin{array}{rcl} & t_1 \sim_p t_2 & \\ & t_1.c(i).h = t_2.c(i).h & \text{Def. } \sim_p \text{ and (4)} \\ & t_1'.c(i).h = t_2'.c(i).h & \text{by C 2 and CA 2} \end{array}$$

■

9 Related Work

Information flow security policies that require a complete absence of leakage are usually specified using different variations of noninterference following the approach pioneered by Goguen and Meseguer [22]. Various verification methods for noninterference have been developed including the self-composition method pioneered by Hähnle and others [19].

For realistic systems we need to support communication beyond the multilevel security model. IPR allows to transfer arbitrary information flow properties from specification (the abstract model) to implementation (the refined model) without the need of a specific mechanism for declassification, like the ones proposed in [42, 37, 5, 8, 6, 15, 37]. This allows to verify countermeasures against side-channels without the need of taking into account the mechanism used to analyze declassification in the abstract model.

The intuition behind IPR has been used to analyze information flow security in presence of speculative processors. Both *conditional noninterference* [26] and *speculative noninterference* [28] are restricted forms of IPR that formalize absence of speculative side channels by requiring that states non-interfering under non-speculative semantics are non-interfering under speculative semantics.

A precursor to IPR is the work of Cohen et al. [16] on abstraction in multi-agent systems. They introduce an epistemic simulation relation that is essentially a state-based version of IPR, and use this to show preservation of formulas in the epistemic temporal logic ACTLK.

Morgan and McIver [38, 35] propose an instrumented *shadow* semantics for ignorance-preserving program refinement, constructing the ignorance set explicitly for the final values of hidden variables. Moreover, their refinement requires equality for all global variables, allowing the introduction of new observations only as local variables within the scope of the refined program. These local variables cease to exist after the scope of the program has ended, hence the approach does not allow for persistent implementation variables (e.g. state of caches) that can carry data between invocations of different program segments.

One of the features that differentiate IPR from other works is that IPR supports introducing secret dependent observations together with sequential compositionality. Similarly to Section 6.2, a compiler may shuffle an array using a random key r and introduce a memory look-up dependent on $s \oplus r$ without compromising the secrecy of s. This type of refinement is not considered in [9, 23, 17] because it violates the assumption requiring that all refined runs of the same abstract state have the same leakage. Similarly, this is not allowed in [39], since modified variables must either be private, which is not the case for τ_{p_2}, or have their classification decreased, which would prevent the indirect flow of $s \oplus r$ to τ_{p_2}. This makes IPR a more permissive condition justifying information-theoretically secure refinement. Therefore it allows us to raise the question when it is possible to infer that an entire implementation is information flow secure in terms of the information flow security of the specification and the information flow preservation of each step.

Heifer et all [29] have investigate how to formally verify prevention of timing channels in seL4. Similarly to Section 8, they provide an abstracted representation of the hardware resources and postulated how they cause timing channels. Their work is specific for seL4 and depends on specific kernel designs. In contract, this work provides a blueprint for this type of analyses by using the general framework of IPR.

Security of constant time programming has been investigated in [10], where Barthe et. al. show that this policy protects against cache based side channels in virtualized environments. In [9] the authors show that several compiler optimizations do not affect the constant time policy, by demonstrating that observational noninterference w.r.t. a source state relation results in observational noninterference w.r.t. a target state relation after compilation.

10 Concluding remarks

We have analyzed a provably secure separation kernel using a compositional approach to information flow preserving refinement that is based on observer ignorance. In our abstract model, we have formalized a flat memory machine and axiomatized kernel properties that entail functional correctness. Once we refined the model to include caches and timing effects, we have shown information leakage for a simple key manager, i.e. observer ignorance is not preserved. Deploying two widely-adopted countermeasures such as cache coloring or a constant-time programming policy recovers ignorance preservation.

A possible extension of this work is to consider different attacker models. If extended to multiple attackers, constant time execution does not guarantee ignorance preservation individually for each attacker due to the presence of covert channels. However, it may be possible to reduce this case to a single, distributed attacker. In addition, one can handle access-driven attackers, i.e., attackers that cannot perform observations while suspended, by introducing a weak transition system that hides attacker-inactive steps.

It would also be interesting to extend the framework to handle other types of system features. For example, to handle cache flushes we would require constant time execution only for addresses that collide with cache entries that have not been previously cleaned. This approach can work if one can demonstrate that cache metadata is correctly reset after a flush, though unfortunately hardware vendors do not usually provide enough details about implementations to conclude this. To handle dynamic cache partitions at the kernel level, we should ensure the abstract model reflects any side channels that arise from the partitioning mechanism.

A final and more difficult problem concerns attackers potentially abusing low level "features" such as Rowhammer [31], mismatched cache attributes [27], speculative execution, or self-modifying code to induce changes in program behavior that are not visible at the abstract level where the effect of these features may not be reflected. We call such attacks *behavior morphing*. In such cases the machinery in this paper no longer works fully as intended. On the other hand, countermeasures against vulnerabilities like Rowhammer do exist [14] that suitably confine the effects of any behavior morphing and should be verifiable using techniques akin to the ones we present in this paper. We leave a proper treatment of ignorance-preserving refinement in the presence of behavior morphing for future work.

Acknowledgment. This work partially was supported by grants from the Swedish Foundation for Strategic Research and the Swedish Civil Contingencies Agency, and the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

Bibliography

[1] seL4 Project, available from: `http://sel4.systems/`. Accessed: 2017-04-21

[2] Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P

[3] Acıiçmez, O., Koç, Ç.K., Seifert, J.P.: Predicting secret keys via branch prediction. In: Cryptographers' Track at the RSA Conference. pp. 225–242. Springer (2007)

[4] Alur, R., Cerný, P., Zdancewic, S.: Preserving secrecy under refinement. In: International Colloquium on Automata, Languages and Programming. pp. 107–118. Springer (2006)

[5] Askarov, A., Chong, S.: Learning is change in knowledge: Knowledge-based security for dynamic policies. In: Computer Security Foundations Symposium (CSF). pp. 308–322. IEEE (2012). https://doi.org/10.1109/CSF.2012.31

[6] Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: Symposium on Security and Privacy. pp. 207–221. IEEE (2007). https://doi.org/10.1109/SP.2007.22

[7] Balliu, M., Dam, M., Guanciale, R.: Automating information flow analysis of low level code. In: Proceedings of the Conference on Computer and Communications Security. pp. 1080–1091. CCS'14, ACM (2014). https://doi.org/10.1145/2660267.2660322

[8] Balliu, M., Dam, M., Le Guernic, G.: Epistemic temporal logic for information flow security. In: Workshop on Programming Languages and Analysis for Security. pp. 6:1–6:12. ACM (2011). https://doi.org/10.1145/2166956.2166962

[9] Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In: IEEE 31st Computer Security Foundations Symposium (CSF). pp. 328–343 (July 2018). https://doi.org/10.1109/CSF.2018.00031

[10] Barthe, G., Betarte, G., Campo, J., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conbference on Computer and Communications Security. pp. 1267–1279. ACM (2014)

[11] Baumann, C., Dam, M., Guanciale, R., Nemati, H.: On compositional information flow aware refinement. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00010, `https://doi.org/10.1109/CSF51468.2021.00010`

[12] Benton, N.: Simple relational correctness proofs for static analyses and program transformations. SIGPLAN Not. **39**(1), 14–25 (Jan 2004). https://doi.org/10.1145/982962.964003

[13] Blazy, S., Pichardie, D., Trieu, A.: Verifying constant-time implementations by abstract interpretation. In: European Symposium on Research in Computer Security. pp. 260–277. Springer (2017)

[14] Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R.: Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 117–130 (2017)

[15] Chong, S., Myers, A.C.: Security policies for downgrading. In: Conference on Computer and Communications Security. pp. 198–209. ACM (2004). https://doi.org/10.1145/1030083.1030110

[16] Cohen, M., Dam, M., Lomuscio, A., Russo, F.: Abstraction in model checking multi-agent systems. In: Proc. 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Vol, 2. pp. 945–952 (2009)

[17] Costanzo, D., Shao, Z., Gu, R.: End-to-end verification of information-flow security for C and assembly programs. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 648–664 (2016). https://doi.org/10.1145/2908080.2908100

[18] Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple ARM-based separation kernel. In: Proceedings of the Conference on Computer and Communications Security. pp. 223–234. CCS'13, ACM (2013)

[19] Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3450, pp. 193–209. Springer (2005). https://doi.org/10.1007/978-3-540-32004-3_20, `https://doi.org/10.1007/978-3-540-32004-3_20`

[20] Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about knowledge. MIT Press (1995)

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

[22] Goguen, J.A., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy. pp. 11–20. IEEE (1982). https://doi.org/10.1109/SP.1982.10014

[23] Graham-Cumming, J., Sanders, J.W.: On the refinement of non-interference. In: Proceedings Computer Security Foundations Workshop IV (CSFW). pp. 35–42 (1991). https://doi.org/10.1109/CSFW.1991.151567

[24] Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A remote software-induced fault attack in Javascript. In: Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 300–321. Springer (2016)

[25] Gu, R., Shao, Z., Chen, H., Wu, X., Kim, J., Sjöberg, V., Costanzo, D.: Certikos: An extensible architecture for building certified concurrent os kernels. In: Proceedings of the 12th USENIX Conference on Operating Systems

Design and Implementation. pp. 653–669. OSDI'16, USENIX Association, Berkeley, CA, USA (2016)

[26] Guanciale, R., Balliu, M., Dam, M.: Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020. pp. 1853–1869 (2020). https://doi.org/10.1145/3372297.3417246, `https://doi.org/10.1145/3372297.3417246`

[27] Guanciale, R., Nemati, H., Baumann, C., Dam, M.: Cache storage channels: Alias-driven attacks and verified countermeasures. In: Symposium on Security and Privacy. pp. 38–55. IEEE (2016). https://doi.org/10.1109/SP.2016.11

[28] Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: Spectector: Principled detection of speculative information flows. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1–19 (2020). https://doi.org/10.1109/SP40000.2020.00011, `https://doi.org/10.1109/SP40000.2020.00011`

[29] Heiser, G., Klein, G., Murray, T.: Can we prove time protection? In: Proceedings of the Workshop on Hot Topics in Operating Systems. pp. 23–29 (2019)

[30] Kessler, R.E., Hill, M.D.: Page placement algorithms for large real-indexed caches. Transactions on Computer Systems **10**(4), 338–359 (1992). https://doi.org/10.1145/138873.138876

[31] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. pp. 361–372. ISCA '14, IEEE Press, Piscataway, NJ, USA (2014)

[32] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. ArXiv e-prints (Jan 2018)

[33] Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Annual International Cryptology Conference. pp. 104–113. Springer (1996)

[34] Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Cryptology Conference on Advances in Cryptology. pp. 388–397. Springer (1999)

[35] McIver, A., Morgan, C.C.: *Sums and Lovers:* Case studies in security, compositionality and refinement. In: Formal Methods. pp. 289–304. Springer (2009)

[36] McLean, J.: The specification and modeling of computer security. Computer **23**(1), 9–16 (Jan 1990). https://doi.org/10.1109/2.48795

[37] van der Meyden, R.: What, indeed, is intransitive noninterference? In: European Symposium on Research in Computer Security (ESORICS). pp. 235–250. Springer (2007)

[38] Morgan, C.: *The Shadow Knows*: Refinement and security in sequential programs. Science of Computer Programming **74**(8), 629–653 (2009). https://doi.org/10.1016/j.scico.2007.09.003

[39] Murray, T.C., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: IEEE 29th Computer Security Foundations Symposium, (CSF). pp. 417–431 (2016). https://doi.org/10.1109/CSF.2016.36

[40] Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: RSA Conference on Topics in Cryptology. pp. 1–20. Springer (2006)

[41] Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive **2002**, 169 (2002)

[42] Rushby, J.: Noninterference, transitivity and channel-control security policies. Tech. rep., SRI International (1992)

[43] Santen, T., Heisel, M., Pfitzmann, A.: Confidentiality-preserving refinement is compositional - Sometimes. In: Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS). p. 194–211. Springer-Verlag (2002)

[44] Stefan, D., Buiras, P., Yang, E.Z., Levy, A., Terei, D., Russo, A., Mazières, D.: Eliminating cache-based timing attacks with instruction-based scheduling. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) European Symposium on Research in Computer Security. pp. 718–735. Springer (2013)

[45] Van der Meyden, R., Zhang, C.: Information flow in systems with schedulers, Part II: Refinement. Theor. Comput. Sci. **484**, 70–92 (May 2013). https://doi.org/10.1016/j.tcs.2013.01.002