Practical Probability: Applying pGCL to Lattice Scheduling

David Cock

NICTA and University of New South Wales
David.Cock@nicta.com.au

Abstract. Building on our published mechanisation of the probabilistic program logic pGCL we present a verified lattice scheduler, a standard covert-channel mitigation technique, employing randomisation as an elegant means of ensuring starvation-freeness. We show that this scheduler enforces probabilistic non-leakage, in addition to non-starvation. The refinement framework employed is compatible with that used in the L4.verified project, supporting our argument that full-scale verification of probabilistic security properties for realistic systems software is feasible.

1 Introduction

In this paper, we demonstrate that mechanically verifying realistic probabilistic software, with probabilistic properties, is feasible. Our 'realistic probabilistic program' is a hybrid probabilistic lattice-lottery scheduler, designed to mitigate information flow through a shared cache, while guaranteeing fairness and remaining simple and efficient. The probabilistic properties are: stochastic fairness — that the probability of starvation for any domain is zero, and non-leakage — that the distribution of observable outputs is independent of hidden inputs. Finally, we make our argument for feasibility by proving our results in a refinement framework compatible with the L4.verified [KEH$^+$09] proof stack, which established by refinement that the seL4 microkernel faithfully implements its specification. We are able to restrict probabilistic reasoning to small regions, allowing the remainder of the proof to proceed in a traditional manner.

We begin with an abstract, nondeterministic specification, which we refine iteratively. Our first refinement is to a probabilistic version, and then to a practical implementation based on lottery scheduling. We demonstrate that this refinement could be continued using the L4.verified results. Finally, we attach a hardware model, allowing us to demonstrate that we do in fact eliminate leakage through the cache.

We express our results using the probabilistic Guarded Command Language (pGCL) of McIver and Morgan [MM04], previously mechanised in HOL4 [HMM05]. This language has been applied in a practical context, namely analysing parts of the FireWire protocol [FS03]. We demonstrate that it is equally applicable to systems-level software. Our own mechanization of pGCL in Isabelle/HOL, specifically aimed at the lightweight integration of existing results, has been described previously [Coc12].

We assume a lattice-based security policy, an established idea, motivated by institutional classification policies [DoD86], and formally treated by authors including Denning [Den76]. Lattice scheduling, due to Hu et al. [Hu92], arose from the VAX VMM project [KZB+91], and is intended to mitigate precisely the kind of leakage that we consider. Lottery scheduling is an established technique for efficient hierarchical allocation of execution time, introduced by Waldspurger et al. [WW94] Our approach uses it only as an elegant way to implement probabilistic scheduling: we do not take advantage of hierarchical resource distribution.

While the threat from cache-based channels has long been recognised, interest has been spurred by recent work demonstrating the feasibility of attacking cryptographic algorithms in a co-hosted system [Ber04, Per05]. Alternative mechanical approaches include that of Barthe et al. [BBCL12] Our work contrasts with this by incorporating probability, and interfacing with a large existing verification effort [KEH+09].

Many authors have analysed the leakage properties of scheduling algorithms [CM07,HN12,GKV11], some employing mechanical proof. Most existing analyses focus on leakage due to the actions of the scheduler itself, or due to the order of updates to shared variables. We are specifically concerned with mitigating a side channel, outside any explicitly shared state. The absence of unintended channels through explicit mechanisms in seL4 has already been established [MMB+12].

1.1 pGCL in Isabelle

We first summarise pGCL, noting small variations in syntax relative to the standard presentation. This summary is naturally incomplete, and the interested reader is directed to the aforementioned work of McIver and Morgan [MM04], and to our own previous summary of the mechanisation [Coc12].

Programs in pGCL have two interpretations: The first is as a probabilistic state transformer, taking a given starting state to one of several possible final states, with well-defined probability. The second interpretation is as an *expectation transformer*, mapping a real-valued function on final states (a post-expectation), to one on initial states (a pre-expectation). The *weakest pre-expectation* (wp) of a post-expectation, under a program, and evaluated at some initial state is the smallest *expected value* (minimised over demonic choices) of the post-expectation in the final state, if the program were to execute from the given initial state. For example, the weakest pre-expectation of the expression x, under the program

$$(x := 1\ {}_{1/2}\!\oplus\ x := 0) \sqcap (x := 2\ {}_{1/3}\!\oplus\ x := 1) \qquad ,$$

(where $a\ {}_p\!\oplus\ b$ is probabilistic choice and $a \sqcap b$ demonic) is

$$\min\ \left(\frac{1}{2}\times 1+\frac{1}{2}\times 0\right)\ \left(\frac{1}{3}\times 2+\frac{2}{3}\times 1\right)=\frac{1}{2} \qquad .$$

While our mechanisation is in terms of expectation transformers, the two interpretations are equivalent, and the forward transformer is generally more intuitive, giving the most straightforward way to visualise results.

Programs are constructed using several operators, including:

– Sequential composition: $a\,;;\,b$.
– Name binding: n is f in $a\ n$, where f is a function from state to value.
– Demonic choice: $x :\in S$, where S is a set-valued function, and x a variable (field) name.
– Probabilistic choice: $x :\in S$ at p, where p is a distribution over S.
– Finite repetition: $a^n = \underbrace{a\,;;\,\ldots\,;;\,a}_{n}$.
– Lifting from a non-probabilistic monad: Exec M.
– Applying a state transformer: Apply f.

While programs may operate on any state type, in practice we use Isabelle's *record types*: tuples with labelled fields, similar to C structures. The advantage is that with support from our mechanisation, we are able to use Isabelle field identifiers directly as pGCL variable names. For example we may write $x := v$, which is translated internally to Apply $\lambda s.\ x\_update\ (\lambda\_.\ v)\ s$, an Isabelle record update. This program could be applied to the following state, expressing a record of two fields, of types τ and μ:

$$\begin{aligned} \textbf{record}\ \text{state} = {} & x :: \tau \\ & y :: \mu \end{aligned}$$

The assertion language is shallowly embedded, and closely resembles the predicate-transformer semantics of Dijkstra's GCL [Dij75]. There are a few novel probabilistic constructions, including:

– Entailment: $P \Vdash Q = \forall s.\ P\ s \leq Q\ s$ (standard syntax $\Rrightarrow$).
– Conjunction: $P\ \&\&\ Q = \lambda s.\ \max 0\ (P\ s + Q\ s - 1)$ (standard syntax $\&$).
– Embedding: $«P» = \lambda s.$ if $P\ s$ then 1 else 0 (standard syntax $[P]$).

Probabilistic entailment is a straightforward generalisation of predicate entailment, $P \vdash Q \Leftrightarrow \forall s.\ P\ s \rightarrow Q\ s$, while embedding is simply syntactic sugar. The form of probabilistic conjunction is chosen for compatibility with its boolean equivalent i.e. $«P»\ \&\&\ «Q» = «\lambda s.\ P\ s \wedge Q\ s»$. That we use this particular form (rather than, for example $P\ \&\&\ Q = \lambda s.\ P\ s \times Q\ s$, which gives the same results on embedded predicates) is for technical reasons concerning the underlying semantic interpretation[1].

The following is an example specification in expectation-entailment style that illustrates the essential features of the logic:

$$«P»\ \&\&\ (\lambda\_.\ p) \Vdash \text{wp}\ (a\,;;\,b)\ «Q»$$

[1] Briefly, the definition given is the only option that is *sub-linear*, a generalisation (to real-valued functions), and weakening, of the *linearity* condition required of expectation transformers in pure GCL. All sub-linear transformers are linear, and sub-linearity reduces to linearity in the case of embedded boolean predicates, but (for example) demonic choice, $a \sqcap b = \lambda s.\ \min\ (a\ s)\ (b\ s)$, is not linear if a or b take values other than 0 and 1. All pGCL primitives are sub-linear, this being the healthiness condition for transformers. For further details, see McIver & Morgan [MM04].

This states that from any initial state satisfying P, after executing a followed by b, we reach a state satisfying Q with probability *at least* p.

Conventions For simplicity, we employ the following conventions throughout: Unbound variables are implicitly universally quantified, and all expectations are non-negative, and bounded by 1.

2 Security Policies and Covert Channels

$$\begin{tikzcd}
 & 3:\{A,B\} & \\
1:\{A\} \arrow[ur] & & 2:\{B\} \arrow[ul] \\
 & \bot:\{\} \arrow[ul] \arrow[ur] \arrow[uu] &
\end{tikzcd}$$

Fig. 1. The classification/clearance lattice

We consider a hierarchically partitioned system, as depicted in Figure 1. Here, all data is classified with one (or both) of the labels A and B. An agent (or program) may be cleared to process one, both, or neither of these, giving rise to 4 *clearance domains*: 1 for A only, 2 for B only, 3 for both and $\bot$ for neither. Our goal is to ensure that information derived from labelled data can only flow into a domain cleared to process it. The formulation of access control policies for such systems, encompassing explicit channels, is a well-studied problem [Den76]. This work is concerned with formalising implementation techniques to prevent leakage through unintended, implicit channels (either covert- or side-channels).

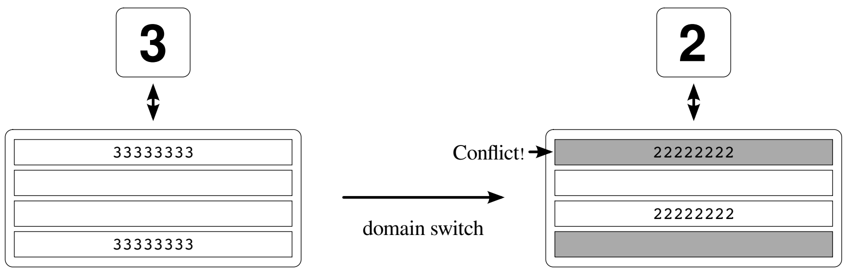

Fig. 2. The cache-contention channel

Figure 2 outlines such a channel, exploiting *cache contention*. The cache is represented by an array of cells (lines), and 3 and 2 are two of our supposedly isolated domains. We ignore the associativity of the cache (one cell may in fact hold several lines), as it only serves to reduce contention.

As domain 3 executes and accesses memory, it gradually fills the cache with its own data. On switching to domain 2, domain 3's data remains in the cache, but is now inaccessible (the greyed-out cells). This access control is enforced by the hardware. As 2 starts to fill the cache, it may eventually attempt to store a value in one of the grey squares, encountering a *conflict*, as indicated.

When a conflict occurs, the cache silently writes (cleans) the old value (domain 3's) into main memory, before storing domain 2's new value[2]. This process is in principle invisible to domain 2. However, if 2 is able to measure its own execution time, the delay caused by writing (cleaning) the cache line to memory can be detected. Domain 2 can thus infer which cache lines 3 has accessed, in violation of the security policy.

The leakage is dramatic: On a uniprocessor, where domains cannot execute concurrently, the bandwidth tops 10kb/s, while on a multiprocessor, bandwidths in excess of 1Mb/s are easily achieved.

3 Countermeasures Through Refinement

$$\begin{tikzcd}
 & 3 \arrow[dd, dashed] & \\
1 \arrow[ur] & & 2 \arrow[ul] \arrow[dl, dashed, shift left] \\
 & \perp \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Fig. 3. The scheduling graph, S

The simplest countermeasure to the cache channel is to flush the cache on every domain switch, returning it to a known secure state. This is, however, a very expensive option: A modern processor, for example an Intel Xeon E7-8870, might have a 30MiB cache, taking $\approx 2.5 \times 10^6$ cycles to refill (at the peak theoretical bandwidth of the memory subsystem), or 89% of the 2.8×10^6 cycles per preemption interval at 1000Hz.

A simple optimisation [Hu92] is to clear the cache only when essential. Considering our partitioned system, it is acceptable to permit leakage from a domain to any other domain whose clearance includes that of the first. Thus it is only

[2] In a write-back cache with write-allocate. Read contention occurs in all caches.

necessary to flush when decreasing clearance level. This is the essence of *lattice scheduling*: Transition upward in the classification lattice for as long as possible, before finally starting again at the bottom, employing countermeasures to protect the downward transition.

To implement this, we construct the *scheduling graph* in Figure 3; consistent with the classification graph in Figure 1. The scheduling graph gives valid domain transitions for the system, and contains only edges from the classification graph, or transitions to the *downgrader*, $\perp$. These latter are emphasised with a dashed arrow. In the implementation, the shared cache must be flushed on entering the downgrader. We omit the edges from $\perp$ to 3, and from 1 to $\perp$, to emphasise that not all edges need be included.

The conditions on the scheduling graph (modelled as a relation) are captured as assumptions on S (encapsulated within an Isabelle locale), with the most important being downgrading:

Lemma 1 (Downgrading). *If S allows a downward transition, it is to the downgrader, $\perp$:*

$$\frac{(c, n) \in S \quad \mathit{clearance}\ c \nsubseteq \mathit{clearance}\ n}{n = \perp}$$

We specify the scheduler nondeterministically over the valid transitions from the current domain, using the unconstrained demonic choice operator:

```
**record** stateA = current_domain :: *dom_id*
scheduleS =
   $c$ is current_domain in
   current_domain :∈ ($\lambda$_. $\{n.\ (c, n) \in S\}$)
```

3.1 A Randomised Scheduler

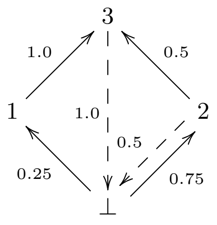

Fig. 4. The transition graph, T

The classically nondeterministic specification of scheduleS, together with the downgrading property, capture the requirement that all downward transitions pass through the downgrader. As a practical specification however, it has a disadvantage: it allows starvation. A refinement of this specification is free to follow any trace within the graph, for example $(\bot, 2, \bot, 2, \dots)$, never scheduling domain 3.

We could extend the specification to guarantee starvation freeness, by dictating its behaviour over traces in a modal logic. This would risk obscuring the present simplicity of the specification, and would require a more complex implementation, needing to take into account more than just the current domain.

Randomisation provides an elegant alternative: By assigning a probability to each edge in Figure 3, we produce the *transition graph* in Figure 4. We only require that the outgoing probabilities from each node sum to 1, and that any transition with non-zero probability appears as an edge in Figure 3. Implementation remains simple, needing only to consider the current state in choosing a transition. More importantly, with appropriately chosen transition probabilities, the probability of starvation can be made zero. We specify the new scheduler using the probabilistic choice operator:

$$\begin{aligned}
&\text{scheduleT} = \\
&\quad c \text{ is current\_domain in} \\
&\quad \text{current\_domain} :\in (\lambda\_.\ \{\bot, 1, 2, 3\} \text{ at } (\lambda\_\ n.\ T\ (c, n))
\end{aligned}$$

The scheduler is now a Markov process, with T fixing its transition rule. Under the appropriate conditions (strong-connectedness, or positive recurrence interval for all states), there exists an asymptotic equilibrium distribution. These conditions are satisfied by T, and thus in addition to avoiding starvation, the randomised lattice scheduler guarantees statistical fairness, over the long run.

3.2 Program Refinement and Starvation-Freedom

In order to eventually show non-leakage, we need to demonstrate that the downgrading property is also shared by scheduleT. We do so by establishing that scheduleT is a *probabilistic refinement* of scheduleS.

Definition 1. *Program b* refines *program a, written $a \sqsubseteq b$, exactly when all expectation-entailments on a also hold on b:*

$$\frac{P \Vdash wp\ a\ Q}{P \Vdash wp\ b\ Q}$$

Lemma 2. *The transition scheduler refines the lattice scheduler:*

$$scheduleS \sqsubseteq scheduleT$$

Note that, in the terminology of pGCL, the specification of scheduleT is completely ‘deterministic’, referring to the absence of demonic nondeterminism.

This terminology makes sense in light of the refinement order: Demonic nondeterminism can be restricted by refinement, whereas probabilistic choice cannot. Once a specification is fully probabilistic, it is maximal in the refinement lattice, and one can take it no further. This implies that any further refinement is, in fact, semantic equivalence. We make use of this fact shortly, as a shortcut to establishing program correspondence.

Having fixed transition probabilities, we can establish non-starvation. Proceeding in stages, we first show that starting in *any* domain, the probability of ending in domain $\perp$ after 4 steps is at least 1/64:

$$(\text{in\_dom } d_i) \,\&\&\, \left(\lambda\_.\ \frac{1}{64}\right) \Vdash \text{wp scheduleT}^4\ (\text{in\_dom } \perp)$$

where

$$\text{in\_dom } d \quad \leftrightarrow \quad «\lambda s.\ \text{current\_domain } s = d»$$

We further establish that from domain $\perp$, after a further 4 steps, there is a non-zero probability of ending in any given final domain:

$$(\text{in\_dom } \perp) \,\&\&\, \left(\lambda\_.\ \frac{1}{64}\right) \Vdash \text{wp scheduleT}^4\ (\text{in\_dom } d_f)$$

Combining these, we have:

$$\left(\lambda\_.\ \frac{1}{4096}\right) \Vdash \text{wp scheduleT}^8\ (\text{in\_dom } d_f) \tag{1}$$

Finally:

Lemma 3 (Non-starvation). *Taking at least 8 steps from* any *initial domain, we reach* any *final domain with non-zero probability:*

$$\forall s.\ 0 < wp\ scheduleT^{8+n}\ (in\_dom\ d_f)\ s$$

Proof. By induction on n. Equation 1 establishes the result for $n = 0$. By inspection of Figure 4, we see that every domain is reachable in one step, and with non-zero probability, from at least one other, and thus if all domains are reachable after n steps then all are reachable after $n + 1$. □

Figure 5 summarises these results. We have downgrading for scheduleS by assumption, and non-starvation for the probabilistic scheduleT, as indicated by the dotted arrows. Refinement is depicted as a solid arrow. The arrow directions summarise the compositionality of results: composing with refinement, downgrading also holds for scheduleT, but non-starvation does not hold for scheduleS.

$$\begin{tikzcd}
\textsc{Downgrading} \arrow[r, dotted] & \text{scheduleS} \arrow[d] \\
\textsc{Non-Starvation} \arrow[r, dotted] & \text{scheduleT}
\end{tikzcd}$$

Fig. 5. First Refinement Diagram

3.3 Data Refinement and the Lottery Scheduler

It is not sufficient to have an elegant specification, unless that specification can be practically implemented. Therefore we implement our randomised lattice scheduler as a *lottery scheduler*. We only require the assumption of randomness for a single operation: drawing a ticket.

We extend the abstract state with a *lottery* for each domain. Every possible successor domain holds a certain set of tickets, given by the function 'lottery'. To transition, the scheduler draws a ticket (32 word) and consults the table to choose a successor. To emphasise that the probabilistic component can be isolated, and to demonstrate compatibility with our existing framework, we divide the implementation into a core, in the nondeterministic state monad [CKS08], which is then lifted into pGCL using the Exec operator, allowing us to employ probabilistic choice. Both scheduleC and scheduleM operate on the same state space: stateC. The syntax $r(\!|x := y|\!)$ is an Isabelle record update, assigning value y to field x of record r.

$$\begin{aligned}
\textbf{record}\ \text{domain} &= \text{lottery} :: 32\ \text{word} \Rightarrow \mathit{dom\_id} \\
\textbf{record}\ \text{stateC} &= \text{current\_domain} :: \mathit{dom\_id} \\
&\ \text{domains} :: \mathit{dom\_id} \Rightarrow \text{domain} \\
\text{scheduleM}\ t &= \text{do}\ c \leftarrow \text{gets current\_domain} \\
&\qquad dl \leftarrow \text{gets domains} \\
&\qquad \text{let}\ n = \text{lottery}\ (dl\ c)\ t\ \text{in} \\
&\qquad \text{modify}\ (\lambda s.\ s(\!|\text{current\_domain} := n|\!)) \\
&\quad \text{od} \\
\text{scheduleC} &= t\ \text{from}\ (\lambda s.\ \text{UNIV})\ \text{at}\ 2^{-32}\ \text{in} \\
&\quad \text{Exec}\ (\text{scheduleM}\ t)
\end{aligned}$$

Having moved to a new state space, we cannot have direct program refinement between scheduleT and scheduleC. Noting, however, that the abstract state can be recovered from the concrete by projection, we instead have *(projective) probabilistic data refinement*:

Definition 2 (Probabilistic Data Refinement). *Program b, on state type σ, refines program a, state τ, given precondition $G : \sigma \to Bool$ and under projection*

$\theta : \sigma \to \tau$, written $a \sqsubseteq_{G,\theta} b$, exactly when any expectation entailment on a implies the same for b, on the projected state and with a guarded pre-expectation:

$$\frac{P \Vdash wp\ a\ Q}{\text{«}G\text{»}\ \&\&\ (P \circ \theta) \Vdash wp\ b\ (Q \circ \theta)}$$

Lemma 4. *Let $\|S\|$ be the cardinal measure (element count) of set S.*
Under condition LR, that 'lottery' reflects the transition matrix,

$$T\ (c, n) = 2^{-32} \|\{t.\ lottery\ (domains\ s\ c)\ t = n\}\|$$

then under projection ϕ, which extracts the current domain,

$$current\_domain\ (\phi\ s) = current\_domain\ s$$

scheduleC is a data refinement of scheduleT:

$$scheduleT \sqsubseteq_{LR,\phi} scheduleC$$

3.4 Probabilistic Correspondence

$$\begin{tikzcd}
\textsc{Downgrading} \arrow[r, dotted] & \text{scheduleS} \arrow[d] \\
\textsc{Non-Starvation} \arrow[r, dotted] & \text{scheduleT} \arrow[d, Rightarrow, "\phi\text{,LR}"] \\
& \text{scheduleC}
\end{tikzcd}$$

Fig. 6. Second Refinement Diagram

As mentioned in Section 3.2, scheduleT is maximal in the refinement order, and thus any refinement is an equivalence. This is *probabilistic correspondence*:

Definition 3 (Probabilistic Correspondence). *Programs a and b are said to be in* probabilistic correspondence*, $pcorres\ \theta\ G\ a\ b$, given condition G and under projection θ if, for any post-expectation Q, the guarded pre-expectations coincide:*

$$\text{«}G\text{»}\ \&\&\ (wp\ a\ Q \circ \theta) = \text{«}G\text{»}\ \&\&\ wp\ b\ (Q \circ \theta)$$

Probabilistic correspondence is guarded equality on distributions: From an initial state satisfying G, a and b establish Q with equal probability. The advantage of detouring via refinement, rather than directly showing correspondence, is that the proof is simpler; the next result follows directly from Lemma 4:

Lemma 5. *The specifications scheduleT and scheduleC correspond given condition LR and under projection ϕ:*

$$\mathit{pcorres}\ \phi\ \mathit{LR}\ \mathit{scheduleT}\ \mathit{scheduleC}$$

This extends Figure 5 to Figure 6, with correspondence indicated by the double arrow. As correspondence implies refinement, both downgrading and non-starvation hold for scheduleC, as implied by the arrows. Properties represented by a single dotted arrow (e.g. downgrading), are preserved by both refinement (single arrow) and correspondence (double arrow).

3.5 Proof Reuse: Composing with seL4

$$\begin{tikzcd}
 & \textsc{Downgrading} \arrow[r, dotted] & \text{scheduleS} \arrow[d] \\
 & \textsc{Non-Starvation} \arrow[r, dotted] & \text{scheduleT} \arrow[d, Rightarrow, "\phi{,}\text{LR}"] \\
\text{callKernelD} = = = \text{stepKernel};;\text{scheduleC} \arrow[d, Rightarrow] & & \text{scheduleC} \arrow[ll, Rightarrow, "\phi{,}\text{LR}"'] \\
\text{callKernelH} \arrow[d, Rightarrow] & & \\
\text{callKernelC} & &
\end{tikzcd}$$

Fig. 7. Composed Refinement Diagram

Our argument for the feasibility of this approach rests on the compatibility of probabilistic correspondence with its non-probabilistic equivalent at the heart of the L4.verified proof. We have previously demonstrated the ease with which monadic specifications, in the style of seL4, can be re-used in a probabilistic setting [Coc12], automatically lifting Hoare triples to probabilistic predicate entailment relations. With the following result we go further, and lift the bulk of the refinement stack. The predicate corres_underlying in the following lemma is the fundamental definition which underlies the refinement results at all levels of the L4.verified proof [CKS08]. Here, we need only note that this is the form of the top-level theorem[3].

Lemma 6 (Lifting Correspondence). *Given correspondence between monadic programs M and M', with precondition G and projective state relation ϕ,*

$$\textit{corres\_underlying}\ \{(s, s').\ s = \phi\ s'\}\ \textit{True}\ \textit{rrel}\ G\ (G \circ \phi)\ M\ M'$$

where M does not fail given G,

$$\textit{no\_fail}\ G\ M$$

and neither diverges without failing,

$$\textit{empty\_fail}\ M \quad \textit{empty\_fail}\ M'$$

and that M is deterministic on the image of the projection,

$$\forall s.\ \exists (r, s').\ M\ (\phi\ s) = \{(\textit{False}, (r, s'))\}$$

then we have probabilistic *correspondence between their lifted counterparts:*

$$\textit{pcorres}\ \phi\ (G \circ \phi)\ (\textit{Exec}\ M)\ (\textit{Exec}\ M')$$

Note that the final assumption is exactly the determinism[4] condition that we previously established for scheduleT, restricted to the components of interest. M is free to behave nondeterministically on components which are masked by the projection.

Thus we may compose our probabilistic results with the deterministic levels of the L4.verified proof (the executable, or more recent deterministic abstract [MM12], specification). For the problem at hand, it is only necessary to make a few assumptions on the kernel:

---

[3] Briefly, corres_underlying *srel nf rrel G G′ m m′* is defined as:

$$\begin{aligned}&\forall (s, s') \in \textit{srel}.\ G\ s \wedge G'\ s' \rightarrow (\forall (r', t') \in \text{fst}\ (m'\ s').\\&\quad \exists (r, t) \in \text{fst}\ (m\ s).\ (t, t') \in \textit{srel} \wedge \text{rrel}\ r\ r' \wedge (\textit{nf} \rightarrow \neg\text{snd}\ (m'\ s')))\end{aligned}$$

Where guards G and G' hold on initial states s and s' satisfying state relation *srel*, for any pair of (result, final state) obtained by executing m', there exists a corresponding pair obtainable by executing m. If the non-failure flag, *nf* is set, then the predicate additionally asserts that m' does not fail.

We use the predicate with a projective relation derived from ϕ, no failure, an arbitrary result relation, and a concrete guard which is the anti-projection of the abstract guard $(G \circ \phi)$.

[4] Determinism gives us correspondence, rather than just refinement. Consider monads A and A', and variable $x : \mathbb{N}$, preserved by projection ϕ_A. Let $A\ s$ be nondeterministic, giving either$s(\!|x := x\ s + 1|\!)$ or $s(\!|x := x\ s + 2|\!)$, while $A'\ s$ is deterministic, giving $s(\!|x := x\ s + 2|\!)$. All behaviours of A' are included in A, and thus corres_underlying holds. However, wp $A\ x = \lambda s.\ x\ s + 1$ whereas wp $A'\ (x \circ \phi_A) = \lambda s.\ x\ s + 2$: a refinement, but not correspondence. As previously mentioned, if A were deterministic then by maximality, this refinement would be correspondence.

Lemma 7. *If the kernel preserves the lottery relation,*

$$\{\!|LR|\!\}\ \mathit{stepKernel}\ \{\!|\lambda\_.\ LR|\!\}$$

and the current domain,

$$\{\!|\lambda s.\ CD\ s = d|\!\}\ \mathit{stepKernel}\ \{\!|\lambda\_\ s.\ CD\ s = d|\!\}$$

and is total,

$$\mathit{no\_fail}\ \top\ \mathit{stepKernel} \quad \mathit{empty\_fail}\ \mathit{stepKernel}$$

then with the concrete scheduler, it refines the transition scheduler:

$$\mathit{scheduleT} \sqsubseteq_{LR,\phi} \mathit{stepKernel;;scheduleC}$$

With this (again using refinement to show correspondence), Figure 6 becomes Figure 7, now including the lifted kernel. The L4.verified refinement stack is depicted on the left to indicate how the results would compose, to take our result down to the real, executable kernel. Here callKernelD is the deterministic refinement of original abstract specification of seL4, callKernelH is the executable model derived from the Haskell prototype, and callKernelC is the concrete model, comprising the final C and assembly language implementation

So far, we have only shown that our results are *compatible*: we do not yet have a mechanised proof. The remaining results are the first two assumptions of Lemma 7, which will hold by construction as the existing kernel clearly cannot modify the additional scheduler state, and the fact that the state relation is projective: that is, that the abstract state is uniquely recoverable from the concrete state. This is the intended behaviour of the state relation, and we have no reason to suspect that this result will not hold.

3.6 Non-leakage with a Concrete Machine Model

Our ultimate goal is to show the absence of information leakage via shared state (specifically the processor cache), and so we extend our scheduler with a simple hardware model. We model a private state per domain (memory), and a single shared state (cache):

$$\begin{aligned}\textbf{record}\ (sh, pr)\ \text{machine} = {} & \text{private} :: \mathit{dom\_id} \Rightarrow pr\\ & \text{shared} :: sh\end{aligned}$$

The action of a domain is modelled by the underspecified function runDom :: $sh \times pr \Rightarrow sh \times pr$, acting on both the current domain's private state and the shared state. Only the action of domain $\perp$ is specified, and then only on the shared state, resetting it.

The model exposes the essential information-flow characteristics of the cache channel, as illustrated by Figure 8. Initially, the states associated with domain 3 (black) and 2 (grey) are isolated. After a single step, domain 3's influence propagates to the cache (S), but as yet no other private state has been affected.

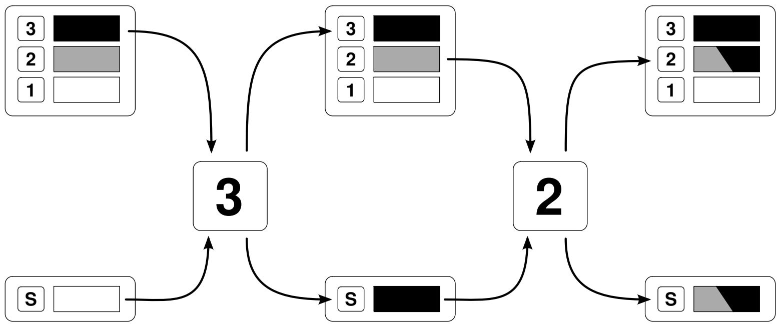

Fig. 8. A schematic depiction of flow from 3 to 2, via shared state S

It is only after the second step that influence propagates to 2's private state, it and the cache now being influenced by both 2 and 3's initial states. As this mixing of private states cannot occur in less than 2 steps, and may take an unbounded time (2's state cannot be influenced until 2 is scheduled), we cannot formulate a one-step security property. Instead we have a trace property, enforcing that after any number of steps, the distribution of outcomes visible to a low observer is independent of any initial high state, a form of *probabilistic non-leakage* [vO04]:

Lemma 8 (Non-leakage). *If the clearance of domain h is not entirely contained within that of domain l,*

$$\mathit{clearance}\ h \nsubseteq \mathit{clearance}\ l$$

then any function of the state after execution, which depends only on elements within l's clearance,

$$Q \circ \mathit{mask}\ l$$

is invariant under modifications to h's private state (as represented by replace):

$$\begin{aligned} &\mathit{wp}\ (\mathit{runDom;;scheduleT})^n\ (Q \circ \mathit{mask}) = \\ &\qquad\qquad (\mathit{wp}\ (\mathit{runDom;;scheduleT})^n\ (Q \circ \mathit{mask})) \circ (\mathit{replace}\ h\ p) \end{aligned}$$

We also have correspondence between scheduleT and runDom;;scheduleC:

Lemma 9. *Assuming that the lottery relation LR holds, then under projection ψ, which drops the machine state, we have the correspondence:*

$$\mathit{pcorres}\ \mathit{LR}\ \psi\ (\mathit{scheduleT})\ (\mathit{runDom;;scheduleT})$$

and thus by compositionality,

$$\mathit{pcorres}\ \mathit{LR}\ (\psi \circ \phi)\ (\mathit{scheduleT})\ (\mathit{runDom;;scheduleC})$$

$$\begin{tikzcd}
\textsc{Downgrading} \arrow[r, dotted] & \text{scheduleS} \arrow[d] & \textsc{Non-Leakage} \arrow[d, dotted, Rightarrow] \\
\textsc{Non-Starvation} \arrow[r, dotted] & \text{scheduleT} \arrow[r, Rightarrow, "\psi{,}\text{LR}"'] \arrow[d, Rightarrow, "\phi{,}\text{LR}"] & \text{runDom ;; scheduleT} \arrow[d, Rightarrow, "\phi{,}\text{LR}"] \\
\text{stepKernel ;; scheduleC} & \text{scheduleC} \arrow[l, Rightarrow, "\phi{,}\text{LR}"] \arrow[r, Rightarrow, "\psi{,}\text{LR}"'] & \text{runDom ;; scheduleC}
\end{tikzcd}$$

Fig. 9. The Complete Refinement Diagram

Therefore, finally, we have all three results: downgrading, non-starvation and non-leakage, on the concrete lottery scheduler composed with the hardware model, as depicted in Figure 9. Here, non-leakage is shown using a double dotted arrow to emphasise that it is only preserved by correspondence, and not by refinement.

4 Conclusions

We have presented a hybrid probabilistic lattice-lottery scheduler, which allows efficient mitigation of the cache channel, while simultaneously guaranteeing non-starvation. Working in pGCL, our system is produced by iterative refinement, supplemented by mechanical proof. This demonstrates that given adequate tool support (namely Isabelle/HOL and our mechanisation of pGCL), refinement-driven development and verification of realistic *probabilistic* systems software is no more difficult than the existing non-probabilistic case. We have shown that our refinement framework is compatible with that of the L4.verified project, and set out the steps necessary to combine this work with a system such as seL4, giving a mechanical proof down to a real system of probabilistic top-level properties. Above all, we argue that verifying probabilistic security properties on realistic systems software is entirely feasible with current technology.

5 Ongoing & Future Work

We have established non-starvation in Lemma 3 as a property of finite traces (of length at least 8). While weaker than this, it would be nice to derive the standard formulation of non-starvation: that any given domain will eventually be scheduled, or $\forall d.\ \Diamond(\text{current\_domain} = d)$ in the syntax of a boolean modal logic. In our case, of course, the result must necessarily be probabilistic: that it is 'almost certain' that any domain is eventually scheduled. We have already partially mechanised the quantitative temporal logic, qTL, of Morgan and McIver

[MM99], which allows us to express this result as $\forall d.\ \Diamond(\text{current\_domain} = d) = \mathbf{1}$, with boolean predicates generalised to real-valued expectations, as for pGCL. We have so far managed to feed our unmodified pGCL results into qTL, and anticipate that these results will appear in a forthcoming work.

Progress on the assumptions of Lemma 7, required for the connection to seL4, is ongoing. In a separate work, currently under submission, our colleagues Daum & Billing have shown that the seL4 state relation is indeed projective, satisfying the implicit assumption. Formally proving the explicit assumptions (lottery relation and current domain preservation) presents no theoretical challenges, simply requiring a large but trivial proof. Integrating the projectivity result should be similarly straightforward. The more interesting question is what form that the final top-level statement should take to cleanly integrate the probabilistic and classical properties of seL4, and is the subject of ongoing research.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

The author is also indebted to the anonymous reviewers for their insightful and constructive feedback, and to Toby Murray and Matthias Daum for reviewing draft copies of this paper.

References

BBCL12. Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient os isolation in an idealized model of virtualization. In *25th Comp. Security Foundations WS*, pages 186–197, 2012. 2

Ber04. D.J. Bernstein. Cache-timing attacks on AES, 2004. 2

CKS08. David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182, Montreal, Canada, Aug 2008. Springer-Verlag. 9, 12

CM07. Han Chen and Pasquale Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS '07, pages 31–40, New York, NY, USA, 2007. ACM. 2

Coc12. David Cock. Verifying probabilistic correctness in isabelle with pGCL. In *Systems Software Verification, Sydney, Australia, pp. 10, November, 2012.*, 11 2012. 1, 2, 11

Den76. Dorothy. E. Denning. A lattice model of secure information flow. *CACM*, 19:236–242, 1976. 2, 4

Dij75. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, Aug 1975. 3

DoD86. US Department of Defence. *Trusted Computer System Evaluation Criteria*, 1986. DoD 5200.28-STD. 2

FS03. Colin Fidge and Carron Shankland. But what if i don't want to wait forever? *Formal Aspects of Computing*, 14:281–294, 2003. 1

GKV11. Xun Gong, N. Kiyavash, and P. Venkitasubramaniam. Information theoretic analysis of side channel information leakage in FCFS schedulers. In *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, pages 1255 –1259, Aug 2011. 2

HMM05. Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. *Theoretical Computer Science*, 346(1):96 – 112, 2005. 1

HN12. Marieke Huisman and Tri Minh Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In *Proceedings of the 2011 international conference on Formal Verification of Object-Oriented Software*, FoVeOOS'11, pages 178–195, Berlin, Heidelberg, 2012. Springer-Verlag. 2

Hu92. W.M. Hu. Lattice scheduling and covert channels. In *IEEE Symp. Security & Privacy*, pages 52–61, 1992. 2, 5

KEH$^+$09. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM. 1, 2

KZB$^+$91. P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A retrospective on the VAX VMM security kernel. *Trans. Softw. Engin.*, 17(11):1147–1165, Nov 1991. 2

MM99. Carroll Morgan and A. K. Mciver. An expectation-based model for probabilistic temporal logic. *Logic Journal of the IGPL*, 7:779–804, 1999. 16

MM04. Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer, 2004. 1, 2, 3

MM12. Daniel Matichuk and Toby Murray. Extensible specifications for automatic re-use of specifications and proofs. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 333–341, Oct 2012. 12

MMB$^+$12. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 126–142. Springer-Verlag, Dec 2012. 2

Per05. Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005. 2

vO04. David von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In Pierangela Samarati, Peter Ryan, Dieter Gollmann, and Refik Molva, editors, *9th ESORICS*, volume 3193 of *LNCS*, pages 225–243, 2004. 14

WW94. Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *1st OSDI*, pages 1–11, Monterey, CA, USA, Nov 1994. 2