Towards a Comprehensive Model of Isolation for Mitigating Illicit Channels*

Kevin Falzon[1] and Eric Bodden[2]

[1]Technische Universität Darmstadt
[2]Universität Paderborn & Fraunhofer IEM
kevin.falzon@ec-spride.de, eric.bodden@uni-paderborn.de

Abstract. The increased sharing of computational resources elevates the risk of side channels and covert channels, where an entity's security is affected by the entities with which it is co-located. This introduces a strong demand for mechanisms that can effectively isolate individual computations. Such mechanisms should be efficient, allowing resource utilisation to be maximised despite isolation.
In this work, we develop a model for uniformly describing isolation, co-location and containment relationships between entities at multiple levels of a computer's architecture and at different granularities. In particular, we examine the formulation of constraints on co-location and placement using partial specifications, as well as the cost of maintaining isolation guarantees on dynamic systems. We apply the model to a number of established attacks and mitigations.

1 Introduction

Side and *covert channels* (collectively, *illicit channels*) are fundamentally the result of imperfect isolation, where information regarding an entity's internal and potentially secret state leaks to an observer through an unregulated interface.

The position of two entities relative to each other determines the type of illicit channel that can be formed between them. For example, two processes sharing a physical core may form a channel over the memory subsystem, whereas processes on separate machines may form a network-based illicit channel. This leads to the notion of *co-location*, where entities are said to be co-located within a medium if they can leverage it to form illicit channels.

Co-location is often considered at the virtual-machine level in the context of cloud computing, yet the notion of co-location as a precursor to illicit channels extends to multiple levels of a computer's architecture. Isolation at the virtualisation level is limited in that it is *coarse-grained*, whereas one often only has to isolate parts of a virtual machine. In addition, the mechanisms used to build illicit channels can operate at a fine granularity, and their effects may not be correctly or precisely encompassed by a coarse-grained model.

* This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE. At the time this research was conducted, Eric Bodden was at Fraunhofer SIT and Technische Universität Darmstadt.

In this work, we develop a holistic model of locality that considers multiple levels of an architecture at varying granularities. This offers numerous advantages over a single-level model. Finer granularity can lead to an improvement in hardware utilisation, as fewer resources are committed to providing isolation guarantees. The ability to compare the cost of maintaining different isolation levels also allows resource allocation to be optimised dynamically, further improving utilisation. Apart from being quantifiable, the cost of maintaining isolations must be attributable, particularly in the case of cloud computing.

As scheduling and placement play a central role in co-location, the locality model must also be able to describe both temporal as well as spatial aspects of a system. Another aspect addressed by the model is the notion of *partial specification*, where entities within a system (such as tenants on a cloud) only have a partial view of their environment, and must be able to delegate their isolation requirements to external entities.

In summary, this work:

- reconciles different aspects of isolation and co-location into a unified model that can describe both temporal and spatial properties of a system at multiple architectural levels,
- examines the different levels and confinement types, and their use in defining partial specifications and isolation requirements,
- provides an operational model for migration and cost estimation, allowing different system configurations and real-world architectures to be compared and optimised, and
- demonstrates various applications of the model in analysing illicit channels.

2 Confinements

A modern computer architecture consists of a multitude of isolated environments, which are themselves contained within isolations, forming a hierarchy. The following section introduces the notions of confinement and containment.

2.1 Isolation and Containment

A computer architecture comprises a number of logical and physical *confinements*. For example, processes execute within the confines of a CPU. Confinements must themselves exist within an environment, which leads to a notion of hierarchical *containment*. Extending the previous example, multiple CPUs may be confined by a single machine, which can itself form part of a network.

Definition 1 (Confinement). *A* confinement *(equivalently,* isolation *or* locality*) denotes a boundary within which a number of sub-confinements exist. A confinement of type* $\boldsymbol{\Gamma}$ *with a name* N *and capability set* $\mathtt{C}$ *containing a set of sub-confinements* Sb *is denoted as* $\boldsymbol{\Gamma}{:}N(\mathtt{C})\,[\textsc{Sb}]$.

A confinement's name is typically dictated by its type, and serves to identify it from amongst its siblings. Capabilities are used to limit how confinements can interact and modify each other, as will be seen in Section 3. The capability set can be omitted when it is empty.

Illicit channels exploit the fact that certain confinements are imperfect, and do not keep their sub-confinements completely isolated from each other. Thus, confinements can be seen as introducing *locality*, where confinements that should theoretically be disjoint are connected through a channel exploiting some characteristic of their parent confinement.

Definition 2 (Containment and Co-Location). *A confinement* X *is* contained *within a confinement* $\boldsymbol{\Gamma}{:}\mathrm{D}(\mathtt{C})\,[\textsc{Sb}]$ *if* $\mathrm{X} \in \textsc{Sb}$. *This is denoted as* $\mathrm{X} \in \mathrm{D}$. X *is said to be* co-located *with* Y *through* D*, written as* $\mathrm{X} \overset{\mathrm{D}}{\leftrightarrow} \mathrm{Y}$*, if* $\mathrm{X} \in \mathrm{D} \wedge \mathrm{Y} \in \mathrm{D}$.

The state leaked within a confinement can potentially be observed both by its direct sub-confinements as well as their members. This gives rise to the notion of *nested containment*, where $\mathrm{X} \in^{+} \mathrm{D} \overset{\text{def}}{=} \mathrm{X} \in \mathrm{D} \vee \exists \mathrm{D}' \in \mathrm{D}.\ \mathrm{X} \in^{+} \mathrm{D}'$, and *nested co-location*, where $\mathrm{X} \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{Y} \overset{\text{def}}{=} \mathrm{X} \in^{+} \mathrm{D} \wedge \mathrm{Y} \in^{+} \mathrm{D}$.

Example 1 (Parallel Execution). Consider a CPU package with two cores (**C**) sharing an **L3** cache, each of which employs *simultaneous multithreading* (SMT) to expose two hardware threads sharing an **L1** and **L2** cache. This can be modelled as:

$$\mathrm{CPU} \overset{\text{def}}{=} \mathbf{L3}{:}0\,[\mathbf{L2}{:}0\,[\mathbf{L1}{:}0\,[\mathbf{C}{:}0\,[]\,,\mathbf{C}{:}1\,[]]]\,,\mathbf{L2}{:}1\,[\mathbf{L1}{:}0\,[\mathbf{C}{:}2\,[]\,,\mathbf{C}{:}3\,[]]]]$$

Two processes X and Y can be susceptible to an attack via **L1** cache [28] if $\exists \mathbf{L1}{:}\mathrm{L} \in^{+} \mathrm{CPU}.\ \mathrm{X} \overset{\mathrm{L}}{\Leftrightarrow} \mathrm{Y}$, or via **L3** cache [36] if $\exists \mathbf{L3}{:}\mathrm{L} \in^{+} \mathrm{CPU}.\ \mathrm{X} \overset{\mathrm{L}}{\Leftrightarrow} \mathrm{Y}$. The latter will hold whenever the processes execute simultaneously. □

Note that proximity, or the depth at which two processes are co-located within the model, does not necessarily correlate with an illicit channel's bandwidth. That is, while processes that are closer to each other can generally communicate at a faster rate or perform more events per unit time than others that are further away (for example, processes sharing a cache interact with their shared resource at a higher frequency than if they were co-located through a network), not every interaction carries information relevant to the channel.

2.2 Types of Isolation

Illicit channels occur either at the software or hardware level [24], the former being a product of the algorithms used, while the latter emerge from the characteristics of a system's hardware. When considering hardware-based channels, an additional distinction between *soft* and *hard isolation* can be made [32]. Hard isolation implies that co-locations are broken by using distinct physical hardware locations, whereas soft isolation simulates distinct hardware locations by

Hard Isolation			Soft Isolation		
Type	Description	Can Contain	Type	Description	Can Contain
Net	Network	**Net**, **M**	**VM**	Virtual machine	**VC**, **OS**
M	Machine	**L3**, **OS**	**VC**	Virtual CPU	**VC**, $\mathbf{P_E}$, **Con**, **VM**
L3	L3 Cache	**L2**	**Con**	Container	**P**
L2	L2 Cache	**L1**	$\mathbf{P_E}$	Control group	**Con**, **P**
L1	L1 Cache	**C**	**P**	Process	-
C	Physical core	**VC**, $\mathbf{P_E}$, **Con**, **VM**	**OS**	Operating Sys.	$\mathbf{P_E}$, **Con**, **VM**

Table 1: Types of soft and hard isolation, and their typical containments.

arbitrating access to resources, hiding their characteristics. Soft isolation is guaranteed with respect to a defined attribute. For example, a timing channel can be closed by masking the timing characteristics of caches [28], yet such a mitigation may not effectively address other potential illicit cache-level channels. Hard isolation is comprehensive, but is limited by capacity [27].

Table 1 lists the soft and hard isolation types with which this work is primarily concerned. Other granularities and isolation types can also be modelled. For example, as will be seen in Section 5.2, monolithic caches can be decomposed into cache sets. The confinement model places no restrictions on the types of sub-confinements, which allows the description of partial specifications and incomplete system hierarchies. In practice, it follows that certain containment patterns do not occur, and that the presence of certain confinements imply the existence of a parent of a specific type. For example, a virtual CPU (**VC**) confinement would imply the existence of a **VM** to which it belongs.

Hard isolations are passive elements of a system. Conversely, certain soft isolations must be upheld through an active and ongoing process, or through a change in policy. For example, early implementations of x86 virtualisation incurred a constant overhead through dynamic binary rewriting [3], which has nowadays been significantly reduced via hardware-assisted virtualisation. Similarly, software-based approaches to securing AES added overheads [28] that were eliminated through their implementation as a special hardware-level confinement [23].

3 Managing Isolations

The core operations for modifying a containment hierarchy are confinement *creation*, *destruction* and *migration*. The latter is modelled as moving an isolation from one containment to another. The implementation of these operations varies based on the isolations involved, and may require a series of compound actions that incur multiple changes at different parts of the hierarchy. Changes to the hierarchy are effected by *agent* processes.

Definition 3 (Agent). *An agent is a confinement* $\mathbf{A}{:}N(\mathtt{C}^{Ag})^{\rightarrow}_{\mathrm{T}}[\mathrm{Q}]$*, where* $\mathbf{A}$ *denotes an agent type,* N *is the agent's name,* $\mathtt{C}^{Ag}$ *is its capability set,* T *is a set of confinements visible to the agent,* $\rightarrow \subseteq \mathrm{T} \times \mathrm{T}$ *is a mapping defining legal containments, and* Q *is a queue of idle confinements.*

$$\text{L-Sc}\ \frac{\mathbf{A}{:}\mathrm{Ag}(\mathtt{C}^{\mathrm{Ag}})^{\rightarrow}_{\mathrm{T}}\,[\mathrm{Q}\cup\{\mathrm{X}\}] \qquad \boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,[\mathrm{S_B}] \qquad \mathrm{Ag}\equiv \mathrm{X}\curvearrowright \mathrm{N}.\mathrm{Ag'} \qquad \mathtt{C}^{\mathrm{Ag}}\Cap\mathtt{C} \qquad \mathtt{C}^{\mathrm{Ag}}\Cap\mathtt{cap(X)} \qquad (\mathrm{X},\mathrm{N})\in\rightarrow}{\mathbf{A}{:}\mathrm{Ag'}(\mathtt{C}^{\mathrm{Ag}})^{\rightarrow}_{\mathrm{T}}\,[\mathrm{Q}] \qquad \boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,[\mathrm{S_B}\cup\{\mathrm{X}\}]}$$

$$\text{L-Ds}\ \frac{\mathbf{A}{:}\mathrm{Ag}(\mathtt{C}^{\mathrm{Ag}})^{\rightarrow}_{\mathrm{T}}\,[\mathrm{Q}] \qquad \boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,[\mathrm{S_B}\cup\{\mathrm{X}\}] \qquad \mathrm{Ag}\equiv \mathrm{X}\curvearrowright \mathrm{Ag}.\mathrm{Ag'} \qquad \mathtt{C}^{\mathrm{Ag}}\Cap\mathtt{C} \qquad \mathtt{C}^{\mathrm{Ag}}\Cap\mathtt{cap(X)}}{\mathbf{A}{:}\mathrm{Ag'}(\mathtt{C}^{\mathrm{Ag}})^{\rightarrow}_{\mathrm{T}}\,[\mathrm{Q}\cup\{\mathrm{X}\}] \qquad \boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,[\mathrm{S_B}]}$$

Fig. 1: Local migration rules.

Agents represent scheduling components that manage confinements. For example, an operating system's process scheduler can be modelled as an agent confinement that regulates movements between an idle queue and core confinements. Agents can be embedded at any part of the hierarchy. For example, a network domain controller can be modelled as an agent embedded within the network's hardware layer. Agents can move (or *migrate*) confinements using *local* and *global* scheduling operations, described in the following sections.

3.1 Local Scheduling

Local scheduling moves a confinement between an agent's idle queue and a target confinement via the *local-schedule* (L-Sc) and *local-deschedule* (L-Ds) rules, the general forms of which are defined in Figure 1, where $\mathtt{cap}(\boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,[\mathrm{S_B}]) \rightarrow \mathtt{C}$, and $A \Cap B \stackrel{\text{def}}{=} A \cap B \neq \emptyset$. For a local-schedule operation, the agent process Ag issues a migration operation ($\mathrm{X}\curvearrowright\mathrm{N}$) that moves X from its idle queue to the target locality N, provided that the allocation is permitted (as defined by $\rightarrow$), and that the agent holds the appropriate capabilities. Descheduling is similar, but returns the locality from a target confinement to the agent's idle queue.

An agent must share a capability with the target confinement, as well as the confinement being moved. Capability checking is modelled as an abstract operation (an intersection between capability sets), as the concrete implementation varies by confinement. For example, destroying a process requires the agent to have the process owner's user rights. Similarly, virtual machines can only be modified by agents holding the appropriate rights, which can be granted through a number of authorisation mechanisms, such as user groups, passwords, or *polkit* [2] policies.

Mutability is not modelled as an intrinsic property of a confinement, rather it is determined by the availability of its capability to agents. While hardware confinements such as caches are not typically disabled at runtime, an agent may want to delete their representation from the model if it is certain that the threat of a channel through that confinement has been neutralised.

Example 2 (Round Robin Scheduler). Consider the CPU hierarchy defined in Example 1. An agent implementing a simple round-robin scheduler with a shared run queue can be defined as $\mathbf{A}{:}\mathrm{rr}(\mathtt{C}^{\mathrm{Ag}})^{\rightarrow}_{\mathrm{T}}\,[\mathrm{Q}]$, where Q contains an ordered

list of processes, and $\rightarrow$ defines the allowed mapping of processes to physical cores. The default behaviour is to map all processes to all available cores, giving $\rightarrow \stackrel{\text{def}}{=} \{(\mathrm{X}, \mathrm{Y}) \mid \mathrm{X} \in \mathrm{Q}, \mathrm{Y} = \mathbf{C}{:}\mathrm{N}(\mathtt{C})\,[\textsc{Sb}]\,, \mathrm{Y} \in^{+} \mathrm{CPU}\}$. Given that $\uparrow(\mathrm{X}) \stackrel{\text{def}}{=} \{\mathrm{Y} \mid (\mathrm{X}, \mathrm{Y}) \in \rightarrow\}$, the scheduler can be defined as a CSP-like process as follows:

$$\textsc{rr}_{\mathrm{Q}}([\mathrm{P} \mid \textsc{Ps}]\,, \mathrm{C_A}, \mathrm{C_F}) \equiv \mathop{\sqcap}^{\mathbf{C}:\mathrm{X} \in \uparrow(\mathrm{P}) \cap \mathrm{C_F}} \mathrm{P} \curvearrowright \mathrm{X}.\textsc{rr}_{\mathrm{Q}}(\textsc{Ps}, \mathrm{C_A}, \mathrm{C_F} \setminus \{\mathrm{X}\}) \sqcap \mathop{\sqcap}^{\mathbf{P}:\mathrm{P'} \in \mathbf{C}:\mathrm{Y} \in \mathrm{C_A}} \mathrm{P'} \curvearrowright \mathrm{rr}.\textsc{rr}_{\mathrm{Q}}(\textsc{Ps} \mid [\mathrm{P'}]\,, \mathrm{C_A}, \mathrm{C_F} \cup \{\mathrm{Y}\})$$

where $\mathrm{C_A}$ is the set of all cores being managed by the scheduler, $[\mathrm{P} \mid \textsc{Ps}]$ is an ordered list of processes with P as its head and Ps as its tail, and $\mathrm{C_F}$ is the set of idle cores. The process would thus be initialised as $\textsc{rr}_{\mathrm{Q}}(\mathrm{Q}, \textsc{Cs}, \textsc{Cs})$, where $\textsc{Cs} = \{\mathrm{X} \mid \mathbf{C}{:}\mathrm{X} \in^{+} \mathrm{CPU}\}$.

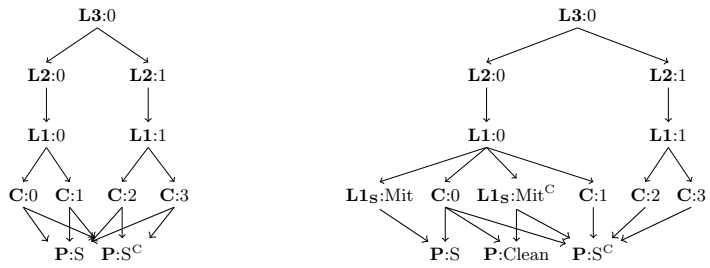
Next, consider the scenario where a security-sensitive process S is added to Q. If the process is susceptible to a cache-level synchronous attack [28], then one must avoid co-locating S with other processes during its execution. As formulated, the scheduler will execute processes in the order specified by the idle queue, but processes can be descheduled pre-emptively at will, meaning that every other process can potentially execute in parallel with S. Forcing processes to execute for an equal and fixed time-slice will cause S to potentially be co-scheduled with the $|\textsc{Cs}| - 1$ processes that appear before and after it in the idle queue. Finally, changing $\rightarrow$ to ensure that S always executes by itself will prevent spatial co-location, at the cost of underutilised hardware. As a compromise, $\rightarrow$ can be varied dynamically, with the number of processes that can share cores growing proportionately to the time elapsed since the last scheduling of S. □

Configurations Reasoning about temporal locality requires the ability to describe how a model evolves from one *configuration* to the next, where a configuration is defined as a set of confinements. The evolution of a configuration is determined by the agents it contains. The presence of multiple agent and varying scheduling policies mean that, in general, there is more than one legal next configuration. This leads to the notion of a $\mathtt{next}(\mathcal{C})$ function, which returns the set of possible configurations that can be reached from a configuration $\mathcal{C}$ through a single application of a local schedule or deschedule operation (Figure 1). This is extended to the iterated next configuration function $\mathtt{next}^{n}(\mathcal{C})$, which returns the set of configurations reachable from $\mathcal{C}$ in n steps, defined as follows:

$$\begin{aligned} \mathtt{next}^{0}(\mathcal{C}) &\stackrel{\text{def}}{=} \{\mathcal{C}\} \\ \mathtt{next}^{n}(\mathcal{C}) &\stackrel{\text{def}}{=} \left\{\mathtt{next}^{n-1}(\mathcal{C}') \mid \mathcal{C}' \in \mathtt{next}(\mathcal{C})\right\} \end{aligned}$$

Finally, the configuration combination operator $\mathtt{next}^{n}_{\cup}(\mathcal{C})$ is defined as:

$$\begin{aligned} \mathtt{next}^{n}_{\cup}(\mathcal{C}) &\stackrel{\text{def}}{=} \left\{\boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,[\textsc{Sb}] \mid \boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,\left[\textsc{Sb}'\right] \in^{+} \textsc{Cfs}\right\} \\ \text{where } \textsc{Sb} &\stackrel{\text{def}}{=} \bigcup \left\{\textsc{Sb}'' \mid \boldsymbol{\Gamma}{:}\mathrm{N}(\mathtt{C})\,\left[\textsc{Sb}''\right] \in^{+} \textsc{Cfs}\right\} \\ \text{and } \textsc{Cfs} &\stackrel{\text{def}}{=} \bigcup \bigcup_{0 \leq i \leq n} \mathtt{next}^{i}(\mathcal{C}) \end{aligned}$$



(a) No isolation at **L1** (b) Creation of $\mathbf{L1_S}$ via soft isolation

Fig. 2: Cache-level co-location and mitigation via soft isolation, with arrows denoting containment.

This effectively performs a union of every possible configuration reachable within n local scheduling operations, including intermediate configurations. The result is a graph that shows every containment combination attainable in a set sequence of steps. This can be used to represent a system's temporal behaviour as a static spatial graph. A related graph can be achieved by combining each agent's containment mapping, giving a graph of potential containments, yet this would over-approximate containments, as a scheduling policy may opt to only use a subset of mappings available to it. To simplify the operation, it is assumed that confinements can be uniquely identified by their name. Otherwise, an additional preprocessing step can be introduced.

Example 3 (Round Robin Scheduler, revisited). In Example 2, co-location with a security-sensitive process S was only considered with respect to a single moment in time, yet an access-based cache-level side channel's effects persist beyond a process' execution [28] until the security-sensitive memory blocks have been flushed. Thus, simply disabling co-scheduling during S's execution would not be sufficient to break the channel reliably.

The duration of the residual effects of caches is independent of real time, and is determined by cache evictions. For the pre-emptive round robin scheduler described earlier, the position of S in the idle queue relative to an attacker process will generally affect the illicit channel's quality, as the probability that S's sensitive cache blocks become clobbered increases with the number of processes that execute in the interim. If cache eviction patterns and process quanta are irregular, or if a fully pre-emptive scheduling policy is used, then each core in $\texttt{next}^{\infty}_{\cup}(\{\texttt{CPU}\})$ will contain Q.

Residual effects can be explicitly removed through a *cache-cleaning process* [37] that invalidates cache blocks, masking their timing variations. The process (henceforth referred to as Clean) must execute after each de-scheduling of S. Any process using the same cache that executes concurrently with S can potentially infer the timing state up to the point of Clean's completion. Thus, one must place an additional restriction on concurrent execution. If these two conditions can be guaranteed as invariants, then the cache has effectively been

partitioned into two sub-confinements of type $\mathbf{L1_S}$ (a soft-isolated **L1**), transforming the hierarchy described in Figure 2a to that illustrated by Figure 2b (for simplicity, S is pinned to **C**:0). The partitioning serves to isolate the process S from the other processes S^C (the latter being the complement of S). Note that the processes remain co-located within **L1**:0, as they are still ultimately sharing hardware locality. If the soft isolation is deemed perfect, then the **L1** confinement can be destroyed. Removing CLEAN would lead to the partitions being destroyed, and the **L1**:0 confinement being recreated. □

3.2 Global Scheduling

Local scheduling limits an agent in its procurement of isolation, as it can only make use of confinements under its direct control. An agent can be supported by additional agents external to its scope in two ways. First, an external agent can provide isolation guarantees on the parents of confinements that are being managed by an agent. For example, if an agent running within a virtual machine requires a hard isolation guarantee that a process executes alone on a core, then it must query an agent in the underlying hypervisor's scope to ensure that the **VC** confinement is placed in a dedicated **C** confinement. Secondly, an external agent serves to extend the pool of available confinements, allowing confinements to be *migrated* to a different scope. Building on the previous example, the hypervisor agent can migrate **VC** confinements amongst cores until an isolated core is provisioned. If the agent finds that all of its resources are committed, it can query additional external agents for isolations on different machines.

Migrating from one agent's scope to the next leads to the notion of *global* scheduling. Broadly, global scheduling involves two steps, namely (a) identifying a target agent which can procure the required level of isolation, and (b) migrating the confinements required to achieve isolation. The following section details how these tasks are performed.

Scopes and Renaming In general, an agent will only have a partial view of a system. Consider the containment hierarchy illustrated in Figure 3, which represents a minimal model $\texttt{next}^{\infty}_{\cup}(\{\texttt{Machine}\})$ of a two-core compute node over which two tenant virtual machines are executing. In this model, each **VM** has an agent TA0 and TA1 running within it, whereas the infrastructure provider has an agent HYP running on the base system. The virtualisation confinement prevents a tenant's agents from enumerating the parent's confinements through standard operating-system interfaces. In addition, even if the details of the parent's confinements can somehow be inferred (for instance, through illicit channels), the tenant's agents would not have the necessary capabilities to alter them. For instance, mere knowledge of the existence of additional co-located tenants would not automatically grant a tenant's agent control over them.

While a tenant agent may be unaware of its parent environment's confinements, the converse does not hold. Containment relationships crossing a boundary still require that the sub-confinement be exposed to its parent. For example,

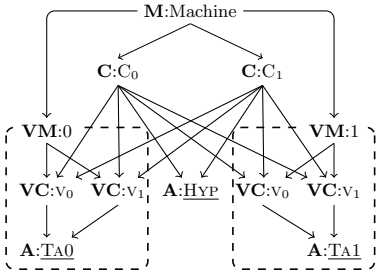


Fig. 3: Partial model showing agent scopes and boundaries.

while tenants in Figure 3 might not be aware of the number of physical cores on the machine, the hypervisor must have a handle to the tenants' **VC** structures in order to manage their core pinnings[1].

The agent's position within a hierarchy also determines its view of a confinement. For example, **VC** confinements managed by HYP are seen as **C**s by processes within the tenants' **VM**s. Thus, isolation requests across scopes must be accompanied by a mechanism to rename confinements. Confinement renaming is not always straightforward, such as in the case of processes, which have a significant amount of state dispersed within their parent **OS** confinement that has to be translated on migration. For instance, a process' PID may have to be changed on migrating to a new **OS** environment [1], which would alter its internal system view. A common workaround is to employ *namespace* mechanisms, commonly in conjunction with containers [2], to encapsulate structures such as PIDs and network interfaces and separate them from the common namespace of the base **OS**. This ensures that a migrated process' structures remain internally consistent.

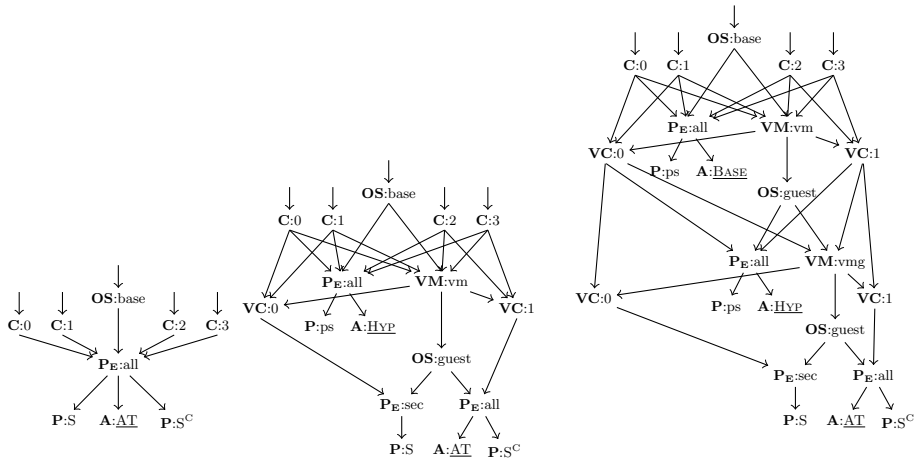
Isolation Constraints A consequence of agent scoping is that changes to external confinements need to be delegated to an agent. In addition, changes cannot refer to specific external confinements, both due to scoping and security reasons.

Consider a simple isolation condition $\texttt{isol}_{\mathbf{P}}()$, which checks whether a process exists by itself in a **C** environment, defined as follows:

$$\texttt{isol}_{\mathbf{P}}(\mathbf{P}{:}\mathrm{X}, \mathbf{C}{:}\mathrm{D}) \stackrel{\text{def}}{=} \neg\exists \mathbf{P}{:}\mathrm{Y} \in^{+} \mathrm{D}.\ \mathrm{X} \neq \mathrm{Y}$$

The evaluation of $\texttt{isol}_{\mathbf{P}}()$ varies based on the underlying system assumptions. Figure 4a illustrates a partial $\texttt{next}^{\infty}_{\cup}()$ graph of the CPU hierarchy from the perspective of an agent running within a virtualised environment (or equivalently, a non-virtualised, bare-metal environment). In this case, for D quantified over all visible confinements, $\texttt{isol}_{\mathbf{P}}(\mathrm{S}, \mathrm{D})$ will fail (return false) due to processes sharing a process control group ALL. To comply with the isolation requirement,

[1] *Introspection* [19] can be used to characterise sub-confinements of a **VM**.



(a) Bare metal (b) Virtualisation (c) Nested virtualisation

Fig. 4: Environment nesting and indirection.

processes must be partitioned into two process groups contained in disjoint sets of cores.

Subsequently adding a virtualisation layer produces the containment tree shown in Figure 4b. If multiple **VM**s execute in parallel, then the $\texttt{isol}_{\mathbf{P}}()$ predicate may fail. Thus, the hypervisor agent HYP must be queried to ensure that cores are allocated exclusively to the **VC** containing S. Given that $\mathrm{X} \mapsto \mathrm{X'}$ renames a confinement X into a locally-scoped confinement X', a second isolation condition $\texttt{isol}_{\mathbf{VC}}()$ is defined and sent to HYP, where:

$$\texttt{isol}_{\mathbf{VC}}(\mathbf{C}{:}\mathrm{X}, \mathbf{C}{:}\mathrm{D}) \stackrel{\text{def}}{=} \mathrm{X} \mapsto \mathbf{VC}{:}\mathrm{X'} \wedge \neg\exists \mathbf{VC}{:}\mathrm{Y} \in \mathrm{D}.\ \mathrm{X'} \neq \mathrm{Y} \wedge \mathrm{X'} \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{Y}$$

In this case, D is a free variable which must be bound by HYP. As described in the previous section, the **C** confinement must be renamed to a structure visible to HYP, namely X'. As virtualisation and containments can potentially be nested to an arbitrary depth (Figure 4c), the $\texttt{isol}_{\mathbf{VC}}()$ isolation request must be pushed upwards in the hierarchy, until the base confinement is reached. This ensures that the intermediate levels of indirection do not lead to co-locations. While the use of nested virtual machines might not currently be widespread, the growing adoption of *containers* will increase the occurrence of such topologies.

Finally, an isolation request may place additional constraints on co-location. For example, tenants may request that **VM**s can only be co-located on a machine if they are all owned by the same tenant. Given that X is the tenant's machine from its scope, and D is the base machine, this can be expressed as:

$$\begin{aligned}\texttt{isol}_{\mathbf{VM}}(\mathbf{M}{:}\mathrm{X}, \mathbf{M}{:}\mathrm{D}) \stackrel{\text{def}}{=}\ & \mathrm{X} \mapsto \mathbf{VM}{:}\mathrm{X'} \wedge \neg\exists \mathbf{VM}{:}\mathrm{Y} \in \mathrm{D}.\ \mathrm{X'} \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{Y} \wedge \\ & \texttt{tenant(X')} \neq \texttt{tenant(Y)}\end{aligned}$$

$$\text{G-Sc}\ \frac{\begin{array}{c}\mathbf{A}{:}\mathrm{Src}(\mathtt{C}^{\mathrm{Src}})^{\rightarrow_{\mathrm{Src}}}_{\mathrm{TS}}\,[\mathrm{Q}_{\mathrm{Src}}\cup\{\mathrm{X}\}] \qquad \mathbf{A}{:}\mathrm{Dst}(\mathtt{C}^{\mathrm{Dst}})^{\rightarrow_{\mathrm{Dst}}}_{\mathrm{TD}}\,[\mathrm{Q}_{\mathrm{Dst}}] \\ \mathrm{Dst}\in\mathrm{TS} \qquad \mathrm{Src}\equiv \mathrm{X}\overset{\mathtt{isol}()}{\curvearrowright}\mathrm{Dst.Src'} \qquad \mathrm{D}\in^{+}\{\mathrm{D'}\mid \mathrm{D'}\in\mathrm{TD}\} \\ \mathtt{C}^{\mathrm{Src}}\Cap\mathtt{C}^{\mathrm{Dst}} \qquad \mathtt{C}^{\mathrm{Src}}\Cap\mathtt{cap(X)} \qquad \mathtt{C}^{\mathrm{Dst}}\Cap\mathtt{cap(D)} \qquad \mathtt{isol}(\mathrm{X},\mathrm{D})\end{array}}{\begin{array}{c}\mathbf{A}{:}\mathrm{Src'}(\mathtt{C}^{\mathrm{Src}})^{\rightarrow_{\mathrm{Src}'}}_{\mathrm{TS}\setminus\{\mathrm{X}\}}\,[\mathrm{Q}_{\mathrm{Src}}] \qquad \rightarrow_{\mathrm{Src}'}\equiv\rightarrow_{\mathrm{Src}}\setminus\{(\mathrm{X},\mathrm{Y})\mid(\mathrm{X},\mathrm{Y})\in\rightarrow_{\mathrm{Src}}\} \\ \mathbf{A}{:}\mathrm{Dst}(\mathtt{C}^{\mathrm{Dst}})^{\rightarrow_{\mathrm{Dst}'}}_{\mathrm{TD}\cup\{\mathrm{X}\}}\,[\mathrm{Q}_{\mathrm{Dst}}\cup\{\mathrm{X}\}] \qquad \rightarrow_{\mathrm{Dst}'}\equiv\rightarrow_{\mathrm{Dst}}\cup\{(\mathrm{X},\mathrm{D})\}\end{array}}$$

Fig. 5: Global migration rule.

Global Migration Global migration changes a confinement's place within a hierarchy by placing it under another agent's control and modifying its mapping rules. Consequently, migration changes a system's infinite configuration $\mathtt{next}^{\infty}_{\cup}\mathtt{()}$.

Figure 5 defines the general rule for migrating a confinement X globally. The source agent Src initiates a migration request to a destination agent Dst with an isolation criterion `isol`(), which Dst attempts to match against its known and controllable confinements. Following the migration, each agent updates its containment mapping rules, with Src removing the associated mappings, and Dst adding a rule for X's allowed containments. The source and destination agents can be the same, allowing confinements to be created, destroyed, or simply remapped. The rule can be modified so that X is assigned multiple parent confinements at its destination. This allows a confinement to maintain the same number of allocated resources across migrations.

A target agent must be within the source agent's scope. Logics such as the *cloud calculus* [25] make use of a `parent()` operator, which returns a handle to a confinement's parent. Agent discovery varies depending on the confinement level being considered, but it generally involves mapping an agent's identifier to its actual address. Discovery mechanisms include broadcasts, distributed keystores and centralised repositories. Each method has its own drawbacks in query time and consistency. Depending on the frequency of agent discovery operations and actual migrations, one may also consider propagating notifications of topology changes down a hierarchy following a migration, with lower-level agents subscribing to their parent agents and receiving notifications whenever their scopes have been altered.

4 Cost Functions and Metrics

Different configurations vary in the degree of isolation that they offer and the cost required to maintain them. The ability to quantify these factors is essential to the process of provisioning isolation, as it allows configurations to be compared, and enables allocations to be optimised. When comparing system hierarchies containing long-lived processes, one must consider the cost of maintaining a configuration over time, rather than simply comparing a system's instantaneous configuration. Thus, metrics and costs should be evaluated over the $\mathtt{next}^{\infty}_{\cup}\mathtt{()}$ of a given hierarchy.

4.1 Metrics

Several metrics and notions of cost can be defined, including, but not limited to:

Utilisation measures the aggregate usage of a system's capacity. Certain confinements can only contain a number of sub-confinements before the system's overall performance begins to drop. For example, consider the scenario of a process scheduler allocating processes to cores evenly. Given that `load(Y)` returns the average CPU utilisation of a process Y expressed as a fraction, one can measure CPU utilisation for a hierarchy $\mathcal{C}$ as a dimensionless unit as follows:

$$\begin{aligned}\texttt{util}(\mathcal{C}) &= \sum_{\mathbf{C}:\mathrm{X} \in^{+} \mathcal{C}} \texttt{min}\left(\sum_{\mathbf{P}:\mathrm{Y} \in \mathrm{X}} \frac{\texttt{load(Y)}}{|\,\{\mathrm{D} \mid \mathbf{C}:\mathrm{D} \in^{+} \mathcal{C} \wedge \mathrm{Y} \in \mathrm{D}\}\,|}\ ,\ 1.0\right) \\ &\approx \sum_{\mathbf{C}:\mathrm{X} \in^{+} \mathcal{C}} \texttt{min}\left(k \sum_{\mathbf{P}:\mathrm{Y} \in \mathrm{X}} |{\uparrow}(\mathrm{Y})|^{-1}\ ,\ 1.0\right)\end{aligned}$$

The second formula is an approximation that can be computed statically given an average processor usage k and the **P**-to-**C** mapping defined by an agent's $\rightarrow$ structure ($\uparrow()$ is defined in Example 2). The `min` function caps each **C**'s usage value.

Capacity is the number of confinements of a given type in a configuration, while **total capacity** is the total number of confinements in the hierarchy.

Consolidation factor is defined as **capacity**/**utilisation**, and represents the ratio between the system's utilisation and the number of confinements of a given type within a hierarchy.

Pairwise co-locations counts the total number of pairs of co-located confinements in a given hierarchy, and is defined as:

$$\texttt{pairs}(\mathcal{C}) = \frac{1}{2}\,\left|\left\{\langle \mathrm{X}, \mathrm{D}, \mathrm{Y}\rangle \mid \mathrm{X}, \mathrm{Y}, \mathrm{D} \in^{+} \mathcal{C}, \mathrm{X} \neq \mathrm{Y}, \mathrm{X} \overset{\mathrm{D}}{\leftrightarrow} \mathrm{Y}\right\}\right|$$

Containment hierarchies can be topologically sorted, and metrics can be computed by performing a breadth-first search and evaluating each sub-graph, provided that costs are compositional. The evaluation of metrics is complicated by agents' partial system specifications. For example, a tenant can compute `pairs()` within its own **VM**, yet this will only serve as a lower-bound, and would have to be combined with additional information from the parent confinement.

In a cloud scenario, tenants and the cloud provider may attempt to optimise their configurations with respect to different metrics. For example, a tenant will want to compromise between pairwise co-locations and total capacity. Conversely, while a cloud provider will attempt to maximise consolidation so as to maintain a smaller deployment, it has a lower incentive to minimise a tenant's total capacity if it bills its clients on the basis of committed resources.

Example 4 (Comparing architectures). A system's containments can vary across vendors. To illustrate, we examine two different CPUs, namely an Intel i7-4790

(INTEL) with 8 hardware threads using SMT, and a hex-core AMD Phenom II X6 (AMD). Apart from cache exclusivity, the architectures vary in that the former has two hardware threads to each **L1** containment, whereas the latter has per-core **L1** and **L2** caches. This results in the following models:

$$\text{INTEL} \stackrel{\text{def}}{=} \mathbf{L3}{:}0\,[\{\mathbf{L2}{:}\mathrm{i}\,[\mathbf{L1}{:}\mathrm{i}\,[\mathbf{C}{:}\mathrm{i}\,[]\,,\mathbf{C}{:}\mathrm{i}{+}4\,[]]] \mid 0 \le i \le 3\}]$$
$$\text{AMD} \stackrel{\text{def}}{=} \mathbf{L3}{:}0\,[\{\mathbf{L2}{:}\mathrm{i}\,[\mathbf{L1}{:}\mathrm{i}\,[\mathbf{C}{:}\mathrm{i}\,[]]] \mid 0 \le i \le 5\}]$$

Consider the case where processes must never be co-located through **L1** or **L2**. For the INTEL hierarchy, this effectively halves the **C** capacity[2]. Assuming that each system divides P processes amongst its **C**s equally, $\texttt{util(AMD)} = \texttt{min}\,(k_{\text{AMD}}P/6, 6.0)\,,$ and $\texttt{util(INTEL)} = \texttt{min}\,(k_{\text{INTEL}}P/4, 4.0)$. Thus, INTEL's process execution time k_{INTEL} must be two thirds of k_{AMD} in order to have equal utilisation rates. □

4.2 Ongoing and Migration Costs

Configurations offer different security guarantees at different costs. Evaluating costs and metrics on a configuration's $\texttt{next}^{\infty}_{\cup}\texttt{()}$ is a tradeoff between performance and precision, as it avoids recomputing costs after each local migration operation.

Given a static model, a configuration can be progressively modified until it reaches an optimal state with respect to a property of the system. For example, tenants within a cloud have an incentive to use resources efficiently, and cloud providers generally attempt to provide resources to tenants with a minimum of overhead. Thus, if no confinements are created or destroyed by the tenants' agents, a cloud provider can alter the system's configuration incrementally until it reaches its lowest cost state.

The fluidity of cloud architectures necessitate a dynamic model, which limits the time allowed for a system to converge to an optimum. More generally, assuming that a system will remain in configuration $\mathcal{C}$ for a duration τ, one should temporarily move to $\mathcal{C}'$ if the cost of $\tau\mathcal{C}$ is greater than that of migrating to and from $\mathcal{C}'$ combined with the cost of maintaining $\tau\mathcal{C}'$. An accurate characterisation of τ enables configurations to be optimised with a minimum of migrations, yet a system in constant flux or with very small values of τ can potentially negate gains in migrating. Cheap migration operations can help offset the effects of τ.

Example 5. Figure 6 models migrations between various $\texttt{next}^{\infty}_{\cup}\texttt{()}$ states of a system's **L1** caches with three processes, where one of the caches has deployed the soft isolation strategy described in Example 3. Utilisation rates are given in brackets, assuming that (a) each **L1** confinement is shared between two cores and has a total capacity of 2, (b) each process has a utilisation factor of 1, (c) **L1** confinements have zero cost, as they are built into the architecture, and (d) non-empty $\mathbf{L1_S}$ confinements reduce their core's capacity to α (overhead values can

[2] Disabling hyperthreading was once common amongst cloud providers [33], although Amazon EC2 has recently foregone this practice [5].

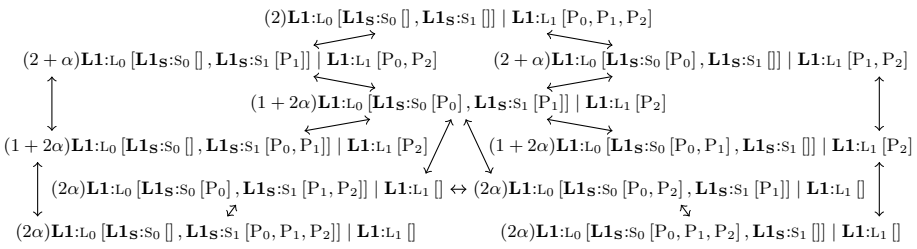


Fig. 6: A subset of possible global migrations between configurations.

reach up to 7% [37]). Disabling co-scheduling on the partitioned core will cause its capacity to be halved. Utilisation is highest $(2 + \alpha)$ when the unmitigated cache is at full capacity, with additional processes running within soft isolations. The configurations with the lowest `pairs()` are obtained for $1 + 2\alpha$. □

Metrics can also be extended to encompass *special purpose confinements* [9] and heterogeneous deployments, with certain configurations being cheaper or more secure to maintain on machines with dedicated hardware.

4.3 Automatically Generating Migration Sequences

The allocation of isolations to locations within a computational hierarchy is ultimately an exercise in scheduling. In its most general form, determining where confinements should be placed within a system is equivalent to bin-packing, thus eluding an efficient solution. The problem of placement is further complicated by the addition of quality of service predicates, which would typically include limits on capacity and utilisation. Finally, the hierarchical nature of the systems being investigated introduces its own nuances. For example, migrating an intermediate node within a containment graph will have a cascading effect on the constraints of its constituents.

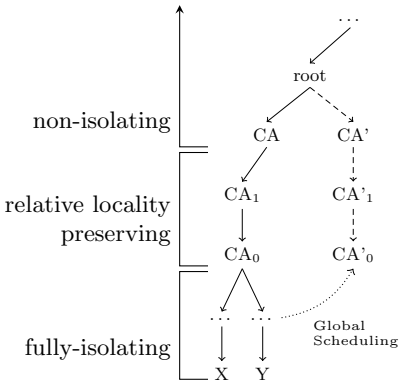
The task is thus to determine a sequence of migration operations that will move a system from a configuration $\mathcal{C}$ to a new configuration $\mathcal{C}'$ that satisfies the isolation and quality of service criteria that are being requested. If $\mathcal{C}'$ is known, then one can compute a sequence of migration operations leading to it using a minimum edit distance algorithm for graphs, with migrations corresponding to edit operations that are weighted according to the migration mechanisms' costs. One drawback of such an approach is that the minimum graph edit distance cannot always be calculated efficiently [22]. More crucially, this approach requires that $\mathcal{C}'$ be identified beforehand, whereas one typically has to compute both the migration sequence as well as the final configuration.

Figure 7a provides a general outline of the steps required to break the co-location of X and Y via a common ancestor CA within a partially-specified hierarchy described in Figure 7b. In the absence of efficient and exact oracles, several steps must be approximated by heuristics, as will be discussed in the remainder of this section. Note that the process of releasing or removing isolation

To break condition $X \overset{CA}{\Leftrightarrow} Y$:

1. find $D \in^{+} CA$.
   $X \in^{+} D \vee X = D \vee Y \in^{+} D \vee Y = D$
2. find/create CA'. $\neg CA' \in^{+} CA$
3. replicate path from CA to D in CA'
4. check isolation constraints and migrate D to new parent in CA'

(a) Migration procedure outline



(b) Migration effects by graph height

Fig. 7: Computing migration paths for breaking $X \overset{CA}{\Leftrightarrow} Y$.

constraints is similar to this procedure, with a greater focus on consolidating previously-isolated confinements back into existing confinements so as to lead to a cheaper configuration.

Finding a source The impact that the migration of a confinement D will have on a graph's isolation constraints will vary based on the position of D within that graph. For instance, migrating a process from one CPU core to the next will break locality at the core level, but not at the machine level.

When attempting to reconfigure the configuration illustrated by Figure 7b to comply with the constraint $\neg(X \overset{CA}{\Leftrightarrow} Y)$, one finds that individual migration operations moving confinements outside of CA can take one of three forms, namely: (a) *fully isolating*, where X and Y share no common ancestor up to the depth of CA, (b) *relative locality preserving*, whereby co-location through CA is broken, yet the confinements are still co-located within an intermediate common confinement, and (c) *non-isolating*, where the structures producing co-location through CA are preserved by the migration. Hence, the depth within the graph at which the confinement being migrated exists determines how many co-locations will survive migration. Consequently, for isolation to be achieved, one must migrate a confinement on the containment path leading from CA to X or Y. Note that in the case of multiple separate routes for co-location through CA, one may have to migrate more than one confinement to fulfil a single isolation constraint.

Migrations that preserve relative locality may be insecure and must be performed with caution, as attacks on the locality type of CA may still be viable were one to migrate to a location of the same type (such as the sibling CA'). Conversely, one cannot rely entirely on fully-isolating migrations due to the finiteness of physical infrastructures. In the case of migrations at the same depth, such as

when migrating either X or Y, one should ideally choose a migration that results in the lowest cost.

Finding a target Given that an appropriate confinement $D \in^+ CA$ has been marked for migration, the next step is to determine a suitable destination. Trivially, this must exist outside of CA. Referring to Figure 7b, the earliest depth within the graph to which the localities can be migrated is CA', a confinement directly co-located with CA.

Provided that it is of the correct type, any confinement CA'. $\neg(CA' \in^+ CA)$ can serve as a destination confinement, yet a heuristic may find it reasonable to attempt to keep migrations as local as possible. In broad terms, migration amongst smaller localities (**VC** to **C**, or **P** to **C**) can be performed in milliseconds, as opposed to the migration of larger structures (**P** to **OS**, or **VM** to **M**), which can be a thousand times slower, principally due to the involvement of the network layers and shared storage [21].

Creating an equivalent environment When migrating a confinement to a new parent, one would generally have to create a containment graph at the destination that matches the source's nesting structure. In certain cases, it may not be necessary to duplicate the full environment at the destination. For example, when migrating a **VM** that is running within a second **VM**, one may opt to migrate the former directly to bare metal.

Satisfying constraints When executing a sequence of migration operations, one must ensure that both the end state as well as the intermediate configurations do not violate any constraints that have previously been placed on the system. Ideally, constraints are checked before any migrations are performed, and migrations are only carried out once it has been established that they respect all isolation constraints. Failing this, transaction semantics must be added to migration sequences, giving the ability to dynamically roll back migration operations and attempt to identify an alternative path.

Backtracking will introduce delays in the servicing of isolation requests, which may not always be tolerable. If the workloads are well characterised, one may determine that certain constraints can be temporarily relaxed. For example, a tenant may tolerate a short-lived dip in performance, which would in turn allow a machine to be temporarily over-provisioned whilst performing a sequence of reconfiguration operations.

5 Applications

The following section investigates various contexts in which the model can be applied, including *runtime enforcement*, as well as in the modelling and analysis of an access-based side-channel and a replication-based timing channel mitigation.

5.1 Runtime Isolation

While co-location properties can be verified for specific scopes, the guarantees may no longer hold after a system has been reconfigured. Runtime monitoring serves to dynamically resolve isolation predicates that depend on confinements at the edges of a configuration's scope. The model can be used to define policies within a runtime monitoring framework, where declarative restrictions on co-locations are used to define invalid configurations. Once a bad state is detected (such as on detecting suspicious memory access patterns [34]), the system can be reconfigured to a correct state using migration, leading to a reactive architecture. Alternatively, the framework can be driven by a system of leases, with isolation being procured before a security-sensitive process executes.

5.2 Pre-emption Rate Limiting

The presented model can be used to reason about attacks at different granularities, which we demonstrate by modelling an access-driven cross-VM side-channel attack developed by Zhang et al. [35], and its scheduler-based mitigation [32]. The attack relies on a Prime-Probe cache access pattern, similar to the attack described in Example 2.

Consider a hypervisor managing two virtual machines, namely a victim $\mathrm{VM_v}$ and attacker $\mathrm{VM_a}$. Both machines (collectively referred to as $\overrightarrow{\mathrm{V}}$) share a core $\mathrm{C_0}$. The hypervisor agent Hyp is defined as:

$$\mathbf{A}{:}\mathrm{Hyp}(\{\mathtt{C}^{\mathrm{VM_v}}, \mathtt{C}^{\mathrm{VM_a}}, \mathtt{C}^{\mathrm{C_0}}\})^{\substack{\{(\mathrm{VM_v},\mathrm{C_0}),(\mathrm{VM_a},\mathrm{C_0})\} \\ \{\mathrm{VM_v},\mathrm{VM_a},\mathrm{C_0}\}}} \left[\overrightarrow{\mathrm{V}}\right]$$

and implemented as a process $\mathrm{Hyp^I}$ defined as:

$$\mathrm{Hyp^I} \equiv \prod^{\mathbf{VM}:\mathrm{X} \in \overrightarrow{\mathrm{V}}} \mathrm{X} \curvearrowright \mathrm{C_0}.\mathrm{X} \curvearrowright \mathrm{Hyp}.\mathrm{Hyp^I}$$

The $\mathtt{next}^{\infty}_{\cup}()$ graph of the system at this coarse level of granularity would reveal that the virtual machines are co-located through $\mathrm{C_0}$, yet the mechanism by which they interfere with each other is not immediately apparent. The hierarchy can be defined at a finer granularity by modelling **L1** as a confinement of N *cache-line sets* (**CLS**), giving $\mathbf{L1}{:}\mathrm{CLS_0}\,[\{\mathbf{CLS}{:}\mathrm{CS_i}\,[] \mid 0 \leq i < N\}]$. Cache-lines are invalidated as processes execute within a **VC**. In a fine-grained model, the agent process is modified to map **VC**s to **CLS** confinements, signifying that an operation running within that **VC** has disturbed the cache set in question (more precise models of cache eviction policies may also be defined, yet this is unnecessary for the purposes of this exposition). A process carried out by an agent Ag which schedules a **VC** to a **C**, models the **VM**'s interactions with **CLS** for R times, and then yields control of the scheduler is defined as:

$$\mathtt{run}(\mathbf{A}{:}\mathrm{Ag}, \mathbf{L1}{:}\mathrm{L}, \mathbf{C}{:}\mathrm{c}, \mathbf{VC}{:}\mathrm{vc}, \mathtt{R}) \equiv \mathrm{vc} \curvearrowright \mathrm{C}.(\prod^{\mathbf{CLS}:\mathrm{cs} \in L} \mathrm{vc} \curvearrowright \mathrm{cs})^{R}.\mathrm{vc} \curvearrowright \mathrm{Ag}$$

The attack is access-based, where the attacker attempts to determine the pattern of a victim's memory accesses. The attacker achieves this by priming

the cache and checking its access times after the victim executes, placing its **VC** VC_a within a cache set previously occupied by VC_v, leading to the sequence:

$$\texttt{run}(\text{Ag}, \text{CLS}_0, \text{C}_0, \text{VC}_\text{a}, \texttt{N}).\texttt{run}(\text{Ag}, \text{CLS}_0, \text{C}_0, \text{VC}_\text{v}, \texttt{R})$$

The attacker's resolution of the victim's intermediate cache states is greatly influenced by R. If a victim can be pre-empted frequently, then the attacker can build a more precise memory access model. Conversely, large values of R will increase the probability that other cache regions unrelated to the security-sensitive computation under attack will have been accessed, leading to noise. Thus, the victim VM_v attempts to choose a value of R such that it *maximises* the value of `pairs()` formed over an execution.

A mitigation against this attack [32] places a minimum running time on virtual machines, which stops an attacker from forcing deschedules and limiting its ability to profile a victim. By knowing the number of cache invalidations required to achieve the desired level of isolation and the cost of performing cache operations, one can determine a minimum **VM** scheduling quantum length.

A similar fine-grained cache analysis can be performed for cache colouring [26], where scheduling must guarantee disjoint cache sets. An additional related mitigation is that of the cache cleaning process (Example 3), which is effectively a solution for the same problem using a different scheduling level.

5.3 Timing Channel Elimination

StopWatch [27] is a collection of mitigations designed to reduce the information content of timing channels in the cloud. The approach's mitigation centres on the use of *replication* to create `R` copies of each virtual machine ($\texttt{R} \geq 3$), each of which is placed on a different machine containing other tenants' replicated **VM**s. Clock sources on a **VM** are then modified to report time as a median of its local time and that of the replicas. This ensures that a co-located attacker will observe the same timing behaviour. Several aspects of the mitigation can be modelled, including event synchronisation and OS-level soft isolations. This section will focus on the **VM** replication and placement aspects of StopWatch.

Given a network Net of machines, the **VM** placement requirements of StopWatch can be modelled as three invariant conditions, namely:

$$\forall \mathbf{VM}{:}\text{v} \in^+ \text{Net}.\ |\left\{\text{v'} \mid \mathbf{VM}{:}\text{v'} \in^+ \text{Net}, \texttt{is\_replica}(\text{v'}, \text{v})\right\}| = \texttt{R} \quad (1)$$

$$\forall \mathbf{VM}{:}\text{v}, \mathbf{M}{:}\text{M} \in^+ \text{Net}.\ |\{\text{v'} \mid \mathbf{VM}{:}\text{v'} \in \text{M}, \texttt{tenant(v')} = \texttt{tenant(v)}\}| \leq 1 \quad (2)$$

$$\begin{aligned}\forall \mathbf{VM}{:}\text{v}_1, \mathbf{VM}{:}\text{v}_2, \mathbf{M}{:}\text{M} \in^+ \text{Net}.\ \text{v}_1 \neq \text{v}_2 \wedge \text{v}_1 \overset{\text{M}}{\longleftrightarrow} \text{v}_2 \rightarrow \\ \neg\exists \mathbf{VM}{:}\text{v}_3, \mathbf{VM}{:}\text{v}_4, \mathbf{M}{:}\text{M'} \in^+ \text{Net}.\ \text{v}_3 \neq \text{v}_4 \wedge \text{v}_3 \overset{\text{M'}}{\longleftrightarrow} \text{v}_4 \wedge \text{M} \neq \text{M'} \wedge \\ \texttt{tenant(v}_1\texttt{)} = \texttt{tenant(v}_3\texttt{)} \wedge \texttt{tenant(v}_2\texttt{)} = \texttt{tenant(v}_4\texttt{)}\end{aligned} \quad (3)$$

The first invariant ensures that there are `R` replica machines within the network. The second invariant checks that each machine has at most one virtual machine belonging to the same tenant. The final invariant checks that any given pair of tenants can be co-located in at most one machine.

6 Related Work

Ambient Models. A seminal work in modelling hierarchical architectures was the calculus of *mobile ambients* [17], which extended process calculi with the *ambient* process construct. Ambients specify boundaries within which other ambients exist and migrate. Several extensions to the original calculus were subsequently defined, including the ability to define security zones to detect confidentiality breaches [15], as well as to model resource allocation through a system of markers [8]. An additional extension is the *cloud calculus* [25].

Graph Models. Graphs allow the definition of many-to-many relationships between a system's entities. Graph models for **VM** networks can be generated automatically [11, 16]. These can then be checked statically [14] to detect violations in operational correctness, failure resilience and isolation. Additional work focuses on making the analysis of dynamic systems more efficient through incremental analysis [13]. The creation and application of deltas is event-driven, triggered using hooks to a hypervisor. Challenges in dynamic monitoring include asynchronous updates, non-atomic actions, unordered events and blocking behaviour introduced by instrumentation [12]. Other approaches group resources into *colours* within which data can be shared, and employ a system of roles that can modify colour groupings and conflict rules [10].

Scheduler-based mitigations. Scheduling policies can be exploited to form illicit channels [32] or steal computational resources [31]. Setting a minimum time between deschedules can undermine a side-channel by obscuring residual cache effects [32]. Global scheduling can be used to reduce contention [31]. Efficiently choosing migration targets is non-trivial, as placement can be constrained by several factors in addition to isolation requirements [30]. The problem can thus be formulated as one of *constraint satisfaction*. Other approaches address placement as a bin-packing problem to guarantee different degrees of isolation whilst upholding a system's functional constraints [6]. The approach is evaluated in terms of a *competitive ratio*, comparing the cost of configurations produced by on-line scheduling against optimal placement, where cost is the number of bins used. Heuristics can aid migration and placement [18]. Another approach uses leases and deadlines to reserve resources and prioritise migrations [4].

Detection and generation. One challenge of policy-based defences is to create policies. Methods have been developed for detecting certain types of leaks through various techniques, including information flow analysis [7], abstract interpretation [20], and data tagging and tracking [29].

7 Conclusions and Future Work

This work has investigated the modelling of temporal and spatial co-location within the context of illicit channels, examining the issues of cost, scoping and

migration. It considered the creation of a model that can consistently reason about a variety of heterogeneous systems through a uniform notion of containment. It also examined the challenges in allocating resources within a hierarchical architecture. These concepts were applied to the modelling and analysis of several established attacks and defences, giving insight into their inner workings.

Future work will focus on the automated synthesis of runtime enforcement monitors, and the integration of the model into simulation frameworks.

References

1. CRIU project page (Jan 2016), http://criu.org/Main_Page
2. Libvirt project page (Jan 2016), http://libvirt.org/
3. Adams, K., Agesen, O.: A comparison of software and hardware techniques for x86 virtualization. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 2–13. ASPLOS XII, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1168857.1168860
4. Afoulki, Z., Rouzaud-Cornabas, J.: A security-aware scheduler for virtual machines on iaas clouds. Tech. rep., LIFO, ENSI de Bourges (2011)
5. Amazon: Amazon ec2 instances. https://aws.amazon.com/ec2/instance-types/ (Apr 2015)
6. Azar, Y., Kamara, S., Menache, I., Raykova, M., Shepard, B.: Co-location-resistant clouds. In: Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security. pp. 9–20. CCSW '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2664168.2664179
7. Backes, M., Kopf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. pp. 141–153. SP '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/SP.2009.18
8. Barbanera, F., Bugliesi, M., Dezani-Ciancaglini, M., Sassone, V.: A calculus of bounded capacities. In: Saraswat, V.A. (ed.) Advances in Computing Science ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation, Lecture Notes in Computer Science, vol. 2896, pp. 205–223. Springer Berlin Heidelberg (2003), http://dx.doi.org/10.1007/978-3-540-40965-6_14
9. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 267–283. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann
10. Bijon, K., Krishnan, R., Sandhu, R.: A formal model for isolation management in cloud infrastructure-as-a-service. In: Au, M., Carminati, B., Kuo, C.C. (eds.) Network and System Security, Lecture Notes in Computer Science, vol. 8792, pp. 41–53. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-11698-3_4
11. Bleikertz, S., Groß, T., Mödersheim, S.: Automated verification of virtualized infrastructures. In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop. pp. 47–58. CCSW '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2046660.2046672

12. Bleikertz, S., Groß, T., Mödersheim, S.: Modeling and analysis of dynamic infrastructure clouds. Tech. rep., IBM Zurich (12 2013)
13. Bleikertz, S., Vogel, C., Groß, T.: Cloud radar: Near real-time detection of security failures in dynamic virtualized infrastructures. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 26–35. ACSAC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2664243.2664274
14. Bleikertz, S., Gro, T.: A virtualization assurance language for isolation and deployment. In: POLICY. pp. 33–40. IEEE Computer Society (2011), http://dblp.uni-trier.de/db/conf/policy/policy2011.html#BleikertzG11
15. Braghin, C., Cortesi, A., Focardi, R.: Security boundaries in mobile ambients. Computer Languages, Systems & Structures 28(1), 101 – 127 (2002), http://www.sciencedirect.com/science/article/pii/S0096055102000097, computer Languages and Security
16. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In: IEEE (ed.) PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing. Pisa, Italy (Feb 2010), https://hal.inria.fr/inria-00429889
17. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Proceedings of POPL'98. ACM Press (1998)
18. Caron, E., Rouzaud-Cornabas, J.: Improving users' isolation in iaas: Virtual machine placement with security constraints. Research Report RR-8444, INRIA (Jan 2014), https://hal.inria.fr/hal-00924296
19. Dolan-Gavitt, B., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: Mining memory accesses for introspection. In: ACM CCS'13. pp. 839–850. ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2508859.2516697
20. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. In: Proceedings of the 22Nd USENIX Conference on Security. pp. 431–446. SEC'13, USENIX Association, Berkeley, CA, USA (2013), http://dl.acm.org/citation.cfm?id=2534766.2534804
21. Falzon, K., Bodden, E.: Dynamically provisioning isolation in hierarchical architectures. In: Lopez, J., Mitchell, C.J. (eds.) Information Security, Lecture Notes in Computer Science, vol. 9290, pp. 83–101. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-23318-5_5
22. Gao, X., Xiao, B., Tao, D., Li, X.: A survey of graph edit distance. Pattern Analysis and Applications 13(1), 113–129 (2010), http://dx.doi.org/10.1007/s10044-008-0141-y
23. Gueron, S.: Intel advanced encryption standard (aes) new instructions set. http://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf (May 2010)
24. Hu, W.M.: Reducing timing channels with fuzzy time. In: Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on. pp. 8–20 (May 1991)
25. Jarraya, Y., Eghtesadi, A., Debbabi, M., Zhang, Y., Pourzandi, M.: Cloud calculus: Security verification in elastic cloud computing platform. In: Smari, W.W., Fox, G.C. (eds.) CTS. pp. 447–454. IEEE (2012), http://dblp.uni-trier.de/db/conf/cts/cts2012.html#JarrayaEDZP12
26. Kim, T., Peinado, M., Mainar-Ruiz, G.: Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In: 21st USENIX conference

on Security symposium. Security'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2362793.2362804
27. Li, P., Gao, D., Reiter, M.: Mitigating access-driven timing channels in clouds using stopwatch. In: Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on. pp. 1–12 (June 2013)
28. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of aes. In: 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology. pp. 1–20. CT-RSA'06, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11605805_1
29. Priebe, C., Muthukumaran, D., O'Keeffe, D., Eyers, D., Shand, B., Kapitza, R., Pietzuch, P.: Cloudsafetynet: Detecting data leakage between cloud tenants. In: ACM Cloud Computing Security Workshop (CCSW). ACM, ACM, Scottsdale, Arizona, USA (11/2014 2014)
30. Raj, H., Nathuji, R., Singh, A., England, P.: Resource management for isolation enhanced cloud services. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security. pp. 77–84. CCSW '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1655008.1655019
31. Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., Swift, M.M.: Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 281–292. CCS '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2382196.2382228
32. Varadarajan, V., Ristenpart, T., Swift, M.: Scheduler-based defenses against cross-vm side-channels. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 687–702. USENIX Association, San Diego, CA (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan
33. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In: 21st USENIX Conference on Security Symposium. pp. 9–9. Security'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2362793.2362802
34. Zhang, Y., Juels, A., Oprea, A., Reiter, M.K.: Homealone: Co-residency detection in the cloud via side-channel analysis. In: IEEE S&P'11. pp. 313–328. IEEE Computer Society, Washington, DC, USA (2011), http://dx.doi.org/10.1109/SP.2011.31
35. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-vm side channels and their use to extract private keys. In: ACM CCS'12. pp. 305–316. ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2382196.2382230
36. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-tenant side-channel attacks in paas clouds. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 990–1003. CCS '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2660267.2660356
37. Zhang, Y., Reiter, M.K.: Dppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: ACM CCS'13. pp. 827–838. ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2508859.2516741