Secure Software Engineering Group
European Centre for Security and Privacy by Design

TECHNISCHE
UNIVERSITÄT
DARMSTADT

# On the Use of Migration to Stop Illicit Channels

*Vom Fachbereich Informatik (FB 20)*
*an der Technischen Universität Darmstadt*

*zur Erlangung des akademischen Grades*
*eines Doktor-Ingenieurs (Dr.-Ing.)*

*genehmigte Dissertation von*

Kevin Falzon, M.Sc.
*geboren am 27. Dezember 1987 in Pietà, Malta*

Tag der Einreichung: 7. November 2016
Tag der Disputation: 21. Dezember 2016

Referenten

| | |
|---|---|
| Prof. Dr. Eric Bodden | *Referent* |
| Prof. Dr. Bernd Freisleben | *Korreferent* |
| Prof. Dr. Patrick Eugster | *Korreferent* |

Hochschulkennziffer: D17
Darmstadt, 2017

Kevin Falzon
*On the Use of Migration to Stop Illicit Channels*

PH.D. REFEREES:
Prof. Dr. Eric Bodden (Referent)
Prof. Dr. Bernd Freisleben (Korreferent)
Prof. Dr. Patrick Eugster (Korreferent)

FURTHER PH.D. COMMISSION MEMBERS:
Prof. Dr. Reiner Hähnle (Vorsitz)
Prof. Dr. Marc Fischlin
Prof. Dr. Matthias Hollick

# Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 7. November 2016.

Kevin Falzon

# Abstract

Side and covert channels (referred to collectively as *illicit channels*) are an insidious affliction of high security systems brought about by the unwanted and unregulated sharing of state amongst processes.

Illicit channels can be effectively broken through isolation, which limits the degree by which processes can interact. The drawback of using isolation as a general mitigation against illicit channels is that it can be very wasteful when employed naïvely. In particular, permanently isolating every tenant of a public cloud service to its own separate machine would completely undermine the economics of cloud computing, as it would remove the advantages of consolidation.

On closer inspection, it transpires that only a subset of a tenant's activities are sufficiently security sensitive to merit strong isolation. Moreover, it is not generally necessary to maintain isolation indefinitely, nor is it given that isolation must always be procured at the machine level.

This work builds on these observations by exploring a *fine-grained* and hierarchical model of isolation, where fractions of a machine can be isolated dynamically using *migration*. Using different units of isolation allows a system to isolate processes from each other with a minimum of over-allocated resources, and having a dynamic and reconfigurable model enables isolation to be procured on-demand. The model is then realised as an implemented framework that allows the fine-grained provisioning of units of computation, managing migrations at the core, virtual CPU, process group, process/container and virtual machine level. Use of this framework is demonstrated in detecting and mitigating a machine-wide covert channel, and in implementing a multi-level moving target defence.

Finally, this work describes the extension of post-copy live migration mechanisms to allow temporary virtual machine migration. This adds the ability to isolate a virtual machine on a short term basis, which subsequently allows migrations to happen at a higher frequency and with fewer redundant memory transfers, and also creates the opportunity of time-sharing a particular physical machine's features amongst a set of tenants' virtual machines.

# Zusammenfassung

Seitenkanäle und versteckte Kanäle stellen insbesonders für sicherheitssensible Systeme ein großes Problem dar. Sie entstehen, da Prozesse unbeabsichtigt Informationen über ihren Zustand teilen.

Solche Kanäle kann man mittels Isolation vermeiden, da dadurch der Grad, mit dem Prozesse interagieren können, eingeschränkt wird. Der Nachteil der Isolation ist allerdings, dass sie äußerst ressourcenintensiv ist. Dies gilt besonders dann, wenn man jeden Anwender einer Cloud permanent von allen anderen Anwendern isoliert, seine Anwendungen also auf einer getrennten Maschine ausführt.

Bei genauerer Betrachtung zeigt sich, dass nur ein Teil der Prozesse eines Anwenders so sicherheitssensibel sind, dass eine totale Isolation sinnvoll ist. Außerdem ist es nur selten nötig, Anwendungen zu jedem Zeitpunkt zu isolieren.

Die vorliegende Dissertation baut auf diesen Beobachtungen auf und stellt ein feingranulares, hierarchisches Modell für die Isolation vor. Das Modell ist in der Lage, Teile einer Maschine mittels Migration dynamisch zu isolieren. Dies erlaubt die Isolation unterschiedlicher Prozesse voneinander, ohne dass Ressourcen verschwendet werden, sowie das Starten des Migrationsprozesses auf Abruf. Das Modell wurde in einem Rahmenwerk implementiert, das die Migration von Prozessorkernen, virtuellen Prozessoren, Prozessgruppen, Containern sowie kompletten virtuellen Maschinen erlaubt. Der Nutzen des Rahmenwerks wird anhand der Erkennung und Beseitigung eines systemweiten versteckten Kanals sowie der Implementierung einer mehrstufigen Verteidigung gegen solche Kanäle gezeigt.

Abschließend beschreibt diese Dissertation eine Erweiterung zu Post-Copy Live Migration, welche das temporäre Migrieren virtueller Maschinen zum Ziel hat. Dies erlaubt es, virtuelle Maschinen kurzzeitig zu isolieren, wodurch eine höhere Frequenz von Migrationsvorgängen bei gleichzeitig geringerer Speicherauslastung erzielt wird. Dadurch wird ermöglicht, dass spezielle Funktionen einer physischen Maschine von allen Anwendern beliebig genutzt werden.

# Publications

This document is an original work, except where indicated otherwise. Elements of this work have appeared in a similar form within the following publications, for which I was the lead author:

- Kevin Falzon and Eric Bodden. 'Towards a Comprehensive Model of Isolation for Mitigating Illicit Channels'. In: *5th International Conference on Principles of Security and Trust (POST 2016)*. Ed. by Frank Piessens and Luca Viganò. Vol. 9635. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, Apr. 2016, pp. 116–138. ISBN: 978-3-662-49635-0. DOI: 10.1007/978-3-662-49635-0_7

  *This paper is the foundation of the model described in Chapter 3, and also constitutes parts of Chapter 2.*

- Kevin Falzon and Eric Bodden. 'Dynamically Provisioning Isolation in Hierarchical Architectures'. In: *18th International Conference on Information Security (ISC 2015)*. Ed. by Javier Lopez and Chris J. Mitchell. Vol. 9290. Lecture Notes in Computer Science. Springer International Publishing, Sept. 2015, pp. 83–101. ISBN: 978-3-319-23317-8. DOI: 10.1007/978-3-319-23318-5_5 **– Awarded Best Student Paper.**

  *Elements of this work appear within Chapter 2 and form the basis of the isolation framework described in Chapter 4.*

- Kevin Falzon and Eric Bodden. *There and Back Again: Temporary Virtual Machine Migration via Aborted Post-Copy*. Under submission. 2016

  *This work is the basis of Chapter 5, which concerns the modification and extension of virtual machine live migration mechanisms to allow temporary virtual machine migration.*

# Acknowledgements

First, I would like to thank my supervisor, Professor Eric Bodden, for offering me a position in his group, for his many excellent and perceptive insights, and for his confidence in my work, particularly when treading into waters unknown. I would also like to thank Professors Bernd Freisleben and Patrick Eugster, as well as Professors Reiner Hähnle, Marc Fischlin and Matthias Hollick, for very kindly agreeing to being on my doctoral committee.

I am greatly indebted to Andrea Püchner and Karina Köhres for their unparalleled help with all things administrative and organisational.

Special thanks go to the most versatile Andreas Follner, who, in having the misfortune of sharing an office with me for the greater part of my studies, took on a variety of roles, ranging from disc jockey and cocktail mixer to translator and legal counsel. Above all, he was, and remains, a great friend.

Equally of note are Michael “Honcho” Zohner, Hanne Weismann, Jan Sinschek, Tommaso Gagliardoni, Daniel Demmler and everyone from the SSE and ENCRYPTO group, who all managed to make Darmstadt feel like a home away from home.

Finally, I would like to thank my friends and family for their unwavering support over the years. My unending gratitude goes out to Claire for her inexhaustible patience, kindness and understanding through thick and thin.

# Contents

# List of Figures

# List of Tables

JOSEPH FALZON
† 29.04.2014

JOSEPH VELLA
† 28.04.2014

ID-DAWL TA' DEJJEM
JIDDI LILHOM

CHAPTER 1

# INTRODUCTION

SECURITY IS THE PRODUCT OF MANY FACTORS. A fundamental determinant of a process' security is the *location* and environment in which it runs. This has become increasingly evident with the advent of cloud computing, where consolidation is rampant, and tenants may unwittingly share hardware with hostile processes. New models and mechanisms must be developed to efficiently manage the scarce luxury of *isolation* and prevent attacks on processes through their environment.

## 1.1 Introduction

The security of a process is strongly affected by the environment within which it executes, as many of the security guarantees offered by an application depend on the validity of assumptions on the underlying infrastructure. For example, while the ciphertext produced by an encryption algorithm may be proven resilient to an attack carried out by an external observer, the same cannot always be said when faced with an attacker that can see the encryption operation's internal and intermediate states. More generally, a process' security may be subverted if its internal state can be directly observed or indirectly inferred by an attacker.

The question of how one may limit the visibility of an entity's internal state from an outside observer has historically been referred to as the *confinement problem* [Lam73]. Typically, attempts at confining processes to their environment are conducted through a combination of hardware and the operating system, which enforces strict partitions through many mechanisms, including memory protection and CPU scheduling policies.

### 1.1.1 Problem Definition

In simple terms, the problem with confinements is that they can *leak*. Many confinements are either intrinsically imperfect or can be sabotaged to reveal their internal state. For example, an attacker may exploit a memory leak to learn the memory contents of a victim process, or to escape a virtual machine confinement. Alternatively, an attacker may compromise a virtual machine monitor to intercept commands as they are issued by processes to their underlying hardware. The latter is difficult for a common attacker to carry out, as it would

require integral and systematic changes to the virtualisation platform, which would be tricky to perform without the direct assistance of the infrastructure's owner. Beyond the difficulty of their execution, such brazen attempts at infiltration carry several drawbacks for an attacker. For instance, they lack an element of plausible deniability, being unambiguous in their purpose. Similarly, a noisy or high-profile attack calls attention to itself more readily, and can lead to the creation of patches that neutralise it. This places the durability of the exploit into question, and potentially bounds the period of time over which data can be acquired.

In the previous attack scenarios, the underlying assumption is that confinements are impenetrable and opaque, necessitating the creation of explicit mechanisms for publicising their internal state. Yet when considering real-world systems, one quickly finds that the confinement problem has never been truly and comprehensively solved. The actions performed by a process will often produce a deluge of non-functional side-effects, which may be observable by other processes that share the same infrastructure. In some cases, these side-effects can be correlated, with a high degree of certainty, to the internal operations being carried out by the process. For instance, rather than install a key logger, a process may attempt to record a user's input indirectly by correlating keystrokes to measurable CPU activity bursts, and comparing the durations between bursts to models of people's typing behaviours and patterns. While this approach may appear tortuous and error-prone, it carries one distinct advantage over a key logger, namely that it only uses innate characteristics of the system as it executes within its normal parameters. This foregoes the need to directly interfere with the system being attacked.

In general, one can learn the internal state of a confinement in two ways. The first is by forming an *overt channel* with a process in the outside world, this being any of the standard communication channels such as sockets, files or shared memory. The second and more underhanded option is through a *side-channel*. To slightly paraphrase Lampson [Lam73], a side-channel is formed when information is sent over a medium that was not designed for communication. This includes a wide range of phenomena. For example, side-channels that infer a target process' instruction stream by measuring the variation of a system's power consumption over time have been demonstrated, with different instructions having different signatures on power consumption. Side-channels are more than mere curiosities, and in some instances present the best-known method for subverting a system's security. For example, side-channels have allowed the deduction of secret keys in otherwise opaque embedded systems performing encryptions, as well as keys used by remote servers.

Related to side-channels are *covert channels*, where one process intentionally forms a side-channel with another process in an effort to transfer data through an unmonitored interface [Tir07]. For example, two processes may exchange a stream of bits by modulating accesses to a hard disk, encoding information in the disk's observable seek time. Unlike

side-channels, covert channels imply a degree of conspiracy, as both endpoints are actively trying to communicate with each other. This makes covert channels simpler to implement than side-channels, as processes can modulate an agreed-upon shared resource using a predetermined transfer protocol. The difference between covert channels and side-channels lies primarily in their intent, rather than mechanism. Throughout this document, covert channels and side-channels will be collectively referred to as *illicit channels*.

#### 1.1.1.1 Co-Location and Isolation

In its simplest interpretation, location affects security in that running a process within a compromised execution environment places it in jeopardy. In the case of illicit channels, location takes on a more literal meaning, with the physical location at which a process executes being a critical factor in the viability of a channel.

Imperfections in confinements are the root causes of illicit channels, yet they only manifest into a security concern when coupled with *co-location*. Consider, for example, the key logging attack described earlier. While the process may be leaking information through the CPU utilisation rates, this is of no consequence if the attacker exists outside of that machine, as it cannot observe the phenomenon. Thus, the attacker must be *co-located* with the victim process though the medium over which the confinement leaks.

More precisely, illicit channels are contingent on the relative position of the victim and attacker processes within an infrastructure. For example, a side-channel that measures the CPU activity of a victim process by observing its own scheduling behaviour must necessarily share the same scheduler, and by association, the same pool of CPU cores. This channel cannot be formed across processes executing on different machines, as their schedulers are disjoint, and the scope of the scheduler's effects at the victim do not carry over onto those of the second machine. Instead, one would have to extend the side-channel to correlate CPU usage with effects that can be observed remotely. For instance, if the victim process is a server, an attacker on a different machine may infer the victim's CPU usage by generating requests and measuring their response times, yet even such an attack would require co-location at the network level.

### 1.1.2 Risk

Illicit channels have a number of features that can make them particularly dangerous in the context of security, which can be summarised as follows.

**Ubiquity** The potential for forming illicit channels within a system is high, given that a typical system will leak in different ways. This becomes a problem when coupled with

modern computer systems, which co-locate many processes with different origins over shared infrastructures.

**Stealth** Illicit channels are often built using effects that are abstracted away during the security analysis of a system, and are not properly encompassed by systems' security policies. For example, program analysis techniques may attempt to prove a program's absence of data leaks by focusing on its data structures and language-level program flow, whereas a holistic approach would also demand attention to secondary effects such as power consumption and instructions' running times.

The subtlety by which certain leaks occur, coupled with the lack of adequate safeguards against an attack, gives an attacker an edge over conventional attacks, which, while often effective, are well-characterised, conspicuous, and actively guarded against. This element of stealth can also allow attacks to execute over a long period of time before it is discovered, facilitating longitudinal data collection.

**Versatility** Illicit channels have the ability to expose elements of a process' internal state that could not be retrieved through more traditional attacks.

One area that carries the threat of illicit channels is *cloud computing*. At its heart, a public cloud service is an assembly of processes belonging to mutually-distrusting tenants executing over a shared physical infrastructure. Various mechanisms, notably *virtualisation*, attempt to confine tenants' behaviours, restricting the degree to which they can interfere with other co-located tenants. Nevertheless, virtual machine confinements are not always perfect, and may admit the creation of illicit channels. This is not to say that the problem of illicit channels is unique to the cloud or other large infrastructures. Smart phones, for example, are excellent candidates for illicit channel attacks, and covert channels can subvert the information flow policies of a phone's operating system to leak private information [Cha+15; CQM14].

## 1.2 State of the Art of Mitigations

Mitigations focus on removing the information content of the phenomenon with which the channel is being formed, effectively *isolating* the communicating parties. Isolation can either be procured synthetically by adding noise to the channel or regularising a system's behaviour (*soft* isolation), or by eliminating the pre-requisite co-location between the parties (*hard* isolation) [VRS14].

### 1.2.1 Limits of Isolation

Soft isolation often incurs an ongoing performance overhead, with some fraction of the machine's capacity committed to maintaining the isolation. Hard isolation is an appealing ap-

proach to mitigating illicit channels, as it can comprehensively eliminate entire classes of attacks. The problem with hard isolation is that it can leave hardware underutilised by limiting multiprogramming, as it places restrictions on co-location.

The cloud computing scenario is one of the most compelling examples of a system that would benefit from strong isolation guarantees, yet the aims of the stakeholders in a public cloud are at odds with each other. This is because a cloud provider aims to maximise profits by consolidating as many tenants as possible on a minimum of machines, whereas a security-conscious tenant would ideally execute in isolation. Clouds also present individual tenants with a limited view of their surroundings, making it harder for a given tenant to identify a co-resident attacker.

## 1.3 Proposed Solution

Permanently isolating tenants within a cloud would wreak havoc on the economics of cloud computing, yet in general, only a subset of a tenant's activities in the cloud are sufficiently security sensitive to warrant complete isolation. Similarly, partial isolation is sufficient for eliminating certain illicit channels, and isolating tenants to their own private machine may be unnecessary. Moreover, machine-level isolation would not necessarily suffice for mitigating channels internal to a tenant's own virtual machine. One would thus benefit from a more nuanced and fine-grained approach to isolation and co-location elimination. To illustrate, consider the following scenario:

**Example 1.** A virtual machine $\mathcal{A}$ is suspected to contain a process that launches a cache-timing attack during the execution of a specific section of code in process $P_{\mathcal{V}}$ in virtual machine $\mathcal{V}$. Such an attack can be foiled at different granularities. For instance, starting from the coarsest level, one may break co-location

i) at the machine level, by placing $\mathcal{A}$ and $\mathcal{V}$ on separate physical machines,

ii) at the virtualisation level, by forbidding $\mathcal{A}$ and $\mathcal{V}$ from being co-scheduled on the same core, or

iii) at the process and virtualisation level, by forbidding the execution of $P_{\mathcal{V}}$ alongside any process in $\mathcal{A}$.

### 1.3.1 Objective

The objective of this work is twofold. First, this work will study the decomposition and modelling of confinements into finer-grained hierarchies of co-located confinements. For example, several physical machines may exist on a network, thus acting as confinements

within the network confinement layer. These machines may, in turn, contain sets of virtual machine confinements existing at the virtualisation layer, which could be further decomposed into cores, processes and threads.

The second aspect of this work is to investigate the elimination of co-location by reconfiguring, or *migrating*, confinements at different levels of the hierarchy. The effectiveness of a migration in procuring isolation depends on *three* characteristics, or *traits*, by which a migration's destination differs from its source. These are *passive*, *active*, and *context* traits, defined as follows.

**Passive traits** are architectural properties inherent to a system that are functionally transparent to processes, yet produce an observable phenomenon. For example, while programs can be written independently of a processor's *cache inclusivity model* (Section 2.5.1), certain attacks will only work on inclusive caches [Liu+15].

**Active traits** are machine capabilities that a program must make use of explicitly. These include special-purpose hardware confinements and CPU instruction-set extensions, such as AES-NI [Gue10], or, more recently, the MPX [Int15b] and SGX [BPH14] extensions. A system may also have software-based active traits, such as cache-conscious memory allocators [KPMR12].

**Context traits** are aspects of a machine that vary depending on its runtime configuration and its state in relation to its environment. For example, a machine hosting a single virtual machine will have a temporary context trait of *isolation*, guaranteeing that the tenant is executing alone.

A dynamic and migration-based solution to illicit channels is based on migrating a vulnerable confinement to a target whose traits disallow the channel in question. Such an approach to co-location elimination becomes even more relevant when one considers that typical cloud infrastructures have multitudes of confinements with different traits, and isolation can be procured on-demand through reconfiguration. By combining a finer-grained model of computer systems with methods for migrating confinements at various granularities, one can comprehensively reason about and efficiently manage spatial and temporal isolation. In this context, spatial isolation refers to the separation of entities that exist at the same time using physical boundaries, such as through partitioning. Temporal isolation concerns the separation of entities in time, or equivalently, the use of time-sharing. As will be seen, temporal isolation can go beyond simply forbidding the simultaneous scheduling of entities. For example, it can require the modification of scheduling policies to change the order of scheduling, or the introduction of minimum quanta sizes.

### 1.3.2 Advantages of a Dynamic Approach

Given that a cloud consists of a network of machines with varying traits, one can foil illicit channels by executing tenants' tasks on machines whose traits guarantee a corresponding level of security. This requires some form of distributed computing. One option is to pre-partition a program [Cho+09; Pat+14; Zhe+03] and map the resulting fragments onto different machines. Similarly, a program can be designed to outsource computations [Qia+15].

Pre-partitioning carries a number of drawbacks, foremost of which is that one must know in advance both *when* isolation will be required, as well as the *location* at which the task should execute. This could potentially leave the designated target underutilised, as it must preserve the traits required by the security-sensitive task for as long as the task may require isolation. Alternatively, a system would have to employ a high degree of replication in order to guarantee that a suitable destination is available at all times.

Contrast this with a migration-based approach to distributed computing. Migration, as opposed to pre-partitioning programs or using replication, avoids the substantial complexities brought about by the introduction of multiple program counters, as there will only be a single definitive live version of each confinement. In addition, a migration target can be chosen dynamically at runtime without the need of elaborate initialisation and coordination routines. This also greatly reduces the need to reserve destination confinements in anticipation of an isolation requirement, and leads to a higher degree of multiplexing of traits (where a machine with a given trait can be time-shared amongst multiple entities).

Migration can often be applied transparently to arbitrary workloads without having to modify the tenants' applications, which are not always simple to decompose. It also avoids the complexities of aggregating results, as there will only be a single active instance of the confinement in question. Moreover, a dynamic approach can react to any context trait, and a migration can be triggered at any time, whereas partitioning can only transfer control to a remote server at pre-defined points. This is particularly useful in the case of cloud infrastructures, where workloads are not generally known in advance, and vary dynamically.

The main criticism of migration lies in its associated overhead, yet as will be seen during this work, localised migrations of small confinements can be performed at high frequencies and with very low costs. Similarly, this work investigates methods of reducing the cost of migrating large confinements, particularly virtual machines, to the minimum required by a mitigation.

### 1.3.3 Claims

In summary, this work presents and supports the following claims.

**Claim 1** The problem of illicit channels is fundamentally one of co-location, and can be

modelled as such.

**Claim 2** Containments can be modified efficiently through scheduling and migration at various granularities.

**Claim 3** The impact of migrating large confinements such as virtual machines can be reduced using partial or temporary migrations.

## 1.4 Document Structure

The following is a brief description of this document's structure and the major themes and aims of each chapter.

**Chapter 2** is an inquiry into existing attacks and mitigations at each level. Currently, security is provided through several piecemeal mechanisms that typically guard against very specific instances of attacks, and cannot always be scaled or adapted to cover complete classes of attacks. Knowledge of attack vectors and mitigation techniques aids in deriving a more general approach to managing each confinement's locality, as well as in extracting the hierarchical structure of confinements.

**Chapter 3** describes a formal model for defining the hierarchical nature of confinements and their movement. The model supports a unified representation and treatment of co-location and migration at each level of the hierarchy, allowing attacks and mitigations to be modelled uniformly. Use of the model is demonstrated using several examples of known attacks and mitigations.

**Chapter 4** details the creation and implementation of a framework, dubbed SAFEHAVEN, that allows the creation, management and deployment of mitigations throughout the confinement hierarchy. The framework is applied to two case studies, namely in coordinating virtual machine migration to eliminate a covert channel, and in developing a multi-level moving target defence. The results of the case studies also form the basis of comparisons between the performance of migration mechanisms at the different levels of the hierarchy.

**Chapter 5** builds on the migration mechanisms explored in the previous chapter, and describes the extension of post-copy virtual machine live migration to support the temporary relocation of virtual machines. This allows tasks to be isolated for a period that is shorter than the duration of a full migration whilst retaining the convenience of operating at the granularity of virtual machines.

**Chapter 6** reviews the results obtained throughout the course of this work, discusses related and future work, and ends the document with concluding remarks.

CHAPTER 2

# BACKGROUND

WHILE EACH ILLICIT CHANNEL may at first appear to have its own unique mechanism of action, further investigation reveals that one may still classify channels on the basis of their *scope*, or medium through which they communicate. Knowing the scope of an illicit channel is crucial when adopting a partitioning-based mitigation strategy, as it defines the conditions for isolation with respect to that channel.

## 2.1 Introduction

The genealogy of modern illicit channels can be traced back many decades [Gli93; Lam73], with illicit channels being a persistent and pervasive threat to computer systems. Initially confined to high-security or military settings, the study of illicit channels has been drawn into the limelight due to a number of technological [Ris+09] and political [Nsa] developments.

In the former case, the paradigm of cloud computing has led users of computer systems to, at least partially, relinquish direct control over their activities by moving their computations from self-owned machines to shared public infrastructures. While the scale and business model may be different, the cloud topology harks back to the days of big iron and mainframes, with a dose of replication and other mechanisms designed to improve resilience. The fundamental problem remains the same, namely that a user's tasks are submitted to the great beyond, with limited assurances on the behaviour and intent of those that are sharing the user's infrastructure, as well as an implied trust in the cloud provider (the latter problem meriting its own investigation outside of this work).

On the smaller end of the spectrum lies another technological development, namely the rise of smart phones and their rampant and widespread adoption. While not directly addressed in this work, illicit channels afflict smart phones in ways similar to the cloud scenario. The fundamental difference is that rather than having multiple adversarial tenants sharing an infrastructure, smart phones have multiple *apps* with different origins sharing a device. This opens the avenue for collusion between apps in an effort to subvert the operating system's access control policies and leak sensitive information [Cha+15].

Politics has served as a driving force in illicit channel research and development due to channels' ability to capture and transfer information that could otherwise not be obtained, as well as their capacity for *stealth*. This makes them a lucrative apparatus for espionage.

These drivers have led to the emergence of many different types of illicit channels, attacking different elements of programs and hardware. Unlike overt communication channels such as sockets and IPCs, illicit channels lack a well-defined and auditable interface, and must be identified and regulated using additional security mechanisms. To better understand illicit channels and how they operate, this chapter analyses the many types and forms of channels, and attempts to identify and extract their commonalities. These are then used to construct a comprehensive strategy to counteracting illicit channels.

### Chapter Outline

This chapter is structured as follows:

**Section 2.2** demonstrates the core elements of illicit channels through an experiment, where the mapping of hardware threads to cores is deduced via an illicit channel.

**Section 2.3** identifies the primary stakeholders and scenarios where illicit channels constitute a threat.

**Section 2.4** broadly describes the common classes of attacks and mitigations.

**Section 2.5** is an in-depth investigation into specific known instances of attacks, and their mitigations, ordered by the medium through which the channels concerned are formed.

**Section 2.6** concludes this chapter.

## 2.2 A Simple Side-Channel

To better motivate the discussion, the following section describes the construction of a simple, yet complete, side-channel attack. The side-channel under consideration is designed to infer the cache hierarchy of a given CPU. Knowing the cache structure of a system is the first step in carrying out subsequent cache-level attacks [OST06; Per05].

### 2.2.1 Scenario

Modern multi-core CPUs generally have a hierarchical cache structure, with multiple cache *levels*. Cores within a CPU package will often share specific cache levels.

Consider the Intel i7-2640M CPU (INTEL-$M_T$ in Table 4.2), which consists of a two-core package, with two hardware threads (**HT**, or *hyperthreads* [Int16b]) to each core, as illustrated in Figure 2.1. In the case of this particular architecture, each core has its own **L1**

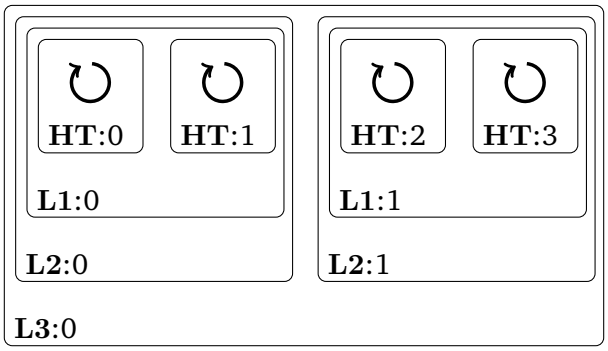



Figure 2.1: Cache hierarchy for CPU with simultaneous multithreading.

and **L2** caches, and both cores share a unified **L3** cache. This cache structure intertwines the hardware threads of each core into pairs that operate within a shared **L1** and **L2** locality, meaning that processes assigned to the same core may perform cache-level side-channel attacks over these levels, in addition to the package-wide **L3** cache.

Knowledge of the mapping between hardware threads and caches (or by implication, cores) aids an attacker in determining the viability of an attack at the **L1** and **L2** cache levels, yet the mapping varies between architectures. Furthermore, operating systems commonly represent hardware threads as full-blown cores, hiding the groupings of hardware threads into physical cores. In addition, the introduction of virtualisation, with its abstraction of *virtual CPUs*, can further obfuscate the relation between computational cores and caches. Thus, an attacker would benefit from a dynamic method of deducing core-to-cache assignments.

Outside of the context of security, Klug et al. [Klu+11] proposed a system to automatically optimise the distribution of processes to cores. Within their work, they identified a phenomenon that enables one to discern the topology of a CPU package, namely the principle that cache contention leads to an observable delay in the servicing of memory modification operations, and that the delay varies depending on the cache level at which contention occurs. Since on-chip coherency updates are faster than off-chip updates, one can deduce whether threads are running over a shared cache by timing repeated memory operations on a shared variable.

### 2.2.2 Experiment

The observation of different timing behaviours for identical operations on different cache-levels is conducive to the formation of a side-channel for determining core-to-cache pinnings, the results of which can then be used for subsequent illicit-channel attacks.

This side-channel was confirmed by the author of this document through the design and implementation of an experiment, which was carried out as follows. To determine the effects

of cache contention, two `pthreads` were started, with each `pthread` executing an identical workload: incrementing and retrieving a shared counter in a tight loop for a large and fixed number of iterations. The total time taken for both threads to execute was recorded. If the hypothesis were true, then one would expect the experiment to have taken longer when the threads were executing on different cores, due to the off-core cache invalidations incurred.

### 2.2.3 Result

Table 2.1 presents the total time observed when running the aforementioned workloads using different thread-to-processor pinnings. As mentioned previously, the operating system represents each hardware thread as a core. Consequently, the CPU was abstracted as a four-core package.

| | | Thread 0 | | | |
|---|---|---|---|---|---|
| | **Core** | **0** | **1** | **2** | **3** |
| Thread 1 | **0** | 1.15 | 1.00 | 2.52 | 2.52 |
| | **1** | 1.00 | 1.15 | 2.51 | 2.52 |
| | **2** | 2.51 | 2.52 | 1.16 | 1.00 |
| | **3** | 2.52 | 2.52 | 1.03 | 1.15 |

Table 2.1: Total time (normalised) for both threads to complete, expressed as a multiple of the shortest observed time.

Total execution times are represented as multiples of the minimum observed running time, which was obtained when the threads were placed onto core pairs $\langle 0, 1\rangle$ and $\langle 2, 3\rangle$. Conversely, the worst execution times were observed when mapping to one core from each of the previous pairs, that is, when mapping to $\langle 0, 2\rangle$, $\langle 0, 3\rangle$, $\langle 1, 2\rangle$ or $\langle 1, 3\rangle$. These timings strongly suggested that cores {0, 1} are actually hardware threads within the same core, as are cores {2, 3}. This hypothesis was verified by using the operating system's official reporting mechanisms (as will be discussed in Chapter 4), as well as by comparing with the CPU topology generated using *hwloc* [Bro$^+$10].

As an aside, perhaps paradoxically, one finds that it is quicker to schedule two threads to the same single hardware thread than it is to place each workload on a different hardware thread in a different core. This is because the time spent performing the actual increment operation is far shorter than the time spent resolving cache invalidations. Consequently, scheduling to the same hardware thread is only $\approx 1.15$ times slower than the optimum running time, as opposed to $\approx 2.52$ times when running on different cores.

## 2.3 Attacker Model

The side-channel presented in Section 2.2 illustrates the fundamental characteristics and components of effective illicit channels.

The first is that an effective channel has a defined *purpose*, meaning that useful conclusions can be drawn by observing the respective phenomenon.

Next is that a channel has two or more *endpoints*, typically an attacker and a victim in the case of a side-channel, and that channels have a *range* (or *scope*) within which the endpoints must exist. In the described example, the endpoints are the incrementing threads, both of which had access to the same observations. Since the threads were co-operating, the example could be classified as a covert channel. Co-operation generally leads to covert channels being simpler to construct than side-channels.

The range, or scope, of a channel depends on the architectural element being attacked, and the mechanism of attack used. As will be seen in Section 2.5, confinements can be broken at different levels of a system architecture, such as the cache level (L1 [OST06], L2 [Xu+11] and L3 [YF14]), virtual machine level [Ris+09], system level [AKS07; WXW12], or network level [CBS04], through various forms of attack.

Finally, an illicit channel has a *mechanism of action*, the choice of which depends on the purpose of the attack and the location of the attacker relative to its victim (the scope).

### 2.3.1 Attack Orchestration

The origins of an illicit channel attack will vary depending on the actors involved. These, in turn, depend on the infrastructure being considered, and who controls the individual elements of the infrastructure.

Figure 2.2 summarises the relationship between the number of separate execution environments and the number of owners (or tenants) that have a stake within the environments, describing the common forms of deployments that result from combining the two variables. These topologies, and the threat of illicit channels to them, are described as follows.

**One environment, one owner** is a typical workstation or smart phone whereby there is a single operating system environment running over a machine owned by a single tenant. The primary attack vector arises from compromised processes or installed malware. Apart from the risk of side-channels, smart phones carry the risk of collusion between apps, which may form covert channels to circumvent the limitations imposed by permission systems, thereby leaking sensitive data [Cha+14; Mar+12].

**Many environments, one owner** refers to a network of machines owned by the same entity, such as an organisation. In this case, the primary risk originates from the use of illicit

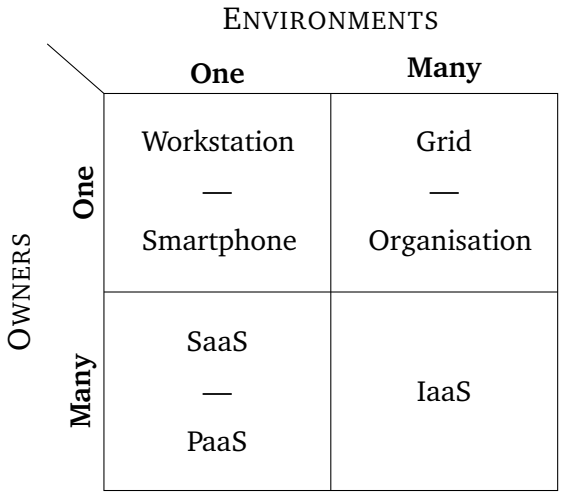



Figure 2.2: Common system topologies and their relation to the number of distinct owners and environments under their control.

channels to bypass data protection policies.

**One environment, many owners** as in the case of Software-as-a-Service or Platform-as-a-Service clouds. These carry the risk of espionage [Zha+14], and the methods available to an attacker vary depending on the control afforded by the cloud service.

**Many environments, many owners** as in the case of Infrastructure-as-a-Service clouds. Channels are again focused on espionage, yet the set of mechanisms at an attacker's disposal are different than the previous scenario. For example, side-channels in the cloud have enabled the inference of tenants' infrastructure and activity [Her+13; Ris+09], and memory contents.

While the principles explored in this work can be applied to all four scenarios, the investigation will focus on the mitigation of illicit channels in the latter. This is because it is effectively a superset, being susceptible to all of the attacks in the previous scenarios, in addition to attacks that can originate from outside of the victim's environment. Furthermore, the cloud infrastructures associated with IaaS are often large, and consequently lend themselves to the use of isolation and partitioning as a general mitigation strategy. In contrast, while partitioning can be applied to single-environment scenarios, these lack the wealth of distinct locations to which attackers can be banished, or within which potential victims can be isolated.

This work assumes a benign cloud provider that, at a minimum, will not actively try to sabotage a tenant's security, and would at best assist in the process of procuring isolation. A cloud provider is practically omnipotent, and can effectively mislead or deceive its tenants

in a multitude of ways, especially by intercepting instruction streams and providing false assurances of isolation to tenants. Conversely, a cloud provider can help a tenant procure isolation faster and with less waste if it can guarantee or enforce conditions on its hardware. For example, if a cloud provider can assure a tenant that it has sole ownership over its cores, then the tenant can procure core-level isolation for its processes without having to migrate to a different machine. A higher degree of co-operation between tenants and their cloud provider thus corresponds to more efficient and secure mitigations.

## 2.4 General Attack and Mitigation Strategies

The task of addressing illicit channels may at first seem daunting when one considers the sheer number of physical side-effects that processes have on their environment during their execution, and that each of these side-effects can potentially be correlated with the process' internal state. This problem is further elaborated when compounded with concerns of implicit data and control flows, where an observable system effect can be directly related to a program's data or execution.

On further analysis, one finds that not all channels are created equal, as they vary in their resilience, robustness, bitrate, practicality, and ultimately, utility. For example, an attacker may find that a side-channel that detects the presence of co-resident virtual machines on a public cloud [Ris+09] without distinguishing between tenants is of limited use, given that co-residency can effectively be assumed. Similarly, the cache-hierarchy distinguishing side-channel described in Section 2.2 may prove useless for an attack on a system whose virtual CPU cores are partitioned over disjoint caches. Thus, out of the set of potential phenomena that leak state, only a subset can be used to form viable illicit channels.

Finally, while the set of observable phenomena may be large, the techniques used to build the known illicit channels can be grouped into families of general strategies. The following section provides an overview of the general classes of attacks that have been observed, and the general forms of the defences that can be employed against them.

### 2.4.1 Attack Types

Attacks are characterised by type (side or covert), scope, bandwidth and feasibility. In terms of mechanism, illicit channels can be broadly categorised into *three* [VRS14] attack classes, as follows:

**Time-driven** attacks rely on measuring variations in the aggregate execution time of operations. These can be categorised into two types of attacks [KPMR12], namely:

- *passive* attacks, where the attacker cannot execute its code in the victim's environment, and
- *active* attacks, where the attacker can invoke or direct routines on the victim's machine.

**Trace-driven** attacks are based on analysing the transformation of the system's state as it evolves over a series of operations. This can be further subdivided into two attack classes [KPMR12], namely:

- *synchronous* attacks that allow for some degree of direct interaction with the victim process, such as the external triggering encryption operations, and
- *asynchronous* attacks that are carried out over shared hardware without directly exercising control over the victim process.

**Access-driven** attacks are possible when an attacker is co-located with a victim, and can detect and correlate the effects of a system's internal state to that of the victim.

*Control flow* can affect a program's timing characteristics, making possible attacks on mechanisms such as branch prediction [AKS07]. Similarly, *data flow* dependencies may expose timing channels through several avenues, including short-circuit expression evaluation, thread contention, and load bypassing [Cop+09].

### 2.4.2 Defences

Mitigations can broadly be categorised as being *passive*, *reactive* or *architectural*.

**Passive countermeasures** attempt to break locality sharing through an indiscriminate process. For example, disallowing multiple hardware threads from executing concurrently will eliminate a class of attacks [OST06] at the cost of performance. Alternatively, one can use a scheduling policy that only co-schedules processes belonging to the same entity [KPMR12; WL06] or *coalition* of virtual machines [Sai+05]. Scheduling policies can also be altered to limit their preemption rate, restricting the granularity of cache-level attacks [VRS14]. Other countermeasures include periodically flushing lower-level caches [ZR13], changing event release rates [AZM10a], and intercepting potentially dangerous operations (such as atomic instructions) through the hypervisor [SXZ13].

**Reactive countermeasures** attempt to detect and mitigate attacks as they emerge. Frameworks for distributed event monitoring [Mdh+13] can be fed events generated via introspection [DG+13], or can ensure that communication complies with a defined information flow

policy [Sai+05], possibly with notions of risk [JSS07]. Event-driven frameworks using profiling from different system levels to direct virtual machine migration have also been employed in the context of load-balancing [Woo+07].

**Architectural mitigations** may either refer to changes in hardware or to the way in which it is used. A notable example is Intel's introduction of specialised AES instructions, which insulate the operations' internal state from external caches. The problem with relying on hardware solutions is that it takes time for them to permeate into the mainstream, and that it would be unrealistic to expect every confinement-sensitive algorithm to be implemented in hardware [KPMR12]. Other solutions include randomly permuting memory placement [WL06], using oblivious data structures to hide memory accesses [GO96], rewriting programs to remove timing variations [Aga00; Cop+09], reducing the precision of system clocks [Hu91; OST06] or normalising timings [LGR13], locking regions of cache to specific processes by avoiding shared cache indices [KPMR12] and by partitioning virtual machines entirely through hardware [Kel+10].

#### 2.4.2.1 Confinement Types

Illicit channels occur either at the software or hardware level [Hu91], the former being a product of the algorithms used, and the latter emerging from the characteristics of a system's hardware. When reviewing the body of known attacks and mitigations, it becomes apparent that there are variations in the efficacy and generality of the established countermeasures combating attacks. In the case of hardware-based channels, these factors are strongly affected by the class of mitigation used, specifically, whether an approach uses a *soft* or *hard isolation* technique [VRS14] to separate an attacker from its victim.

**Hard isolation** entails that co-locations are broken by placing the parties involved at distinct and separate physical hardware locations, whereas

**Soft isolation** mimics the existence of a plurality of distinct hardware locations by arbitrating access to resources, hiding their hardware characteristics and reducing a given channel's information content.

Hard isolation places a strong physical boundary between an attacker and a victim, through what are generally passive elements of a system. For example, the effects of a process on a cache's access times may be hidden from an attacker by placing the latter on a different core that makes use of a different cache hierarchy [OST06]. Conversely, a soft isolation technique allows hardware to be shared between the attacker and victim processes, but will try to extinguish the effect over which a given channel hinges. For example, to counteract the

aforementioned cache-based timing channel using soft isolation, a victim process could attempt to flush its caches using a linear scan prior to its descheduling [OST06], thus masking the access pattern.

Soft isolations are typically only guaranteed with respect to one defined system attribute, and are upheld through active and ongoing processes, or through changes in policy. To elaborate on the previous example, a linear scan may be effective in breaking a cache-based side-channel, but may leave other core-level illicit channels intact. In contrast, migrating a process to a different core will automatically break all subsequent core-level attacks, bar any channels that can be formed using residual effects at the origin produced by its execution prior to migration. In this regard, hard isolation is more comprehensive, but is limited by capacity [LGR13].

Other than their generally narrow scope, soft isolations may require an upkeep, that is they can incur an overhead or lead to lower performance. In the case of the cache-timing obfuscation process, these overheads manifest themselves in the running time of the cleansing process itself, as well as the negation of the advantages of caching due to the forced eviction of memory.

In some cases, soft isolations have been improved or superseded by extensions to hardware, which serve to reduce the methods' speeds or strengthen the degree of isolation offered. For example, early implementations of x86 virtualisation incurred an ongoing and significant overhead due to dynamic binary rewriting [AA06], which has nowadays been drastically reduced through the adoption of hardware-assisted virtualisation. Similarly, software-based approaches to securing AES introduced overheads [OST06] that have been largely eliminated through the implementation of the cryptographic operations as a special hardware-level confinement [Gue10].

#### 2.4.2.2 The Soft/Hard Isolation Trade-off

In the absence of special hardware-level confinements, one must choose between using a (potentially) inefficient soft isolation or dedicating computational capacity to a task, the size of which depends on the scope of the attack. Consider the case of the `clflush` instruction, which flushes all cached versions of a given cache line. This instruction has been demonstrated to be an effective enabler of several cache-level side-channel attacks [YF14; Zha+14], due to its ability to flush all instances of a given cacheline from within a cache hierarchy whilst also avoiding many of the complications of mapping memory addresses to cachelines that an attacker would otherwise face.

Disabling the instruction would be a crude, yet effective, method for impeding such attacks. While `clflush` is an unprivileged instruction that does not generate a hardware trap [Zha+14], closer inspection of its semantics shows that its execution depends upon a

`clflush` flag within the machine's `cpuid` register being asserted [Int15a]. This register is normally immutable, yet a virtual machine would, at least in principle, be able to assign any arbitrary value to its virtualised counterpart [Lib]. Unfortunately, hardware-assisted virtualisation, such as that used by `KVM` [Kvma], bypasses the virtualised `cpuid` register, and `QEMU` will only consult the register when using a translation-based virtualisation method that foregoes hardware acceleration. While this was empirically proven by the author of this document to be effective in disabling `clflush` (an invalid opcode exception was thrown on invoking it during a test), a translation-based VCPU is substantially slower than its `KVM` equivalent, leading to a continuous overhead.

Given that a special-purpose CPU confinement that limits the use of `clflush` does not momentarily exist, and that the **L3** cache can lead to a machine-wide attack, an approach based on hard isolation would have to commission an entire machine to the process in question. As will be seen in Section 3.4, the notion of an upkeep cannot be directly carried over into the realm of hard isolation, as the ongoing computational cost of maintaining passive structural elements such as caches and cores is low. Instead, the primary concern is one of *utilisation*, where overhead is considered in terms of the computational capacity that is left unused during the fulfilment of an isolation requirement.

## 2.5 A Scope-Based Taxonomy

It is instructive to analyse specific instances of illicit channels and attempt to extract their commonalities. Consider the side-channel described in Section 2.2. The channel involves a set of *participating parties* (the attacker and victim thread), a *mechanism* (the timing channel), and *scope* (core or package-level visibility of timing effects).

The received wisdom is to categorise channels on the basis of their mechanism of action, yet this is of limited use when considering a partitioning approach to dissolving channels. Instead, the following exposition attempts to group channels by their *scope*, that is the medium through which the channel is formed.

### 2.5.1 Caches

*Caches* are a pragmatic solution to reducing the average time taken to perform memory operations. They exploit the common observation that real-world data often exhibits *temporal* and *spatial* locality [Ken86], where related data elements often appear close to each other within memory (for example, in the case of consecutive elements in an array), and are typically accessed within similar time frames.

When a processor attempts to access a memory location, it first checks for the element's presence in cache (a cache *hit*), with absence (a cache *miss*) triggering a fetch operation

from main memory, which is slower to service. This idiosyncrasy forms the fundamental principle with which many cache-level illicit channel attacks operate. For the purposes of understanding cache-based illicit channel attacks, one must grasp a few additional concepts, namely:

**Cache lines** The aforementioned principle of locality of reference, coupled with the bus widths of modern architectures, have led to the adoption of *cache lines* as minimum data transfer units. Thus, when requesting a byte of data from main memory, the system loads a block of N consecutive bytes into cache with a single fetch operation (N is determined by the architecture. For example, an Intel i7-4790 CPU uses 64-byte cache lines). This places a practical limit on the resolution with which an access-based side-channel can deduce memory access patterns, as adjacent addresses will produce the same side-effect on memory access times [MKS12].

**Hierarchy** Modern CPUs commonly employ a hierarchical cache architecture with multiple cache levels [Int16b], such as the one illustrated in Figure 2.1. Cache levels are typically denoted as `Ln`, `n` being the level in question, and lower values of `n` denoting smaller but faster caches. The level of a cache can be seen as indicating its proximity to a core, with data to be used by a core filtering down the hierarchy. Certain cache levels may be shared by multiple cores or hardware threads [URv03]. For example, the aforementioned CPU has a per-core **L1** and **L2** cache, and an **L3** cache common to all cores [Int16b]. The cache level affects the scope of an attack [Xu+11], and determines whether an illicit channel is confined to the same core, or can be formed amongst processes running on separate cores.

**Inclusivity** A cache hierarchy can be *inclusive* or *exclusive*. In the former case, the presence of a data element in a lower cache level implies that it is also cached within the higher levels. Thus, for example, if data is contained in **L1**, then it must also exist in **L2**. Conversely, an exclusive cache guarantees that a given data element will always be in at most one cache. Several cache-level attacks can only be carried out reliably on inclusive architectures. Inclusivity varies by hardware vendor and architecture, with most recent CPU architectures from Intel being inclusive, and AMD's offerings being exclusive [Ore+15].

**Eviction and associativity** As caches are smaller than main memory, it follows from the pigeon-hole principle that multiple cache lines in main memory are competing for room in cache. The system is thus faced with the problem of deciding which cache line should be *evicted* in favour of any new data that is being requested.

Caches typically exhibit *cache inertia*, meaning that a cache line is only evicted when an attempt is made to load a new line into it. Often, the policy for choosing between lines in a

cache line set is a variant of the *least recently used* (LRU) policy [AR14], where the first line to be removed is that which has not been accessed the longest [KPMR12].

A related concept is that of *associativity*, which refers to the number of separate locations within a cache to which a given block of main memory can be mapped. The degree of associativity can vary between caching levels [Int16b]. Associativity can increase the noise of an illicit channel by introducing collisions between elements in main memory and obfuscating the memory eviction patterns [MKS12].

**Prefetching** Contemporary architectures may employ additional optimisations such as *prefetching* [Int16b], which attempts to pre-load multiple cache lines based on the predicted memory access patterns. Prefetching may inadvertently hinder the establishment of illicit channels due to the additional noise and loss in accuracy when deriving a memory access map [MKS12].

#### 2.5.1.1 Attacks

Interest in the field of illicit channels has been recently rekindled though a seminal work that detailed a series of methods designed to infer AES encryption keys through cache-level attacks [OST06]. The methods described the exploitation of a feature of many implementations of AES algorithms, namely that they use lookup tables, with elements of the key being used as an index value for the initial round. By triggering an encryption via a known plaintext and observing the algorithm's memory access patterns, one could subsequently derive the key being applied.

Although much of the research in cache-level illicit channels is focused on their use in breaking cryptographic algorithms and stealing keys, in general, any algorithm whose memory access pattern depends on confidential information is at risk of leaking sensitive information through cache-based side channels [KPMR12; Zha+14].

A defining feature of cache-level channels is that they can achieve high bandwidths due to the speed of memory operations and the frequency at which they can be performed [Xu+11]. This can be contrasted with communication via hard-disk access modulation [LMS14], which is inherently slower. Fast channels may be facilitated further with the presence of accurate real-time counters, although timing information can also be acquired through alternate means [OST06].

Cache-level attacks are intimately related to the architecture being attacked. For example, certain synchronous attacks may only be viable on chip multiprocessors [OST06; Per05]. Passive timing attacks are difficult in the context of cache-level illicit channels due to an inability to control the victim process' cache evictions directly. A lack of probes further complicates the formation of an illicit channel due to the lack of local timing information. These

result in increased noise, which may in certain cases be reduced through statistical methods and multiple sampling techniques. In contrast, active attacks afford a greater degree of control over cache evictions, and result in heightened bitrates.

Trace-driven cache attacks analyse memory access patterns [KPMR12; NS07]. Attacks such as PRIME+PROBE [OST06] are also active, as they require the direct manipulation, or *priming*, of the victim's cache. Synchronous trace-driven attacks allow for some degree of direct interaction with the victim process, such as by triggering encryption operations. Conversely, asynchronous attacks are performed on shared hardware without relying on control over the victim process [OST06]. Such attacks often rely on an attacker's ability to execute code during a sensitive process, recovering information on the process' intermediate stages. Asynchronous attacks on single-threaded systems have been demonstrated, with attackers relying on knowledge of the underlying process scheduling algorithm to execute their code at the appropriate stages [KPMR12].

#### 2.5.1.2 Mitigations

The following is an overview of the several soft and hard isolation-based mitigations that have been proposed for combating cache-level illicit channels.

**Data Structures and Memory Layouts** An elementary mitigation against cache-based attacks would be to not use the cache, that is, to remove operations on main memory, rendering attacks through cache profiling irrelevant. In the case of securing AES, accesses to cache may be removed by replacing table lookups with computations, at the cost of performance [OST06]. A slightly more permissive approach is to use registers as caches, keeping values local to the core, yet registers tend to be very finite, and their number and width varies by architecture. Similarly, attacks may be partly mitigated by compressing lookup tables and shrinking their memory footprint. This increases the odds that memory segment will be accessed completely, rendering traces less informative. This approach is based on probability, and would merely slow down an attack. In addition, it is not very effective against asynchronous attacks [OST06].

Rather than modify the underlying data structures, one may opt to change (or *fuzz*) the access method. For example, modifying every table lookup operation to always scan linearly through the entire table would obfuscate the relevant element's index [Pag02]. As an optimisation, one can read a single element from each cache-line block, reducing the number of reads that must be performed to normalise the entire cache. This must be employed with some caution, as certain architectures exhibit timing differences when accessing values within the same block [Cop+09]. Fuzzing can also be used to add random delays to sensitive operations, or to normalise operations to some maximal value, although this would slow

down the process to its worst-case execution time. Reducing the resolution of the system's clock can also hinder tracing [LGR13], yet an attacking process may acquire similar timing information through alternative means, such as by executing a separate timer thread with an internal counter [OST06].

**Hardware** A simple method of hindering several fast attacks is to disable simultaneous multithreading (SMT), disallowing multiple hardware threads from executing concurrently over the same low-level cache and limiting the resolution of synchronous attacks [OST06]. Hardware threads typically share a great degree of state, including caches. By removing concurrency, one removes the risk of having an attacker's thread executing in parallel with the sensitive process on the same core. This comes at the cost of computing capacity, and thus, performance.[1]

On certain architectures, there exists the option of disabling caches entirely, with reads and writes being committed directly to memory. This drastic course of action would negate the benefits of caching. A less aggressive alternative, available on some architectures, is to place caches in a *no-fill mode*, where reads are performed from cache, but subsequent cache evictions are disabled. Thus, in the context of AES, one could load the lookup tables into cache and enter no-fill mode, preserving the speed-up associated with caching during reads whilst stopping external processes from forcing cache evictions. This would require support from the kernel, as well as a way of marking privileged processes. [OST06]

Finally, extensions to a machine's active traits (Section 1.3.1), such as the AES-NI [Gue10] extensions, can counteract algorithm-specific illicit-channels by creating special-purpose hardware confinements.

**Operating System Support** Rather than permanently disabling SMT, a security-sensitive process may opt to temporarily disable SMT at runtime. This would require support at the machine level. Alternatively, an operating system may leave the hardware thread active, yet opt to never schedule processes to it, effectively achieving an equivalent result (a similar concept will be explored in Section 4.2.3). The problem with such an approach is that it could lead to the enabling of denial-of-service attacks, especially if processes are able to disable SMT directly through system calls [OST06].

A more relaxed and less debilitating alternative is gang scheduling, where only threads from the same originating process are ever co-scheduled [KPMR12], allowing processes to make use of multithreading. Under the assumption that a process will not attempt to attack itself, such a policy would not present a higher risk to the process.

[1] Disabling hyperthreading was once common amongst cloud providers [WXW12] for a variety of reasons, chiefly related to power efficiency and bean counting, although Amazon EC2 has recently foregone this practice [Ama15].

Rather than addressing the problem as one of scheduling, one can also attempt to tackle the root cause of the attacks, namely the cache behaviour itself. Operating systems can be extended to partition memory into non-interfering sets using *page colouring* [Erl07]. This service can be offered by the operating system in the form of a dynamic call that allocates regions of *stealth memory* dynamically using page table alerts, automatically flushing cache-lines between context switches amongst processes owned by different entities.

### 2.5.2 Operating System Environments and Machines

Operating systems and physical machines necessarily carry within them an abundance of state. A core activity of operating systems is to limit any given process' view of this global state, yet some sharing will invariably remain.

#### 2.5.2.1 Attacks

Modern operating systems provide processes with many facilities, including memory mapped I/O and hardware access, as well as metrics such as a process' CPU or memory consumption. These are all potentially subject to illicit channel attacks. For example, meta-data attacks [Smi+06] can be used to conduct industrial espionage using tools as basic as `ps`. The types of meta-data attacks available vary based on the operating system's process-facing interface. Thus, for example, an attack with which one can infer a process' GUI state may only work on certain mobile devices [CQM14], as desktop-class operating systems may lack the analogous metric or process behaviour.

Beyond meta-data attacks, a process can have measurable and observable effects on a machine's subsystems. For example, a program's control flow directly affects mechanisms such as *branch prediction*, which in turn affects computation times. Consequently, branch prediction has been used to extract bits of a private cryptographic key within a single encryption by observing the timing effects of the instruction pipeline [AKS07].

Similarly, a program's data flow can influence timing by creating data dependencies between instructions, or by changing a program's execution time in the case of instructions whose execution varies based on its operands' values (such as certain implementations of `DIV` on Intel processors that use early exit). Data flow dependencies also arise between registers and memory. For example, storing and loading values to the same location would cause an out-of-order processor to assume the existence of a data dependency between the elements. *Load bypassing*, especially implementations which only consider parts of the address of the location being loaded, induce further irregularities in memory operation times. Contention between threads also leads to inter-thread timing dependencies. [Cop+09]

#### 2.5.2.2 Mitigations

**Sandboxing** Sandboxing, in its most general sense, refers to the encapsulation of a process within some form of restricted or seemingly isolated environment. *Process containers* [Lxc] have emerged as a robust mechanism for isolation, being particularly effective in the elimination of meta-data attacks through their use of namespaces. The applicability of containers depends on how a deployment is structured, and for maximum effect requires the processes being contained to be decoupled from each other. Note that moving a process to a completely different operating system and machine and letting it execute on its own could be seen as an extreme form of sandboxing.

Combined operating system and hardware support may also allow a process to efficiently move regions of memory to different addresses. As trace-driven attacks profile specific regions in memory, the system would complicate data acquisition by migrating the sensitive region on each access. Migration could, for example, be implemented through dynamic page tables, or by using multiple copies of pages and alternating accesses. Alternatively, the system may permute memory elements within the sensitive section itself. Unfortunately, these approaches require a significant degree of hardware support, and tend to be very application-specific. [OST06].

**Program Transformation** Control-flow illicit channel attacks can be neutralised by eliminating the unbalances brought about by program branching. One option is to transformation a program and to render it compliant with the *program counter security model* [Cop$^+$09].

The program counter security model considers an attack where the only information leaked is the program counter's values during execution [Mol$^+$06], from which an attacker can then determine which branches were taken, and consequently how conditional expressions were evaluated. If the branches of a conditional statement took different times to execute, and the condition depends on a secret key, then one may be able to infer knowledge about the key through a timing attack. Conversely, a timing attack will not work if the program's control flow does not depend on a secret key and if execution time is only dependent on control flow.

Coppens *et al*. [Cop$^+$09] achieve this through *if-conversions*, where the branches of a key-dependent `if` are modified so as to reduce the variance in the execution times of branches, as well as in branch prediction. The core of the construction relies on predicating each instruction in a branch. A fully predicated instruction set, such as that of the EPIC architecture [SRM00], readily accommodates such transformations. However, the x86 architecture only has predicated `MOV` instructions, making the transformation less straightforward. The approach should only be applied to secret keys, which are delineated using `pragma` directives, so as to minimise unnecessary overheads [Cop$^+$09]. Their approach used LLVM to ensure

that the transformation happened beyond the optimisation stage, which would undermine the conversion.

Beyond the complication of having to tag the security sensitive elements of a program, the method has a drawback in that it must be revised with each processor generation, as micro-architectures, and consequently timing behaviours, can change [Cop+09]. Performance can also be impacted, particularly in the case of loops with terminating conditions that depend on a secret value, as these necessitate iterating up to the loops' upperbounds.

Other application-specific program transformations exist. For example, with *algorithmic masking*, one may obfuscate data-dependent operations by applying random transformations to sensitive data. An alternative heuristic is to add noise via spurious memory accesses, such as by running several dummy encryptions in parallel with the true encryption. [OST06]

### 2.5.3 Virtual Machines and Mixed-Level Attacks

Cloud computing presents a somewhat curious security landscape, as it potentially places rivals on a shared platform, as opposed to the comparative safety of an enterprise's internal network. This creates an atypical attack surface, with attacks being launched horizontally across environments, rather than from within a victim's server environment.

Virtualisation is very commonly used to confine the activities of individual tenants and limit interference. While the coarse-grained nature of virtualisation and developments in computer architecture (such as per-core caches, complex memory prefetching protocols and hardware-implemented AES instructions) have complicated the task of reliably forming certain types of side-channels across virtualised platforms [MKS12], breaches are still reported [Ira+14; YF14; Zha+12a]. In addition, virtualisation can enable new classes of illicit channels that act directly on the virtualisation platform itself.

#### 2.5.3.1 Attacks

Cloud providers generally provide tenants with an abstracted view of the infrastructure over which they are executing. Thus, while data centres may correspond to broad geographical locations, tenants are not given the precise location at which their virtual machines execute.

Nevertheless, certain indicators may allow a virtual machine to extract information about its underlying infrastructure. For example, attacks have managed to co-locate specific virtual machine instances within the Amazon EC2 cloud via a series of heuristics and side-channels [Ris+09]. As EC2 is based on Xen [Bar+03b], co-location was verified by comparing the `dom0` IP address visible to each machine, and further reaffirmed using hard-disk based covert channels. Additional methods for confirming co-location were also identified, including the use of network-based side-channels and by timing the response times of com-

munication over external interfaces. The risk introduced by being able to reliably co-locate virtual machines with targets is that it enables the deployment of subsequent attacks.

Other attacks may be used to detect the rate of web traffic on an adversary's machine, aiding espionage [Ris+09]. By probing the virtual machine's load whilst subjecting the victim to input, one may infer a correlation between computational load and traffic. Load measurements can also potentially be used to record keystroke timings from interactive secure shell sessions. Combined with keyboard usage models, one could determine the input with varying degrees of uncertainty based on the fingers' travel times. Admittedly, this attack was demonstrated under laboratory conditions, and may not necessarily be feasible in a cloud setting due to the associated noise [Ris+09].

Covert channels in the cloud can be used to clandestinely export data from one tenant's machine to another's [WXW12]. Knowledge of the underling scheduling algorithm can used to form a channel by modulating the time for which a machine [OO10] or process [Hu92] is scheduled. For example, two co-located virtual machines may communicate via load-based channels, such as through the *Covert Channels using CPU loads between Virtual machines* (CCCV) scheme [OO10]. CCCV games the Xen scheduler by encoding a message through the modulation of a process' CPU load. A second process executing at the corresponding target can infer the message based on the length of the scheduling quantum allocated to it. For multi-core architectures, the sender and receiver endpoints may deploy multiple communicating processes, in an attempt to land a pair of sending and receiving processes on the same core, relaying a message between groups.

A similar covert channel can be formed by having the sender process invalidate cache lines, making a measurable impact on the receiver's memory access times [Xu+11]. The magnitude of the performance impact can be directly controlled by the number of cache lines that the sender invalidates, allowing more complex encodings. Other covert channels may attempt to attack the system's data bus to modulate memory access times [WXW12].

An advantage of load-based side channels is that they do not require any elevated privileges on the system, as load can be modulated by changing the volume of computations. In addition, loads can be very easily and finely controlled, with synchronisation being simpler to achieve, especially when compared to side-channels formed through other mechanisms such as page-fault rates [OO10].

Finally, one potential source of illicit channels endemic to virtualisation is memory deduplication [HPSP10]. Since tenants often have significant portions of their environment in common, deduplication can tangibly reduce memory consumption, yet it also enables attackers to infer memory contents of their co-located tenants by checking for collisions.

#### 2.5.3.2 Mitigations

**Isolation** The most direct way of combating attacks crossing virtual machines is to isolate each tenant to its own physical machine [LGR13; MSR15; Ris+09]. The principal criticism against such an approach is that it is, in the general case, wasteful.

As a compromise, a mitigation may opt to periodically isolate different tenants on a temporary basis, yet one must decide when isolation should be employed. For example, a virtual machine may be temporarily migrated in response to a perceived attack, such as on detecting suspicious cache activity [Zha+11].

The approach becomes harder to apply when adversarial behaviours are either inadequately characterised, or cannot be efficiently detected. One option is to periodically migrate virtual machines indiscriminately as part of a *moving target defence* [MSR15]. Using virtual machine migration and game theory, a fair scheme was developed that denies malicious virtual machines enough time to conduct an attack through continuous migration [Zha+12b]. This scheme considers the scenario where a secret value is split amongst a set of $M$ virtual machines, each of which privately stores its part of the secret. The original secret value cannot be inferred from a single virtual machine's private store, and can only be reconstructed by combining $k$ machines' values, $k$ being a constant chosen when the secret was split. Given that the secret has been split amongst $M$ virtual machines, a malicious virtual machine would have to compromise $k$ machines[2].

To offset the penalty of virtual machine migration, the scheme uses game theory to determine an incentive that is given to tenants that opt into performing a migration. The scheme relies on a number of assumptions, namely that the migration operations are secure and incur a constant cost, and that the cloud provider has the capacity to hand out the promised rewards. [Zha+12b]

While periodic migration may obfuscate attacks, a malicious virtual machine may still be able to predict tenant placement given enough time. From a performance perspective, migrations should ideally occur infrequently, yet the longer a virtual machine lingers at a single location, the greater the chances of information leakage. Thus, the system must be able to determine an optimal interval for migrations which is shorter than the attacker's setup time.

Finally, certain high-frequency and high-resolution attacks can be foiled by simply setting a lower-bound on the minimum scheduling quanta's size [VRS14]. While this soft isolation

[2]Secrets are split using Shamir's secret sharing [Sha79] technique (a *threshold scheme*). A polynomial $f(x)$ of degree $k-1$ is first defined with random coefficients with the secret to be shared appearing in the polynomial as the constant term. The polynomial is then evaluated for each integer $i \in [1, M]$, with each virtual machine $i$ being given the point $(i, f(i))$. Given $k$ points, one can then reconstruct the polynomial $f(x)$, recovering the constant term.

approach does not completely nullify illicit channels, it can greatly diminish the information content of many incarnations of cross-VM cache attacks.

**Hardware partitioning** One approach to reducing potential avenues of attack on the virtualisation platform is to make it smaller and use strict hardware partitioning. *NoHype* [Kel+10] is an extreme expression of this principle, whereby virtualisation is performed entirely in hardware, bar the use of a *cloud manager* to handle administrative tasks for cloud deployments.

Mediation to resources in NoHype must be done entirely through hardware. For a user in a cloud setting, hardware is typically limited to cores, main memory, permanent storage and a network interface. Under NoHype, cores are pinned to virtual machines on initial bootup, and regions in memory are allocated to virtual machines through partitioning using *hardware assisted paging* (such as Extended Page Tables for Intel VT architectures). Devices such as a network card may be used through virtualised device support, which exposes the device through multiple, virtual interfaces, or generally via IOMMU support.

NoHype eliminates several classes of cache-level attacks by never allocating fractions of a CPU to a virtual machine. This is justified by asserting that one core is an insignificant unit of computational capacity in modern architectures, moreso fractions of a core. A distinct advantage of this allocation policy is that active attacks on L1 cache are avoided entirely.

The architecture also fends against memory access violations through paging, and page tables can only be compromised by subverting the cloud manager. Registers are also hidden as there is no underlying hypervisor executing throughout the virtual machine's lifetime, avoiding cross-VM leakages. Side-channel bandwidth is also reduced through private L1 caches, I/O rate limiting and fair access to memory. [Kel+10]

Resource partitioning and limiting communication between virtual machines in a structured manner is non-trivial. *sHype* [Sai+05] extends the Xen hypervisor by allowing the regulation of explicit data flow and limiting covert channel capacity by simplifying resource management, allowing administrators to define security policies dictating sharing using a variety of property languages.

Policies in sHype can be defined in terms of individual virtual machines, or named groups (*coalitions*) of virtual machines executing at the same security and access levels. These allow the definition of *Mandatory Access Control* (MAC) policies that are translated into reference monitors, which mediate security-critical operations which may leak information across virtual machines. For example, MAC can be used to regulate Xen's inter-machine communication mechanisms, namely event channels and shared pages. Apart from mediating all security-critical operations, reference monitors should also be tamper-proof as well as minimal, the latter rendering them more amenable to verification. Verification is paramount, as

MAC domains form part of the *Trusted Computing Base* (TCB), and constitute a security risk.

Security-critical operations are defined by inserting *security enforcement hooks* that delineate regions within the hypervisor that allow cross-VM sharing. The use of hooks is guarded by MAC policies. Hooks are *functionally transparent*, executing if legal, and failing with an error code otherwise. In sHype, security enforcement hooks are defined for overt communication channels [JSS07], namely domain management operations (such as suspending and migrating virtual machines), inter-domain event channel communication and shared memory pages.

MAC for sHype may still admit covert channels, as they remain hard to identify and prevent. Thus, policies are extended to include notions of risk with *risk flow policies* [JSS07]. Assuming that covert communication may only be carried out between concurrently executing virtual machines, risk flow analysis ensures that virtual machines are never scheduled in such a way that information may leak. First, one defines the possible overt flows between entities, followed by potential covert flows. Risk flows are consequently defined as the union of such flows. Information may cross between flows transitively, restricting schedules based on which flows have already occurred. Flows may be restricted by labelling *conflict sets* which partition virtual machines.

MAC and risk flow allow the definition of various policies. For example, the Chinese wall security model was formulated as a policy [JSS07], stating that an object can only be read if a prior object in its permission group has already been read, or if nothing has been read to this point. This implemented *freedom of choice*, as the system locks into parts of the policy based on the initial state chosen. The basic Chinese wall policy does not support unidirectional flows, and may thus be reformulated as the *Aggressive Chinese wall* policy, defining conflict sets for each machine.

In general, Chinese wall policies were found to be too restrictive [JSS07]. An alternative approach to defining policies that was analysed was the use of the *Bell-LaPadua* model, which can be used to define lattice structures allowing information from low-security zones to leak into higher-security zones, but not vice versa. A shortcoming of this approach is that one is only able to specify the most limited level of risk. A similar lattice-based policy that was analysed was the *Caernarvon policy*, which allows subjects flows within a range of labels defining lower and upperbounds on segments within a lattice.

The work concluded with three primary observations [JSS07]. Firstly, as risk flow may span across systems, MAC has to be coordinated between the different participants. Next is that strict partitioning may hinder a system's execution by denying applications legitimate access to resources, an issue which may further be aggravated by the choice and restrictions of the policy language used. Finally, risk flow policies for a system cannot intersect, as this would signify that information may leak transitively between partitions.

**Memory partitioning** The abundance of attacks that attempt to subvert virtual machine confinements by targeting memory have led to the development of a number of tailored countermeasures. For example, *StealthMem* [KPMR12] provides virtual machines with private memory pages that map uniquely to lines in last level caches (L2/3), thus eliminating the ability for attackers to force cache collisions. This is achieved by locking a *stealth page* of a virtual machine into shared cache, forbidding other virtual machines from paging it out. Given the principle of cache inertia and predictable cache associativity, modifications to a cache set can be regulated by restricting access to each index's pre-image set. A virtual machine, or a coalition of machines, may then be allocated sole access to colliding pages. Reserving lines in cache contracts the cache's size, which will affect performance, and limits its applicability to L1 caches.

Internally, StealthMem binds physical page tables to cores rather than virtual machines. Thus, StealthMem must also save and restore stealth pages during vCPU switches, copying the contents of a stealth page and ensuring that the pinned stealth page's contents are flushed out from cache before the next virtual machine is scheduled. Another consequence of having stealth pages pinned to cores is that processes will see different stealth page contents depending on which vCPU they are running. While this may be inconsequential for lookup tables (for example, AES tables would remain identical to all processes), migration will be hindered in cases where processes write to the stealth page.

The approach of StealthMem is similar to page colouring [Raj+09], yet the overheads of the former are significantly smaller than those of the latter, which grow with the number of virtual machines [KPMR12].

### 2.5.4 Networks

Most machines form part of a local or a wide-area network, which can serve as a medium over which illicit channels can be formed. Networks enable *remote* attacks, whereby a process on one machine can form a channel with a second on a different machine.

#### 2.5.4.1 Attacks

Illicit network-level channels broadly fall under the category of storage or timing channels [CBS04]. For example, covert storage channels may hide information in unused packet header fields or as steganographically obscured data encoded in the payload [HS96]. Channels based on unused header fields are of questionable reliability, both because they are an established avenue of attack, as well as the fact that the handling of such fields varies across routers and network interfaces.

Network timing channels are far more insidious, as they are formed through measuring or

modulating the order, frequency and timing of packets as they are transmitted [CBS04]. For example, timing side-channels have been used to infer secret encryption keys [BB03; Ber05]. More recent attacks are also capable of inferring a connection's internal state and sequence numbers, and whether a given pair of hosts is communicating over a TCP channel [Cao+16]. Covert channels can also leverage timing information. For example, a *packet sorting channel* encodes messages into the order in which packets arrive (the true packet order is defined by the sequence number in packet headers). The arrival of out-of-order packets is not an unexpected occurrence under normal operation, thus complicating the detection of such a side-channel. Similarly, a *general timing channel* operates by alternating between sending bursts of traffic (such as a succession of `ping` operations) and no traffic, and must be detected through traffic analysis [CBS04].

#### 2.5.4.2 Mitigations

Illicit channels at the network level, while considered dangerous, are susceptible to noise via traffic congestion and poor quality of service guarantees [CBS04]. Intermediate packet processing and congestion control procedures, such as batching and caching, alter the timing properties of a transmission, and degrade a channel. The principle of batching can also be pursued as an active countermeasure to normalise transmission times [AZM10b; Gor+12]. Note that this will not completely eliminate timing attacks, rather it can be used to lower any potential channels' bitrates and render them impractical. Conversely, a traffic normaliser that sanitises traffic and standardises all unused or redundant fields in headers would completely eliminate a class of storage channels [CBS04]. *Packet sanitisers* may also be employed to remove sensitive data which crosses from high to low security levels.

Network jitter can cause bits to arrive outside of their expected sampling window, introducing errors and breaking synchronisation. Ultimately, a transmitter must also be able to communicate the interval times being used, although these may be decided through different channels. Countermeasures to synchronisation loss include *start-of-frame* synchronisation packets sent by the transmitter prior to transmission, and *silent intervals* with no packet transfer (particularly during periods of heavy network load). Receivers may apply *interval adjusting* by modelling the network conditions under ideal conditions and comparing them with actual reception times, deriving time offset values and applying them as necessary. Alternatively, the transmitter may attempt to monitor incremental changes in the network and adjust its speeds based on the feedback loop formed. [CBS04]

## 2.6 Conclusion

The key observation of this chapter is that illicit channels are formed between entities through a shared medium. The position of two entities relative to each other determines the type of illicit channels 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. A general countermeasure against illicit channels is thus to break co-location, or isolate the given endpoints from each other. In this regard, isolation means that the endpoints should no longer share their communication medium.

Isolation within a cloud infrastructure is a scarce commodity due to the cloud's incentive for consolidation. Thus, the heavy-handed approach of isolating every tenant to its own dedicated machine would be wasteful and uneconomical. Instead, a more nuanced approach to isolation should be adopted, whereby every medium is addressed using its own interpretation of isolation. This culminates into the concept of *fine-grained isolation*, which forms the focus of the upcoming chapter.

CHAPTER 3

# MODELLING LOCALITY AND MIGRATION

COMPUTATIONS ARE NOT PERFORMED IN THIN AIR, much as though the cloud tries to hide it, and the increased sharing of computational resources elevates the risk of illicit channels. This creates a demand for efficient mechanisms that can effectively isolate particular computations. Modelling serves to capture the intricacies of sharing within complex infrastructures, and provides the basis over which subsequent mitigation strategies can be built.

## 3.1 Introduction

To devise a comprehensive strategy against illicit channels, one must first derive order and structure from within the zoo of seemingly disjoint attack and defence techniques.

While the channels' mechanism of action is perhaps, at first glance, the most obvious characteristic on which to base a taxonomy of exploits, it is not necessarily the most useful in the context of attack mitigation. Specifically, grouping by mechanism of action limits the capacity for abstraction and uniform reasoning. For example, by grouping all timing channels into a single class of attacks, one is limited in the number of comprehensive mitigations that can be applied effectively to each member of that class, as each timing channel has its own intricacies and will generally require its own mitigation. In addition, such a grouping eliminates all sense of *scope*, that is it hides the range of the illicit channels, and gives no indication as to which processes can communicate with which. Conversely, fracturing the class into an agglomeration of individual illicit channels removes any advantage of grouping, as each channel will have to be counteracted using a tailored mitigation.

It becomes apparent that an approach based on partitioning and isolation would benefit from a characterisation of illicit channels that captures scope. This is because hard isolation will affect all illicit channels at the granularity or medium at which the isolation operation is procured. To illustrate, while fuzzing a timing channel on a machine may leave a cache-based attack intact, isolating the offending process to a different machine will break every machine-wide illicit channel that can be formed with other processes at the origin. This is not to say that a better understanding of an illicit channel's behaviour would somehow be

a detriment. On the contrary, a fuller understanding of the attack surfaces and vectors that afflict a system enables the directed use of soft isolation, which can serve as an effective, if less comprehensive, mitigation against specific illicit channels.

As evidenced by Chapter 2, illicit channels can be formed at various granularities and scopes. More interestingly, the hierarchical nature of modern computer systems and infrastructures is mirrored by illicit channels, where the scopes of channels appear to be delimited by common system boundaries. Thus, for example, one finds attacks that can form channels amongst entities sharing the same core, cache, physical machine or network, amongst other confinements.

Classifying illicit channels into a hierarchy of scopes is not simply an exercise in taxonomy, rather it allows the application of two principles during dynamic isolation and reconfiguration, which in this document are termed *cascading* and *reconfiguration minimisation*.

**Cascading** is the notion that isolating a process to break a channel with a wide scope can also have a knock-down effect and destroy channels with a narrower scope. For example, consider the scenario where two virtual machines are sharing a physical machine. Migrating one of the VMs to a different machine would prevent the formation of machine-wide illicit channels, and would also remove the risk of cross-VM cache-level attacks.

**Reconfiguration minimisation** is a related concept, whereby a mitigation attempts to provision the smallest amount of isolation necessary in order to break a given channel. With reference to the previous example, if a system only needs to be protected against cache-level attacks, then one may opt to only isolate at the cache-level, rather than attempt to isolate entities at the coarser-grained VM-level and relying on cascading.

Both principles are based on more general concepts of *locality*. For instance, when procuring isolation at runtime using migration, one finds that the performance impact of migration grows with the size of the structure being migrated, as well as the distance between the source and destination. The full effects of this phenomenon will be quantified in Chapter 4, yet even intuitively, one would for example expect a virtual machine migration from one machine to another to take longer than repinning a process to a different core on the same machine, as this would require a larger state transfer over a greater distance. In this regard, the principle of reconfiguration minimisation would be to keep migrations as local as possible. Note that minimisation must be taken within the context of the entire system. Specifically, it may transpire that several locally-scoped reconfigurations can be subsumed by a single migration at a higher level with cascaded effects, and that the latter will have a lower performance impact on the system than executing each of the former reconfigurations individually. As will be seen in this chapter, not all isolation operations have a cascading effect, as this depends on

the relative positions of the endpoints of the channel in question. Thus, for example, isolating a virtual machine will still preserve the internal process structure, meaning that an illicit channel formed between processes within the virtual machine cam be reconstructed at the destination if the channel's medium lies within the machine.

Reasoning about isolation and dynamic reconfiguration is non-trivial, particularly when considering modern systems of networked machines. This chapter explores the development of a holistic model of locality and isolation that describes the movement and isolation of computations within hierarchical systems. This model serves to rigorously define *co-location*, which is a pre-requisite for the construction of illicit channels defined by the channels' scope, as well as formalise the use of *migration* to reconfigure systems dynamically in order to provide a requested level of isolation.

Several additional demands must be fulfilled by the model for it to be considered holistic with respect to its ability to represent the attacks and mitigations explored in the previous chapter. First, it must be able to express the fundamental notions of confinement and locality sharing, which serve to define whether or not a given computation is isolated relative to other entities within the system. This must allow both soft and hard isolations to be handled uniformly. In addition, so as to correctly reflect the hierarchical nature of modern systems, it must also be capable of expressing *nested* confinements.

As scheduling and placement play a central role in co-location, the model must 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.

The ability to simultaneously model different parts of a system using different granularities can lead to an improvement in hardware utilisation, as fewer resources are committed to providing isolation guarantees. Being able 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.

## Chapter Outline

This chapter is structured as follows:

**Section 3.2** defines the fundamental notions of confinement, containment and co-location, with which illicit channels can be described.

**Section 3.3** describes how *agents* can be used to dynamically manipulate a system model, reconfiguring a system using local and global migration operations.

**Section 3.4** explores different ways in which a model of a system can be analysed and evaluated.

**Section 3.5** defines the considerations that must be made when choosing the subjects and targets of migration operations.

**Section 3.6** demonstrates the model's application to the modelling of several illicit channels and channel mitigations.

**Section 3.7** concludes this chapter.

## 3.2 A Hierarchy of Isolation

Modern computer architectures consist of a multitude of logical and physical environments. For example, an operating system is an environment which controls the physical resources that have been assigned to it. The resources comprising an environment may be further sub-divided into constituent groups of finer-grained environments, forming a hierarchy. Expanding on the previous example, an operating system generally contains several process environments that it manages and schedules.

A fundamental task in security is to enforce boundaries on environments, regulating access to their underlying resources. This is done through a variety of mechanisms, such as memory protection and context switching. Thus, environments also serve as *confinements*, with different environments placing restrictions on access to their internals. Alternatively, they can be seen as *isolations*, with the effects occurring in one being invisible to the other.

### 3.2.1 Confinement and Containment

The fundamental unit of computation and boundary delineation in this work is the *confinement*, defined as follows.

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

A confinement's name is typically dictated by its type, and serves to identify it from amongst its siblings. It is assumed that confinements can be uniquely identified by their name. In the case of conflicts and duplicate names, one can either associate a generated unique identifier to the confinements through an additional preprocessing step, or differentiate between confinements using the prefix of containments leading to them.

Capabilities are used to limit how confinements can interact and modify each other, as will be seen in Section 3.3.1. The capability set can be omitted when it is empty.

The notion of confinement automatically entails one of *containment*, where a confinement X contains Y when Y is a sub-confinement of X. Since Y is itself a confinement, this enables the representation of hierarchical containment. For example, processes execute within the confines of a CPU, and multiple CPUs are confined to a single machine, which can itself form part of a network.

**Definition 2** (Containment)**.** A confinement X is *contained* within a confinement $\mathbf{\Gamma}{:}\mathrm{D}(\mathtt{C})\ [\mathrm{S_B}]$ if $\mathrm{X} \in \mathrm{S_B}$. This is denoted as $\mathrm{X} \in \mathrm{D}$.

Illicit channels exploit the fact that certain confinements are imperfect, and do not keep their sub-confinements completely isolated from each other. An illicit channel is thus formed when members of a confinement can communicate with each other over an unregulated medium. Thus, imperfect 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. The principle of containment can thus be extended to express *co-location*, which is defined as follows.

**Definition 3** (Co-Location)**.** 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. For example, a thread confinement running within a process confinement may communicate with another thread via the process' parent operating system. This is expressed through an extension of the basic co-location and containment predicates, in the form of the *nested containment* and *nested co-location* predicates, which are defined as follows.

**Definition 4** (Nested Containment)**.** A confinement X is *recursively contained*, or *nested*, within a confinement D (denoted as $\mathrm{X} \in^{+} \mathrm{D}$) if it is contained within any of its sub-members, that is, $\mathrm{X} \in^{+} \mathrm{D} \overset{\mathrm{def}}{=} \mathrm{X} \in \mathrm{D} \vee \exists \mathrm{D}' \in \mathrm{D}.\ \mathrm{X} \in^{+} \mathrm{D}'$.

**Definition 5** (Nested Co-Location)**.** A confinement X is *recursively co-located* with Y via a confinement D (denoted by $\mathrm{X} \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{Y}$) if they share a common ancestor. This can be expressed formally as $\mathrm{X} \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{Y} \overset{\mathrm{def}}{=} \mathrm{X} \in^{+} \mathrm{D} \wedge \mathrm{Y} \in^{+} \mathrm{D}$.

Together, these predicates can be used to model various architectures and attack scenarios. The following is a simple example of how one can model a cache hierarchy, and how co-location would manifest itself in such a model.

**Example 2** (Parallel Execution)**.** Consider a CPU package with two cores (**C**) sharing an **L3** cache, with each core having its own **L1** and **L2** cache. Moreover, each core employs *simultaneous multithreading* (SMT) (or *hyperthreading*, in the case of Intel machines) and

exposes two hardware threads, which share a given core's **L1** and **L2** caches. This can be modelled as:

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

Figure 2.1 illustrates the nesting of confinements specified by CPU. Two processes X and Y can be susceptible to an attack via an **L1** cache [OST06] if

$$\exists \mathbf{L1}{:}\mathrm{L} \in^{+} \text{CPU}.\ \mathrm{X} \not\Leftrightarrow \mathrm{Y}$$

or via the **L3** cache [Zha+14] if

$$\exists \mathbf{L3}{:}\mathrm{L} \in^{+} \text{CPU}.\ \mathrm{X} \not\Leftrightarrow \mathrm{Y}$$

For the given hierarchy, the latter will hold whenever processes execute simultaneously. □

Note that while the physical proximity between an attacker and victim (or equivalently, the depth at which the two entities appear within the model) can determine the feasibility of a channel [Gur+14; Gur+15], it does not necessarily imply that an illicit channel's useful bandwidth will be higher. 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) [Gli93], not every interaction carries information relevant to the channel.

#### 3.2.1.1 Modelling Soft and Hard Isolation

Soft and hard isolation are represented within the model in an identical fashion, namely as a confinement following the specification given in Definition 2. The nature of the isolation provided by a confinement is indicated by the confinement's type, with no explicit mention as to whether the confinement implements soft or hard isolation. Instead, the distinction appears when considering a confinement's semantics and behaviour during a hierarchy's lifetime.

Table 3.1 lists a number of confinement types that can be used to provide hard isolation at various system granularities, along with their common unique identifiers. 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. To that end, the table also lists the common types of sub-confinements that each confinement type can typically accept in a fully-specified

| Type | Description | Identifier | Can Contain |
|---|---|---|---|
| **Net** | Network | Network ID | **Net**, **M** |
| **M** | Machine | IP/Hostname | **L3**, **OS** |
| **L3** | L3 Cache | Index | **L2** |
| **L2** | L2 Cache | Index | **L1** |
| **L1** | L1 Cache | Index | **C** |
| **C** | Physical core | Index | **vC**, $\mathbf{P_E}$, **Con**, **VM** |

Table 3.1: Examples of hard isolations, and their containments.

| Type | Description | Identifier | Can Contain |
|---|---|---|---|
| **VM** | Virtual machine | VM name/UUID | **vC**, **OS** |
| **vC** | Virtual CPU | VCPU ID | **vC**, $\mathbf{P_E}$, **Con**, **VM** |
| **Con** | Container | Container name/UUID | **P** |
| $\mathbf{P_E}$ | Control group | Group name | **Con**, **P** |
| **P** | Process | Process ID (PID) | - |
| **OS** | Operating System | Hostname | $\mathbf{P_E}$, **Con**, **VM** |

Table 3.2: Examples of soft isolations, and their containments.

hierarchy. Note that **M** and **C** also admit the containment of several types of soft isolation confinements, such as processes and virtual machines.

Table 3.2 tabulates the set of soft isolation confinements with which this work is primarily concerned. These types, in combination with the hard isolation types defined in Table 3.1, form the core components of the systems for which the overall mitigation approach is designed, and serve as the fundamental building blocks for a cloud-based deployment scenario. Note that hardware can be partitioned into finer granularities. For example, as will be seen in Section 3.6.2, one can further decompose monolithic cache confinements into their finer constituent cache sets. Nevertheless, the hierarchy as described is sufficient to model the principal actors within a network of machines, and additional confinement layers and partitionings such as software-defined networks or subnets can easily be incorporated by extending the type hierarchy. Conversely, the hierarchy can be simplified by restricting analysis to the coarser-grained virtual machine level, reverting to the more traditional and less nuanced approaches to isolation.

Predictably, the hard isolation confinements listed in Table 3.1 are predominantly rigid architectural elements such as caches and networks, which, while offering some degree of configuration, exist at fixed locations in relation to each other. Conversely, the soft isolations described in Table 3.2 are dynamic, and can be created, destroyed, or in some cases, moved.

## 3.3 Migration and Reconfiguration

The ultimate aim of the model is to be able to model the dynamic elimination of co-location between confined entities. This requires the ability to model the modification of containment hierarchies through operations enabling the *creation*, *destruction* and *migration* of confinements. 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.

### 3.3.1 Agents

Changes to the hierarchy are effected by *agent* processes. 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 are represented within the model as extended confinements, and are defined as follows.

**Definition 6** (Agent)**.** An agent is a confinement $\mathbf{A}{:}\mathrm{N}(\mathrm{C}^{\mathrm{AG}})^{\rightarrow}_{\mathrm{T}}[\mathrm{Q}]$, where $\mathbf{A}$ denotes an agent type, N is the agent's name, T is a set of confinements visible to the agent, $\mathrm{C}^{\mathrm{AG}}$ is its capability set, $\rightarrow \subseteq \mathrm{T} \times \mathrm{T}$ is a mapping defining legal containments, and Q is a queue of idle confinements.

Agent confinements serve to make the scheduling aspects of a system explicit. While agents are given the abstract agent type $\mathbf{A}$, they can be implemented in a variety of ways. For example, as will be seen in Section 4.4, an agent can be implemented as a user-level process that issues scheduling commands to the underlying operating system environment. Similarly, an agent can be an externalised representation of a scheduling procedure that is embedded within a larger system. For example, an agent can serve to expose the behaviour of an operating system's scheduler, allowing its semantics to be incorporated into the model. Agents may also be embedded at different levels of a hierarchy. For example, a network domain controller or router can be modelled as an agent embedded within the network's hardware layer.

The set of confinements T contains a subset of the confinements that constitute the hierarchy being modelled. While this could theoretically be fixed as the universal set of possible confinements, it is assumed that T is the set of legal confinements with which an agent can interact.

An agent's control over its set of known confinements T is limited through a system of *capabilities*. An agent can create, destroy or migrate a confinement if it shares a capability with the confinements involved. This gives rise to the following notion of capability checking.

**Definition 7** (Capability set comparison)**.** Two confinements with capability sets $\mathtt{A}$ and $\mathtt{B}$ can interact if their capability sets intersect, that is, if $\mathtt{A} \Cap \mathtt{B}$ is true, where $\mathtt{A} \Cap \mathtt{B} \stackrel{\text{def}}{=} \mathtt{A} \cap \mathtt{B} \neq \emptyset$.

Capabilities serve to describe the extent of an agent's influence, being restricted to the capabilities denoted by $\mathtt{C}^{\mathrm{AG}}$. The task of checking capabilities is intentionally kept abstract, as the actual authorisation operation varies depending on the confinements involved. For example, within a single operating system, a root password can serve as a single capability that allows all of its sub-confinements to be managed. Other mechanisms include certificates, user groups, and policies defined via *polkit* (as is done in the case of virtual-machine management using `libvirt` [Lib]). The concept of a capability set is used, rather than simply using the confinement name as a capability (as in the case of mobile ambients [CG98]), as it separates the identification aspect of a confinement from its control and policy mechanisms. In addition, the use of a set more closely reflects the principle that there are multiple and different control mechanisms present in a complex system. Mutability is not modelled as an intrinsic property of a confinement, rather it is determined by the availability of a confinement's capability to agents.

The relation $\rightarrow$ specifies the containments that are allowed by the agent's scheduling policy. This is used during analysis when determining potential co-locations between confinements. Pairs in the mapping are of the form $\langle \mathrm{X}, \mathrm{D} \rangle$, which represents that a confinement X can be contained by a parent confinement D.

Finally, the idle queue Q is a set of confinements that are controlled by the agent, and that have been temporarily removed from the active parts of a hierarchy during the system's execution. The idle queue effectively functions as a limbo, or holding area, for confinements that are not executing. For example, an agent representing a process scheduler would transfer processes between $\mathbf{C}$ confinements and its idle queue, placing the active processes within the former and confining the idle processes in the latter.

#### 3.3.1.1 Probes

A *probe* is an abstraction of an event source. This is modelled as a confinement, and is effectively an agent that lacks migration capabilities.

### 3.3.2 Communication and Scoping

For any non-trivial hierarchy of confinements, it is generally the case that no one confinement will have complete knowledge of the topology of which it forms part. Consequently, an agent will only have a partial view of a system.

Consider the simplified cloud infrastructure illustrated in Figure 3.1, which consists of a minimal model of a single node MACHINE with two cores, over which two tenant virtual

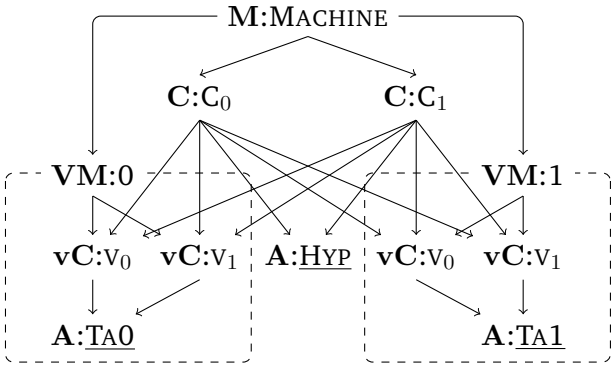



Figure 3.1: Partial model showing agent scopes and boundaries.

machines are executing. In this model, each **VM** has an agent TA0 and TA1 running within it, while the infrastructure provider has an agent HYP running on the base system.

The virtualisation confinements within MACHINE prevent the tenants' agents from enumerating their parent confinements through standard operating system interfaces. In addition, even if the details of the parent confinements can somehow be inferred (for instance, through illicit channels), the tenants' agents would not have the necessary capabilities to alter them directly. For instance, mere knowledge of the existence of additional co-located tenants would not automatically grant a tenant's agents control over them.

Thus, in this scenario, the constituents of each virtual machine are hidden from the entities, or agents, outside of their boundaries. For a given agent to have full knowledge and control over the entire hierarchy, it would have to collude with the other agents. Unanimous collaboration between rational agents would be unreasonable in the case of a public cloud, since

i) tenants are adversaries (or at the very least, mutually-distrusting neighbours) that will not willingly disclose their internal state,

ii) the cloud provider will not advertise details regarding its infrastructure due to the associated security risks, as well as a general lack of incentive, and

iii) tenants would not willingly expose their internal state to a cloud, and suspicions that a cloud provider is performing introspection [DG+13] will quickly lead to the tenants' confidence in the platform being undermined.

Collusion would thus be limited to cases where multiple virtual machines are owned by the same tenant, or when a virtual machine is compromised and is intentionally attempting to leak data to an outside entity.

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, while tenants in Figure 3.1 might not know the number of physical cores that are available on the machine, the hypervisor must have a handle to the tenants' **vC** structures in order to manage their core pinnings. Furthermore, the agent's position within a hierarchy also determines its view of a confinement, and can alter its observed type, particularly in the case of confinements on the fringes of an agent's scope. 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, as evidenced by the migration 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 [Cri], which would alter its internal system view. A prevalent workaround is to employ *namespace* mechanisms, commonly in conjunction with containers [Lib], 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.

When migrating a confinement, an agent must be able to access both the source and destination confinement. Scoping complicates migration, as these confinements may lie outside of the original agent's control. For example, consider the case where an agent on one machine is attempting to migrate a virtual machine onto another physical machine. In this scenario, the agent would require control over the target. One option would be to temporarily transfer a capability over the destination to the agent performing the migration, yet this would pose a security risk. Instead, to exert influence on locations outside its scope, an agent must proxy its requests through an external agent that controls the target scope. Using the previous example, rather than allowing the initiating agent to directly control the virtual machine's receiving endpoint at the destination, the agent forwards a migration request to the destination's controlling agent. The receiving agent then proceeds in creating the destination's virtual machine process whilst retaining control over the new virtual machine. By approaching the problem of migration in terms of provisioning isolation and regulating migrations crossing control boundaries at the agent level, one simplifies the secure and reliable management of capabilities and allows for the modelling of restrictive sharing policies.

### 3.3.3 Scheduling

Agents are scheduling manifest, modelling the execution and placement of confinements. An agent may perform two fundamental forms of scheduling, namely it can

i) transfer confinements directly between confinements that it controls and its idle queue, giving rise to *local scheduling*, and

$$\text{L-Sc}\ \frac{\begin{array}{c}\mathbf{A}{:}\text{Ag}(\mathtt{C}^{\text{Ag}})^{\rightarrow}_{\text{T}}\,[\text{Q}\cup\{\text{X}\}]\\ \mathbf{\Gamma}{:}\text{N}(\mathtt{C})\,[\text{Sb}] \qquad \text{Ag}\equiv \text{X}\curvearrowright \text{N}.\text{Ag'} \qquad \mathtt{C}^{\text{Ag}}\Subset\mathtt{C} \qquad \mathtt{C}^{\text{Ag}}\Subset\mathtt{cap(X)} \qquad (\text{X},\text{N})\in\rightarrow\end{array}}{\mathbf{A}{:}\text{Ag'}(\mathtt{C}^{\text{Ag}})^{\rightarrow}_{\text{T}}\,[\text{Q}] \qquad \mathbf{\Gamma}{:}\text{N}(\mathtt{C})\,[\text{Sb}\cup\{\text{X}\}]}$$

$$\text{L-Ds}\ \frac{\begin{array}{c}\mathbf{A}{:}\text{Ag}(\mathtt{C}^{\text{Ag}})^{\rightarrow}_{\text{T}}\,[\text{Q}]\\ \mathbf{\Gamma}{:}\text{N}(\mathtt{C})\,[\text{Sb}\cup\{\text{X}\}] \qquad \text{Ag}\equiv \text{X}\curvearrowright \text{Ag}.\text{Ag'} \qquad \mathtt{C}^{\text{Ag}}\Subset\mathtt{C} \qquad \mathtt{C}^{\text{Ag}}\Subset\mathtt{cap(X)}\end{array}}{\mathbf{A}{:}\text{Ag'}(\mathtt{C}^{\text{Ag}})^{\rightarrow}_{\text{T}}\,[\text{Q}\cup\{\text{X}\}] \qquad \mathbf{\Gamma}{:}\text{N}(\mathtt{C})\,[\text{Sb}]}$$

Figure 3.2: Local migration rules.

ii) migrate a confinement to a different part of the containment hierarchy that is managed by another agent, leading to *global scheduling*.

Local migration limits the pool of targets to which an agent can schedule a confinement to its set of known confinements, assuming that it also owns the corresponding capabilities. On migrating a confinement globally, one changes the set of potential parent confinements to those that are owned by the destination's agent and allowed by its containment mapping $\rightarrow$.

The following is a description of the two forms of scheduling, as well as a definition of a hierarchy's *configuration*, or state at a given point in time.

#### 3.3.3.1 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 3.2.

The local-schedule rule describes the movement of a confinement X from within an agent Ag's idle queue to a confinement N, triggered on the issue of a migration instruction ($\text{X}\curvearrowright\text{N}$) that moves X from its idle queue to the target locality N. For the local-schedule to execute, the agent must share capabilities with the target confinement, as well as the confinement being moved. This is represented using the capability comparison operator (Definition 7) on $\mathtt{C}^{\text{Ag}}$ and the capability set $\mathtt{C}$, as well as the capability set of X. The latter is represented using the helper function `cap()` that returns a given confinement's capability set, that is, $\mathtt{cap}(\mathbf{\Gamma}{:}\mathtt{N}(\mathtt{C})\,[\text{Sb}]) \rightarrow \mathtt{C}$. Finally, the rule checks that the allocation is permitted (as defined by $\rightarrow$). Given that all the conditions have been met, the confinement X becomes a sub-confinement of N.

The local-deschedule operation is similar, with the key difference that the confinement in question (X) is moved from a confinement within the locally-scoped containment hierarchy

back into the agent's idle queue.

Confinement creation and destruction can be modelled as the addition and removal of confinements from an agent's idle queue, respectively. The destruction of non-idle confinements would thus have to be preceded by a local-deschedule operation. Modelling destruction in this manner avoids having to assign a different set of semantics for rendering a confinement inactive due to destruction or suspension.

Creation and destruction would generally be reserved for soft isolations, as hard isolations are normally fixed confinements dictated by hardware. While hardware confinements such as caches are not typically disabled at runtime, an agent may nevertheless want to delete their representation from the model if it is certain that the threat of a channel through that confinement has been neutralised.

In theory, one could model the deployment of soft isolation techniques for separating a given set of confinements as the destruction of the parent confinement through which they are co-located. In reality, soft isolation techniques are generally imperfect, and can themselves introduce information leaks. For example, a work-conserving scheduling policy may reveal details such as the number of processes co-located with an attacker. Instead, soft isolations should be modelled as the creation of a new confinement, the existence of which depends on a separate set of conditions holding, as will be seen shortly.

**Example 3** (Round Robin Scheduler)**.** Consider the CPU hierarchy defined in Example 2. An agent implementing a simple round-robin scheduler with a shared run queue can be defined as $\mathbf{A}{:}\textsc{rr}(\mathtt{C}^{\textsc{Ag}})^{\rightarrow}_{\mathrm{T}}\,[\mathrm{Q}]$, 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 $\rightarrow \stackrel{\text{def}}{=} \{(\mathrm{X},\mathrm{Y}) \mid \mathrm{X} \in \mathrm{Q} \wedge \mathrm{Y} = \mathbf{C}{:}\mathrm{N}(\mathtt{C})\,[\textsc{Sb}] \wedge \mathrm{Y} \in^{+} \mathrm{CPU}\}$. Given that $\uparrow()^{\rightarrow}$ is a function that returns the allowed parent containments within which a given confinement can be placed, that is, $\uparrow(\mathrm{X})^{\rightarrow} \stackrel{\text{def}}{=} \{\mathrm{Y} \mid (\mathrm{X},\mathrm{Y}) \in \rightarrow\}$, the scheduler can be defined as a CSP-like process as follows:

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

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 [OST06], 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 $|\text{Cs}| - 1$ 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. □

#### 3.3.3.2 Configurations

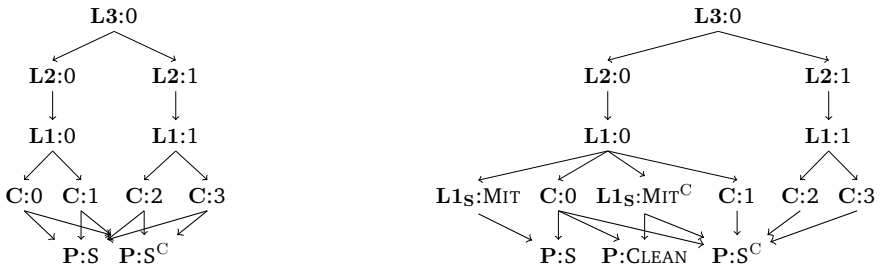
Reasoning about temporal locality requires the ability to describe how a model evolves from one *configuration* to the next, where a configuration is defined as a set of confinements describing a snapshot of the system as a given point in time. The evolution of a configuration is determined by the agents 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 $\texttt{next}(\mathcal{C})$ 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 3.2). This is extended to the iterated next configuration function $\texttt{next}^n(\mathcal{C})$, which returns the set of configurations reachable from $\mathcal{C}$ in $n$ steps, defined as follows:

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

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

$$\begin{aligned}\texttt{next}^n_\cup(\mathcal{C}) &\stackrel{\text{def}}{=} \left\{\mathbf{\Gamma}{:}\text{N}(\text{C})\,[\text{SB}] \mid \mathbf{\Gamma}{:}\text{N}(\text{C})\,[\text{SB}'] \in^+ \text{CFS}\right\}\\ \text{where SB} &\stackrel{\text{def}}{=} \bigcup\left\{\text{SB}'' \mid \mathbf{\Gamma}{:}\text{N}(\text{C})\,[\text{SB}''] \in^+ \text{CFS}\right\}\\ \text{and CFS} &\stackrel{\text{def}}{=} \bigcup \bigcup_{0\le i\le n} \texttt{next}^i(\mathcal{C})\end{aligned}$$

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.




(a) No isolation at $\mathbf{L1}$

(b) Creation of $\mathbf{L1_S}$ via soft isolation

Figure 3.3: Cache-level co-location and mitigation via soft isolation, with arrows denoting containment.

#### 3.3.3.3 Isolation using Local Scheduling

Soft isolation can be obtained in a number of ways, often by using some form of *normaliser*. In this scenario, a normaliser is a process or procedure that, through its execution, produces a new confinement that exposes less information than its non-normalised counterpart. For example, normalisers can reduce the information content of a side-channel that observes triggering times of several mechanisms, such as packet delivery times [Gor+12] and hard disk usage.

Both soft and hard isolations are represented as confinements (Section 3.2.1.1). Scheduling a confinement into a soft isolation confinement would imply that the conditions for that confinement to exist must hold. For example, by never co-scheduling processes, one creates a mutually-exclusive containment at the cache level. This allows a dynamic scheduling policy to be treated as a static guarded resource, simplifying the representation.

**Example 4** (Round Robin Scheduler, revisited)**.** In Example 3, 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 [OST06] 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* [ZR13] 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 $\mathbf{L1}$), transforming the hierarchy described in Figure 3.3a to that illustrated by Figure 3.3b (for simplicity, S is pinned to $\mathbf{C}$: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 $\mathbf{L1}$:0, as they are still ultimately sharing hardware locality. If the soft isolation is deemed perfect, then the $\mathbf{L1}$ confinement can be destroyed. Removing CLEAN would lead to the partitions being destroyed, and the $\mathbf{L1}$:0 confinement being recreated. □

#### 3.3.3.4 Global Scheduling

Local scheduling limits an agent in its ability to procure isolation, as it can only move entities amongst confinements that are 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 $\mathbf{vC}$ confinement is placed in a dedicated $\mathbf{C}$ 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 $\mathbf{vC}$ 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

i) identifying a target agent which can procure the required level of isolation, and

ii) migrating the confinements required to achieve isolation.

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 $\texttt{next}^{\infty}_{\cup}()$.

Figure 3.4 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 $\texttt{isol}()$, which DST attempts to match against its known and controllable confinements. Following the migration, each agent updates its containment mapping rules, with SRC removing

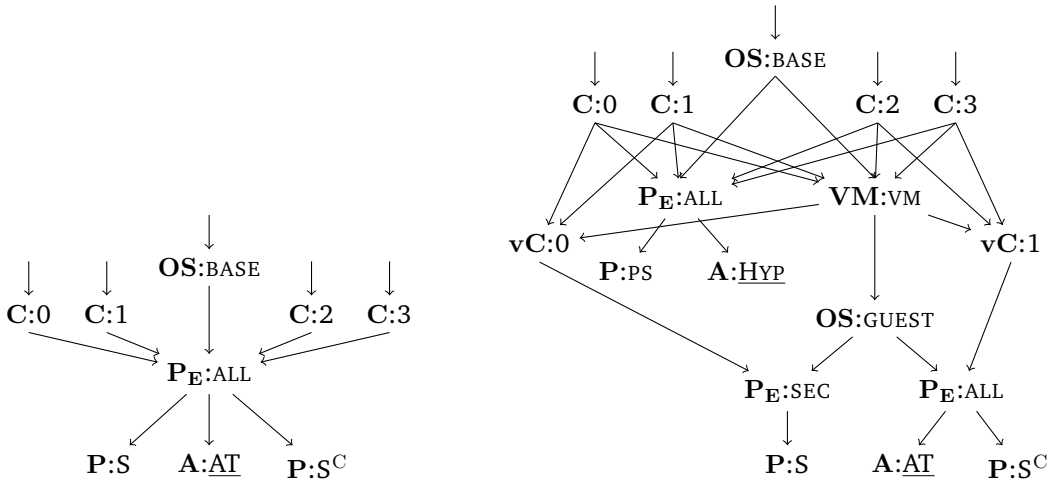
$$\text{G-SC}\ \frac{\begin{array}{c}\mathbf{A}{:}\textsc{Src}(\mathtt{C}^{\textsc{Src}})^{\rightarrow_{\textsc{Src}}}_{\textsc{TS}}\,[\mathrm{Q}_{\textsc{Src}}\cup\{\mathrm{X}\}] \qquad \mathbf{A}{:}\textsc{Dst}(\mathtt{C}^{\textsc{Dst}})^{\rightarrow_{\textsc{Dst}}}_{\textsc{TD}}\,[\mathrm{Q}_{\textsc{Dst}}]\\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

Figure 3.4: Global migration rule.

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 potential parent confinements at its destination. This allows a confinement to maintain the same number of allocated resources across migrations, for instance, it would allow a virtual machine to preserve its ratio and mapping of virtual CPUs to physical CPUs at the destination.

As will be seen in Chapter 4, global migration is implemented through a variety different mechanisms, the choice of which depends on the containment level being considered, as well as the position of the agent relative to the confinements being migrated. The choice of migration approach affects the performance impact of reconfiguration, due to varying amount of state that will have to be moved about. As a general rule, reconfiguration costs can be minimised by favouring migrations between nearby confinements over migrations to distant destinations. This is also affected by the confinements' depth within the hierarchy. Conversely, it may occasionally prove to be advantages to perform multiple local reconfigurations over a single large migration, particularly in the case of cascaded confinements. In addition, other factors such as data locality within a data centre can serve to restrict the set of viable destinations that a confinement may have.

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. The difficulty then lies in the choice of communication interfaces that are made available to sub-confinements. For example, while logics such as the *cloud calculus* [Jar+12] make use of a `parent()` operator, which returns a handle to a confinement's parent, such a predicate is not generally available out of the box. On the contrary, there are several cases




(a) Bare metal

(b) Single-Layer Virtualisation

Figure 3.5: Introducing indirection through virtualisation.

where the parent confinement maintains an active effort to fully encapsulate its child confinements and remain transparent. This is particularly evident in the case of virtualisation, where the virtualisation platform generally limits direct interaction from its guest virtual machines to a set of defined drivers, and makes a conscious effort to provide guests with the illusion that they are running directly on the underlying hardware. As will be seen in Section 5.2.3, this problem can be overcome through the creation of additional explicit interfaces through which an agent within the guest virtual machine can pass up requests to a hypervisor-level agent.

As a confinement may potentially be migrated across hostile boundaries, additional measures must be adopted to secure communications between the endpoints of an implementation of the system.

#### 3.3.3.5 Isolation Constraints

A consequence of agent scoping is that changes to external confinements need to be delegated to an agent. In addition, operations that modify the hierarchy cannot directly 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 $\mathbf{C}$ 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 3.5a illustrates a partial $\texttt{next}^{\infty}_{\cup}()$ graph of the CPU hierarchy from the perspective of

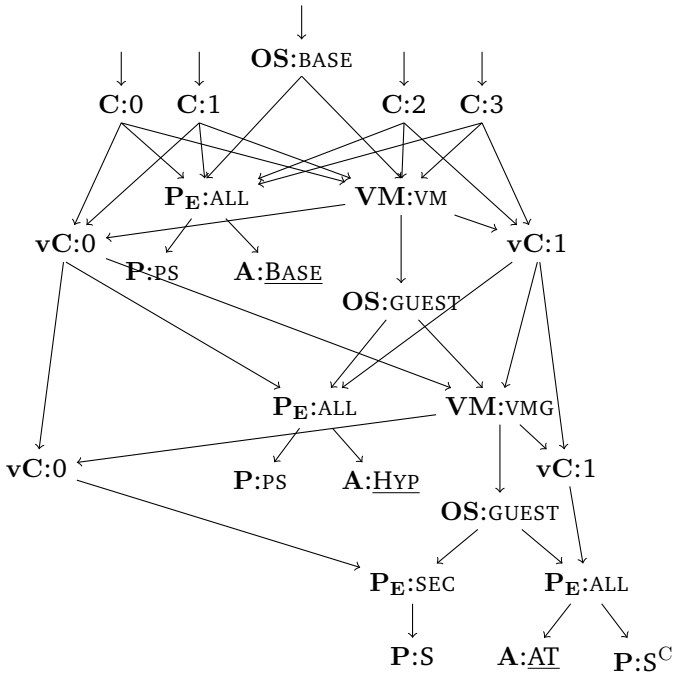



Figure 3.6: Nested virtualisation (two layers).

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, 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 3.5b. 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 X $\mapsto$ 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 3.6), 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* may 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:

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

## 3.4 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. Cost can be expressed with respect to different resource types, including power consumption and computational costs, yet these can often be derived from the more general measurement of utilisation.

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, accurate measurements of costs should be evaluated over the $\texttt{next}^{\infty}_{\cup}\texttt{()}$ of a given hierarchy.

What constitutes cost depends on the isolation approach being used. For example, while the upkeep cost of a soft isolation technique may be represented by its overhead, or the computing capacity committed to the active process that is preserving the isolation, the cost of hard isolation is that it can lead to unused computational capacity. The impact of unused capacity, especially in the context of cloud computing, materialises in the form of lost potential profits and unnecessary maintenance costs. Thus, the former is quantified through the presence of higher workloads, whereas the latter leads to the reduction or absence of computation. Depending on the degree of isolation involved, one can expect a crossover point, where the cost associated with committing resources to a soft isolation outweigh the cost of maintaining the capacity that would be left unused using hard isolation. This crossover point varies when one considers that freeing resources committed to maintaining soft isolation will also make them available for the actual workloads that require isolation. The composition of isolation strategies will also affect scalability and the total cost of isolation. For example, a compound action such as creating a virtual machine and migrating a process that requires isolation to it may prove cheaper than migrating the process' original virtual machine, due to the larger granularity of the latter confinement.

### 3.4.1 Core Metrics

The following defines a number of fundamental metrics and expressions for analysing configurations and evaluating costs.

#### 3.4.1.1 Capacity

A basic metric of a model is its *total capacity*, or the number of confinements that a given configuration contains. This is defined as

$$\texttt{capacity}(\mathcal{C}) \stackrel{\text{def}}{=} |\left\{ \mathrm{X} \mid \mathrm{X} \in^{+} \mathcal{C} \right\}|$$

Rather than the total capacity, one is generally interested in the capacity of a hierarchy with respect to a given confinement type $\mathbf{\Gamma}$, which allows the enumeration of elements such as $\mathbf{VM}$ and $\mathbf{M}$. This is defined as

$$\texttt{capacity}_{\mathbf{\Gamma}}(\mathcal{C}) \stackrel{\text{def}}{=} |\left\{ \mathrm{X} \mid \mathrm{X} \in^{+} \mathcal{C} \wedge \mathrm{X} = \mathbf{\Gamma}'{:}\mathrm{N} \wedge \mathbf{\Gamma}' = \mathbf{\Gamma} \right\}|$$

#### 3.4.1.2 Utilisation

Certain confinements can only contain a number of sub-confinements before the system's overall performance peaks or 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:

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

The second formula is an approximation that can be computed statically given an average processor usage $k$ and the $\mathbf{P}$-to-$\mathbf{C}$ mapping defined by an agent's $\rightarrow$ structure (the relation ${\uparrow}()^{\rightarrow}$ being defined in Example 3). The `min` function caps each $\mathbf{C}$'s usage value to 100% (1.0), as each $\mathbf{C}$ can only work at its maximum. In the case of a hierarchy containing a mixture of core architectures and performance ratings, a more precise utilisation metric can be derived by parametrising the capped value and changing it based on the core's type. These values can be expressed as a ratio with a basic core type, or using an absolute performance metric such as instructions per second. While the expression is designed for the context of core utilisation, this metric can also be extended to other forms of bounded containment.

#### 3.4.1.3 Consolidation factor

Beyond measuring the aggregate utilisation of a system's capacity, cloud providers are interested in their infrastructure's *consolidation factor*. This is represented as **utilisation**/**capacity**, or the ratio between the system's utilisation and the number of confinements of a given type within a hierarchy. This figure encompasses both underutilisation as well ask skewed load distributions.

#### 3.4.1.4 Pairwise co-locations

Assuming that every co-location is an equally-large threat to a tenant, one would reduce the attack surface by minimising the total number of co-located pairs in a given hierarchy, which can be computed as follows:

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

When forming a coherent defence against illicit channels, a tenant may opt to prioritise certain isolations over others. For example, a tenant may prefer to minimise co-locating processes with other tenants prior to isolating its own local structures, based on the assumption that the risk of an external attack is greater than an internal one. In this case, `pairs()` can be extended to only consider subsets of parent confinement types.

### 3.4.2 Applying Metrics

As containment hierarchies are acyclic, they 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 are fundamentally at odds and will 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, the latter having a material financial impact on its operations. Conversely, while a cloud provider will attempt to optimise 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 5** (Comparing architectures)**.** A system's containments can vary across vendors. To illustrate, consider two different CPUs, namely an Intel i7-4790 ($\textsc{Intel}_{\mathrm{T}}$) with 8 hard-

ware threads using SMT[1], and a hex-core AMD Phenom II X6 ($\text{AMD}_\text{T}$). 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:

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

Consider the case where processes must never be co-located through **L1** or **L2**. For the $\text{INTEL}_\text{T}$ hierarchy, this effectively halves the **C** capacity. Assuming that each system divides $P$ processes amongst its **C**s equally, $\texttt{util(AMD}_\text{T}\texttt{)} = \texttt{min}\,(k_{\text{AMD}_\text{T}}P/6, 6.0)\,,$ and $\texttt{util(INTEL}_\text{T}\texttt{)} = \texttt{min}\,(k_{\text{INTEL}_\text{T}}P/4, 4.0)$. Thus, $\text{INTEL}_\text{T}$'s process execution time $k_{\text{INTEL}_\text{T}}$ must be two thirds of $k_{\text{AMD}_\text{T}}$ in order to have equal utilisation rates. □

### 3.4.3 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 $\tau$, 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 $\tau$ enables configurations to be optimised with a minimum of migrations, yet a system in constant flux or with very small values of $\tau$ can potentially negate gains in migrating. Cheap migration operations can help offset the effects of $\tau$.

**Example 6.** Figure 3.7 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 4. Utilisation rates are given in brackets, assuming that

i) each **L1** confinement is shared between two cores and has a total capacity of 2,

[1]For this architecture, the indices for hardware threads sharing the same core are not consecutive.

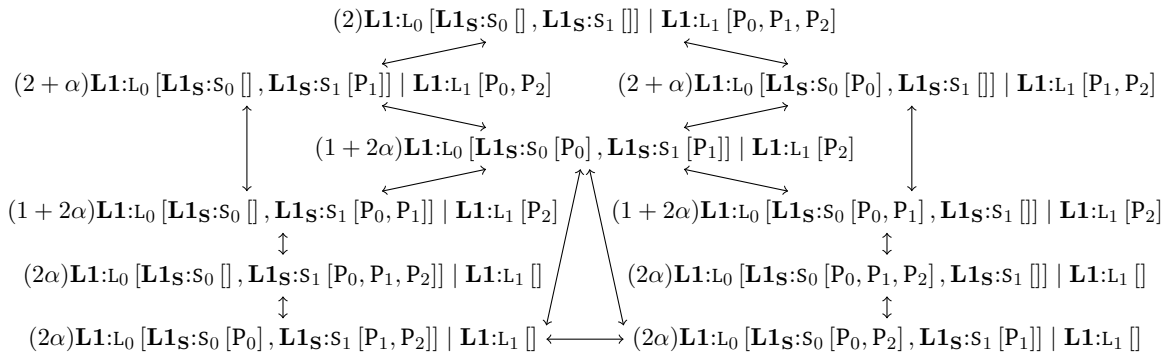



Figure 3.7: A subset of possible global migrations between configurations.

ii) each process has a utilisation factor of 1,

iii) **L1** confinements have zero cost, as they are built into the architecture, and

iv) non-empty $\mathbf{L1_S}$ confinements reduce their core's capacity to $\alpha$ (overhead values can reach up to 7% [ZR13]).

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* [BPH14] and heterogeneous deployments, with certain configurations being cheaper or more secure to maintain on machines with dedicated hardware.

## 3.5 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 and eludes an efficient solution [Aza+14]. 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

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

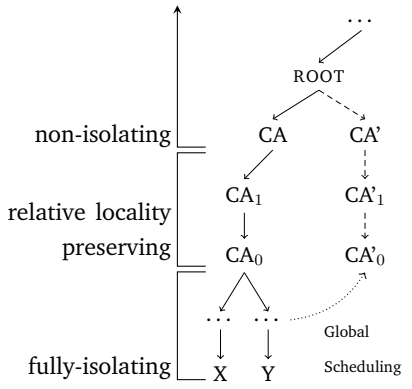
i) find D $\in^+$ CA.
   X $\in^+$ D $\vee$ X = D $\vee$ Y $\in^+$ D $\vee$ Y = D

ii) find/create CA'. $\neg$CA' $\in^+$ CA

iii) replicate path from CA to D in CA'

iv) check isolation constraints and migrate D to new parent in CA'

(a) Migration procedure outline




(b) Migration effects by graph height

Figure 3.8: Computing migration paths for breaking X $\overset{CA}{\Longleftrightarrow}$ Y.

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 [Gao$^+$10]. 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 3.8a 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 3.8b. 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 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.

### 3.5.1 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. The choice of confinement source (and similarly, destination) is limited by the set of agents available to the orchestrator of the migration, as this directly affects the visible confinement scopes.

When attempting to reconfigure the configuration illustrated by Figure 3.8b to comply with the constraint $\neg$(X $\overset{CA}{\Longleftrightarrow}$ Y), one finds that individual migration operations moving confinements outside of CA can take one of three forms, namely:

i) *fully isolating*, where X and Y share no common ancestor up to the depth of CA,

ii) *relative locality preserving*, whereby co-location through CA is broken, yet the confinements are still co-located within an intermediate common confinement, and

iii) *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 through the effects of cascading (Section 3.1). 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.

### 3.5.2 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 3.8b, 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 ($\mathbf{vC}$ to $\mathbf{C}$, or $\mathbf{P}$ to $\mathbf{C}$) can be performed in milliseconds, as opposed to the migration of larger structures ($\mathbf{P}$ to $\mathbf{OS}$, or $\mathbf{VM}$ to $\mathbf{M}$), which can be a thousand times slower, principally due to the involvement of the network layers and shared storage, as will be seen in Chapter 4.

### 3.5.3 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 $\mathbf{VM}$ that is running within a second $\mathbf{VM}$, one may opt to migrate

the former directly to bare metal. Conversely, a process will not operate correctly unless its environment's assumptions are correctly mirrored at the destination.

### 3.5.4 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.

## 3.6 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.

### 3.6.1 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 [Zha+11]), the system can be reconfigured to a correct state using migration, leading to a reactive architecture. An implementation of this principle will be expounded upon at greater lengths in Section 4.4.2. Alternatively, the framework can be driven by a system of leases, with isolation being procured before a security-sensitive process executes.

### 3.6.2 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*. [Zha+12a], and its scheduler-based mitigation [VRS14]. The attack relies on a PRIME-PROBE cache access pattern, similar to the attack described in Example 3.

Consider a hypervisor managing two virtual machines, namely a victim $\text{VM}_\text{v}$ and attacker $\text{VM}_\text{a}$. Both machines (collectively referred to as $\vec{\text{V}}$) share a core $\text{C}_0$. The hypervisor agent HYP is defined as:

$$\mathbf{A}{:}\text{HYP}(\{\text{C}^{\text{VM}_\text{V}}, \text{C}^{\text{VM}_\text{a}}, \text{C}^{\text{C}_0}\})^{\{(\text{VM}_\text{V},\text{C}_0),(\text{VM}_\text{a},\text{C}_0)\}}_{\{\text{VM}_\text{V},\text{VM}_\text{a},\text{C}_0\}}[\vec{\text{V}}]$$

and implemented as a process $\text{HYP}^\text{I}$ defined as:

$$\text{HYP}^\text{I} \equiv \prod^{\mathbf{VM}:\text{X} \in \vec{\text{V}}} \text{X} \curvearrowright \text{C}_0.\text{X} \curvearrowright \text{HYP}.\text{HYP}^\text{I}$$

The $\texttt{next}^\infty_\cup()$ graph of the system at this coarse level of granularity would reveal that the virtual machines are co-located through $\text{C}_0$, 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:

$$\mathbf{L1}{:}\text{CLS}_0\,[\{\mathbf{CLS}{:}\text{CS}_\text{I}\,[] \mid 0 \leq i < N\}]$$

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:

$$\texttt{run}(\mathbf{A}{:}\text{AG}, \mathbf{L1}{:}\text{L}, \mathbf{C}{:}\text{C}, \mathbf{vC}{:}\text{VC}, \text{R}) \equiv \text{VC} \curvearrowright \text{C}.(\prod^{\mathbf{CLS}:\text{CS} \in L} \text{VC} \curvearrowright \text{CS})^R.\text{VC} \curvearrowright \text{AG}$$

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** $\text{VC}_\text{a}$ within a cache set previously occupied by $\text{VC}_\text{v}$, leading to the sequence:

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

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 $\text{VM}_\text{v}$ attempts to choose a value of $R$ such that it *maximises* the value of `pairs()` formed over an execution.

A mitigation against this attack is to place a minimum running time on virtual machines [VRS14], which stops an attacker from forcing deschedules and limits 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 $\mathbf{VM}$ scheduling quantum length.

A similar fine-grained cache analysis can be performed for cache colouring [KPMR12], where scheduling must guarantee disjoint cache sets. An additional related mitigation is that of the cache cleaning process (Example 4), which is effectively a solution for the same problem using a different scheduling level.

### 3.6.3 Timing Channel Elimination

STOPWATCH [LGR13] is a collection of mitigations designed to reduce the information content of timing channels in the cloud. The approach centres on the use of *replication* to create `R` copies of each virtual machine (`R` $\geq 3$), each of which is placed on a different machine containing other tenants' replicated $\mathbf{VM}$s. Clock sources on a $\mathbf{VM}$ 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 $\mathbf{VM}$ replication and placement aspects of STOPWATCH.

Given a network NET of machines, the $\mathbf{VM}$ placement requirements of STOPWATCH can be modelled as three invariant conditions, namely:

$$\forall \mathbf{VM}\text{:v} \in^+ \text{NET}.\ |\left\{\text{v'} \mid \mathbf{VM}\text{:v'} \in^+ \text{NET}, \texttt{is\_replica}(\text{v'}, \text{v})\right\}| = \texttt{R} \tag{3.1}$$

$$\forall \mathbf{VM}\text{:v}, \mathbf{M}\text{:M} \in^+ \text{NET}.\ |\{\text{v'} \mid \mathbf{VM}\text{:v'} \in \text{M}, \texttt{tenant(v')} = \texttt{tenant(v)}\}| \leq 1 \tag{3.2}$$

$$\begin{aligned}\forall \mathbf{VM}\text{:}\text{v}_1, \mathbf{VM}\text{:}\text{v}_2, \mathbf{M}\text{:M} \in^+ \text{NET}.\ \text{v}_1 \neq \text{v}_2 \wedge \text{v}_1 \overset{\text{M}}{\longleftrightarrow} \text{v}_2 \rightarrow \\ \neg\exists \mathbf{VM}\text{:}\text{v}_3, \mathbf{VM}\text{:}\text{v}_4, \mathbf{M}\text{:M'} \in^+ \text{NET}.\ \text{v}_3 \neq \text{v}_4 \wedge \text{v}_3 \overset{\text{M'}}{\longleftrightarrow} \text{v}_4 \wedge \text{M} \neq \text{M'} \wedge \\ \texttt{tenant(}\text{v}_1\texttt{)} = \texttt{tenant(}\text{v}_3\texttt{)} \wedge \texttt{tenant(}\text{v}_2\texttt{)} = \texttt{tenant(}\text{v}_4\texttt{)}\end{aligned} \tag{3.3}$$

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.

### 3.6.4 Other Properties

The following section outlines several additional scenarios that can be characterised using the model and concepts explored in this chapter.

**Basic Chinese Wall Policy** The elimination of cache-level contention can be modelled at a number of granularities. For example, the individual cache-lines can be modelled as fine-grained confinements, contained within a larger cache-set parent confinement. Isolation can also be enforced at the CPU package and process level.

**Trusted/Allowed Co-Schedulings** The model may be used to define groups of trusted confinements that can be co-scheduled within time or space, either by extending confinements to allow for attributes, or by simply replicating the trust groupings using containments. In the latter case, groups can be broken down into finer-grained federations of confinements, which may also span across different confinement types.

**Grouping Address Ranges** Due to the circuitry and algorithms involved with memory address alias resolution, accesses to certain memory elements may influence the time taken to perform memory operations on other, essentially unrelated, memory elements [Cop+09]. In the case of systems using *pessimistic load bypass* [Cop+09], this occurs amongst elements interspersed by a fixed width. These groupings can be modelled by partitioning memory into a series of buckets, the size of which varies based on the architecture in question.

**Delayed Scheduling** Due to the effects of a channel potentially persisting across time, one can postpone the scheduling of a sensitive task, or prefix it with a normalising process (Example 4). In the case of cache-based channels, one can postpone scheduling until a given degree of cache entropy has been reached, thus avoiding the use of an explicit cache cleaning process. Tracking of cache pollution can be done using *hardware event counters* (these will be covered in greater detail in Section 4.4.2.2). In the case of time-critical processes, the system can set an upperbound on time, after which a high-entropy process is forced into existence. This can also be extended to the virtual machine level, or a mixed-level mitigation between processes and virtual machines belonging to a different entity. A related mitigation is to schedule and migrate processes or virtual machines in such a way as to maximise the periodicity, or the time before any given confinement is co-scheduled with a previously co-scheduled confinement.

## 3.7 Conclusion

This chapter has investigated the modelling of temporal and spatial co-location within the context of illicit channels, examining the issues of cost, scoping and migration. Migration was subsequently split into *local* and *global* migration classes, with the former representing scheduling operations, and the latter enabling the transfer of confinements between agent scopes.

While it is assumed that the model will be used in the context of cloud infrastructures, there is nothing within the model that inherently restricts its application to such scenarios. Cloud architectures have primarily been singled out due to their diversity of confinement types, and because the scale of cloud infrastructures translates into an abundance of potential targets to which a confinement can be isolated.

Based on the established attacks and mitigations studied in the previous chapter, it appears that some degree of co-location is always necessary for an illicit channel to be feasible. That is, one cannot claim that a system is completely air-gapped whilst simultaneously admitting an illicit channel. For example, physical phenomena such as electromagnetic emanations may allow a channel to be formed over physical spans [Gur+14], subject to physical co-location. In the case of remote attacks, although the attacker and victim may not share a single global and authoritative real-time clock, the former may indirectly correlate progress at the victim's end with its own local clock [BB03]. Similarly, other remote attacks may still rely on the execution of code at the victim to directly exploit co-location [Ore+15]. This co-location prerequisite is useful as it simplifies the model by ensuring that the same metaphors of co-location and containment can be uniformly applied to different attack and mitigation scenarios.

The next chapter concerns the transition from model-based mitigations to concrete implementations using SAFEHAVEN, with particular attention to the performance of reconfiguration and migration operations.

CHAPTER 4

# THE SAFEHAVEN FRAMEWORK

WITH THE TERMINOLOGY AND NOTATION required to model confinement and co-location defined, attention is now turned towards SAFEHAVEN, a framework designed to facilitate the creation, deployment and evaluation of isolation policies.

## 4.1 Introduction

Physical isolation provides tenants in a cloud with strong security guarantees against hardware illicit channels. The viability of hard isolation as a general mitigation technique depends on three factors, namely the availability of distinct hardware locations, the degree of underutilisation tolerated by the owner of the infrastructure, and the cost of dynamic reconfiguration.

The number of available isolated confinements depends on the size of the physical infrastructure under consideration. If a cloud provider were to provision isolation at the machine level, then it would very quickly run out of computational capacity, as well as obliterate the system's consolidation factor (and, by association, its profitability). This is closely tied to the second factor, in that the granularity at which isolation is procured also determines the granularity at which underutilisation occurs. Thus, for example, provisioning isolation at the hardware thread level will block the utilisation of a co-located hardware thread, while a machine-wide isolation guarantee will deny other tenants use of that machine.

The previous chapter demonstrated, by means of a hierarchical model, how a finer-grained approach to isolation enables higher rates of utilisation by minimizing unused capacity and effectively multiplying the number of candidate confinements for isolation. This chapter details the implementation of the model into a framework that allows the dynamic provisioning of isolation at various levels of a system's architecture, primarily at the core, cache, and machine level, as well as their virtualised equivalents. Each confinement type is studied separately, and is followed by a description of their integration into a unified framework, dubbed SAFEHAVEN.

SAFEHAVEN is a framework that assists in the creation and deployment of networks of

communicating *probe* and *agent* processes. Sophisticated system-wide detectors can be built by cascading events from various probes at different system levels. A crucial aspect of this approach is that detectors can be both *anticipatory* as well as *reactive*, meaning that they can either trigger isolations as a precaution or as a countermeasure to a detected attack.

SAFEHAVEN is implemented in Erlang [Erl] due to its language-level support for many of the framework's requirements, with probes and agents as long-lived distributed actor processes communicating their stimuli through message passing. Other innate language features include robust process discovery and communication mechanisms and extensive support for node monitoring and error reporting. SAFEHAVEN was developed in lieu of adapting existing cloud-management suites such as OpenStack [Ope] so as to focus on the event signalling and migration aspects of the approach. Erlang's functional nature, defined communication semantics and use of *generic process behaviours* [Erl] help to simplify the automatic generation and verification of policy enforcement code, paving the way for future formal analysis.

The framework is demonstrated using two case studies, showing its efficacy both in a reactive, as well as an anticipatory, role. Specifically, SAFEHAVEN is used to detect and foil a *system-wide covert channel* in a matter of seconds, and to implement a *multi-level moving target defence policy*. The results from these case studies are used to quantify the last of the three factors determining the approach's viability, namely the overheads of migration and reconfiguration.

### Chapter Outline

This chapter is structured as follows:

**Section 4.2** investigates concrete and real-world analogues to the confinements considered in the previous chapter, and details their migration and management.

**Section 4.3** illustrates the core elements of SAFEHAVEN agents by means of an example.

**Section 4.4** evaluates the application and performance of different migration mechanisms when used to counteract a machine-wide covert-channel attack and when implementing a moving target defence.

**Section 4.5** concludes this chapter.

## 4.2 An Instantiated Hierarchy

The following section details the implementation of a framework for managing an instantiation of the model described in the previous chapter. The framework, dubbed SAFEHAVEN, is designed to control a number of confinement types, a list of which can be found in Table 4.1.

The technologies and confinement types that were incorporated within SAFEHAVEN were chosen as analogues to the confinement types described previously (Section 3.2.1.1), and form the fundamental components of cloud environments and workstations.

| Model | Technology | Reconnaissance | Identifier |
|---|---|---|---|
| **OS** | Linux | `uname` | Host name |
| **VM** | `QEMU` | `virsh` | VM name |
| **vC** | `KVM` or emulated | `virsh`, `/proc/` | vCPU ID |
| **Con** | `LXC` | `lxc-info` | Container name |
| $\mathbf{P_E}$ | `cgroups` | `cset` | cgroup name |
| **P** | System process | `ps` | Process ID |

Table 4.1: Summary of technologies used to implement confinement stack and the methods by which confinements are enumerated at each level.

As part of its duties in managing containments and allocations, SAFEHAVEN also allows the enumeration of confinements through a series of *reconnaissance* (or `recon`) functions, the implementation of which varies based on the hierarchy level in question. Using `recon` functions, an agent can query the underlying system at runtime and build a partial model of the infrastructure, translating it into a graph of first-class Erlang objects representing confinements. This facilitates the creation of dynamic policies. Table 4.1 summarises the core enumeration mechanisms that are used at each confinement level being considered.

Figure 4.1 illustrates a containment hierarchy built primarily using the aforementioned confinement types, as well as the core migration pathways that are considered in this work (depicted through arrows 1 to 7). The remainder of this section provides a detailed description of the modelling and implementation of each migration path.

### 4.2.1 Cores (C) and Virtual CPUs (vC)

Virtual CPUs (**vC**) are an abstraction of physical cores (**C**). In the case of `QEMU` virtualisation, each virtual machine is assigned a subset of the **C** confinements that are physically available on the underlying machine. In turn, each of the **VM**'s virtual CPU confinements can be assigned (or pinned) to a subset of these allocated cores, either directly via QMP [Qema], or through libvirt [Lib]. This means that a given **vC** can only execute within one of the cores to which it is assigned, that is it can only be locally scheduled to cores within its group. Similarly, a **vC** cannot be directly migrated to a core external to its parent **VM**'s assigned set of **C**s.

Figure 4.2 shows the series of migration operations required to migrate a **vC** from one **C** allocation to another. The **vC** is first moved to an agent AG that executes in parallel to the

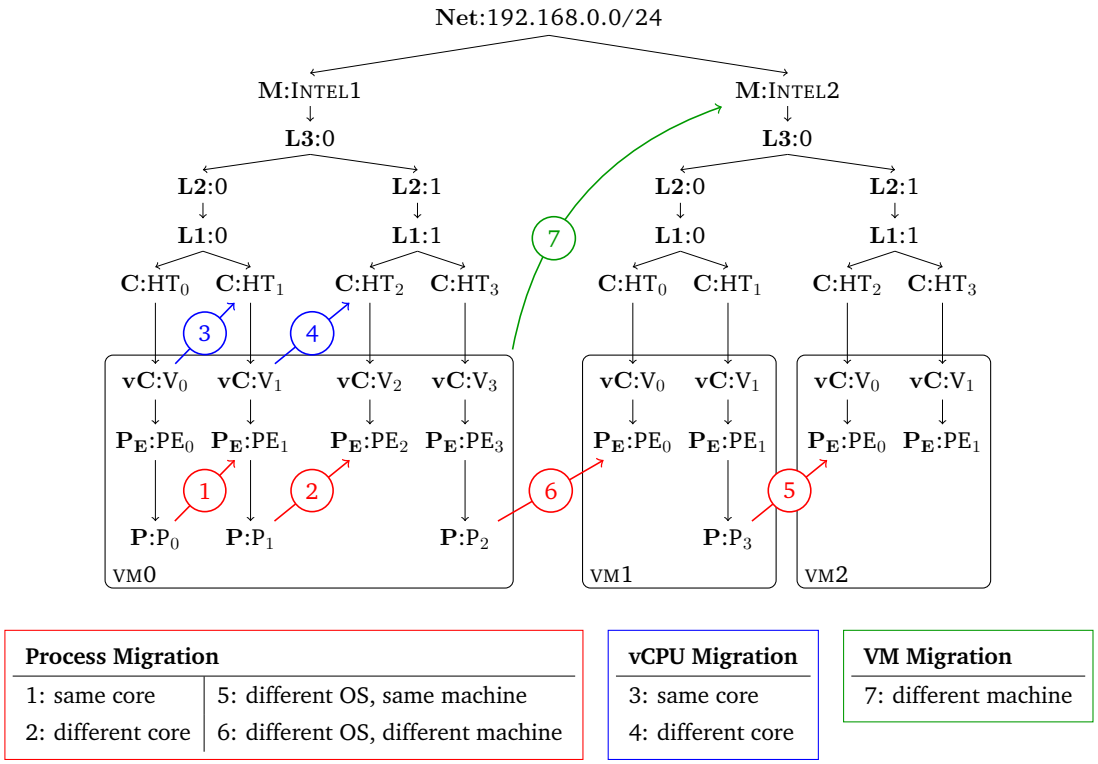



Figure 4.1: Configuration of two physical machines running three virtual machines, with edges denoting containment. The paths labelled 1 through 7 denote routes for migration, with the source confinement changing its parent to the destination.

**VM** to which the **vC** belongs, after which the mapping rules are modified, and the **vC** is migrated to its destination core `C1`. **vC** migration within a nested virtualisation environment follows a similar sequence of operations, except that the confinement type of `M` becomes **VM**, and `C0` and `C1` become **vC** confinements.

The agent process shares the same operating system as the virtual machine, and exists at the same level within the containment hierarchy. While the procedure may initially appear to be a local scheduling operation, it entails a modification to the mapping rules, and consequently modifies the system's $\texttt{next}^{\infty}_{\cup}()$. Thus, the operation is still a global migration, yet its source and destination agents are the same.

With regards to capabilities, an agent requires authorisation over the hypervisor and **vC**s to reconfigure the **vC**-to-**C** mapping. Note that the existence of a **vC** automatically implies the existence of a **VM** further up the hierarchy, that is,

$$\mathbf{vC}{:}\text{VCPU} \in \mathbf{C}{:}\text{CPU} \Rightarrow \exists \mathbf{VM}{:}\text{VM}.\ \text{VCPU} \in \text{VM} \wedge \text{VM} \in \text{CPU}$$

`QEMU` **vC**s can either operate as emulators, providing a fully-virtualised confinement, or with hardware acceleration, typically using `KVM` [Kvma]. As will be seen in Section 5.3.6, the latter may admit *unsafe migrations*, with virtual machines being migrated to physical

$$\frac{\begin{array}{l}[\text{OS} \in^{+} \mathbf{M}{:}\text{M},\\ \text{C0} \in^{+} \mathbf{M}{:}\text{M},\\ \text{C1} \in^{+} \mathbf{M}{:}\text{M},\\ \text{VM} \in \text{C0},\\ \text{VM} \in \text{C1}]\end{array} \quad \begin{array}{c}\mathbf{OS}{:}\text{OS}\left[\mathbf{VM}{:}\text{VM}(\mathtt{C}^{\text{D}})\left[\mathbf{vC}{:}\text{VCPU}(\mathtt{C}^{\text{D}})\right], \mathbf{P_E}{:}\text{PGA}\left[\mathbf{A}{:}\text{AG}(\mathtt{C}^{\text{AG}})^{\{\langle \text{VCPU, C0}\rangle\}\cup\rightarrow_{\text{AG}}}_{\{\text{VCPU, C0, C1, VM}\}}[]\right]\right]\\ \mathbf{C}{:}\text{C0}(\mathtt{C}^{\text{X}})\,[\text{VCPU}] \qquad \mathbf{C}{:}\text{C1}(\mathtt{C}^{\text{Y}})\,[] \qquad \text{AG} \equiv \text{VCPU} \curvearrowright \text{AG.AG'}\\ \mathtt{C}^{\text{AG}} \Subset \mathtt{C}^{\text{D}} \qquad \mathtt{C}^{\text{AG}} \Subset \mathtt{C}^{\text{X}}\end{array}}{\begin{array}{c}\mathbf{OS}{:}\text{OS}\left[\mathbf{VM}{:}\text{VM}(\mathtt{C}^{\text{D}})\,[], \mathbf{P_E}{:}\text{PGA}\left[\mathbf{A}{:}\text{AG'}(\mathtt{C}^{\text{AG}})^{\rightarrow_{\text{AG}}}_{\{\text{VCPU, C0, C1, VM}\}}[\mathbf{vC}{:}\text{VCPU}]\right]\right]\\ \mathbf{C}{:}\text{C0}(\mathtt{C}^{\text{X}})\,[] \qquad \mathbf{C}{:}\text{C1}(\mathtt{C}^{\text{Y}})\,[]\\ \mathtt{C}^{\text{AG}} \Subset \mathtt{C}^{\text{D}} \qquad \mathtt{C}^{\text{AG}} \Subset \mathtt{C}^{\text{Y}} \qquad\qquad \text{AG'} \equiv \text{VCPU} \curvearrowright \text{C1.AG"}\end{array}}$$

$$\mathbf{OS}{:}\text{OS}\left[\mathbf{VM}{:}\text{VM}(\mathtt{C}^{\text{D}})\left[\mathbf{vC}{:}\text{VCPU}(\mathtt{C}^{\text{D}})\right], \mathbf{P_E}{:}\text{PGA}\left[\mathbf{A}{:}\text{AG"}(\mathtt{C}^{\text{AG}})^{\{\langle \text{VCPU,C1}\rangle\}\cup\rightarrow_{\text{AG}}}_{\{\text{VCPU, C0, C1, VM}\}}[]\right]\right]$$
$$\mathbf{C}{:}\text{C0}(\mathtt{C}^{\text{X}})\,[] \qquad \mathbf{C}{:}\text{C1}(\mathtt{C}^{\text{Y}})\,[\text{VCPU}]$$

Figure 4.2: $\mathbf{vC}$ migration.

machines that have a different set of active CPU traits.

#### 4.2.1.1 Discovery

The set of $\mathbf{vC}$s assigned to a machine can be enumerated from within the virtual machine's operating system by querying `/sys/devices/system/cpu/`. The pool of $\mathbf{C}$s available to a virtual machine is set using *control groups* (Section 4.2.2), which, when using libvirt, are mounted at `/machine/vm-libvirt-qemu/emulator/`, and can be modified by an agent with capabilities over the mount point.

### 4.2.2 Process/Control Groups ($\mathbf{P_E}$)

An agent process running within an operating system can limit the execution of its co-located processes to a set of $\mathbf{C}$s in a number of ways. Notably, the `sched_setaffinity()` system call [Sch] allows a process to set a scheduling mask, which defines the set of cores to which a process can be scheduled. The drawback of restricting execution through affinities is that unprivileged processes can change their own mappings at will, subverting their confinement.

Consequently, SAFEHAVEN uses *control groups* (managed via `cpusets` [Cpu]) to define a hierarchy of $\mathbf{C}$ partitions ($\mathbf{P_E}$). Each $\mathbf{P_E}$ defines a group of cores, and assigning a process to a $\mathbf{P_E}$ confinement limits its execution to the partition's associated cores. Collections of cores can be further partitioned into subgroups, and all processes are initially placed within a default *root* control group. Crucially, processes cannot exit their $\mathbf{P_E}$ confinement by changing affinities, and can only be reassigned to another $\mathbf{P_E}$ by a privileged process. Control groups are manipulated using `cpusets` [Cpu], which allows the creation and destruction of $\mathbf{P_E}$ confinements, as well as the transferring of processes between confinements.

Figure 4.3 defines the sequence of transformations that must be undertaken when migrating a process group PG from one $\mathbf{C}$ allocation to another. The agent executes within the operating system of which PG makes part. As in the case of $\mathbf{vC}$ migration, since the effects are local to the operating system, both the source and destination agents are the same agent,

$$\frac{\begin{array}{c}\mathbf{OS}{:}\textsc{os}\left[\mathbf{P_E}{:}\textsc{pg}(\mathtt{C}^{\mathrm{D}}),\mathbf{P_E}{:}\textsc{pga}\left[\mathbf{A}{:}\textsc{ag}(\mathtt{C}^{\mathrm{AG}})^{\{\langle \textsc{pg},\mathtt{c0}\rangle\}\cup\rightarrow_{\textsc{ag}}}_{\{\mathtt{c0},\ \mathtt{c1},\ \textsc{pg}\}}[]\right]\right]\\ \mathbf{C}{:}\mathtt{c0}(\mathtt{C}^{\mathrm{X}})\,[\textsc{pg}]\qquad \mathbf{C}{:}\mathtt{c1}(\mathtt{C}^{\mathrm{Y}})\,[]\qquad \textsc{ag}\equiv\textsc{pg}\curvearrowright\textsc{ag}.\textsc{ag}'\\ \mathtt{C}^{\mathrm{AG}}\Subset\mathtt{C}^{\mathrm{D}}\qquad \mathtt{C}^{\mathrm{AG}}\Subset\mathtt{C}^{\mathrm{X}}\end{array}}{\begin{array}{l}[\textsc{os}\in\mathbf{M}{:}\textsc{m},\\ \mathtt{c0}\in\mathbf{M}{:}\textsc{m},\\ \mathtt{c1}\in\mathbf{M}{:}\textsc{m}]\end{array}\quad\begin{array}{c}\mathbf{OS}{:}\textsc{os}\left[\mathbf{P_E}{:}\textsc{pga}\left[\mathbf{A}{:}\textsc{ag}'(\mathtt{C}^{\mathrm{AG}})^{\rightarrow_{\textsc{ag}}}_{\{\mathtt{c0},\ \mathtt{c1},\ \textsc{pg}\}}\left[\mathbf{P_E}{:}\textsc{pg}(\mathtt{C}^{\mathrm{D}})\right]\right]\right]\\ \mathbf{C}{:}\mathtt{c0}(\mathtt{C}^{\mathrm{X}})\,[]\qquad \mathbf{C}{:}\mathtt{c1}(\mathtt{C}^{\mathrm{Y}})\,[]\\ \mathtt{C}^{\mathrm{AG}}\Subset\mathtt{C}^{\mathrm{D}}\qquad \mathtt{C}^{\mathrm{AG}}\Subset\mathtt{C}^{\mathrm{Y}}\end{array}\quad\begin{array}{l}\\ \\ \textsc{ag}'\equiv\textsc{pg}\curvearrowright\mathtt{c1}.\textsc{ag}''\end{array}}{\mathbf{OS}{:}\textsc{os}\left[\mathbf{P_E}{:}\textsc{pg}(\mathtt{C}^{\mathrm{D}}),\mathbf{P_E}{:}\textsc{pga}\left[\mathbf{A}{:}\textsc{ag}''(\mathtt{C}^{\mathrm{AG}})^{\{\langle \textsc{pg},\mathtt{c1}\rangle\}\cup\rightarrow_{\textsc{ag}}}_{\{\mathtt{c0},\ \mathtt{c1},\ \textsc{pg}\}}[]\right]\right]\\ \mathbf{C}{:}\mathtt{c0}(\mathtt{C}^{\mathrm{X}})\,[]\qquad \mathbf{C}{:}\mathtt{c1}(\mathtt{C}^{\mathrm{Y}})\,[\textsc{pg}]}$$

Figure 4.3: $\mathbf{P_E}$ migration.

yet the intermediate migration is nevertheless global. Process group migrations are always internal to a machine, although cross-machine migrations can be approximated by first creating a process group at the target, and then subsequently migrating the processes from the source machine.

#### 4.2.2.1 Discovery

Control groups, their **C** allocations and their assigned processes can be enumerated through the `cpusets` interface.

### 4.2.3 Processes and Containers (P, Con)

Using SAFEHAVEN, processes (**P**) and LXC [Lxc] containers (**Con**) are always placed within a control group, and are migrated from one control group to another. Nevertheless, two separate mechanisms are used, the choice of which depends on whether the target **Con** shares the same **OS** as the origin, or if the target exists within a different **OS**.

$$\frac{\begin{array}{c}\mathbf{OS}{:}\textsc{os}\left[\mathbf{P_E}{:}\textsc{pgs}(\mathtt{C}^{\mathrm{X}})\,[\mathbf{P}{:}\textsc{p}],\mathbf{P_E}{:}\textsc{pgd}(\mathtt{C}^{\mathrm{Y}})\,[],\mathbf{P_E}{:}\textsc{pga}\left[\mathbf{A}{:}\textsc{ag}(\mathtt{C}^{\mathrm{AG}})^{\{\langle \textsc{p},\textsc{pgs}\rangle\}\cup\rightarrow_{\textsc{ag}}}_{\{\textsc{pgs},\ \textsc{pgd},\ \textsc{p}\}}[]\right]\right]\\ \mathtt{C}^{\mathrm{AG}}\Subset\mathtt{C}^{\mathrm{X}}\qquad \textsc{ag}\equiv\textsc{p}\curvearrowright\textsc{ag}.\textsc{ag}'\end{array}}{\begin{array}{c}\mathbf{OS}{:}\textsc{os}\left[\mathbf{P_E}{:}\textsc{pgs}(\mathtt{C}^{\mathrm{X}})\,[],\mathbf{P_E}{:}\textsc{pgd}(\mathtt{C}^{\mathrm{Y}})\,[],\mathbf{P_E}{:}\textsc{pga}\left[\mathbf{A}{:}\textsc{ag}'(\mathtt{C}^{\mathrm{AG}})^{\rightarrow_{\textsc{ag}}}_{\{\textsc{pgs},\ \textsc{pgd},\ \textsc{p}\}}[]\right]\right]\\ \mathtt{C}^{\mathrm{AG}}\Subset\mathtt{C}^{\mathrm{Y}}\end{array}\quad\begin{array}{l}\\ \textsc{ag}'\equiv\textsc{p}\curvearrowright\mathtt{c1}.\textsc{ag}''\end{array}}{\mathbf{OS}{:}\textsc{os}\left[\mathbf{P_E}{:}\textsc{pgs}(\mathtt{C}^{\mathrm{X}})\,[],\mathbf{P_E}{:}\textsc{pgd}(\mathtt{C}^{\mathrm{Y}})\,[\mathbf{P}{:}\textsc{p}],\mathbf{P_E}{:}\textsc{pga}\left[\mathbf{A}{:}\textsc{ag}''(\mathtt{C}^{\mathrm{AG}})^{\{\langle \textsc{p},\textsc{pgd}\rangle\}\cup\rightarrow_{\textsc{ag}}}_{\{\textsc{pgs},\ \textsc{pgd},\ \textsc{p}\}}[]\right]\right]}$$

Figure 4.4: $\mathbf{P}$ migration, intra-OS.

Figure 4.4 represents the migration sequence invoked when migrating a process from a group PGS to PGD, where $\textsc{pgs}\overset{\textsc{os}}{\longleftrightarrow}\textsc{pgd}$, leading to a migration that is internal to the parent operating system. Arbitrary processes can be moved directly amongst $\mathbf{P_E}$ groups within

$$\frac{\begin{array}{c}\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})\left[\mathbf{P_E}{:}\textsc{pgs}(\mathtt{C}^{\mathrm{X}})\,[\mathbf{P}{:}\textsc{p}]\,,\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src}(\mathtt{C}^{\textsc{Src}})^{\{\langle \textsc{p},\textsc{pgs}\rangle\}\cup\rightarrow_{\textsc{Src}}}_{\{\textsc{pgs},\ \textsc{Dst},\ \textsc{p}\}}\,[]\right]\right]\\ \mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})\left[\mathbf{P_E}{:}\textsc{pgd}(\mathtt{C}^{\mathrm{Y}})\,[]\,,\mathbf{P_E}{:}\textsc{pgda}\left[\mathbf{A}{:}\textsc{Dst}(\mathtt{C}^{\textsc{Dst}})^{\rightarrow_{\textsc{Dst}}}_{\{\textsc{pgd},\ \textsc{Src}\}}\,[]\right]\right]\\ \mathtt{C}^{\textsc{Src}}\Cap\mathtt{C}^{\mathrm{X}}\qquad \textsc{Src}\equiv\textsc{p}\curvearrowright\textsc{Src.Src'}\end{array}}{\frac{\begin{array}{c}\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})\left[\mathbf{P_E}{:}\textsc{pgs}(\mathtt{C}^{\mathrm{X}})\,[]\,,\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src'}(\mathtt{C}^{\textsc{Src}})^{\rightarrow_{\textsc{Src}}}_{\{\textsc{pgs},\ \textsc{Dst},\ \textsc{p}\}}\,[\mathbf{P}{:}\textsc{p}]\right]\right]\\ \mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})\left[\mathbf{P_E}{:}\textsc{pgd}(\mathtt{C}^{\mathrm{Y}})\,[]\,,\mathbf{P_E}{:}\textsc{pgda}\left[\mathbf{A}{:}\textsc{Dst}(\mathtt{C}^{\textsc{Dst}})^{\rightarrow_{\textsc{Dst}}}_{\{\textsc{pgd},\ \textsc{Src'}\}}\,[]\right]\right]\\ \mathtt{C}^{\textsc{Src}}\Cap\mathtt{C}^{\textsc{Dst}}\qquad \textsc{Src'}\equiv\textsc{p}\curvearrowright\textsc{Dst.Src''}\end{array}}{\frac{\begin{array}{c}\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})\left[\mathbf{P_E}{:}\textsc{pgs}(\mathtt{C}^{\mathrm{X}})\,[]\,,\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src''}(\mathtt{C}^{\textsc{Src}})^{\rightarrow_{\textsc{Src}}}_{\{\textsc{pgs},\ \textsc{Dst}\}}\,[]\right]\right]\\ \mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})\left[\mathbf{P_E}{:}\textsc{pgd}(\mathtt{C}^{\mathrm{Y}})\,[]\,,\mathbf{P_E}{:}\textsc{pgda}\left[\mathbf{A}{:}\textsc{Dst}(\mathtt{C}^{\textsc{Dst}})^{\rightarrow_{\textsc{Dst}}}_{\{\textsc{pgd},\ \textsc{Src''},\ \textsc{p}\}}\,[\mathbf{P}{:}\textsc{p}]\right]\right]\\ \mathtt{C}^{\textsc{Dst}}\Cap\mathtt{C}^{\mathrm{Y}}\qquad \textsc{Dst}\equiv\textsc{p}\curvearrowright\textsc{pgd.Dst'}\end{array}}{\begin{array}{c}\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})\left[\mathbf{P_E}{:}\textsc{pgs}(\mathtt{C}^{\mathrm{X}})\,[]\,,\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src''}(\mathtt{C}^{\textsc{Src}})^{\rightarrow_{\textsc{Src}}}_{\{\textsc{pgs},\ \textsc{Dst},\ \textsc{p}\}}\,[]\right]\right]\\ \mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})\left[\mathbf{P_E}{:}\textsc{pgd}(\mathtt{C}^{\mathrm{Y}})\,[\mathbf{P}{:}\textsc{p}]\,,\mathbf{P_E}{:}\textsc{pgda}\left[\mathbf{A}{:}\textsc{Dst'}(\mathtt{C}^{\textsc{Dst}})^{\{\langle \textsc{p},\textsc{pgd}\rangle\}\cup\rightarrow_{\textsc{Dst}}}_{\{\textsc{pgd},\ \textsc{Src''},\ \textsc{p}\}}\,[]\right]\right]\end{array}}}}$$

Figure 4.5: **P** migration, inter-OS.

the same **OS** using `cpusets`, which is fast and can be performed in bulk. As with previous instances, the fact that the source and destination process groups share the same environment typically entails that both the source and destination confinements are managed by the same agent process.

Figure 4.5 describes the second type of process migration operation, where the participating process groups exist within separate operating systems. There is no requirement that the operating systems exist within different physical machines, and processes may be migrated between virtual machine environments co-located over the same machine.

Transferring processes across operating systems is significantly more complex than internal migrations, as additional mechanisms must be used to package, translate and rebuild the process' data structures at the destination. In SAFEHAVEN, this is handled using `criu` [Cri], which enables process checkpoint/restore from within user-space. Recent versions of the Linux kernel (3.11 onwards) have built-in support for the constructs required by `criu`.

To perform a migration, the source agent SRC deschedules the process being migrated (P), and transfers it to the idle queue of DST, which resides within a different **OS** environment. This was implemented by having SAFEHAVEN's agents negotiate a migration, which then delegate the process transfer to the `criu` daemon.

Cross-**OS** process migration comes with some limitations. Trivially, processes that are critical to their parent **OS** cannot be migrated away. Other restrictions stem from a process' use of shared resources. For instance, the use of interprocess communication may result in unsafe migrations, as the process will be disconnected from its endpoints. Similarly, a process cannot be migrated if it would cause a conflict at the destination, such as in the case of overlapping process IDs or changing directory structures. This problem is addressed by launching a process with its own *namespaces*, or more generally, by using a container [Cri].

Migration preserves a process' $\mathbf{P_E}$ containment structure.

At the time that this research was conducted, LXC live migration was still under active development. Consequently, a stop-gap measure was used to implement migration, whereby a full checkpoint and restore were performed, transferring the frozen image in a separate step using `rsync` [Tyc14]. As will be seen in Section 4.4.3.7, this has a severe impact on performance, yet it allows the SAFEHAVEN approach to be realised, and future improvements to the migration method can be easily dropped in without necessitating any significant changes to the architecture.

#### 4.2.3.1 Discovery

Process enumeration is performed through the `ps` command, which lists the processes within the invoker's environment. As will be seen in Section 4.4.2.4, a process may also scan the `/proc` directory, which, being a synthetic file system, allows for a scanning process to quickly and efficiently sweep through a process list.

### 4.2.4 Virtual Machines (VM)

Similar to process migration, **VM**s can be migrated locally (changing **C** pinnings) using $\mathbf{P_E}$ groups and `cpusets`, or at the global level (moving to another **OS**). The latter is performed in SAFEHAVEN using `QEMU`'s *live migration* operators, backed by a *Network File System* (NFS) server storing **VM** images.

$$\frac{\begin{array}{c}\mathbf{M}{:}\textsc{ms}\left[\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})\left[\mathbf{VM}{:}\textsc{vm}(\mathtt{C}^{\mathrm{D}})\left[\mathbf{vC}{:}\textsc{v}(\mathtt{C}^{\mathrm{D}})\right]\right],\mathbf{C}{:}\textsc{cs}(\mathtt{C}^{\mathrm{X}})\left[\mathbf{vC}{:}\textsc{v}(\mathtt{C}^{\mathrm{D}})\right],\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src}(\mathtt{C}^{\textsc{Src}})^{\{\langle \textsc{v},\textsc{cs}\rangle,\langle \textsc{vm},\textsc{oss}\rangle\}\cup\rightarrow_{\textsc{Src}}}_{\{\textsc{Dst},\ \textsc{vm},\ \textsc{cs},\ \textsc{v},\ \textsc{oss}\}}[]\right]\right]\\ \mathbf{M}{:}\textsc{md}\left[\mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})[],\mathbf{C}{:}\textsc{cd}(\mathtt{C}^{\mathrm{Y}})[],\mathbf{P_E}{:}\textsc{pgsb}\left[\mathbf{A}{:}\textsc{Dst}(\mathtt{C}^{\textsc{Dst}})^{\rightarrow_{\textsc{Dst}}}_{\{\textsc{Src},\ \textsc{cd},\ \textsc{osd}\}}[]\right]\right]\\ \mathtt{C}^{\textsc{Src}}\Cap\mathtt{C}^{\mathrm{X}}\qquad\mathtt{C}^{\textsc{Src}}\Cap\mathtt{C}^{\mathrm{D}}\qquad\textsc{Src}\equiv\textsc{vm}\curvearrowright\textsc{Src}.\textsc{Src}'\end{array}}{\begin{array}{c}\mathbf{M}{:}\textsc{ms}\left[\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})[],\mathbf{C}{:}\textsc{cs}(\mathtt{C}^{\mathrm{X}})[],\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src}'(\mathtt{C}^{\textsc{Src}})^{\rightarrow_{\textsc{Src}}}_{\{\textsc{Dst},\ \textsc{vm},\ \textsc{cs},\ \textsc{v},\ \textsc{oss}\}}\left[\mathbf{VM}{:}\textsc{vm}(\mathtt{C}^{\mathrm{D}})\left[\mathbf{vC}{:}\textsc{v}(\mathtt{C}^{\mathrm{D}})\right]\right]\right]\right]\\ \mathbf{M}{:}\textsc{md}\left[\mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})[],\mathbf{C}{:}\textsc{cd}(\mathtt{C}^{\mathrm{Y}})[],\mathbf{P_E}{:}\textsc{pgsb}\left[\mathbf{A}{:}\textsc{Dst}(\mathtt{C}^{\textsc{Dst}})^{\rightarrow_{\textsc{Dst}}}_{\{\textsc{Src}',\ \textsc{cd},\ \textsc{osd}\}}[]\right]\right]\\ \mathtt{C}^{\textsc{Src}}\Cap\mathtt{C}^{\textsc{Dst}}\qquad\textsc{Src}'\equiv\textsc{vm}\curvearrowright\textsc{Dst}.\textsc{Src}''\end{array}}$$

$$\frac{\begin{array}{c}\mathbf{M}{:}\textsc{ms}\left[\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})[],\mathbf{C}{:}\textsc{cs}(\mathtt{C}^{\mathrm{X}})[],\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src}''(\mathtt{C}^{\textsc{Src}})^{\rightarrow_{\textsc{Src}}}_{\{\textsc{Dst},\ \textsc{cs},\ \textsc{oss}\}}[]\right]\right]\\ \mathbf{M}{:}\textsc{md}\left[\mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})[],\mathbf{C}{:}\textsc{cd}(\mathtt{C}^{\mathrm{Y}})[],\mathbf{P_E}{:}\textsc{pgsb}\left[\mathbf{A}{:}\textsc{Dst}(\mathtt{C}^{\textsc{Dst}})^{\rightarrow_{\textsc{Dst}}}_{\{\textsc{Src}'',\ \textsc{vm},\ \textsc{cd},\ \textsc{v},\ \textsc{osd}\}}\left[\mathbf{VM}{:}\textsc{vm}(\mathtt{C}^{\mathrm{D}})\left[\mathbf{vC}{:}\textsc{v}(\mathtt{C}^{\mathrm{D}})\right]\right]\right]\right]\\ \mathtt{C}^{\textsc{Dst}}\Cap\mathtt{C}^{\mathrm{Y}}\qquad\mathtt{C}^{\textsc{Dst}}\Cap\mathtt{C}^{\mathrm{D}}\qquad\textsc{Dst}\equiv\textsc{vm}\curvearrowright\textsc{osd}.\textsc{Dst}'\end{array}}{\begin{array}{c}\mathbf{M}{:}\textsc{ms}\left[\mathbf{OS}{:}\textsc{oss}(\mathtt{C}^{\mathrm{X}})[],\mathbf{C}{:}\textsc{cs}(\mathtt{C}^{\mathrm{X}})[],\mathbf{P_E}{:}\textsc{pgsa}\left[\mathbf{A}{:}\textsc{Src}''(\mathtt{C}^{\textsc{Src}})^{\rightarrow_{\textsc{Src}}}_{\{\textsc{Dst}',\ \textsc{cs},\ \textsc{oss}\}}[]\right]\right]\\ \mathbf{M}{:}\textsc{md}\left[\mathbf{OS}{:}\textsc{osd}(\mathtt{C}^{\mathrm{Y}})\left[\mathbf{VM}{:}\textsc{vm}(\mathtt{C}^{\mathrm{D}})\left[\mathbf{vC}{:}\textsc{v}(\mathtt{C}^{\mathrm{D}})\right]\right],\mathbf{C}{:}\textsc{cd}(\mathtt{C}^{\mathrm{Y}})\left[\mathbf{vC}{:}\textsc{v}(\mathtt{C}^{\mathrm{D}})\right],\mathbf{P_E}{:}\textsc{pgsb}\left[\mathbf{A}{:}\textsc{Dst}'(\mathtt{C}^{\textsc{Dst}})^{\{\langle \textsc{v},\textsc{cd}\rangle,\langle \textsc{vm},\textsc{osd}\rangle\}\cup\rightarrow_{\textsc{Dst}}}_{\{\textsc{Src}'',\ \textsc{vm},\ \textsc{cd},\ \textsc{v},\ \textsc{osd}\}}[]\right]\right]\end{array}}$$

Figure 4.6: **VM** migration.

Figure 4.6 describes the sequence of steps required to migrate a virtual machine from one physical machine to another. This operation is a compound action, as it migrates the **VM**'s **vC**s in addition to the **VM**'s constituents. The **vC**s are represented explicitly within the transformation rules as they lie on the boundary between the virtualisation layer and the underlying hardware, and interface directly with the **C** layer.

SAFEHAVEN uses KVM for virtualisation, managed via libvirt and QMP. In the case of a cloud infrastructure, the provider's agents exist within the base **OS**, running alongside a tenant's **VM**. The framework can easily be retargeted to Xen or Xen-like virtualisation platforms, with hypervisor-level agents residing within dom0. The choice of hypervisor largely determines what type of instrumentation can be made available to probes.

#### 4.2.4.1 Live Migration Modes

Virtual machines are typically large structures, and transmitting their state data from one machine to another over a network takes an appreciable amount of time, despite several attempts to minimise the amount of state that must be transferred [HSS15; Jo+13].

Modern virtualisation platforms such as QEMU [Qemb] implement *live migration*, where a VM is moved to a different physical machine with minimal downtime by letting the virtual machine execute throughout the greater part of the transfer process. Migration is carried out in two phases, namely

i) a *state* transfer phase, where a VM's memory contents are sent to the destination, and

ii) a *control* transfer phase, where a VM stops executing at the source and resumes at the destination.

Control transfer also includes setup and tear-down procedures such as device initialisation and network announcements.

The order in which these phases are carried out leads to two approaches to live migration, namely *pre-copy* and *post-copy* migration [Ahm+15]. Pre-copy works by iteratively transferring a VM's pages, with each iteration transmitting pages that were dirtied while sending the previous iteration. This process is repeated until the number of dirty pages falls below a set threshold, at which point the VM is paused and resumed at the target following a final transfer.

*Post-copy* swaps the transfer phases, transferring a minimum amount of control state and resuming execution immediately at the target machine, with memory pages being pulled on request from the source using demand paging [Mil+00].

Pre-copy migration has two main drawbacks, namely that the machine's entire state must be transferred before control can resume at the target, and that data-intensive processes may invalidate pages at a faster rate than that at which they are being transferred, resulting in *non-convergence*. In contrast, post-copy migration will transfer each page at most once, guaranteeing convergence [HDG09].

The key drawback of post-copy migration is that the VM's state is split between two machines, and a link or node failure during a migration can corrupt the VM. This makes the method less robust than pre-copy migration, which can tolerate failures at the destination

during a migration (the operation is simply aborted, and execution continues at the source machine). This limitation can be mitigated in part using *hybrid migration*, whereby a migration is initiated in pre-copy and switches to post-copy on detecting non-convergence, which would happen when memory pages are being modified faster than they can be transferred. Additional countermeasures will be elaborated upon in Section 5.4.2.

Pre-copy migration support comes as standard with current versions of `QEMU` [Qemb]. Post-copy support requires that the base operating system hosting the virtual machines in question are equipped with the `userfaultfd` kernel extensions, which are implemented within a fork [Arc16]. These extensions expose a file descriptor from kernel-space to user-level processes over which the kernel announces the addresses of faulting pages, and must be present both at the source and destination environments. Post-copy functionality in `QEMU` making use of these extensions was also implemented as a fork [Orb16]. Similarly, libvirt was extended [Kle16] to expose the post-copy migration functionality to system processes via the virtualisation driver. All testing and implementation was performed on QEMU v2.2.92.

Beyond its guarantee for convergence, post-copy virtual machine migration is especially valuable in the context of illicit channel mitigation, as it rapidly breaks a virtual machine's co-locations with other entities executing at the source environment. Conversely, a virtual machine being evicted using pre-copy migration will keep executing at the source for the duration of the migration operation, leaving the VM vulnerable to attacks at the source. This is particularly detrimental in the case of time-critical or high-bandwidth channels. This notion of responsiveness, and the boon of convergence, will be explored in further detail later on in this chapter.

#### 4.2.4.2 Discovery

The list of running and idle virtual machines can be produced via `virsh`, which integrates with the libvirt virtualisation driver.

### 4.2.5 Additional Operations

In addition to admitting migration, **VM**, **P** and **Con** confinements can be paused in memory, which can serve as a temporary compromise in cases where an imminent threat cannot be mitigated quickly enough through migration.

## 4.3 Agents

The core operations of an agent process are confinement reconnaissance and migration. Agents perform both *objective* and *subjective moves* [CG98], as they can migrate confinements

**Algorithm 4.1** SAFEHAVEN confinement data structures.

```
1: % Confinement types (atoms)
2: -type loc_t() ::  l_network | l_machine | l_l3 | l_l2 | l_l1
3:                 | l_cache | l_cpu | l_kvm_vcpu | l_os | l_kvm_vm
4:                 | l_container | l_cgroup | l_proc | l_nfs.
5:
6: -type name_t():: string().       % Confinement name data type
7: -type id_t()  :: term().         % Confinement handle data type
8:
9: -record(locality,                % Confinement data structure
10:   { type   :: loc_t(),                  % Confinement type
11:     name   :: name_t(),                 % Symbolic name
12:     id     :: id_t(),                   % Handle (IP, PID, etc.)
13:     caps   :: term(),                   % Capabilities
14:     sublocs = [] :: [locality_t()] }).  % Sub-confinements
15:
16: -type locality_t() :: #locality{}.
```

to which they belong as well as external confinements. The remainder of this section will demonstrate these fundamental agent duties by means of an example.

### 4.3.1 Data Structures

Algorithm 4.1 defines the fundamental Erlang data structure used within SAFEHAVEN, namely the `locality` record type. These records are used as references to confinements that exist within the system, and allow properties and agents to handle confinements uniformly. The `loc_t()` data type lists the types of localities that are understood by the framework. Defining the type of a `locality` structure allows SAFEHAVEN to determine which migration method to apply when moving a confinement.

### 4.3.2 A Process-Level Agent

Algorithm 4.2 defines a prototypical SAFEHAVEN agent that partitions a set of processes amongst a set of cores depending on their owner. This requires both enumeration and migration, which are employed as follows.

**Algorithm 4.2** Agent partitioning processes between CPUs by owner via SAFEHAVEN.

```
 1: Procs = process:recon(),          % Get system processes
 2: [CR, CA|Cs] = cpu:recon(),        % Get available CPUs
 3: lists:foreach(
 4:   fun(P = #locality{type = l_proc, id = ID}) ->
 5:     User = owner(ID),
 6:     Dest = case User of           % Choose destination CPU
 7:              "root"   -> CR;
 8:              "apache" -> CA;
 9:              _        -> Cs
10:            end,
11:     mig_process:migrate(P, Dest)  % Pin process
12:   end, Procs).
```

#### 4.3.2.1 Confinement Discovery and Enumeration

The view of an arbitrary agent within a cloud is generally limited to its immediate environment and that of other agents with which it is co-operating. For example, a tenant's agents will be restricted to the processes and structures of their **OS** environment. Similarly, the cloud provider views **VM**s as black boxes. Knowledge of their internal structures is limited to what is exposed by the tenants' agents, bar the use of introspection or disassembly mechanisms.

The agent described in Algorithm 4.2 can directly query its **OS** environment to retrieve the list of running processes (Line 1) and available cores (Line 2). Both `recon()` operations return lists of confinements. In the latter case, the resultant list is split into three sub-lists, namely

1. a single element list `CR` to which processes owned by *root* are to be migrated,
2. a single element list `CA`, designated for processes owned by the *apache* user, and
3. `CS`, a group of cores over which the rest of the processes are to execute.

Note that the above partitioning assumes that the system has a minimum of three cores. Both reconnaissance operations can be run using ordinary user privileges.

As discussed earlier, a subtle consequence of virtualisation is that correctly virtualised cores are seen by the confined processes as being actual, physical cores. Thus, for the agent being considered, the `cpu:recon()` function would always return a set of seemingly physical cores, yet an external agent would correctly discern them as being virtual.

While side-channels and advanced fingerprinting techniques may allow an agent to determine additional confinement attributes (Section 2.2), SAFEHAVEN restricts itself to overt channels and named interfaces.

#### 4.3.2.2 Migration

The second component of the agent iterates through the list of environment processes `CR` and determines the destination of each process based on its owner, which is determined via a helper function (Line 5). The agent then invokes process migration (Line 11), moving the process under consideration to its target core. Internally, the migrate process translates the set of cores to a process group, and migration to a $\mathbf{P_E}$ group is carried out as detailed in Section 4.2.3.

Note that in this example, the agent directly invokes the process migration routine, as the object type of P is known beforehand. Alternatively, a meta-level and generic `migrate()` operation can be invoked, which executes the relevant migration operation based on the types of the confinements constituting its arguments. While this allows properties to be expressed using more general types, one cannot always design agents that are completely oblivious to the migration method being used. For example, while an agent may afford to block processes' execution pending short migrations, long lived migrations (such as when moving virtual machines) may be executed in blocking or non-blocking modes, necessitating an additional argument.

### 4.3.3 Communication

Communication within SAFEHAVEN is carried out using Erlang's message passing facilities. Processes can only message others that share a token (a *magic cookie* [Erl]) that serves as a communication capability.

Erlang's message passing enables agents to communicate uniformly with each other, and hides many of the intricacies of the underlying network topology. The primary concern of agents is thus how best to advertise each other's addresses (or Erlang PIDs). One simple and effective method is to have every agent participating in implementing a policy connect to a known Erlang node. By enabling automatic announcements, each additional agent dialling into the known node will be added to a network of processes, each of which can communicate with each other, provided that their cookies also match. Finer grained agent networks can be created by assigning per-connection cookies, and disabling automatic node propagation.

The choice of using *avahi* [Ava] within SAFEHAVEN was based on the principle of decentralisation and the aim of supporting highly fluid infrastructures. While broadcasts may simplify the setup process of a confinement following a migration, it may lead to scalability

issues, and one may have to resort to alternative nameserver schemes.

### 4.3.4 Allocation

To determine a destination for a confinement that must be migrated, an agent broadcasts an isolation request to its known agents. If one of these agents finds that it can serve the request whilst maintaining its existent isolation commitments, it authorises the migration. The problem of placement is equivalent to the *bin-packing problem* [Aza+14], and a greedy allocation policy will not produce an optimal allocation. Nevertheless, the SAFEHAVEN architecture is sufficiently general so as to allow different allocation strategies. For example, targets can be prioritised based on their physical distance. Prioritisation can also be used in hybrid infrastructures, where certain targets may be more effective at breaking specific types of co-locations than others. For example, a cloud provider can opt to mix in a number of machines with various hardware confinements and lease them on demand. This principle will be explored in greater detail in Section 5.3.6.

## 4.4 Implementing Detection and Mitigations

The fine-grained approach to isolation advocated by SAFEHAVEN serves two purposes. First, it increases utilisation by promoting the use of the minimum unit of isolation required to secure a confined entity. This has the knock-on effect of reducing the amount of hardware that is left underutilised due to restrictions on co-locations. The second is that smaller confinements are faster to migrate, and incur fewer overheads than larger confinements. The following section attempts to substantiate the latter claim by examining the performance impact of different migration mechanisms. It also describes how SAFEHAVEN can be used to mitigate a machine-wide covert channel attack, as well as to implement a multi-level moving target defence.

### 4.4.1 Experimental Setup

Table 4.2 provides a summary of the core attributes of the different physical and virtual machine configurations used throughout the following experiments.

All experiments were carried out on two Intel i7-4790 machines (4 cores $\times$ 2 hardware threads) with 8GB RAM ($\text{INTEL}_\text{T}$). A third computer ($\text{AMD}_\text{T}$), an AMD Phenom II X6 1090T with 8GB RAM, served as an NFS server hosting the VMs' images (average sequential speeds: 81MB/s read, 76 MB/s write), and was also used in the experiment described in Section 5.3.6. Machines communicated via a consumer-grade gigabit switch. All machines ran Ubuntu 14.04 LTS with the 3.19.0-rc2+ kernel patched for post-copy support (Section 4.2.4.1), and libvirt

| Machine ID | CPU | Cores | Threads per Core | RAM (Gb) |
|---|---|---|---|---|
| INTEL-$M_T$ | Intel i7-2640M | 2 | 2 | 4 |
| $INTEL_T$ | Intel i7-4790 | 4 | 2 | 8 |
| $AMD_T$ | AMD Phenom II X6 1090T | 6 | 1 | 8 |
| $VM_T$ | - | 2 | 1 | 2 |

Table 4.2: Hardware configurations used during evaluation.

version 1.2.11. Each VM ($VM_T$) was allocated two vCPUs and 2GB of RAM, and had a 20GB image. VMs used bridged network interfaces, and host discovery was performed using *avahi* [Ava].

#### 4.4.1.1 Benchmarks

The evaluation of the performance aspects of migration call for repeatable workloads that are representative of the deployment scenarios under consideration, namely, clouds. The PARSEC [Bie+08] benchmark suite was used to generate a variety of intense and mixed workloads, with varying pressures on memory and computational capacity. The evaluation focuses on a subset of the workloads present in the suite, namely

- `blackscholes`, a financial analysis application with small working sets,
- `canneal`, a simulated annealing benchmark with large working sets,
- `streamcluster`, a data mining and clustering application with moderate working sets,
- `dedup`, a storage deduplicator with large working sets,
- `raytrace`, a rendering engine with large working sets, and
- `bodytrack`, a computer vision application with moderate working sets.

In addition, the `all` benchmark was defined. This consists of the consecutive execution of a single run of each of the aforementioned benchmarks, and was used for benchmarks requiring a long-running and mixed workload.

### 4.4.2 System-Wide (Cross-VM) Covert Channel

The following section describes the use of SAFEHAVEN as an active countermeasure to thwart a system-wide covert-channel.

#### 4.4.2.1 Overview

Wu *et al*. [WXW12] demonstrated that performing an atomic operation spanning across a misaligned memory boundary will lock the memory bus of certain architectures, inducing a system-wide slowdown in memory access times. This effect could then be used to implement a cross-VM covert channel by modulating the observed memory access speeds.

#### 4.4.2.2 Detection

Detecting the channel's reader process is difficult, as it mostly performs low-key memory and timing operations, and would execute in a co-located **VM**, placing it outside the victim tenant's scope. Conversely, writer processes are relatively conspicuous, in that they perform memory operations that are *atomic* and *misaligned*. Atomic instructions are used in very restricted contexts, and compilers generally align a program's memory locations to the architecture's native width. Having both simultaneously can thus be taken as a strong indication that a program is misbehaving.

Although an attack can be detected by replicating a reader process, a much more direct, precise and efficient method is to use *hardware event counters* [Int11] to measure the occurrence of misaligned atomic accesses. Recent versions of `KVM` virtualise a system's performance monitoring unit, allowing **VM**s to count events within their domain [DSZ10]. One limitation of hardware counters is that their implementation is not uniform across vendors, complicating their use in heterogeneous systems. In addition, while event counters are confined to their **VM** and can only be used by privileged users, one must ensure that they do not themselves enable attacks (for instance, by exposing a high resolution timer).

#### 4.4.2.3 Policy

Algorithm 4.3 and Algorithm 4.4 outline the behaviour of the agents operated by the tenant and cloud provider, respectively, while Figure 4.7 illustrates how the components taking part in the mitigation interoperate. Each agent takes two arguments, namely the isolation that they are monitoring and a list of additional cooperating agents. When a probe detects that a process P is emitting events at a rate exceeding a threshold $\epsilon$, it notifies its local agent. If the environment is not already isolated, then the agent attempts to locate an isolated resource amongst its own existing tenants. Failing this, the cloud provider is co-opted into finding an isolated machine and resolving the request at the virtual machine level. If a process is mobile, then the cloud provider can opt to create a new isolated **VM** to which the process can be migrated, rather than migrating the source machine.

The degree of isolation required is regulated by the $\texttt{isol}_{\mathrm{D}}(\mathrm{X})$ predicate, which checks whether X is isolated within D. Evaluating this accurately from within the tenant's scope

**Algorithm 4.3** Tenant agent for covert channel mitigation.

**Require:** An event rate threshold $\epsilon$

**Require:** $A_T$ set of tenant-owned agents

**agent** TENANT(**OS**:X, $A_T$)
  **for all** **P**:P $\in^+$ X **do**
    **if** $\texttt{evs}(P) \geq \epsilon \wedge \neg\texttt{isol}_X(P)$ **then**
      $D \leftarrow \bot$
      **if** $\texttt{mobile}(P)$ **then**
        $D \in \{D' \mid \text{TENANT}(D', *) \in A_T \wedge \texttt{isol}_{D'}(P)\}$
      **if** $D \neq \bot$ **then**
        $P \curvearrowright D$
      **else if** $X \in$ **VM**:V **then**
        forward isolation request to CLOUD(Y, $*$). $V \in^+ Y$
  TENANT(X, $A_T$)
**end agent**

**Algorithm 4.4** Cloud agent for covert channel mitigation.

**Require:** $A_C$ set of cloud-owned agents

**agent** CLOUD(**M**:Y, $A_C$)
  receive isolation request for **VM**:X $\in^+$ Y
  **if** $\neg\texttt{isol}_Y(X)$ **then**
    $D \in \{D' \mid \text{CLOUD}(D', *) \in A_C \wedge \texttt{isol}_{D'}(X)\}$
    **if** $D \neq \bot$ **then**
      $X \curvearrowright D$
    **else**
      fallback strategies
  CLOUD(Y, $A_C$)
**end agent**

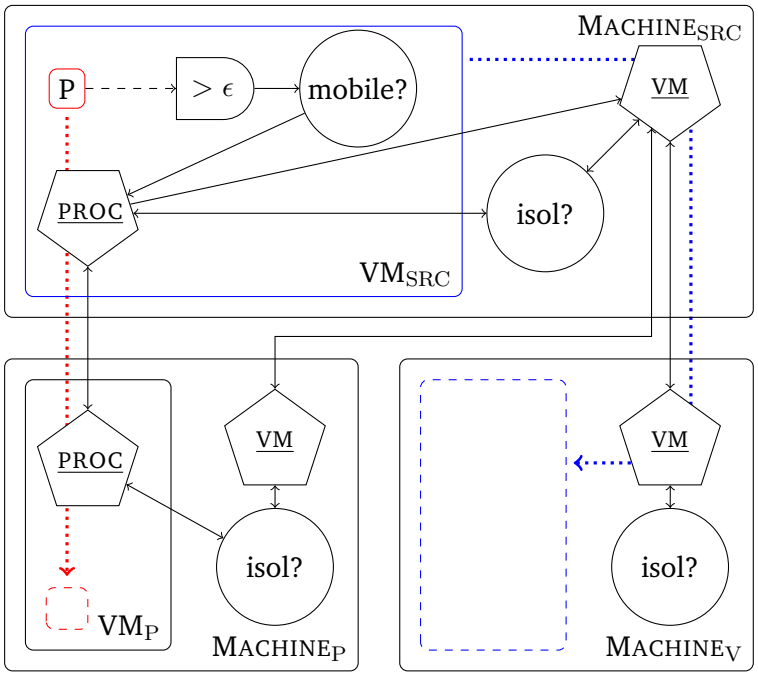



Figure 4.7: Policy showing mitigation for processes ($VM_{SRC}$ [P] $\rightarrow$ $VM_P$ [P]) and virtual machines (MACHINE$_{SRC}$ [$VM_{SRC}$] $\rightarrow$ MACHINE$_V$ [$VM_{SRC}$]). Arrows denote communication channels, with dotted paths denoting migration paths. Rectangles denote locality boundaries, circles are predicate evaluation processes, while pentagons signify agents.

requires additional information from the cloud agent regarding its neighbours. The strictest interpretation of isolation would be to allocate a physical machine to each **VM** requesting isolation. Another approach is to stratify isolation into different classes determined by user access lists [CDRC14], or to only allow a tenant's isolated **VM**s to be co-located with each other.

If an isolated destination cannot be found immediately, then soft isolation must be used as a fallback strategy. Note that soft isolation only has to disrupt the channel until hard isolation is achieved. For example, rather than migrating the locality requesting isolation, one can evict its co-residents, applying soft isolation during their eviction. A simple, general but intrusive method would be to pause the process until isolation is obtained. This should be reserved for creating temporary isolations during fast migration operations. A more targeted mitigation may attempt to degrade the attacker's signal-to-noise ratio by flooding the memory bus with its own misaligned atomic memory accesses. Finally, one may deploy a system such as BusMonitor [SXZ13] on a number of machines and migrate **VM**s requesting isolation to them. The problem with the latter solutions is that they must be changed with each discovered attack, whereas a migration-based approach would only require a change in the detector.

#### 4.4.2.4 Implementation and Evaluation

The policy was implemented in SAFEHAVEN as a network of Erlang server processes, with the detector running as a separate process and taking two parameters, namely

i) a set of system processes $\vec{\mathbf{P}}$ to be scanned, and

ii) a duration $\tau$ within which the scan must be performed.

**Instrumentation** Hardware counters were accessed using the *Performance Application Programming Interface* (PAPI) [Muc+99] library, with calls proxied through an Erlang module using *Native Implemented Functions* (NIF) [Erl]. The $\text{INTEL}_\text{T}$ machines exposed a native event type that counts misaligned atomic accesses (LOCK_CYCLES: SPLIT_LOCK_UC_LOCK_DURATION [Int11]). Conversely, $\text{AMD}_\text{T}$ was found to lack such a combined event type. In this case, one would have to measure misaligned accesses and atomic operations independently, which can lead to a higher rate of false positives.

The procedure for measuring a process' event emission rate is to attach a counter to it, sleep for a sample time $\phi$, and read the number of events generated over that period of time. This is repeated for each process in $\vec{\mathbf{P}}$. The choice of $\phi$ will affect the detector's duty cycle. Setting $\phi = \tau/|\vec{\mathbf{P}}|$ guarantees that each process will have been sampled once within each $\tau$ period, but the sampling window will become narrower as the number of processes increases, raising the frequency of library calls and consequently CPU usage. Setting a fixed $\phi$ produces an even CPU usage, but leads to an unbounded reaction time.

Algorithm 4.5 implements the aforementioned sampling routine as a function named `instrument`, which accepts a $\phi$ (`SampleTime`), a list of processes (`Procs`) and a list of hardware event types (`Events`). The `counters` library is the proxy module to PAPI. This module is used to define the event type to be monitored (Line 4), and to start (Line 8) and stop (Line 10) counting events for a given process. The detector sleeps for the defined $\phi$ (Line 9) between the starting and stopping of the event counter. As defined, the procedure performs a separate sweep for each event in the `Events` set, and tabulates the result in a list. PAPI also allows for multiple event types to be tracked simultaneously. The number of events that can be tracked concurrently is limited by the hardware over which monitoring is taking place. If this limit is exceeded, PAPI resorts to multiplexing [Muc+99].

**Detection Feature** The hypothesis regarding the infrequency of misaligned atomic accesses was tested by sampling each process within virtualised and non-virtualised environments present in the test bed over a minute during normal execution. Most processes produced no events of the type under consideration, with the exception of certain graphical applications such as VNC, which produced intermittent spikes on the order of a few hundreds per

**Algorithm 4.5** Counting process events using PAPI.

```
 1: instrument(SampleTime, Procs, Events) ->
 2:   lists:foldl(
 3:     fun(E, EvAcc) ->
 4:       % Set monitoring event type; 0 on success
 5:       CReads = case counters:papi_init([E]) of
 6:         0 ->
 7:           Samples = lists:foldl(
 8:             % Monitor each process in sequence
 9:             fun(P, CntAcc) ->
10:               [case counters:start_counting(locality:getID(P)) of
11:                  0 -> timer:sleep(SampleTime), % Measure for φ
12:                       [Cnt|_] = counters:stop_counting(),
13:                       Cnt;
14:                  _ -> "-"          % Error
15:                end | CntAcc]
16:             end, [E], Procs),     % Reverse after tail recursion
17:             [lists:reverse(Samples) | EvAcc];
18:         _ ->
19:           EvAcc                   % Bad event; skip
20:         end,
21:
22:         counters:papi_close(),    % Close library
23:         CReads
24:     end, [], Events).
```

second during use. The emission rate of the attack's sender process was then measured using the reference implementation of Wu *et al*. [WXW12], compiled with its defaults. This was found to emit $\approx 1.4 \times 10^6$ events per second in both environments, with attacks for 64-byte transmissions lasting $6 \pm 2$ seconds.

**Efficiency** Invoking PAPI functions via a NIF proxy is convenient, as the detector logic can be completely defined as a high-level SAFEHAVEN agent using Erlang. This enables agents and probes to be quickly prototyped, as they can make use of the full range of functions and data structures made available by the framework.

The drawback of having a thin proxy is that every NIF call triggers a number of intermediate bridging operations, and converting to and from Erlang and C data structures introduces a measurable overhead. Consequently, although the low-level components of the NIF functions used by the probe defined in Algorithm 4.5 execute very quickly, the performance of the probe as a whole is very low. This is because the NIF proxy is called with a very high frequency, with each process triggering two NIF calls, and potentially hundreds of processes being scanned every second.

In order to reduce the number of translations between the SAFEHAVEN proper and the low-level NIF functions, one may opt to transfer the detector logic into the low-level layer, which results in the probe described in Algorithm 4.6. This probe performs the process enumeration directly at the low-level layer by scanning the `/proc/` directory. Communication with SAFEHAVEN is reduced to two functions, namely initialisation (where the events of interest are specified and the monitor is started) and reporting in the event that an offending process is found. Unlike the Erlang-based detector, this approach places the tight loops and polling operations within the native segment, resulting in a massive improvement in performance.

Figure 4.8 shows the detector's CPU usage (measured directly using `top`) against varying $\phi$ using the compiled C probe. To fully characterise the detector's overhead, the virtual machine was pinned to a single vCPU. At $\phi = 10$ms, overhead peaked at a measured 0.3%. This is in contrast to the Erlang-based probe, which would reach an average of 25% utilisation. The performance measurements were further verified by executing the CPU-intensive `blackscholes` computation from the PARSEC benchmark suite [Bie+08] in parallel with the detector. This benchmark ran at full CPU utilisation, and thus directly competed for cycles with the detector, as they both shared a single vCPU. As expected, the reduction in $\phi$ produced a proportional speed up in `blackscholes`'s execution (or, equally, a reduction in the benchmark's total execution time) that follows the direct measurement.

Figure 4.9 considers the *quality* of the detector, in that it describes how the detector's reaction time varied against the number of processes being monitored, where reaction time

**Algorithm 4.6** Shifting detector into native code.

```
 1: static int event_set;   // Event mask; populated during setup
 2: static int ev_cnt;      // Size of event_set
 3:
 4: int scan(int threshold, int sample_time_us, int sleep_time_s) {
 5:   DIR *d;
 6:   int foundpid = 0;                 // !0 => Attacker PID
 7:   long long values[ev_cnt];
 8:   if ((d = opendir("/proc/"))) {    // Query proc directly
 9:     while (!foundpid) {             // Break on violation
10:       struct dirent *dir;
11:       while (!foundpid && (dir = readdir(d)) != NULL) {
12:         int attachpid = atoi(dir->d_name);  // PID
13:         if (attachpid > 0) {        // Start counting events
14:           if ((PAPI_attach(event_set, attachpid) == PAPI_OK) &&
                  (PAPI_start(event_set) == PAPI_OK)) {
15:             usleep(sample_time_us);          // φ
16:             PAPI_stop(event_set, values);   // Stop and record
17:             int iter;       // Look for violation
18:             for (iter = 0; iter < ev_cnt; iter++) {
19:               if (values[iter] > threshold) {
20:                 foundpid = attachpid; break;
21:               }
22:             }
23:             PAPI_detach(event_set);     // Remove watches
24:           }
25:         }
26:         if (!foundpid && sleep_time_s)  // Sleep between scans?
27:           sleep(sleep_time_s);
28:       }
29:       rewinddir(d);
30:     }
31:   }
32:   closedir(d);
33:   return foundpid;
34: }
```

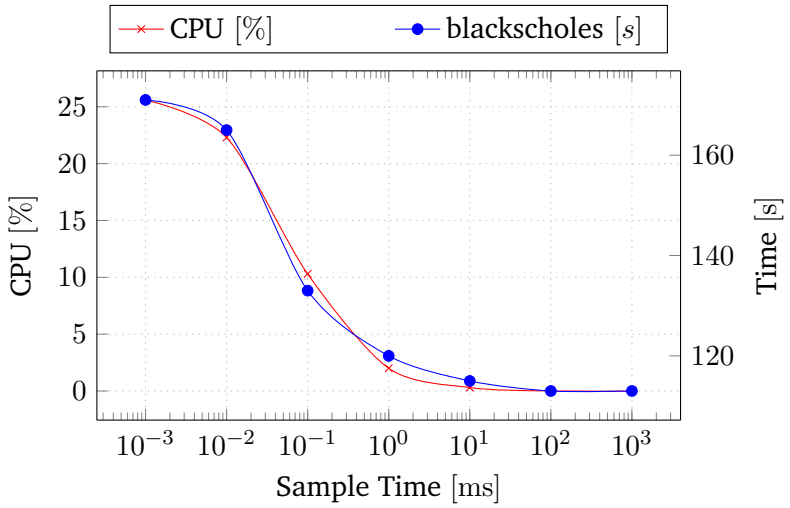



Figure 4.8: Detector overhead against $\phi$.

was measured as the time elapsed between the start of an attack and its detection. The reaction time was measured for $133 \leq |\vec{\mathbf{P}}| \leq 200$. The size of $\vec{\mathbf{P}}$ was raised by spawning additional processes that periodically wrote to an array. The attack was started at random points in time.

#### 4.4.2.5 Mitigation

Once a potential attack is detected, it must be isolated. As the attack is machine-wide, co-location must be broken through virtual machine or process migration. The following section investigates the former, whereas the latter will be discussed in Section 4.4.3.

Figure 4.10 illustrates the worst case times taken to perform a single **VM** live migration using pre-copy, hybrid and post-copy while it executed various workloads from the PARSEC suite. Migrations were triggered at random points during the benchmark's execution, with 6 readings per benchmark and migration mode. The host machines were left idle to reduce additional noise. Solid bars represent the time taken for the **VM** to resume execution at the target machine, and the shaded area denotes the time spent copying over the remainder of the **VM**'s memory pages after it has been moved.

Pre-copy migration's performance was significantly affected by the workload being executed, with `canneal` never converging. Hybrid migration fared better, as it always converged and generated less overall traffic than pre-copy migration. Post-copy exhibited the most consistent behaviour, both in terms of migration time as well as generated traffic.

To perform a post-copy migration in `QEMU`, one must first initiate a normal pre-copy migration and subsequently send a command to the virtualisation platforms to switch to post-copy migration. During the course of the experiments, it was found that attempting to start a mi-

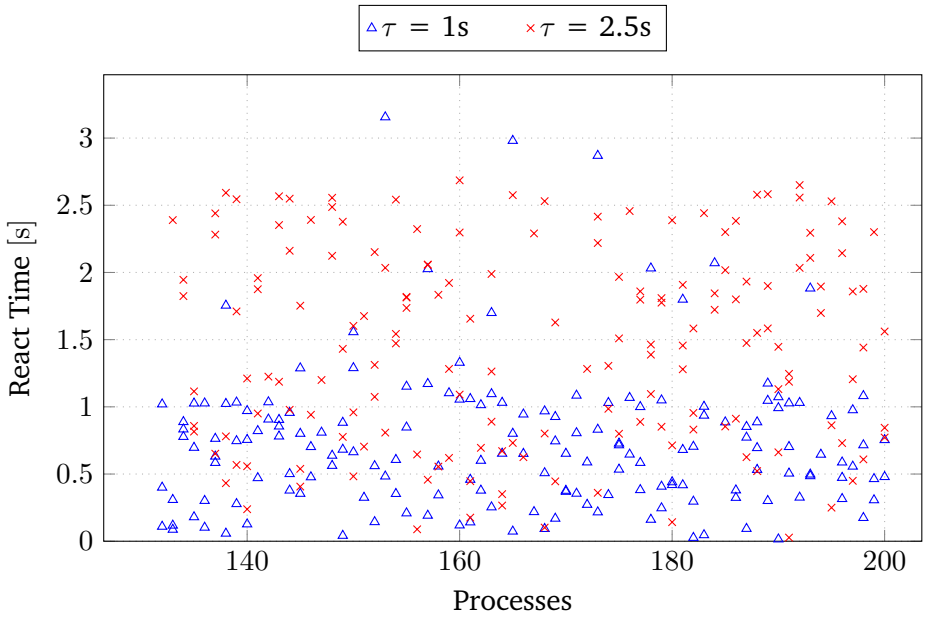



Figure 4.9: Reaction time on varying $\vec{\mathbf{P}}$, $\tau = 1$s and 2.5s.

| Phase | Parameters | Min | Max | Geometric Mean | Arithmetic Mean |
|---|---|---|---|---|---|
| Detect | $\tau = 1$s | 0.0148 | 3.16 | 0.54 | 0.72 |
| | $\tau = 2.5$s | 0.0272 | 2.69 | 1.20 | 1.46 |
| Migrate | Post-copy | 1.2813 | 2.13 | 1.47 | 1.48 |
| Detect & Migrate | Post-copy & $\tau = 1$s | 1.296 | 5.29 | 2.01 | 2.20 |
| | Post-copy & $\tau = 2.5$ | 1.309 | 4.82 | 2.67 | 2.93 |

Table 4.3: Summary of detection and mitigation times (s).

gration immediately in post-copy mode would occasionally trigger a race condition. This was remedied by adding a one second delay before switching to post-copy. As will be seen in the next chapter, this delay can be almost completely eliminated, yet this requires modifications to the `QEMU` migration mechanisms.

Nevertheless, even with the added delay, **VM**s migrated using post-copy resumed execution at the target in at most 2.13 seconds, and 1.51 seconds on average. Total migration time and data transferred were also consistently low, averaging 20 seconds and 2GB, respectively.

Table 4.3 summarises the results. Based on the detector's reaction times and post-copy's switching time, and assuming that a target machine has already been identified, a channel can be mitigated in around 1.3 seconds under ideal conditions, 5.3 seconds in the worst case, and in just under 3 seconds on average.

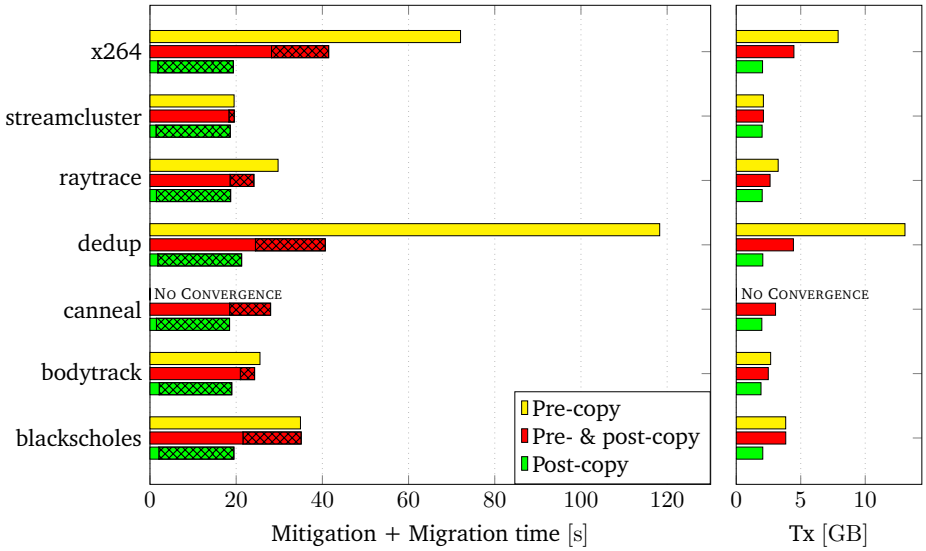



Figure 4.10: Comparison of pre-copy, hybrid and post-copy migration.

#### 4.4.2.6 Conclusion

This section has demonstrated how hardware event counters can be used to detect an attack efficiently, quickly and precisely, and how post-copy migration considerably narrows an attacker's time window. Additional improvements can be obtained by integrating event counting with the scheduling policy, where the event monitor's targets are changed on context switching. This would eliminate the need to sweep through processes and avoids missing events.

### 4.4.3 Moving Target Defence, Revisited

The following describes the use of SAFEHAVEN in implementing a passive and preventive mitigation, specifically, a *moving target defence*.

#### 4.4.3.1 Overview

The risk of an illicit channel being formed between a set of processes increases in proportion to the time they spend co-located [Aza+14; MSR15; Zha+12b]. Similarly, the greater the number of tenants sharing an infrastructure (the *consolidation factor*), the higher the odds of being co-located with a malicious tenant. Consequently, a tenant's security with respect to illicit channels is inversely proportional to the cloud's consolidation factor and inertia.

The moving target defence [Zha+12b] is based on the premise that an attacker co-located with a victim within a confinement D requires a minimum amount of time $\alpha(\mathrm{D})$ to set up and perform its attack. Attacks can thus be foiled by limiting continuous co-location with every other process to at most $\alpha(\mathrm{D})$.

The defence is notable in that it does not attempt to identify a specific attacker, being driven entirely on the basis of co-location. The principle difficulty lies in choosing a minimum value of $\alpha(\mathrm{D})$ which will guarantee the best level of security without any excess migrations.

#### 4.4.3.2 Policy

**Algorithm 4.7** General form of the moving target defence.

**Require:** A root locality R

**for all** $\mathbf{T}{:}\mathrm{L}_0, \mathbf{T}{:}\mathrm{L}_1 \in^+ \mathrm{R}.\ \mathrm{L}_0 \neq \mathrm{L}_1$ **do**

**if** $\exists \mathrm{D} \in^+ \mathrm{R}.\ \tau(\mathrm{L}_0 \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{L}_1) + H(\mathbf{T}) \geq \alpha(\mathrm{D})$ **then**

$\mathrm{L}_{i\,\in\,\{0,1\}} \curvearrowright \mathrm{S}.\ \mathrm{S} \in^+ \mathrm{R} \wedge \neg \mathrm{L}_0 \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{L}_1$

Algorithm 4.7 describes the moving target defence as a generalisation of the formulation given by Zhang *et al*. [Zha+12b]. The policy assumes the existence of three predicates, namely

i) $H(\mathbf{T})$, the time required to migrate a locality of type $\mathbf{T}$,

ii) $\alpha(\mathrm{D})$, the time required to attack a process through D, and

iii) $\tau(\mathrm{P})$, the duration for which a supplied predicate P holds.

The remainder of this section attempts to establish practical approximations for the aforementioned predicates.

#### 4.4.3.3 Defining $H()$

$H()$ must be able to predict the cost of a future migration. In addition, $H()$ varies based on the destination of a migration, thus requiring that the predicate be refined. As an estimate, the next value of $H()$ can be approximated using an *exponential average* [SGG05], expressed as the following recurrence relation:

$$H_{n+1}(\mathrm{T} \curvearrowright \mathrm{D}) = h\eta_n(\mathrm{T} \curvearrowright \mathrm{D}) + (1-h)H_n(\mathrm{T} \curvearrowright \mathrm{D})$$

where $\eta_n()$ is the measured duration of a migration, and $0 \leq h \leq 1$ biases predictions towards historical or current migration times. By convention [SGG05], schedulers take $h = 0.5$, yet it may be constructive to consider other values of $h$ in the case of highly unstable or fluctuating migration times.

#### 4.4.3.4 Defining $\alpha()$

A precise predicate for $\alpha()$ is difficult to define, as it would require a complete characterisation of the potential attacks that a system can face, with knowledge of the state of the art at most bounding the predicate. In the absence of a perfect model, one must adopt a pragmatic approach, whereby the duration of co-locations (and, by association, the migration rate) is determined by the overhead that a tenant will bear, as this is ultimately the limiting factor.

#### 4.4.3.5 Defining $\tau(\Leftrightarrow)$

A tenant can determine the co-location times for processes within its domain, but is otherwise oblivious to other tenants' processes. In the absence of additional isolation guarantees from the cloud provider, $\tau(\Leftrightarrow)$ must be taken as the total time spent at a location, timed from the point of entry, this being the worst-case value of co-location time.

#### 4.4.3.6 Propagating resets

The hierarchical nature of confinements can be leveraged to improve the moving target defence. Migrations at higher levels will break co-locations in their constituents. Thus, following a migration, an agent can propagate a directive to its sub-localities, resetting their $\tau(\Leftrightarrow)$ predicates. Propagation must be selective. For example, while process migration to another machine will break locality at the **OS** and **C** level, **VM** migration only breaks cache and machine-wide locality, and leaves the **OS** hierarchy intact (Section 3.5). Similarly, a lower locality can request isolation from a higher-level parent to trigger a bulk migration action, which can resolve multiple lower-level migration deadlines.

#### 4.4.3.7 Implementation and Evaluation

Similarly to the previous case study, a two-tiered system of agents is used. Agents are given a set of distinct locations which are guaranteed to be disjoint, which is necessary for the mitigation to work, as otherwise migrations would not break co-location.

Table 4.4 lists the migration times measured when migrating containers and **VM**s through each migration path (paths 1-7 in Figure 4.1) whilst executing various benchmarks from PARSEC, with the hosts being otherwise idle. Migrations between targets sharing a core or not sharing a core were evaluated separately. Given its consistent behaviour, and its rapid transfer of control, only post-copy migration was considered when moving **VM**s. The timings for **Con** migration were broken down into its phases. To keep **Con** migration independent from the cloud provider, container images were transferred to their target using `rsync`. This was by far the dominant factor in **Con** migration times, and can largely be eliminated through

| | | Con ↷ vC | | vC ↷ C | | Con ↷ OS | | | Con ↷ OS | | | VM ↷ OS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Mig. Path | 1 | 2 | 3 | 4 | 5 | | | 6 | | | 7 |
| | | | | | | rsync | Check | Rest | rsync | Check | Rest | |
| Benchmark | blackscholes | 24.14 | 24.07 | 26.84 | 26.93 | 32,508 | 13,695 | 2,027 | 31,235 | 13,636 | 1,876 | 18,781 |
| | bodytrack | 23.99 | 24.61 | 26.80 | 26.99 | 15,442 | 4,895 | 1,018 | 18,596 | 4,539 | 899 | 19,069 |
| | canneal | 25.03 | 25.20 | 27.00 | 27.29 | 68,972 | 24,562 | 7,950 | 55,831 | 21,936 | 6,399 | 18,748 |
| | dedup | 26.81 | 26.79 | 26.99 | 26.98 | 71,563 | 10,888 | 3,396 | 56,422 | 11,021 | 2,712 | 19,469 |
| | streamcluster | 24.70 | 24.79 | 26.79 | 26.96 | 19,215 | 5,048 | 842 | 13,016 | 5,104 | 797 | 18,654 |
| | raytrace | 24.30 | 24.85 | 26.92 | 26.96 | 66,881 | 18,668 | 4,804 | 53,223 | 17,057 | 4,255 | 18,841 |
| | x264 | 25.65 | 25.56 | 26.99 | 27.04 | 56,224 | 4,262 | 1,095 | 47,580 | 4,392 | 1,228 | 19,410 |
| | $H_0()$ (Geo.) | 24.93 | 25.11 | 26.90 | 27.02 | 40,510 | 9,542 | 2,197 | 34,678 | 9,233 | 1,986 | 18,994 |

Table 4.4: Migration times for different isolation types and paths (ms).

shared storage. The initial value of $H_0()$ for each path was derived from the geometric mean of the migration times.

Next, the relationship between performance and migration frequency was evaluated on the system when running at capacity. On the first machine, three **VM**s were assigned benchmarks to execute. A fourth was set as a migrating tenant, running each benchmark listed in Table 4.4. A fifth **VM** served as a destination for cross-**VM** process migration, and was kept idle. The second machine was configured with three tenants running benchmarks and two idle **VM**s.

Table 4.5 lists the geometric means of the benchmarks' running times, with the `all` column denoting the time required for all of the migrating tenant's benchmarks to complete. Figure 4.11 shows the predicted and actual migration times for the first migration operations, using the $H_0()$ values derived previously.

Network effects and thrashing on the NFS server introduced a significant degree of variability. In summary, it was found that migration operations generally had no discernible effect on the neighbouring tenants at the frequencies investigated, although it is likely that this would not hold for oversubscribed systems. Migrations at the **C** and **vC** level had no significant effect on performance. **Con** and **VM** migration did not appear to affect neighbouring tenants, but clearly affected their own execution. Migrating the **VM** every 30 seconds more than doubled its benchmark's running time (note that at this migration frequency, the **VM** was involved in a migration operation for two-thirds of its running time).

#### 4.4.3.8 Conclusion

This section has analysed the core components of a multi-level moving target defence, and examined the cost of migration at each level. Lower-level migrations can be performed at high frequency, but break the fewest co-locations, whereas the opposite holds at higher levels. Restricting the moving target defence to a single level limits its ability to break co-location.

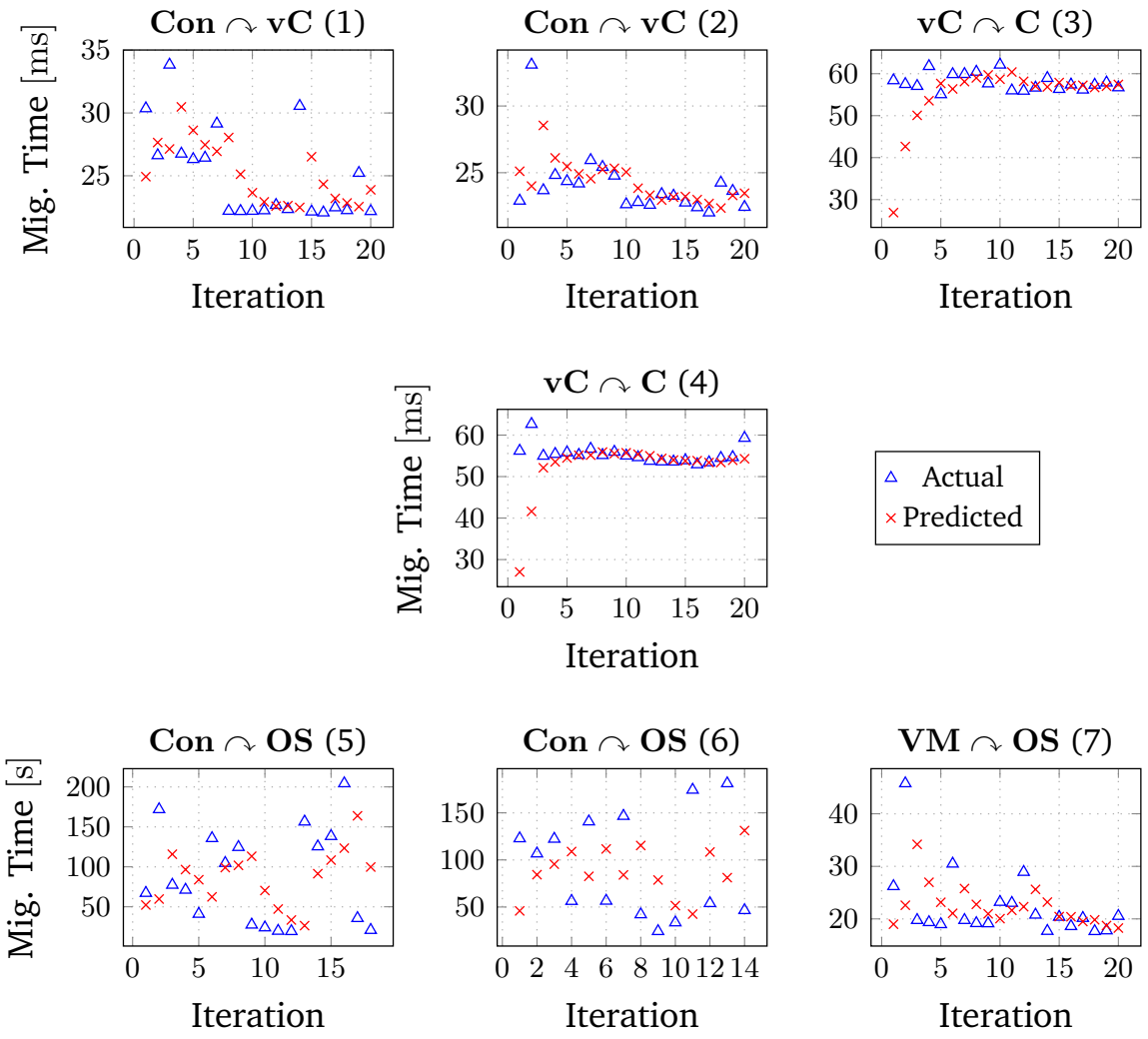



Figure 4.11: Predictions of $H()$ against measured migration times.

For example, while **VM** migration will break co-locations with other tenants, it cannot break the **OS**-level co-locations formed within it. Process and container migration can break co-location through every level, yet offline migration results in a significant downtime, rendering its application to a moving target defence limited. The advent of live process migration will thus help in making this mitigation pathway more viable.

### 4.4.4 Other Policies

The following are two examples of other policies and mechanisms that can be incorporated within the SAFEHAVEN framework.

**HomeAlone** *HomeAlone* [Zha+11] uses a PRIME-PROBE attack to monitor cache utilisation, and a trained classifier to recognize patterns indicative of shared locality. This can be used

| | | | Local | | | | Remote | | |
|---|---|---|---|---|---|---|---|---|---|
| | Dispatch (ms) | Migrations | all | blackscholes | canneal | streamcluster | blackscholes | canneal | streamcluster |
| No migration | - | 0 | 1,612 | 124 | 184 | 397 | 118 | 169 | 367 |
| **Con** $\curvearrowright$ **vC** (1) | 500 | 2,930 | 1,475 | 122 | 168 | 385 | 117 | 153 | 368 |
| | 400 | 3,601 | 1,463 | 120 | 168 | 385 | 117 | 154 | 369 |
| | 300 | 5,230 | 1,567 | 121 | 166 | 383 | 117 | 153 | 369 |
| | 200 | 7,889 | 1,590 | 122 | 170 | 384 | 118 | 153 | 369 |
| **Con** $\curvearrowright$ **vC** (2) | 500 | 3,110 | 1,560 | 124 | 169 | 391 | 118 | 153 | 369 |
| | 400 | 4,062 | 1,656 | 126 | 167 | 388 | 118 | 152 | 371 |
| | 300 | 5,034 | 1,521 | 127 | 171 | 388 | 117 | 154 | 367 |
| | 200 | 7,824 | 1,573 | 126 | 171 | 390 | 117 | 152 | 368 |
| **vC** $\curvearrowright$ **C** (3) | 500 | 3,117 | 1,562 | 123 | 172 | 404 | 118 | 159 | 373 |
| | 400 | 4,020 | 1,609 | 124 | 173 | 387 | 118 | 158 | 374 |
| | 300 | 5,379 | 1,614 | 124 | 174 | 388 | 118 | 160 | 372 |
| | 200 | 7,628 | 1,534 | 126 | 177 | 394 | 118 | 158 | 372 |
| **vC** $\curvearrowright$ **C** (4) | 500 | 3,154 | 1,576 | 125 | 171 | 395 | 118 | 157 | 372 |
| | 400 | 3,995 | 1,598 | 127 | 170 | 393 | 118 | 157 | 372 |
| | 300 | 5,413 | 1,630 | 128 | 173 | 394 | 119 | 159 | 372 |
| | 200 | 8,514 | 1,705 | 128 | 175 | 398 | 118 | 154 | 369 |
| **Con** $\curvearrowright$ **OS** (5) | 210,000 | 14 | 2,886 | 124 | 167 | 380 | 119 | 153 | 369 |
| | 180,000 | 18 | 3,565 | 124 | 165 | 380 | 118 | 152 | 369 |
| **Con** $\curvearrowright$ **OS** (6) | 210,000 | 14 | 2,780 | 122 | 164 | 375 | 119 | 155 | 373 |
| **VM** $\curvearrowright$ **OS** (7) | 120,000 | 17 | 2,028 | 120 | 179 | 392 | 121 | 176 | 375 |
| | 90,000 | 23 | 2,025 | 122 | 170 | 384 | 120 | 162 | 392 |
| | 60,000 | 39 | 2,282 | 121 | 162 | 389 | 122 | 173 | 390 |
| | 30,000 | 125 | 3,770 | 121 | 169 | 384 | 124 | 177 | 394 |

Table 4.5: Effect of migration frequency on performance when running at capacity.

to implement a hypervisor-independent version of the `isol()` predicate described in Section 4.4.2, or to detect adversarial behaviour.

**Network Isolation** Networks can harbour illicit channels [BT11; CBS04]. Isolation at this level can be achieved via a combination of soft and hard isolation, with trusted machines sharing network segments and traffic normalisers [Gor+12] monitoring communication at the edges.

## 4.5 Conclusion

This chapter has examined the use of migration, in its many forms, to dynamically reconfigure a system at runtime. This was achieved using the SAFEHAVEN framework, through which an efficient and timely mitigation against a system-wide covert-channel attack was implemented. The application of SAFEHAVEN was also considered in the context of a moving target defence.

Several points of consideration emerged over the course of the investigation. The first is that low reconfiguration costs are key to the viability of the approach, as they allow migrations to be performed at a high frequency without severely impacting the workloads being migrated. Cheap reconfiguration allows isolation to be procured temporarily and on-demand, further improving utilisation rates by minimising the duration for which resources

are reserved, which translates into lowered operating costs for tenants requesting isolation. Similarly, the *turnaround time* of a migration method, or the time between the reception of a request for isolation and its procurement, is critical for the soundness of the approach. This is because, particularly in the case of reactive mitigations, the turnaround time determines the length of the attack window following the detection of a potential attack.

The results in this chapter confirm an early hypothesis that directed the foray into the use of a hierarchy of confinement granularities, namely that the time taken to migrate a confinement grows with the size of the confinement. This, coupled with utilisation, acts as a further incentive to isolate processes using the smallest confinements possible.

The benchmarks obtained for process migration may initially appear anomalous, as one would expect process and container migration to be quicker than migrating their parent virtual machine. Further investigation reveals that the lacklustre performance emerges from the low throughput of transferring files, rather than the checkpoint and restore procedures themselves. This is purely an implementation issue, and the transition to a full-featured live migration mechanism for `criu/p.haul` is only a matter of time [Cri]. In addition, the ability to track page faults from user-space [Arc16] can also be used to implement post-copy migration for containers.

Finally, this chapter has dispelled the false intuition that virtual machines are too cumbersome to take part in an illicit-channel mitigation strategy that is based on migration. While this may be the case for pre-copy migration, with its potential for non-convergence and long turnaround times, the same cannot be said for post-copy migration. On the contrary, post-copy migration quickly transfers a virtual machine's execution to a target machine, followed by the remainder of its state. This allows machine-level co-locations to be broken quickly and within a consistent time window. Nevertheless, the stock implementation of post-copy migration within `QEMU` has a few limitations with respect to the aims of SAFEHAVEN, key of which is that an ongoing migration must be carried out fully before the virtual machine in question can be migrated to another host. This limitation, and one approach to circumventing it, forms the topic of the next chapter.

CHAPTER 5

# ABORTED POST-COPY MIGRATION

VIRTUAL MACHINES ARE CONVENIENT units of computational capacity, offering strong isolation guarantees and minimal barriers for adoption, yet their heft is a double-edged sword. The stock implementation of post-copy within QEMU does not allow ongoing migrations to be interrupted, limiting the frequency with which a virtual machine can be migrated, and hindering the ability to procure isolation on a short-term and temporary basis. This shortcoming is addressed through the implementation of two-way post-copy migration for QEMU, allowing partial and temporary migrations to and from a target machine.

## 5.1 Introduction

Post-copy live migration is an excellent tool in procuring isolation, as it almost instantaneously separates a confinement from its co-located entities. This is particularly relevant when migrating large confinements, notably virtual machines, where pre-copy migration would leave a tenant requesting isolation co-located with its potential attackers until the migration is completed, assuming that the migration ever converges.

While post-copy migration may solve the issue of urgency in handling isolation requests for virtual machines, it does not entirely eliminate the complications brought about by the virtual machines' size. Granted, post-copy guarantees that pages will only be sent at most once, yet one must still transfer the entire virtual machine once a migration is initiated.

A precise formulation of the problem is that post-copy migration, as implemented in QEMU, lacks semantics for *aborting* a migration. Cancelling a pre-copy migration is straightforward, as the virtual machine is still running in its entirety at the source machine throughout the migration process. Consequently, cancelling a pre-copy simply entails tearing down the migration stream and discarding the transferred state. The version of QEMU used throughout this work employed the same routine for cancelling a post-copy migration, which would naturally corrupt both the source and destination virtual machine processes, being that neither machine has a complete state.

The inability to abort ongoing post-copy migrations places two main limitations on the

use of virtual machine migration in procuring isolation. First, if the destination is found to be unsuitable during the course of a migration (for example, due to an unforeseen change in the machine's context traits), then one is committed to finishing the migration before the virtual machine can be evicted.

Secondly, and more crucially, there are a number of scenarios where one would like to migrate a virtual machine *temporarily*. For example, a tenant may only require isolation for the duration of the execution of a single security-sensitive task. Similarly, when implementing a moving target defence in a setting against attacks with very short setup times, one must be able to migrate at a high frequency. More generally, there are sources of asymmetry between machines in a cloud, including factors such as data locality and node capacity, that a tenant may only need to leverage temporarily. Without the ability to perform a partial migration, one would have to perform two full migrations in order to temporarily migrate a machine to and from a target. This has the effect of placing a upper-bound on the frequency with which migrations can be performed. For example, post-copy virtual machine migrations using the setup described in the previous chapter took around 20 seconds to complete, leading to a minimum 40 second round-trip time. This greatly discourages the use of migration at high frequencies, or for short-lived tasks.

This chapter examines the notion of *temporary* virtual machine migration, whereby a physical machine is dynamically leased to a virtual machine for a short period of time. This is achieved by modifying the post-copy virtual machine live migration mechanisms of `QEMU` to support *aborted post-copy* (or *two-way*) live migration (APC). While post-copy migration allows a virtual machine to immediately resume its execution at a target, two-way post-copy adds the ability to stop an ongoing migration at any point, migrating the remote state back to its origin without losing the virtual machine's progress. By only sending back the pages that were transferred during the outgoing migration, APC allows partial virtual machine migrations to be carried out at high frequencies.

The approach was evaluated by temporarily migrating virtual machines running intensive workloads generated using the PARSEC benchmark suite. In addition, the approach was used to take advantage of heterogeneity within a network of machines. In particular, APC was used to demonstrate how one can improve the performance and security of software implementations of AES operations by temporarily migrating to a machine with hardware AES-NI extensions.

### Chapter Outline

This chapter is structured as follows:

**Section 5.2** explains the theory behind APC, and describes the extensions carried out to `QEMU` to support it.

**Section 5.3** examines the performance of APC when migrating to machines with different context and active traits.

**Section 5.4** discusses several refinements that can be carried out on the method, and describes additional application scenarios as well as special considerations that must be made when deploying APC.

**Section 5.5** concludes this chapter.

## 5.2 Implementing Aborted Post-copy Migration

The task of dynamically migrating a virtual machine $\mathbb{V}$ hosted on a machine $\mathrm{M_{SRC}}$ to a destination $\mathrm{M_{DST}}$ requires a system that, at a minimum, implements

i) *a migration mechanism*, and

ii) *an interface* over which migrations can be triggered.

The standard `QEMU` migration mechanisms are *one-way*, with a machine's state flowing from a source to a destination machine. In addition, migrations are triggered at the source via the QMP protocol, the console-based monitor, or indirectly via libvirt.

Implementing *two-way* (or aborted) post-copy migration requires changes to the mechanism to allow for concurrent transmissions in opposite directions, and modifications to the triggering mechanism, as the triggers initiating and aborting a migration are sent to different host machines. The implementation of the mechanism and interface options for `QEMU` will be described in this section.

### 5.2.1 Dissecting Post-Copy Virtual Machine Live Migration

The following section describes the low-level details of a standard, one-way post-copy migration of a virtual machine $\mathbb{V}$ within `QEMU` to a machine $\mathrm{M_{DST}}$, which are key to understanding the modifications necessary to implement APC. While reference is made to specific internal structures and processes of `QEMU`, the general approach is not inherently restricted to this virtualisation platform.

#### 5.2.1.1 Preparing the Target

Before a migration begins, a **VM** instance $\bar{\mathbb{V}}$ with a specification identical to $\mathbb{V}$'s must first be created at $\mathrm{M_{DST}}$ [Qemb]. Unlike $\mathbb{V}$, $\bar{\mathbb{V}}$ is started in a *listening* state, leaving its virtual CPUs paused while waiting for an incoming migration stream. The task of creating an idle **VM** is extremely lightweight, as no state is transferred or allocated.

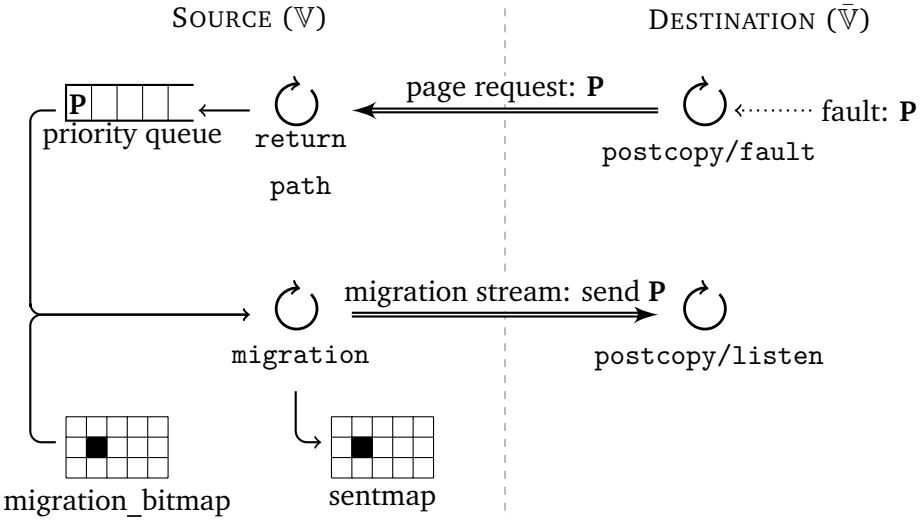



Figure 5.1: Serving a remote page fault during a post-copy migration.

The process of creating the idle **VM** at the target is performed automatically by a daemon process when migration is triggered through libvirt. Nevertheless, in order to reduce any confounding factors, $\mathrm{M_{DST}}$ was created directly throughout the evaluation by having the originating process invoke a script remotely.

#### 5.2.1.2 One-Way Post-Copy

When migrating $\mathbb{V}$ to $\mathrm{M_{DST}}$, $\mathbb{V}$ opens a direct connection to $\bar{\mathbb{V}}$ on a known port, establishing a *migration stream* [Kvma]. Pre-copy migration is fundamentally one-way, with a `migration` thread on $\mathbb{V}$ pushing the VM's memory contents onto the migration stream for reassembly by $\bar{\mathbb{V}}$. Post-copy extends the migration process by introducing a *back-channel* from $\bar{\mathbb{V}}$ to $\mathbb{V}$ over which page faults are announced to the source.

Figure 5.1 illustrates the data flow that takes place when the destination $\bar{\mathbb{V}}$ faults on a page **P** during a post-copy migration. The fault is read by the `postcopy/fault` thread on $\bar{\mathbb{V}}$ and forwarded to $\mathbb{V}$ over the back-channel, where it is consumed by the `return path` thread and placed onto the *priority queue*. The `migration` thread on $\mathbb{V}$ drives the migration, identifying memory pages to be sent and pushing them to $\bar{\mathbb{V}}$. The thread first services all page requests on the priority queue. It then scans for and transmits batches of yet-unsent dirty pages (the queue is serviced in favour of the scan, hence the term *priority* queue), the addresses of which are indicated by the `migration_bitmap` structure. Once a page is transmitted, its corresponding address is marked in the `sentmap` bitmap.

Automatically scanning for and sending as-yet unrequested pages carries a number of advantages in a standard one-way migration, namely it ensures that the migration stream remains saturated, and it generally reduces the number of remote page faults generated at

$M_{DST}$ by increasing the likelihood that a requested page has already been transferred. In addition, sending contiguous sequences of pages in bulk increases efficiency by reducing signalling and header processing.

The advantage of scanning and sending dirty pages in anticipation is less clear-cut for APC, as it has two main disadvantages. First, the priority queue is not pre-emptive, and adding a page will not interrupt an ongoing batch transfer, introducing latency (by default, batches are sent in 50ms iterations). Secondly, the scanning process is speculative, and will potentially transfer pages that will not be used by the remote task during the temporary migration. This makes anticipating transfers potentially wasteful, especially when the task has a small working set

The post-copy migration mechanism was modified to better facilitate the analysis of the impact of batch transfers. Specifically, on triggering a post-copy migration operation, one has the ability to specify an additional *mode* parameter. This determines whether an outgoing post-copy migration is to operate using *Demand Paging with Scanning* (DPwS) or *Pure Demand Paging* (PDP). In the latter case, the migration foregoes automatic `migration_bitmap` scanning and bypasses the priority queue, instead servicing page requests immediately as they arrive at $\mathbb{V}$. In addition, a migration can be started immediately in post-copy mode, avoiding the race condition encountered in Section 4.4.2.5.

### 5.2.2 Adding Cancel Semantics: Aborted Post-Copy Migration

Conceptually, a two-way migration involves stopping the original migration from $\mathbb{V}$ to $\bar{\mathbb{V}}$, and performing a migration in reverse. While one may be tempted to simply cancel the original migration and merge the modified $\bar{\mathbb{V}}$ back into $\mathbb{V}$, the stock `QEMU` cancel operation shuts down all the migration channels and unfreezes $\mathbb{V}$ while destroying $\bar{\mathbb{V}}$. This occurs when cancelling both pre- and post-copy migrations, in the latter case corrupting $\mathbb{V}$, as its state will be split across machines.

Instead, two-way migration was implemented using a post-copy live migration in both directions. This was implemented as a separate migration protocol, which was termed *bounce mode* (the analogy being that a **VM** is being bounced against a target physical machine, returning to its original host). Using live migration on the return leg, as opposed to a simple offline merging of pages, reduces downtime, particularly in the case of long bounce operations.

A bounce-mode migration is started immediately in post-copy mode. On receiving an abort command, the ongoing migration is stalled, rather than cancelled, at a point just prior to commencing cleanup. By stalling, $\bar{\mathbb{V}}$ pauses its execution, and $\mathbb{V}$ transfers its current `sentmap` to $\bar{\mathbb{V}}$. Once received, $\bar{\mathbb{V}}$ initiates a post-copy migration back to $\mathbb{V}$ using the acquired `sentmap` as its starting `migration_bitmap`, limiting transfers to the set of pages that were

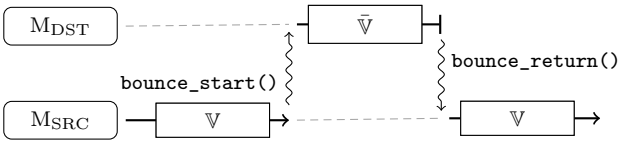



Figure 5.2: Temporarily migrating $\mathbb{V}$ from $M_{SRC}$ to $M_{DST}$.

sent by $\mathbb{V}$. On completion, the original migration is stopped and clean-up is performed. The migration from $\bar{\mathbb{V}}$ to $\mathbb{V}$ is always carried out in DPwS, as the objective is to return back to the origin as quickly as possible, and the set of pages to be returned is known.

While a `QEMU` migration would normally create a fresh **VM** process at its destination, bouncing back will migrate $\bar{\mathbb{V}}$ into the original virtual machine $\mathbb{V}$. This preserves the continuity of the $\mathbb{V}$ process on $M_{SRC}$, with its process identifier persisting through the bounce operation. $\mathbb{V}$ can be bounced or migrated freely following a completed bounce operation, whereas $\bar{\mathbb{V}}$ will be left in an internal QEMU `postmigrate` state, and cannot be reused as a migration target. Rather than modifying QEMU's internal state machine and potentially breaking compatibility with external tools and front-ends, a robust and pragmatic approach was adopted, whereby the $\bar{\mathbb{V}}$ process was simply restarted after each completed bounce operation.

The stock implementation of post-copy migration favours *batching*, both on the migration stream as well as the return path. In the latter case, requests generated by page faults are buffered and periodically flushed, adding a delay between a page fault and its placement onto the remote priority queue. As the approach being considered focuses on leasing hardware on a short-term basis, PDP was modified to eliminate buffering and transmit page faults immediately.

### 5.2.3 Controlling Migrations

The second aspect of the approach concerns the *triggering* of bounce operations. As illustrated by Figure 5.2, the state of a bounce operation is controlled by two signals, namely

i) a `bounce_start()` signal, received at $\mathbb{V}$, that initiates an outgoing bounce operation, and

ii) a `bounce_return()` signal, received at $\bar{\mathbb{V}}$, that bounces $\mathbb{V}$ back to its origin.

Bounce operations must be triggered by communicating with the `QEMU` virtualisation layer of the **VM** in question. Figure 5.3 illustrates the two principal pathways for triggering bounce operations. Bounce operations may either be triggered *internally* by a process within $\mathbb{V}$, or *externally* by a process outside $\mathbb{V}$. The former is used to service a tenant **VM**'s request for

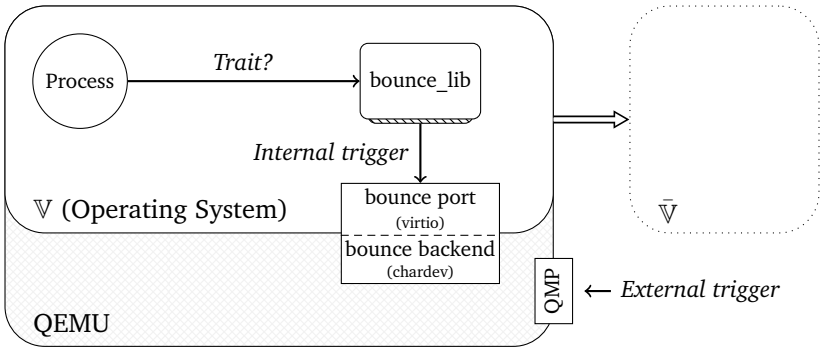



Figure 5.3: Trigger interfaces and flows when triggering a bounce operation.

a specific machine trait, whereas the latter would typically be used by a cloud provider to temporarily displace a **VM**.

`QEMU` does not employ paravirtualisation, and guests lack the ability of communicating directly with their hypervisor via hypercalls [Kvma]. Instead, a character device driver backend was developed, which acts as a device that is exposed to guests as a *virtio* serial port [Kvmb]. This allows guests to communicate with the virtualisation platform, and gives privileged processes within a **VM** the ability to dynamically trigger bounce operations by writing requests to a system file descriptor. These are then picked up and interpreted from within the `QEMU` layer. Bounce directives also accept a *trait* parameter on which the hypervisor can base its choice of migration target. Commands are interpreted asynchronously, as performing blocking or long-lived operations within the device backend can cause the entire virtual machine to block.

The external triggering of operations is enabled by modifying the *QEMU Machine Protocol* (QMP) [Qema] specification, which accepts `QEMU` commands as JSON requests via a defined interface (such as a UNIX socket). Concretely, the QMP specification was extended to interpret *bounce* directives. Care must be exercised when triggering bounce operations on $\mathbb{V}$ through QMP, as $\bar{\mathbb{V}}$ is effectively a separate machine, and existing QMP connections will not be automatically transferred to the target following a migration. This issue does not manifest itself in the case of internally-triggered migrations, as the backend automatically re-establishes a connection to the **VM**'s tenant-facing virtio serial port following a migration, and the frontend remains unaltered from the perspective of processes within the **VM**. Thus, if a process within $\mathbb{V}$ issues a `bounce_start()` to its bounce driver port, it will initiate a migration operation to $\bar{\mathbb{V}}$. The backend will be changed transparently, and a subsequent `bounce_return()` to the same port would be received by $\bar{\mathbb{V}}$'s hypervsior, bouncing the VM back.

## 5.3 Experiments

The performance of APC largely depends on the volume and access pattern of pages that must be transferred. One could easily contrive an ideal use case, bouncing a **VM** running an application with a minimal working set. While such applications exist, the strength of APC lies in its versatility and generality. To that end, the evaluation centred on the bouncing of heavy workloads, produced via the PARSEC [Bie+08] benchmark suite (Section 4.4.1.1).

The following section attempts to quantify the effects of APC while running intensive workloads under various scenarios. In particular, it analyses the effects of bouncing a VM to an idle machine (measuring the use case of obtaining *temporary isolation*, a context trait) as well as an occupied target (to quantify more generic applications such as a moving target defense). This is followed by an investigation into the use of APC to bounce a VM between machines with different active traits, performing encryptions by temporarily migrating to a machine with AES-NI [Gue10] extensions.

### 5.3.1 Baseline

As seen in Section 4.4.2.5, migrating $\text{VM}_\text{T}$ between two $\text{INTEL}_\text{T}$ machines using an unmodified DPwS migration takes $\approx 20$ seconds. Memory *ballooning*, whereby a VM's unallocated memory is freed back to the base system, can lower migration times, but varies depending on the workload. The following baseline can thus be established: in the absence of APC, a 2GB VM can be migrated temporarily using two full back-to-back post-copy migrations, with a round-trip taking at least 40 seconds, and at most 4GB of memory being transferred.

### 5.3.2 Test Parameters

Tests are carried out on a single *bouncing VM* $\mathbb{V}$, which is migrated temporarily from $\text{M}_\text{SRC}$ to $\text{M}_\text{DST}$. The tests described in this section contain various sources of variability, the foremost being the

i) workload of $\mathbb{V}$,

ii) number of additional VMs executing on $\text{M}_\text{SRC}$ and $\text{M}_\text{DST}$, and their workloads,

iii) *bounce period* $\tau$, which is the time between one outgoing bounce migration and the next, and

iv) *bounce duration* $\delta$, which is the time that $\mathbb{V}$ spends temporarily executing on $\text{M}_\text{DST}$ within a single bounce operation.

A VM's memory is partitioned into a number of *ramblocks*, the largest being the VM's RAM (524,288 pages) and video buffer (2,048 pages). In total, $\mathbb{V}$ has 527,025 4KiB pages that can

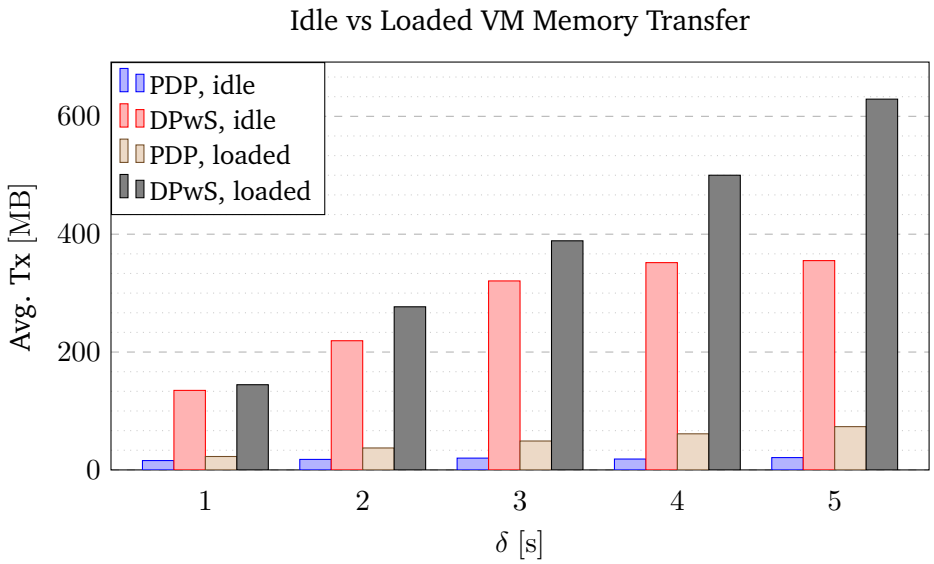



Figure 5.4: Average outgoing memory transfer sizes when bouncing idle and loaded VMs in PDP and DPwS mode.

be transferred using post-copy, requiring a bitmap of 65,880 bytes that must be exchanged on initiating a `bounce_return()` operation (Section 5.2.1).

The VM's RAM is by far the largest component of its state that will be demand paged. Some minor savings may be had by reducing the VM's video memory. Another possible optimisation is to forego the migration of the networking device. In addition to reducing the amount of memory to be transferred, this would avoid having to perform the device setup and network announcement routines at the destination. This may be useful when bouncing to perform very short-lived computations that do not make use of the network, but it would also render the method application dependent. Consequently, this avenue was not pursued in this work.

### 5.3.3 Characterising Workloads

The following experiments investigate the effects that different workloads have on the performance of migration.

#### 5.3.3.1 Activity and Inactivity

The first experiment seeks to establish a baseline between an active and inactive virtual machine with respect to the amount of memory that is transferred during an outgoing bounce operation.

The transfer size for the inactive case was averaged over 30 bounces of a running but otherwise idle VM. For the active case, `all` was executed once in the VM, which was bounced periodically until the benchmark completed. As will be seen, larger values of $\delta$ incur longer execution times. Consequently, the total number of bounce operations for a given $\delta$ varied between 77 and 175. Both experiments were carried out with $\tau = 15$s, and were repeated for values of $\delta \in [1, 5]$ seconds. The VM executed alone and was rebooted between experiments.

Figure 5.4 illustrates the result of running the experiment in PDP and DPwS mode using varying values of $\delta$. As can be seen, the idle VM maintains a small memory footprint, only consuming $\approx 300$MB of its allocated 2GB. This leads to a complete migration in just over $3$s when in DPwS mode, as evidenced by the plateau in transfer sizes. Similarly, the VM's very low internal activity generates a negligible number of page faults at the temporary destination in PDP, and little is transferred beyond an initial 20MB. In contrast, a loaded VM has a much larger memory footprint, and more state must be transferred. DPwS saturates the migration stream ($\approx$ 110MB/s, which approaches the network's nominal capacity) for the duration of the bounce operation, and the volume of memory grows in direct proportion to $\delta$. While the precise value of $\delta$ beyond which the transfer plateaus varies depending on the memory balloon size, growth in the transfer stage is effectively linear, and will take at most $\approx 20$s for a 2GB VM. Conversely, throughput in PDP mode is significantly lower, capping at $\approx 12$MB/s, or $11\%$ of the saturated channel capacity. This figure embodies the drawbacks of a pure demand-based migration strategy with no batching, whereby each page request has an associated round-trip time. Given that this is well below the channel's bandwidth, and that the throughput is constant for each tested value of $\delta$, we can deduce that this is a soft limit set by the machinery of PDP, and that the benchmark is requesting pages at a higher rate than that at which it can be served. This was further reinforced through the execution of additional benchmarks, as well as by directly analysing the CPU consumption rates on both machines during a bounce operation.

Note that the values cited in Figure 5.4 are for the outgoing migration, and the same volume of pages must be sent back on performing `bounce_return()`. In terms of memory transfers, PDP has the advantage of being more economical and only sending requested pages, although some waste may still occur, as will be discussed in Section 5.4.1.1.

#### 5.3.3.2 Locality and Page Spread

The comparatively poor throughput of PDP can be attributed to the large turnaround times involved between submitting a fault and receiving the page, coupled with the relatively small page size being used (4KB). Throughput can be improved through batching, provided that an application's memory access patterns are sufficiently local or regular. This section attempts to empirically quantify the spatial locality of each benchmark, which aids in gauging the

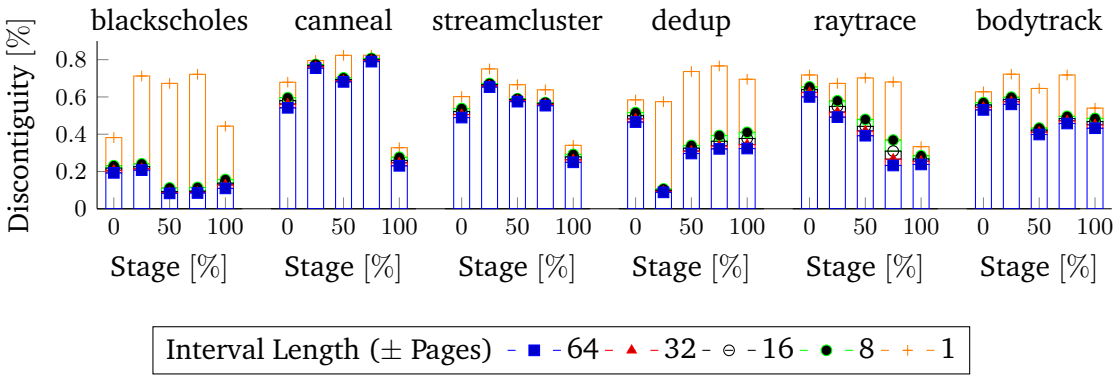



Figure 5.5: Percentage of pages in a PDP bounce migration that are not contiguous, for bounce migrations triggered at 5 stages of a benchmark's execution. $\tau = 10$s, $\delta = 5$s.

feasibility of batching and in interpreting subsequent evaluation results.

To characterise the spatial locality of a **VM**, a PDP bounce operation was triggered, and the sequence $\mathtt{S}$ of $n$ emitted pages was recorded, where $\mathtt{S} \equiv \{P_1, P_2 \dots P_n\}$, and $P_i$ is the address of the $i^{\text{th}}$ page in the sequence. The number of instances where an interval of pages does not contain pages with adjacent memory addresses were then counted. More formally, given a distance $\mathtt{I}$, a set of discontinuous pages $\mathtt{D} \equiv \{i \mid P_i, P_j \in \mathtt{S} \wedge \neg\exists P_j.\ |P_i - P_j| = 1 \wedge |i - j| \leq \mathtt{I}\}$ is extracted. Finally, the *discontinuity ratio* $|\mathtt{D}|/n$ was computed for the identified sets.

Figure 5.5 illustrates the result of calculating the discontinuity ratio for each benchmark using $\mathtt{I} \in \{1, 8, 16, 32, 64\}$. The ratio is calculated using page sequences generated during back-to-back bounce operations, with $\delta = 5$s and $\tau = 10$s. Ratios are shown for bounces at five stages of each benchmark's execution.

The initialisation and termination stages of a benchmark exhibit the highest spatial locality. Sequential spatial locality at the initial stages may be attributed to the benchmarks' unpacking phase, whereas the final stages may be the product of reduced activity. Of note is that each observed page sequence has at least 20% of its pages neighbouring others that are adjacent in memory. Increasing the interval length to $\pm 8$ significantly lowers the discontinuity ratio for localised workloads, yet the effect is less pronounced for benchmarks with large working sets. Raising interval lengths further did not appreciably reduce discontinuity. This would imply that discontinuities are primarily the result of page requests with sporadic addresses, rather than a switch to a different sequential region of memory. Thus, while transferring both pages adjacent to a requested page should improve usable throughput, moving to larger intervals may be counterproductive when migrating for very short durations.

| Benchmark | Running Time (s) | | | | | TX (MB) | | RX Throughput (MB/s) | |
|---|---|---|---|---|---|---|---|---|---|
| | Base | DPwS | | PDP | | DPwS | PDP | DPwS | PDP |
| blackscholes | 113 | 119 | (×1.05) | 160 | (×1.42) | 2003 | 548 | 939.2 | 923.2 |
| bodytrack | 95 | 100 | (×1.05) | 115 | (×1.21) | 1554 | 643 | 936.7 | 925.2 |
| canneal | 128 | 152 | (×1.19) | 229 | (×1.79) | 1536 | 1003 | 928.1 | 933.5 |
| dedup | 19 | 31 | (×1.63) | 79 | (×4.16) | 1987 | 823 | 922.1 | 939.1 |
| raytrace | 150 | 155 | (×1.03) | 239 | (×1.59) | 1551 | 943 | 937.7 | 934.0 |
| streamcluster | 340 | 399 | (×1.17) | 412 | (×1.21) | 483 | 149 | 864 | 920.6 |

Table 5.1: Remote execution of single benchmarks.

### 5.3.4 Migration to Idle Target

This section examines the migration scenario where the target physical machine is not hosting any virtual machines, that is, when $\mathrm{M_{DST}}$ is idle.

#### 5.3.4.1 One-Shot Bounce Migration

The following experiment attempts to quantify the overhead of executing remotely during a bounce migration. This is done by performing an outward bounce operation on a **VM** shortly after a benchmark is started, and only returning once the benchmark completes. This procedure was carried out for each benchmark in both bounce modes, and the execution times and memory transfer sizes were recorded.

Table 5.1 lists the running times for each benchmark. The *TX* column refers to the volume of data transferred during the outgoing leg of the operation. The *RX* column lists the throughput of the return operation for the associated outgoing mode. Since return operations are always performed in DPwS, identical throughput values are obtained for the return leg of both PDP and DPwS bounces.

The triggering point of `bounce_start()` affects the workload's total running time. An early trigger, particularly in DPwS mode, can take advantage of memory ballooning, leading to a VM being transferred before the test ramps up. Triggering later reduces the benefit of ballooning, yet it leads to a greater fraction of the workload having executed prior to bouncing. In the case of the benchmarks in Table 5.1, `bounce_start()` was triggered after executing for $\approx 30\%$ of the benchmarks' base time.

With the exception of `streamcluster`, each benchmark's memory footprint had ballooned to the point of consuming the majority of the **VM**'s allocation, which resulted in large transfers being performed in DPwS mode. In contrast, PDP migrations transferred an average of 25-50% of their DPwS equivalents. Since only the outgoing leg is being performed, and the workloads execute for a significant amount of time, DPwS carries a distinct advantage over

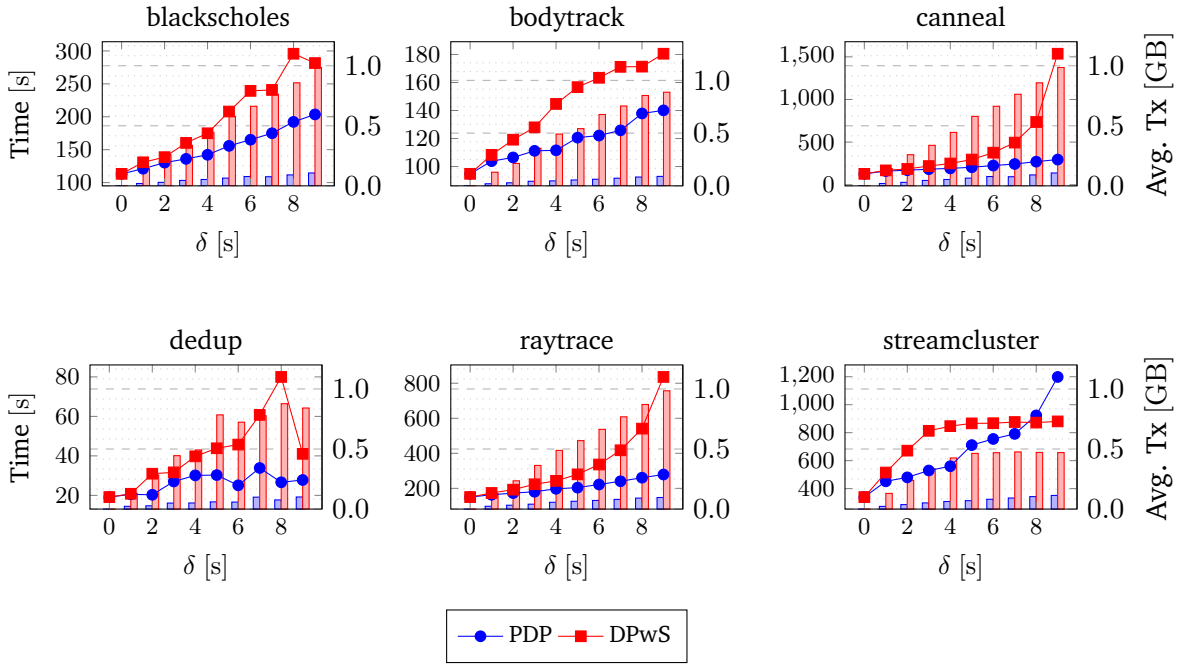



Figure 5.6: Iterated bounce migration to idle host for varying $\delta$, $\tau = 20$. Points denote the total execution time (left axis), whereas bars denote the average transfer size of outgoing bounce operations of the corresponding $\delta$ (right axis).

PDP in terms of running time. This is because once DPwS transfers the **VM**'s entire state, there will no longer be any involvement from the source machine, and no further performance penalties will be incurred. Conversely, PDP leaves the migration channel active for the duration of the bounce operation, and any remote page faults will cause a stall. This effect becomes increasingly apparent when bouncing larger workloads for longer durations.

Most workloads only experienced a minor slowdown in DPwS mode. The `dedup` benchmark was heavily impacted by bouncing, being a short-lived computation with a large working set. The `streamcluster` workload is particular in that it produced the least amount of network traffic, yet experienced the largest absolute slowdown (59s) out of the benchmarks bounced using DPwS.

#### 5.3.4.2 Iterated Bounce Migration

The following experiments evaluate the effect of performing back-to-back bounce operations on a **VM** when running each benchmark separately. As before, the host and destination base machines are otherwise idle, and the VM is relaunched between tests.

Figure 5.6 shows the total running times (shown as points) when bouncing with $\tau = 20$s using different values of $\delta$. The outgoing transmission size for a given $\delta$ (shown as bars)

is derived by taking the average transfer size of each bounce operation performed within that run. The value of $\delta$ was kept constant throughout each individual execution of each benchmark. The maximum tested $\delta$ (9s) was kept to just below half of the set $\tau$ (20s) to ensure that a **VM** had returned to its source before another bounce operation is initiated. Note that the VM is effectively in a state of constant migration when performing DPwS bounces with $\delta = 9$s. In contrast, a PDP bounce operation with the same $\delta$ will cause $\approx 100$MBs of pages to be transferred on the outgoing leg, which can be returned to the source machine in $\approx 1$s using a DPwS migration. This results in the **VM** being involved with a migration for roughly half of its execution time.

Several factors have to be taken into consideration when comparing DPwS and PDP migrations. As evidenced by the previous test, more work per unit time will be performed at the destination using DPwS rather than PDP when bouncing workloads with non-trivial working sets using moderate values of $\delta$. The exponential slowdown of `canneal` and `raytrace`'s performance against $\delta$ in DPwS mode may be attributed to their locality model. The performance of the `dedup` benchmark is somewhat erratic due to its relatively short execution time, where the choice of $\delta$ can change the execution time to a multiple of $\tau$, thus changing the total number of bounce operations performed in a run.

The performance impact of bouncing will always be bounded, as it will level off once the migration completes. This can be observed for relatively low values of $\delta$ when bouncing the `streamcluster` benchmark using DPwS. In this case, the working set is transferred within the first five seconds of `bounce_start()`, and the performance impact plateaus. An intersection with PDP's running times can be observed around the $\delta = 8$s mark, at which point the cost of stalling on pages exceeds the cost of transferring the machine's complete state via DPwS.

In summary, larger values of $\delta$ lead to slower execution times, up to the point where a VM's working set is migrated completely. This is primarily the result of two factors, namely

i) larger values of $\delta$ generally lead to larger outgoing memory transfers, which must be sent back, and

ii) the longer a task executes remotely, the greater the odds of stalling on a page, particularly in the case of PDP.

### 5.3.5 Migration with Co-Residency

APC is ultimately designed for deployment within a cloud infrastructure, where the source and destination machines may host multiple, loaded virtual machines. This section evaluates the behaviour of APC with co-resident VMs. This can guide the use of the method in other contexts, such as in implementing a moving-target defence, or in temporarily balancing loads.

| | Totals | | | $M_{SRC}$ | | | | |
|---|---|---|---|---|---|---|---|---|
| | Bounces | Tx (GB) | $\delta$ | `all` | blackscholes | canneal | streamcluster | dedup |
| | 0 | 0 | - | 948 | - | - | - | - |
| | 0 | 0 | - | 1299 | - | 149.82 | 384.14 | 22.69 |
| | 0 | 0 | - | 1337 | 121.93 | - | 366.21 | 22.51 |
| | 0 | 0 | - | 1375 | 120.88 | 143.42 | - | 23.00 |
| | 0 | 0 | - | 1099 | 120.52 | 146.45 | 381.97 | - |
| **PDP** | 151 | 8.04 | 5 | 2287 | - | 149.45 | 382.09 | 22.21 |
| | 139 | 7.57 | 5 | 2096 | 120.88 | - | 362.36 | 22.07 |
| | 152 | 8.21 | 5 | 2296 | 121.36 | 143.83 | - | 22.55 |
| | 138 | 9.91 | 5 | 2079 | 121.02 | 146.85 | 380.25 | - |
| **DPwS** | 262 | 119.93 | 5 | 4031 | - | 149.38 | 379.69 | 21.32 |
| | 250 | 112.42 | 5 | 3901 | 120.74 | - | 362.34 | 21.78 |
| | 258 | 116.53 | 5 | 4032 | 122.60 | 143.80 | - | 21.91 |
| | 217 | 123.49 | 5 | 3357 | 121.71 | 149.04 | 377.34 | - |

Table 5.2: Loaded source, idle destination, $\tau = 15$s. Times are given in seconds.

#### 5.3.5.1 Loaded Source, Idle Destination

The first test evaluated the case where $M_{SRC}$ is loaded and $M_{DST}$ is idle. For each test, three VMs were launched concurrently with a designated bouncing VM at the source, and each VMs was assigned a workload that it executes continuously in a loop. The co-located VMs did not actively participate in any bounce operations, and merely served as loads. Each co-located VM performed a warm-up round prior to the beginning of each test.

The test consisted of executing `all` within the bouncing VM and timing its overall execution time when performing periodic bounce migrations in both modes with $\delta = 5$s and $\tau = 15$s. The execution times of the co-located workloads were also recorded so as to examine the global effects of APC.

Table 5.2 presents the results of bouncing a **VM** in APC and DPwS with an idle $M_{DST}$ using $\delta = 5$s and $\tau = 15$s, as well as the base cases with no bounces. Each row represents a separate test. Each column represents a separate VM, and the average time taken to execute its associated benchmark. The `all` column lists the time taken to complete the bouncing workload. As the other workloads will have executed several times before `all` terminated, the geometric mean of the individual execution times was calculated. The *bounces* column lists the number of bounces performed during that test's execution, and *Tx* gives the total data sent during the outgoing legs of each bounce migration.

The first observation is that simply adding parallel workloads on the source machine

significantly impacts the performance of `all`, and, to a lesser extent, that of the individual workloads. This was found to be, at least in part, the product of several intermittent and intense write bursts to the NFS share, the effects of which are obscured when taking the mean of the individual workloads but accumulate within the `all` benchmark. For the observed runs, DPwS bounces caused the `all` benchmark to take almost twice as long to complete as when bouncing using PDP, and to transfer around 14 times as much data. The extended execution time also led to a greater number of bounce operations being performed, which further slowed down the benchmark. PDP bounces effectively doubled the base execution time of `all`. As discussed earlier, one must keep in mind that given the same $\delta$, a VM bouncing in DPwS mode will spend more time in flight than in PDP.

The effects of bouncing do not appear to spill over to the co-located workloads, which is crucial in a cloud setting. This may not hold true in the case of highly-consolidated machines, or if multiple tenants make heavy use of the network without quotas.

#### 5.3.5.2 Loaded Source, Loaded Destination

The previous test was repeated, with the exception that the destination was no longer kept idle. Four VMs were set to execute concurrently at $\mathrm{M_{DST}}$, with a benchmark assigned to each VM. The bouncing VM executed `all` in PDP mode with $\delta \in [1, 10]$ seconds (Table 5.3), and in DPwS mode with $\delta \in [1, 6]$ seconds (Table 5.4).

Introducing a second set of busy VMs onto the network exacerbated the bottleneck at the NFS server that was observed in the previous test. Comparing with the previous results using an idle destination, bouncing with $\delta = 5$s increased the execution time of `all` by an average of 38% for PDP, and 42% for DPwS. Tests where `dedup` did not execute were markedly faster than other tests, which indicates some degree of interference.

### 5.3.6 Leveraging Active Traits: AES-NI

An interesting quirk of `KVM` acceleration is that it effectively leads to a virtual machine having two `cpuid` registers, namely a virtualised register that is presented to the operating system, and the physical process-level register which is visible to the actual instruction stream [Kvma]. Consequently, instructions issued to a `KVM` vCPU will execute correctly, provided that they form part of the physical core's instruction set, even if `QEMU` reports that the **VM** lacks the associated CPU feature (this behaviour has been discussed in Section 2.4.2.2 in the context of blocking `clflush`). Consequently, apart from bouncing for isolation, a VM $\mathbb{V}$ running on $\mathrm{M_{SRC}}$ can be temporarily migrated to $\mathrm{M_{DST}}$ to leverage differences in the machines' *active traits* (Section 1.3.1), or instruction set extensions.

As a proof of concept, an investigation was carried out with $\mathrm{M_{SRC}}$ as an $\mathrm{AMD_T}$ machine

| | Totals | | | $M_{SRC}$ | | | | | $M_{DST}$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Bounces | Tx (GB) | $\delta$ | all | blackscholes | canneal | streamcluster | dedup | blackscholes | canneal | streamcluster | dedup |
| | 0 | 0 | - | 948 | - | - | - | - | - | - | - | - |
| | 0 | 0 | - | 1742 | - | 150.21 | 376.99 | 27.76 | 121.02 | 171.79 | 392.55 | 28.38 |
| | 0 | 0 | - | 2042 | 119.98 | - | 355.43 | 27.55 | 121.25 | 149.04 | 368.24 | 28.01 |
| | 0 | 0 | - | 1892 | 119.77 | 140.35 | - | 25.08 | 120.39 | 143.57 | 368.97 | 25.04 |
| | 0 | 0 | - | 1367 | 120.85 | 146.14 | 376.15 | - | 123.04 | 145.94 | 372.79 | 22.90 |
| **Pure Demand Paging** | 147 | 3.81 | 1 | 2196 | - | 146.37 | 376.31 | 24.60 | 120.05 | 144.59 | 371.14 | 23.85 |
| | 142 | 3.25 | 1 | 2125 | 119.77 | - | 357.34 | 24.89 | 119.59 | 142.28 | 370.46 | 24.28 |
| | 161 | 3.74 | 1 | 2406 | 118.80 | 138.77 | - | 22.89 | 118.71 | 141.68 | 370.21 | 23.15 |
| | 108 | 2.34 | 1 | 1609 | 121.39 | 146.43 | 377.21 | - | 121.74 | 146.23 | 373.06 | 22.60 |
| | 151 | 5.77 | 2 | 2257 | - | 146.84 | 377.32 | 24.35 | 119.61 | 144.35 | 369.86 | 25.61 |
| | 157 | 4.68 | 2 | 2363 | 118.93 | - | 355.80 | 23.34 | 119.80 | 143.11 | 370.24 | 23.81 |
| | 161 | 5.01 | 2 | 2425 | 120.37 | 138.70 | - | 23.96 | 119.31 | 143.08 | 369.59 | 24.63 |
| | 111 | 3.60 | 2 | 1668 | 121.10 | 145.75 | 378.50 | - | 120.31 | 148.19 | 373.83 | 22.82 |
| | 160 | 6.28 | 3 | 2417 | - | 146.23 | 376.18 | 24.50 | 120.35 | 143.88 | 370.36 | 25.06 |
| | 174 | 6.81 | 3 | 2626 | 119.85 | - | 356.33 | 24.48 | 120.21 | 143.56 | 371.72 | 25.22 |
| | 180 | 7.39 | 3 | 2718 | 119.69 | 138.31 | - | 24.98 | 119.81 | 142.38 | 371.20 | 25.14 |
| | 119 | 4.57 | 3 | 1793 | 121.01 | 146.05 | 376.27 | - | 121.53 | 145.80 | 373.97 | 23.08 |
| | 185 | 8.75 | 4 | 2795 | - | 146.66 | 376.79 | 24.09 | 120.90 | 144.64 | 370.51 | 25.03 |
| | 197 | 9.02 | 4 | 2971 | 119.39 | - | 355.77 | 24.13 | 120.42 | 143.13 | 371.00 | 23.80 |
| | 193 | 9.30 | 4 | 2916 | 120.49 | 138.43 | - | 24.47 | 120.74 | 143.67 | 370.35 | 23.98 |
| | 136 | 6.63 | 4 | 2044 | 121.80 | 145.98 | 378.76 | - | 122.00 | 147.84 | 374.91 | 23.02 |
| | 201 | 10.49 | 5 | 3027 | - | 146.44 | 376.97 | 23.70 | 121.36 | 144.10 | 372.03 | 24.64 |
| | 194 | 10.77 | 5 | 2914 | 119.49 | - | 355.20 | 23.64 | 120.42 | 144.80 | 371.80 | 23.42 |
| | 241 | 12.49 | 5 | 3635 | 120.16 | 138.33 | - | 24.63 | 121.17 | 144.34 | 372.55 | 25.32 |
| | 171 | 10.10 | 5 | 2577 | 121.01 | 144.65 | 378.10 | - | 121.30 | 147.04 | 376.91 | 23.28 |
| | 239 | 15.30 | 6 | 3597 | - | 147.09 | 375.70 | 23.83 | 121.61 | 145.66 | 374.13 | 25.98 |
| | 262 | 15.41 | 6 | 3952 | 119.16 | - | 354.72 | 23.26 | 121.08 | 144.70 | 373.89 | 24.55 |
| | 230 | 14.72 | 6 | 3467 | 120.62 | 140.14 | - | 25.02 | 121.80 | 173.25 | 395.54 | 24.87 |
| | 187 | 13.64 | 6 | 2812 | 121.64 | 147.43 | 376.58 | - | 121.70 | 149.02 | 376.46 | 23.28 |
| | 280 | 19.71 | 7 | 4217 | - | 146.43 | 376.07 | 23.41 | 122.04 | 146.69 | 375.03 | 25.19 |
| | 287 | 19.25 | 7 | 4328 | 119.82 | - | 356.69 | 24.31 | 120.94 | 144.45 | 375.57 | 24.45 |
| | 232 | 17.44 | 7 | 3495 | 120.48 | 140.46 | - | 24.71 | 121.93 | 173.04 | 397.69 | 25.16 |
| | 208 | 16.46 | 7 | 3132 | 121.39 | 146.29 | 377.69 | - | 121.78 | 152.85 | 376.65 | 22.98 |
| | 312 | 24.68 | 8 | 4703 | - | 146.58 | 374.94 | 23.03 | 122.09 | 147.48 | 375.01 | 24.40 |
| | 322 | 25.62 | 8 | 4871 | 120.54 | - | 357.69 | 24.11 | 121.56 | 146.19 | 374.10 | 24.15 |
| | 311 | 24.84 | 8 | 4692 | 120.48 | 140.21 | - | 23.89 | 121.89 | 144.82 | 374.71 | 23.85 |
| | 259 | 22.73 | 8 | 3908 | 121.19 | 145.60 | 376.69 | - | 122.46 | 149.46 | 377.87 | 23.13 |
| | 375 | 32.04 | 9 | 5665 | - | 147.96 | 376.52 | 23.04 | 121.85 | 147.82 | 375.81 | 24.48 |
| | 366 | 32.57 | 9 | 5535 | 120.48 | - | 356.26 | 24.48 | 122.36 | 173.76 | 397.53 | 24.55 |
| | 376 | 32.69 | 9 | 5678 | 120.64 | 139.30 | - | 23.74 | 121.94 | 145.90 | 375.79 | 24.89 |
| | 315 | 29.77 | 9 | 4754 | 121.29 | 146.57 | 376.80 | - | 122.51 | 169.05 | 377.84 | 23.27 |
| | 449 | 45.55 | 10 | 6772 | - | 147.82 | 375.73 | 23.97 | 121.79 | 147.44 | 378.17 | 24.73 |
| | 458 | 45.10 | 10 | 6918 | 120.51 | - | 356.31 | 23.98 | 122.27 | 147.08 | 376.42 | 24.38 |
| | 454 | 44.50 | 10 | 6866 | 120.53 | 138.48 | - | 24.31 | 122.74 | 147.48 | 376.09 | 25.26 |
| | 395 | 41.17 | 10 | 5962 | 121.78 | 146.02 | 375.84 | - | 123.72 | 149.29 | 379.62 | 22.92 |

Table 5.3: Loaded source and destination, PDP, $\tau = 15$s. Times are given in seconds.

| | Totals | | | $M_{SRC}$ | | | | | $M_{DST}$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Bounces | Tx (GB) | $\delta$ | all | blackscholes | canneal | streamcluster | dedup | blackscholes | canneal | streamcluster | dedup |
| | 0 | 0 | - | 948 | - | - | - | - | - | - | - | - |
| | 0 | 0 | - | 1742 | - | 150.21 | 376.99 | 27.76 | 121.02 | 171.79 | 392.55 | 28.38 |
| | 0 | 0 | - | 2042 | 119.98 | - | 355.43 | 27.55 | 121.25 | 149.04 | 368.24 | 28.01 |
| | 0 | 0 | - | 1892 | 119.77 | 140.35 | - | 25.08 | 120.39 | 143.57 | 368.97 | 25.04 |
| | 0 | 0 | - | 1367 | 120.85 | 146.14 | 376.15 | - | 123.04 | 145.94 | 372.79 | 22.90 |
| **Demand Paging with Scanning** | 168 | 25.17 | 1 | 2509 | - | 147.20 | 376.38 | 24.36 | 120.91 | 144.84 | 371.90 | 24.76 |
| | 208 | 30.66 | 1 | 3104 | 119.16 | - | 354.33 | 25.52 | 120.51 | 146.76 | 371.83 | 25.83 |
| | 178 | 26.11 | 1 | 2656 | 120.60 | 138.84 | - | 24.18 | 119.93 | 144.24 | 371.82 | 24.86 |
| | 116 | 15.28 | 1 | 1730 | 121.16 | 147.75 | 379.26 | - | 121.12 | 146.79 | 374.96 | 23.06 |
| | 205 | 41.38 | 2 | 3075 | - | 147.84 | 376.21 | 24.32 | 120.76 | 145.22 | 372.13 | 24.68 |
| | 198 | 44.17 | 2 | 2959 | 119.83 | - | 356.45 | 23.93 | 120.55 | 144.81 | 370.44 | 24.66 |
| | 206 | 45.92 | 2 | 3094 | 121.10 | 140.37 | - | 25.17 | 120.90 | 144.59 | 370.62 | 25.30 |
| | 132 | 29.70 | 2 | 1980 | 121.10 | 147.14 | 376.76 | - | 121.65 | 147.67 | 375.77 | 22.42 |
| | 227 | 67.91 | 3 | 3451 | - | 146.77 | 376.85 | 24.63 | 121.27 | 146.59 | 373.41 | 24.58 |
| | 237 | 73.61 | 3 | 3582 | 120.57 | - | 356.14 | 24.11 | 120.62 | 146.53 | 372.23 | 24.98 |
| | 241 | 75.10 | 3 | 3626 | 121.76 | 141.69 | - | 24.22 | 120.55 | 144.96 | 372.90 | 24.55 |
| | 182 | 61.14 | 3 | 2732 | 121.95 | 147.40 | 378.45 | - | 121.79 | 148.45 | 375.27 | 22.75 |
| | 283 | 114.63 | 4 | 4422 | - | 147.86 | 375.84 | 24.22 | 122.24 | 146.73 | 374.14 | 25.31 |
| | 295 | 123.30 | 4 | 4564 | 120.17 | - | 356.70 | 25.11 | 120.82 | 147.94 | 372.79 | 25.44 |
| | 293 | 118.38 | 4 | 4533 | 121.58 | 141.42 | - | 23.76 | 121.43 | 147.39 | 373.48 | 25.53 |
| | 227 | 100.16 | 4 | 3558 | 121.52 | 147.75 | 377.07 | - | 121.99 | 148.57 | 377.77 | 23.10 |
| | 333 | 159.39 | 5 | 5264 | - | 148.30 | 376.37 | 23.68 | 122.32 | 148.24 | 376.23 | 24.97 |
| | 371 | 192.83 | 5 | 5956 | 120.27 | - | 380.29 | 24.90 | 122.23 | 148.68 | 374.80 | 26.20 |
| | 370 | 180.86 | 5 | 5805 | 121.07 | 140.97 | - | 24.18 | 121.64 | 149.21 | 376.93 | 25.35 |
| | 303 | 157.34 | 5 | 4827 | 122.40 | 148.15 | 376.88 | - | 122.06 | 150.83 | 379.49 | 23.06 |
| | 459 | 248.06 | 6 | 7474 | - | 147.59 | 377.07 | 23.27 | 121.90 | 149.38 | 377.34 | 25.05 |
| | 460 | 252.13 | 6 | 7339 | 120.21 | - | 357.70 | 24.27 | 122.55 | 147.75 | 375.68 | 25.12 |
| | 463 | 255.21 | 6 | 7557 | 120.35 | 142.75 | - | 23.51 | 122.20 | 150.45 | 376.59 | 24.45 |
| | 338 | 197.79 | 6 | 5684 | 122.64 | 148.48 | 376.39 | - | 123.05 | 151.26 | 380.68 | 23.16 |

Table 5.4: Loaded source and destination, DPwS, $\tau = 15$s. Times are given in seconds.

and $\mathrm{M}_{\mathrm{DST}}$ being an $\textsc{Intel}_{\mathrm{T}}$ machine. The active traits under consideration were the AES-NI CPU extensions [Gue10], which $\mathrm{M}_{\mathrm{SRC}}$ lacks. AES-NI operations are faster and more resilient against side-channel attacks than their software-based equivalents [OST06]. As will be seen, this difference in speed can add up when performing many cryptographic operations (such as when implementing an MPC protocol [BSKR13]), to the point where it would be faster to bounce to $\mathrm{M}_{\mathrm{DST}}$ and make use of its AES-NI capabilities rather than to use a software-based implementation on $\mathrm{M}_{\mathrm{SRC}}$.

#### 5.3.6.1 Setup

The performance of AES was evaluated using the *Gladman* benchmarks within Intel's AES-NI reference implementation [Int16a]. These benchmarks measured the time taken to execute a series of iterations of encryptions, with each iteration encrypting 160Kb of data using a 128-bit key. The benchmark can be performed using either a pure assembly-based implementation of the AES operators (referred to as *ASM*), or one that is AES-NI enabled (referred to as *AES-NI*).

Simultaneous multi-threading was disabled for $\mathbb{V}$ so as to match the $\mathrm{AMD}_{\mathrm{T}}$'s CPU hierarchy. As with previous experiments, `KVM` acceleration was enabled. While `QEMU` virtualises the `cpuid` register, `KVM` cuts through `QEMU`'s capability enforcement layer [Kvma]. Consequently, a program can forego sanity checks on the vCPU's capability model and blindly issue AES-NI instructions. This was tested on the aforementioned machines, and it was found that the $\textsc{Intel}_{\mathrm{T}}$ architectures would interpret such commands correctly, whereas the $\mathrm{AMD}_{\mathrm{T}}$ would generate an *illegal instruction* exception. Migrating from the $\mathrm{AMD}_{\mathrm{T}}$ to an $\textsc{Intel}_{\mathrm{T}}$ machine prior to dispatching the instructions results in their correct execution.

#### 5.3.6.2 Experiments

The experiments measured the performance of AES encryption operations performed within $\mathbb{V}$ when

i) hosted on an Intel machine,

ii) hosted on the AMD machine, and

iii) started on the AMD and bouncing to an Intel in PDP mode for the duration of the workload's execution.

Evaluations were carried out for cases (i) and (iii) using both the ASM and AES-NI implementations, whereas only ASM could be used for (ii).

Figure 5.7 illustrates the time taken to execute the benchmark for the aforementioned scenarios while varying the number of encryption iterations. $\textsc{Intel}_{\mathrm{T}}$ is newer and faster than

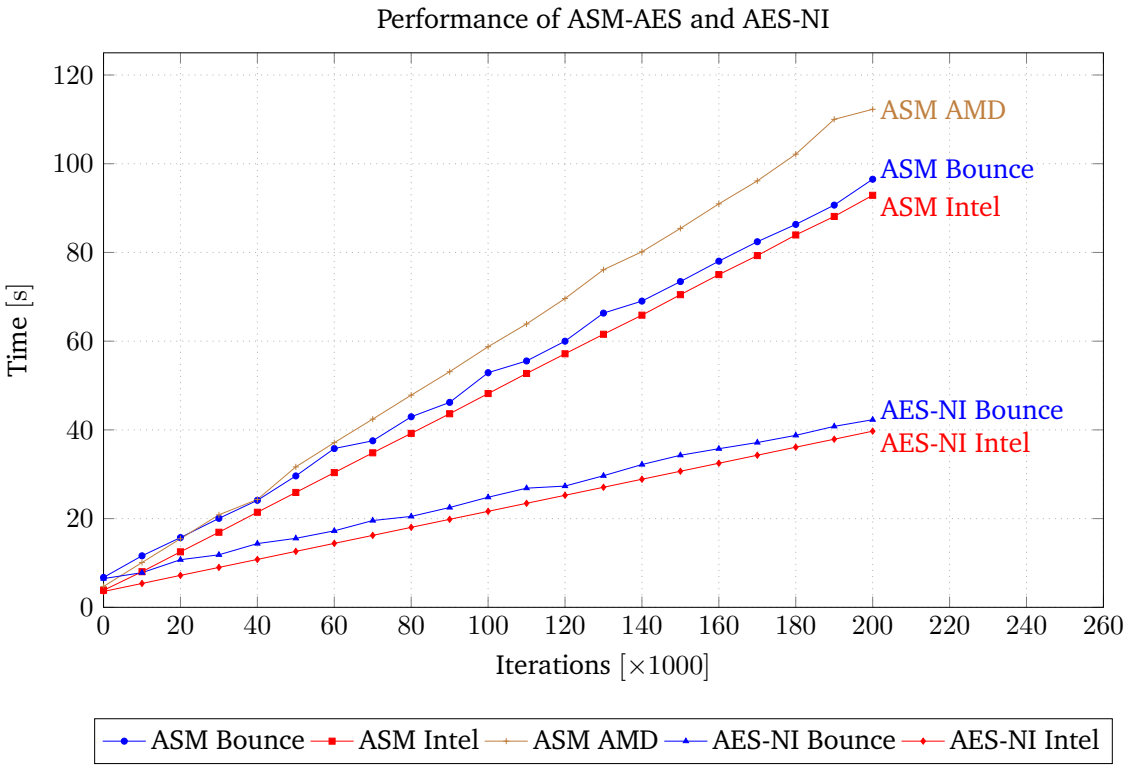



Figure 5.7: Running times using a pure assembler-based implementation (ASM) and one accelerated with AES-NI.

$AMD_T$, leading to a corresponding improvement in the ASM running times. Using AES-NI on the Intel machine halved the running time over ASM. As can be seen, the difference in computing power is such that it can prove faster to bounce from the AMD machine to an Intel machine and back than to execute directly on the AMD. Specifically, the ASM implementation runs faster using bounce mode when performing over 20,000 benchmark iterations, whereas using AES-NI will lead to a speed-up beyond 10,000 iterations.

#### 5.3.6.3 Conclusion

This section has demonstrated an application of APC that takes advantage of asymmetries in active traits, specifically AES-NI support. While this can increase both security and performance, it is anticipated that APC will be of even greater use when leasing active traits that have no adequate analogue at their machine of origin. For example, while a software implementation of AES may approximate the characteristics of AES-NI, this may not be the case for special hardware confinements such as SGX [BPH14].

Note that applying these mechanisms to a production environment would require additional safeguards to prevent a guest VM from attempting to run instructions on unsupported hardware. The "off-label" use of CPU instructions can pose a security risk, potentially introdu-

cing side-channels allowing machines to be fingerprinted, and can also complicate the saving and loading of CPU states.

## 5.4 Discussion

The following section describes the issues affecting the performance of APC, and ways of improving its performance. It also identifies possible additions and extensions to APC that allow its use in additional scenarios.

### 5.4.1 Factors Affecting Performance

The following section describes the principle sources of inefficiency, and possible avenues of improvement.

#### 5.4.1.1 Crosstalk

While one may only be interested in bouncing a **VM** for the benefit of a single process within it, all contained processes are ultimately participating in the migration. This is simultaneously one of the approach's greatest strengths and weaknesses, as migrating a whole VM avoids having to decompose and decouple the internal workings of the hosted application, yet it may also introduce *crosstalk*.

In this context, crosstalk refers to page faults generated by processes unrelated to the task that instigated the bounce operation. The coarse-grained nature of **VM**s may be inadequate for single environments that are executing multiple unrelated and intensive processes, and one may have to resort to finer-grained *container* [Cri] migration (Section 4.2.3). A container-based solution would serve to localise page faults, yet research in live container migration is still ongoing, and deployment in a cloud setting would require the creation of additional interfaces to coordinate deployment and confinement discovery.

#### 5.4.1.2 Locality and Memory Size

Spatial locality and memory requirements may vary widely between applications, which complicates batching transfers and pre-fetching pages during a migration [HDG09]. A direct way of increasing performance is to simply improve the hardware. For example, DPwS is effectively network-bound, and increasing bandwidth (such as by switching to a 10GbE network) will translate to faster migration times. This is not as straightforward in the case of PDP, as the turnaround time is primarily determined by latency, rather than bandwidth.

A more intrusive white-box approach can employ program analysis to provide hints as to how memory will be used. One can subsequently perform a hybrid migration, transferring

memory in anticipation of an imminent bounce operation. Analysis can also serve to identify empty or new pages that can be skipped during migration [HSS15].

### 5.4.2 Limitations

Other than potential performance and capacity issues, which may be mitigated through hardware and software, post-copy migration is inherently less reliable than pre-copy migration [Qemb]. During a post-copy migration, a virtual machine's state is split between the source and destination machines, with the latter hosting the most current updates to memory. Failure of the destination machine or the network places the virtual machine in an inconsistent state, as all the memory operations that occurred during migration will be lost. If the destination terminates gracefully, then one could potentially rescue the VM by aborting the post-copy operation (effectively bouncing back early), yet this would require strong hardware guarantees. In contrast, pre-copy migration always maintains a consistent version of the virtual machine on the source machine, and control is only transferred once the machine's memory contents are synchronised, greatly reducing the time window during which a migration can be corrupted. Should the destination fail during migration, one would simply keep executing at the source.

Post-copy operations can be made more robust by taking snapshots (or *checkpointing* [Cri; Qemb]) prior to a migration, which would act as a backup. Alternatively, the source machine can be kept frozen in memory, and a failing migration can be restarted. This solution would require a notion of transactions and roll-backs, as operations will generally have side-effects, and not all operations can simply be repeated.

### 5.4.3 Extensions

Temporary migration can potentially be applied to other contexts beyond the procurement of isolation, such as in counteracting short-lived load imbalances, or in bringing computations closer to their respective distributed data sources. The following is a brief overview of a number of ways in which aborted post-copy can be improved, as well as extended to support additional deployment scenarios.

#### 5.4.3.1 Migrate-On-Write

The current implementation of aborted post-copy transfers any pages sent during the outgoing operation back during the return leg, yet there is no need to return pages that were not modified during the bounce operation. By tracking page writes at the destination throughout a bounce operation and filtering out non-dirtied pages from the `migration_bitmap`, one can avoid bouncing back read-only pages.

#### 5.4.3.2 Event Sources

In the current implementation, bounce operations are explicitly triggered either from within the **VM** over the driver interface, or externally via QMP. The task of driving the migration, or specifying *event sources*, will ultimately depend on the cloud provider's specific use-case.

For example, a cloud provider may be able to stave off hardware obsolescence using temporary migrations. Instead of upgrading its machines with every new CPU generation, a cloud provider can mix a limited number of new machines into its existing network. Virtual machines are then given a virtual representation of the new architecture, and calls making use of the new features can be dynamically intercepted to transparently trigger a temporary migration to the appropriate hardware. This would require the interception of the instruction stream, either using traps, emulation [Qemb], or rewriting on loading (if the virtualisation platform supports it [SXZ13]). This can prove cost-effective as a transitional technology, or in the case of extensions that are infrequently used. Such asymmetries can be modelled as capabilities [Lib], and could be integrated into existing **VM** management infrastructures.

Alternatively, code can be automatically transformed to trigger migration on the basis of active traits, guarding code blocks via inlined commands to trigger migrations, or dynamically using introspection [FL12]. A heterogeneous cloud will produce variability at the level of passive and active traits that can be leveraged, provided that the security policies are sufficiently fine-grained. Automatically exploiting context traits is less straightforward, as these are generally the subject of more abstract security requirements (for example, one would have to specify the signature of a side-channel attack, or that a certain data source is spatially sensitive).

#### 5.4.3.3 Chained Post-copy

The current implementation of aborted post-copy always returns a bouncing VM back to its original source. This limits the method's ability to *chain* migrations, as a bounce migration's temporary state cannot simply be migrated to a third machine for further processing, rather it must be returned and bounced anew.

Beyond improving the performance of operation pipelining, support for chained post-copy can help redirect execution during hardware failure. In addition, it allows a more efficient implementation of a moving target defence by avoiding unnecessary return migrations.

To implement chained post-copy, the `migration_bitmap` must be augmented to also track the source of each page, allowing the correct machine to be polled once a page fault is generated. Alternatively, faulting **VM**s can be modified to multi-cast their page requests to each machine in the chain.

#### 5.4.3.4 Application to Moving Target Defence

Using the stock migration methods incorporated into SAFEHAVEN, a virtual machine $\mathrm{VM_T}$ could be migrated at maximum rate of around 3 times a minute (Section 4.4.3). This migration frequency may not be considered adequate in the case of very fast attacks.

Bounce migration enables an alternative approach to a moving target defence, where a machine is bounced from a source machine $\mathrm{M_{SRC}}$ to a destination $\mathrm{M_{DST}}$ for a $\delta$ defined by $\alpha()$. Note that the current implementation does not support chained partial migration, meaning that an aborted migration will always return to $\mathrm{M_{SRC}}$. Consequently, an implementation of the moving target defence may want to vary $\mathrm{M_{DST}}$ between bounce migrations, so as to defend against attackers that can exploit intermittent and periodic co-location. Similarly, if the virtual machine remains grounded on $\mathrm{M_{SRC}}$, it runs the risk of being compromised by a co-located process on $\mathrm{M_{SRC}}$ that performs an attack over a series of bounce operations. This can be prevented either by demanding additional security guarantees on $\mathrm{M_{SRC}}$ (such as only scheduling the same tenants' virtual machines), or by occasionally changing $\mathrm{M_{SRC}}$ through a full live migration.

With regards to the model, APC has the effect of altering $h$ and consequently $H()$. This is due to APC operations not having a typical or fixed length. The $H()$ predicate for an outgoing APC migration can be modelled as having zero cost, as control is transferred immediately to the destination. Conversely, the $H()$ value for the return leg varies based on the total amount of state transmitted to the remote machine. If the bounce operation is being carried out using DPwS, then one may assume that the connection is being saturated, and $H()$ can be approximated as a function of the total co-location time at the destination multiplied by the maximum transfer rate. A similar approach can be adopted when operating in APC mode, except that the average throughput value is less consistent, given that it depends on the internal activity of the virtual machine.

## 5.5 Conclusion

This chapter has explored the principle of temporary virtual machine migration, demonstrating how the semantics of cancelling a post-copy migration can be modified to initiate a migration in reverse, enabling short-lived computations to be performed on a remote machine without necessitating two full virtual machine migrations. This concept was actualised into a concrete implementation, enabling two-way post-copy within `QEMU`, and evaluated using workloads provided by the PARSEC benchmarking suite. Finally, several extensions and alternative scenarios to which the method can be applied were examined.

The performance of a temporary virtual machine migration operation was found to depend on several factors, primarily the duration for which the virtual machine is displaced,

and the mode in which the outgoing migration is performed. A workload undergoing a temporary migration using Demand Paging with Scanning is affected in much the same way as one that is migrated using a standard post-copy migration. The advantage of APC is that the performance impact must only be borne for the duration of the computation, as the migration can be abbreviated at will. In contrast, a full migration would always run to completion, needlessly impacting the virtual machine's performance beyond the task's execution, as well as wasting computational capacity.

Temporary migration using Pure Demand Paging aims to reduce the performance impact of migration by only transferring the minimum amount of memory that is requested by the remote task. This frugality comes at the expense of low throughput, and one should at least have an inkling of the workload's behaviour and expected execution time prior to migrating in this mode.

While SAFEHAVEN addresses spatial granularity through the use of fine-grained confinements, aborted post-copy introduces temporal granularity for large structures, allowing virtual machine migrations of varying lengths. Note that the methods described are not exclusive to `QEMU`, and can be implemented within alternative virtualisation platforms that allow page tracking, such as `Xen` [Bar+03b]. Similarly, there are no fundamental barriers to applying APC to process and container migration, especially since the page tracking extensions are available in user-space [Arc16]. This would result in the ability to specify even finer-grained isolation properties.

The primary appeal of the approach explored in this chapter is its ability to be applied to arbitrary workloads without requiring decomposition and with a minimum of preparation. The method can be deployed within a network with no additional infrastructural changes, as it does not introduce any special requirements beyond those of a standard post-copy migration. Nevertheless, the suitability of the approach is ultimately determined by the provider's requirements and intended deployment scenarios, and additional factors such as real-time or capacity constraints may have to be taken into consideration.

CHAPTER 6

# CONCLUSION

THIS WORK IS BUILT UPON THREE intertwined lines of research, namely the modelling of co-location using a fine-grained hierarchical model, the implementation of a migration framework and the evaluation of its migration mechanisms, and the extension of virtual machine post-copy migration to allow temporary migration. This chapter is a retrospective look at the original research questions, and the answers provided by this work.

## 6.1 Introduction

The following chapter is a review of the key concepts and results produced during the course of this work, where a fine-grained model of isolation was realised into a framework for dynamically controlling migration. It also provides an overview of similar work, as well as alternative approaches to illicit channel mitigation.

### Chapter Outline

This chapter is structured as follows:

**Section 6.2** compares various aspects of the approach with other related work.

**Section 6.3** revisits the stated aims of this work and describes how they were met.

**Section 6.4** identifies several lines of future research.

**Section 6.5** concludes this work.

## 6.2 Discussion

The following section provides a review of research that is similar and related to the work presented in this document. In particular, it describes alternative approaches to modelling systems and isolating entities. It also describes the security considerations that one must keep in mind when deploying an isolation-based mitigation.

### 6.2.1 Comparison to Other Formalisations

The following section discusses ambient and graph-based approaches to modelling systems.

#### 6.2.1.1 Ambient Models

A seminal work in modelling hierarchical architectures was the calculus of *mobile ambients* [CG98], 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 [BCF02], as well as to model resource allocation through a system of markers [Bar$^+$03a]. An additional extension is the *cloud calculus* [Jar$^+$12].

Ambients differ from the model explored in this work in a number of ways. For example, the latter uses a graph model, as confinements can be directly co-located within several different parent containments. This has the effect of requiring migration operations that simultaneously affect multiple containments. Another feature is the absence of the *open* operator, which in the ambient calculus is designed to destroy an ambient and release its constituents into the parent ambient. Instead, confinements are introduced into and removed from a hierarchy in a compound action staged through agents' idle queues.

#### 6.2.1.2 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 [BGM11; Bro$^+$10], and can then be checked statically [BG11] 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 [BVG14], where only changes, or deltas, are analysed. The creation and application of deltas is event-driven, triggered using hooks to a hypervisor. This line of research differs from that explored in this document in that its focus is on detection, rather than the online reconfiguration aspects of mitigation. In addition, analysis is confined to the virtual machine level, and does not focus on finer-grained isolation.

Other work has also looked into using graphical representations to assist in the modelling of systems, and automatically translating them into process calculi [PB09]. Challenges in dynamic monitoring include asynchronous updates, non-atomic actions, unordered events and blocking behaviour introduced by instrumentation [BGM13]. 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 [BKS14].

### 6.2.2 Scheduler-Based mitigations

Scheduling policies can be exploited to form illicit channels [VRS14] or steal computational resources [Var+12]. Setting a minimum time between deschedules can undermine certain classes of side-channels by obscuring residual cache effects [VRS14]. The problem with such an approach is that setting a long minimum quantum size and using a non-work conserving schedulers can result in consistently decreased utilisation rates.

Efficiently choosing migration targets is non-trivial, as placement can be constrained by several factors in addition to isolation requirements [Raj+09]. 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 [Aza+14; Kra+10]. 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 [CRC14]. Another approach uses leases and deadlines to reserve resources and prioritise migrations [ARC11].

Scheduling using partial virtual machine migration has also been explored in the context of consolidation [Bil+12], where virtual machines are migrated to a shared host during periods of inactivity to reduce power consumption. While similar in mechanism, this approach has objectives that are diametrically opposite to those of this work, as it favours long-lived migrations with low workloads.

### 6.2.3 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 [BKR09], abstract interpretation [Doy+13], and data tagging and tracking [Pri+14].

Program pre-partitioning and analysis can optimise the exchange of data and limit it to the state required by the remote operation. The problem with pre-partitioning when compared with a fully dynamic solution is that it is not as flexible, both in choosing its isolation target and in responding to changes in the system's configuration. One existing hybrid approach attempts to optimise virtual machine migration by identifying regions of memory that do not have to be transferred, with a proof of concept having been developed using a modified Java Virtual Machine [HSS15].

### 6.2.4 Security

A co-resident attacker can potentially exploit a side-channel (for example, by analysing network traffic or memory access patterns) to determine that a bounce operation is underway.

This could potentially be used to infer some aspect of the victim's internal state, either by associating bounce triggers with specific points in the victim's applications' execution, or by relating packet transmission frequencies during a bounce operation to the virtual machine's memory state. In addition, a cloud provider must carefully regulate bounce triggers to prevent load imbalances.

Note that a cloud provider should never allow a tenant to directly specify its own migration destination, as this would create a significant security risk, and could itself aid in forming illicit channels [Ris+09]. Adopting a declarative approach, whereby the cloud provider interprets a tenant's high-level isolation requests without exposing its internals, can help mitigate such risks. A cloud provider may also have to consider rate-limiting migrations or placing a minimum on the time that a virtual machine spends migrated. This can protect the cloud provider from denial-of-service attacks, as well as prevent resource-stealing attacks [Var+12; Zho+13].

## 6.3 Revisiting Claims

The following section revisits the original aims of this work, as stated in Section 1.3.3, and details the way in which they were addressed.

### 6.3.1 Claim 1

**The problem of illicit channels is fundamentally one of co-location, and can be modelled as such.**

Bar some unknown quantum phenomena, illicit channels appear to always require a medium, and do not exhibit what Einstein once termed "spooky action at a distance". This is reinforced by the instances of illicit channels observed throughout the course of this work, which to date follow causality as per classical computing. Even in the case of attacks that cross air gaps, such as when measuring electromagnetic emanations, one is operating through a medium, and physical co-location correlates with signal quality.

Modern architectures are hierarchical and vast, with different regions of their hierarchy offering varying granularities of isolation. Isolated resources can thus be provisioned at a finer granularity than dedicating machines to each tenant, which enables higher rates of utilisation. The advantage of using a layered and hierarchical model is that it can describe the cascading effects of migration, where movements at a higher granularity also affect its constituent confinements. Conversely, migrations at finer granularities have a limited effect on their parent confinements. In addition, it allows the creation of mitigations against attacks that cross hierarchies and levels.

### 6.3.2 Claim 2

**Containments can be modified efficiently through scheduling and migration at various granularities.**

In the model explored in this work, reconfiguration operations were split into two categories, namely *local* and *global migrations*. The cost of migrations, particularly global migrations, directly affects the viability of the approach explored in this work, as long migration times will lower the quality of service and will delay the fulfilment of an isolation request. When evaluating the impact on quality of service, one should keep in mind that non-migration based mitigations carry their own costs. In particular, terminating processes in response to detected attacks could severely degrade the quality of service when one considers that detectors may produce false positives. In contrast, migration is a lenient and largely transparent approach to restricting resource access, and detectors can afford to be more aggressive in their classifications.

Decomposing systems into hierarchies of containments allowed mitigations to use the cheapest migration necessary to isolate a confinement. On-demand isolation at the larger end of the hierarchy was shown to be viable when aided by a number of technological developments. In the case of virtual machine migrations, post-copy live migration serves to both guarantee convergence, and, crucially, make co-location breaks at the virtualisation level effectively immediate. The use of container and process migration allowed a second intermediate level of provisioning, with tenants having the option of consolidating containers amongst their own virtual machines. Finally, hardware counters and virtualised performance monitoring units allowed the development of cheap and accurate detectors that served to trigger migrations.

When studying the model and implementation, it transpired that process/container migration is, in principle, the most versatile dynamic isolation mechanism in the arsenal of reconfiguration methods. This is because it can be used to break co-location at any level of the hierarchy. From a practical standpoint, container migration is still under development, requires some forethought in deployment, and cannot always be performed at will. In contrast, virtual machines are robust and simple to deploy to, and do not require tenants to decompose their systems to facilitate migration. In addition, they offer strong system-level isolation guarantees between tenants, and are amenable to migration due to their use of virtualised devices and in-built decoupling assumptions.

### 6.3.3 Claim 3

**The impact of migrating large confinements such as virtual machines can be reduced using partial or temporary migrations.**

While post-copy live migration remedies the sluggishness of pre-copy virtual machine migration, it still involves the transfer of large volumes of state from one machine to another. Performing a complete migration would be particularly wasteful if one only needs isolation for the duration of a short security-sensitive computation.

This work has detailed the implementation of *temporary* virtual machine migration, where a physical machine is dynamically leased to a virtual machine for a short period of time. This was demonstrated by modifying the post-copy migration methods of `QEMU` to support two-way, or *aborted*, live migration. This adds the ability to stop an ongoing migration at any point and return the partial remote state back to its origin through a post-copy migration in the reverse direction. This avoids transferring the entire virtual machine's state, and allows partial migrations to be carried out at high frequencies.

Two-way post-copy migration allows machines to be leased to tenants at a high frequency on the basis of their traits. This was evaluated by temporarily migrating a tenant performing a series of encryption operations to a physical machine that supported the AES-NI CPU extensions, an active trait that the original machine lacked. Using two-way post-copy could promote alternative cloud topologies, where collections of machines are assigned one or more idle machines, which are then multiplexed temporarily amongst tenants on a demand basis. Mixing-in designated target machines at the rack level can help leverage high-speed interconnects and keep migrations local, which translates into faster migrations.

## 6.4 Future Work

Future work will focus on the automated synthesis of runtime enforcement monitors from properties expressed using the hierarchical model, and the integration of the model into simulation frameworks. Extensions to the model can also be considered, such as the addition of attributes to confinements to allow co-location to be queried on the basis of a specific attribute, and the use of weights to denote channel capacities at each level.

A natural progression for work on the implementation of the approach is to further develop and improve post-copy live migration at the process and container level, and to introduce two-way migration at this granularity of confinements. In addition, the incorporation of techniques to reduce the amount of state transferred during live migration, such as by detecting duplicate or empty memory pages, should be pursued.

## 6.5 Concluding Remarks

Partitioning computational resources into finer-grained units of computation challenges the notion that hard isolation is a necessarily wasteful and impractical response to illicit chan-

nels. Adopting a correspondingly fine-grained approach to migration further reduces the overheads of procuring hard isolation dynamically.

While future infrastructures may be faster and larger than contemporary deployments, it is unlikely that the fine-grained approach to isolation explored in this work will be invalidated solely through scale. Even if the cost of migration were somehow reduced to nil, one would still opt for a deployment which maximises isolation whilst minimising waste, necessitating fine-grained notions of partitioning. In addition, it has historically been the case that workloads grew in tandem with technological and architectural advancements, and economy in allocation will remain pertinent.

# Bibliography


[AA06] Keith Adams and Ole Agesen. '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*. (San Jose, California, USA). ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 2–13. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168860. URL: http://doi.acm.org/10.1145/1168857.1168860.

[Aga00] Johan Agat. 'Transforming out Timing Leaks'. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. (Boston, MA, USA). POPL '00. New York, NY, USA: ACM, 2000, pp. 40–53. ISBN: 1-58113-125-9. DOI: 10.1145/325694.325702. URL: http://doi.acm.org/10.1145/325694.325702.

[Ahm+15] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab. Hamid, Muhammad Shiraz, Abdullah Yousafzai and Feng Xia. 'A survey on virtual machine migration and server consolidation frameworks for cloud data centers'. In: *Journal of Network and Computer Applications* 52.0 (2015), pp. 11 –25. ISSN: 1084-8045. DOI: http://dx.doi.org/10.1016/j.jnca.2015.02.002. URL: http://www.sciencedirect.com/science/article/pii/S1084804515000284.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç and Jean-Pierre Seifert. 'On the Power of Simple Branch Prediction Analysis'. In: *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. (Singapore). ASIACCS '07. New York, NY, USA: ACM, 2007, pp. 312–320. ISBN: 1-59593-574-6. DOI: 10.1145/1229285.1266999. URL: http://doi.acm.org/10.1145/1229285.1266999.

[Ama15] Amazon. *Amazon EC2 Instances*. Apr. 2015. URL: https://aws.amazon.com/ec2/instance-types/.

[AR14] Andreas Abel and Jan Reineke. 'Reverse Engineering of Cache Replacement Policies in Intel Microprocessors and Their Evaluation'. In: *2014 IEEE Interna-*

*tional Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2014, pp. 141–142.

[ARC11] Zaina Afoulki and Jonathan Rouzaud-Cornabas. *A Security-Aware Scheduler for Virtual Machines on IaaS Clouds*. Tech. rep. LIFO, ENSI de Bourges, 2011.

[Arc16] Andrea Arcangeli. *userfaultd Linux kernel extensions*. May 2016. URL: https://git.kernel.org/cgit/linux/kernel/git/andrea/aa.git/?h=userfault.

[Ava] *Avahi*. Feb. 2016. URL: http://www.avahi.org/.

[Aza+14] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova and Bruce Shepard. 'Co-Location-Resistant Clouds'. In: *Proceedings of the 6th ACM Cloud Computing Security Workshop*. (Scottsdale, Arizona, USA). CCSW '14. New York, NY, USA: ACM, 2014, pp. 9–20. ISBN: 978-1-4503-3239-2. DOI: 10.1145/2664168.2664179. URL: http://doi.acm.org/10.1145/2664168.2664179.

[AZM10a] Aslan Askarov, Danfeng Zhang and Andrew C. Myers. 'Predictive Black-box Mitigation of Timing Channels'. In: *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security*. (Chicago, Illinois, USA). CCS '10. New York, NY, USA: ACM, 2010, pp. 297–307. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866341. URL: http://doi.acm.org/10.1145/1866307.1866341.

[AZM10b] Aslan Askarov, Danfeng Zhang and Andrew C. Myers. 'Predictive black-box mitigation of timing channels'. In: *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security*. (Chicago, Illinois, USA). CCS '10. New York, NY, USA: ACM, 2010, pp. 297–307. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866341. URL: http://doi.acm.org/10.1145/1866307.1866341.

[Bar+03a] Franco Barbanera, Michele Bugliesi, Mariangiola Dezani-Ciancaglini and Vladimiro Sassone. 'A Calculus of Bounded Capacities'. English. In: *Advances in Computing Science – ASIAN 2003*. Ed. by Vijay A. Saraswat. Vol. 2896. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 205–223. ISBN: 978-3-540-20632-3. DOI: 10.1007/978-3-540-40965-6_14. URL: http://dx.doi.org/10.1007/978-3-540-40965-6_14.

[Bar+03b] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield. 'Xen and the Art of Virtualization'. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. (Bolton Landing, NY, USA). SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462. URL: http://doi.acm.org/10.1145/945445.945462.

[BB03] David Brumley and Dan Boneh. 'Remote Timing Attacks Are Practical'. In: *Proceedings of the 12th USENIX Security Symposium*. (Washington, DC). USENIX Security '03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1251353.1251354.

[BCF02] Chiara Braghin, Agostino Cortesi and Riccardo Focardi. 'Security boundaries in mobile ambients'. In: *Computer Languages, Systems and Structures* 28.1 (2002). Computer Languages and Security, pp. 101 –127. ISSN: 1477-8424. DOI: http://dx.doi.org/10.1016/S0096-0551(02)00009-7. URL: http://www.sciencedirect.com/science/article/pii/S0096055102000097.

[Ber05] Daniel J. Bernstein. *Cache-timing attacks on AES*. Tech. rep. 2005.

[BG11] Sören Bleikertz and Thomas Groß. 'A Virtualization Assurance Language for Isolation and Deployment'. In: *POLICY*. IEEE Computer Society, 2011, pp. 33–40. ISBN: 978-1-4244-9879-6. URL: http://dblp.uni-trier.de/db/conf/policy/policy2011.html#BleikertzG11.

[BGM11] Sören Bleikertz, Thomas Groß and Sebastian Mödersheim. 'Automated Verification of Virtualized Infrastructures'. In: *Proceedings of the 3rd ACM Cloud Computing Security Workshop*. (Chicago, Illinois, USA). CCSW '11. New York, NY, USA: ACM, 2011, pp. 47–58. ISBN: 978-1-4503-1004-8. DOI: 10.1145/2046660.2046672. URL: http://doi.acm.org/10.1145/2046660.2046672.

[BGM13] Sören Bleikertz, Thomas Groß and Sebastian Mödersheim. *Modeling and Analysis of Dynamic Infrastructure Clouds*. Tech. rep. IBM Zurich, Dec. 2013.

[Bie+08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li. 'The PARSEC Benchmark Suite: Characterization and Architectural Implications'. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Oct. 2008.

[Bil+12] Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen and Mahadev Satyanarayanan. 'Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration'. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. (Bern, Switzerland). EuroSys '12. New York, NY, USA: ACM, 2012, pp. 211–224. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168858. URL: http://doi.acm.org/10.1145/2168836.2168858.

[BKR09] Michael Backes, Boris Kopf and Andrey Rybalchenko. 'Automatic Discovery and Quantification of Information Leaks'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. S&P '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 141–153. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.18. URL: http://dx.doi.org/10.1109/SP.2009.18.

[BKS14] KhalidZaman Bijon, Ram Krishnan and Ravi Sandhu. 'A Formal Model for Isolation Management in Cloud Infrastructure-as-a-Service'. English. In: *Network and System Security*. Ed. by ManHo Au, Barbara Carminati and C.-C.Jay Kuo. Vol. 8792. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 41–53. ISBN: 978-3-319-11697-6. DOI: 10.1007/978-3-319-11698-3_4. URL: http://dx.doi.org/10.1007/978-3-319-11698-3_4.

[BPH14] Andrew Baumann, Marcus Peinado and Galen Hunt. 'Shielding Applications from an Untrusted Cloud with Haven'. In: *11th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '14. Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[Bro+10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault and Raymond Namyst. 'hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications'. In: *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. PDP 2010. Pisa, Italy, Feb. 2010. DOI: 10.1109/PDP.2010.67. URL: https://hal.inria.fr/inria-00429889.

[BSKR13] Mihir Bellare, Viet Tung Hoang v Sriram Keelveedhi and Phillip Rogaway. 'Efficient Garbling from a Fixed-Key Blockcipher'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. S&P '13. Washington, DC, USA: IEEE Computer Society, May 2013, pp. 478–492. DOI: 10.1109/SP.2013.39. URL: http://dx.doi.org/10.1109/SP.2013.39.

[BT11] Billy Bob Brumley and Nicola Tuveri. 'Remote Timing Attacks Are Still Practical'. In: *ESORICS*. 2011, pp. 355–371.

[BVG14] Sören Bleikertz, Carsten Vogel and Thomas Groß. 'Cloud Radar: Near Real-time Detection of Security Failures in Dynamic Virtualized Infrastructures'. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. (New Orleans, Louisiana, USA). ACSAC '14. New York, NY, USA: ACM, 2014, pp. 26–35. ISBN: 978-1-4503-3005-3. DOI: 10.1145/2664243.2664274. URL: http://doi.acm.org/10.1145/2664243.2664274.

[Cao+16] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy and Lisa M. Marvel. 'Off-Path TCP Exploits: Global Rate Limit Considered Dangerous'. In: *Proceedings of the 25th USENIX Security Symposium*. (Austin, TX). USENIX Security '16. Berkeley, CA, USA: USENIX Association, Aug. 2016, pp. 209–

225. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/cao.

[CBS04] Serdar Cabuk, Carla E. Brodley and Clay Shields. 'IP covert timing channels: design and detection'. In: *Proceedings of the 11th ACM SIGSAC Conference on Computer and Communications Security*. (Washington DC, USA). CCS '04. New York, NY, USA: ACM, 2004, pp. 178–187. ISBN: 1-58113-961-6. DOI: 10.1145/1030083.1030108. URL: http://doi.acm.org/10.1145/1030083.1030108.

[CDRC14] Eddy Caron, Frédéric Desprez and Jonathan Rouzaud-Cornabas. 'Smart Resource Allocation to Improve Cloud Security'. English. In: *Security, Privacy and Trust in Cloud Systems*. Ed. by Surya Nepal and Mukaddim Pathan. Springer Berlin Heidelberg, 2014, pp. 103–143. ISBN: 978-3-642-38585-8. DOI: 10.1007/978-3-642-38586-5_4. URL: http://dx.doi.org/10.1007/978-3-642-38586-5_4.

[CG98] Luca Cardelli and Andrew D. Gordon. 'Mobile ambients'. In: *Proceedings of Principles of Programming Languages*. POPL '98. ACM Press, 1998.

[Cha+14] Swarup Chandra, Zhiqiang Lin, Ashish Kundu and Latifur Khan. 'Towards a Systematic Study of the Covert Channel Attacks in Smartphones'. In: *10th International Conference on Security and Privacy in Communication Networks*. SecureComm. Sept. 2014, pp. 427–435.

[Cha+15] Swarup Chandra, Zhiqiang Lin, Ashish Kundu and Latifur Khan. 'Towards a Systematic Study of the Covert Channel Attacks in Smartphones'. In: *International Conference on Security and Privacy in Communication Networks: 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*. Ed. by Jing Tian, Jiwu Jing and Mudhakar Srivatsa. Cham: Springer International Publishing, 2015, pp. 427–435. ISBN: 978-3-319-23829-6. DOI: 10.1007/978-3-319-23829-6_29. URL: http://dx.doi.org/10.1007/978-3-319-23829-6_29.

[Cho+09] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng and Xin Zheng. 'Building Secure Web Applications with Automatic Partitioning'. In: *Commun. ACM* 52.2 (Feb. 2009), pp. 79–87. ISSN: 0001-0782. DOI: 10.1145/1461928.1461949. URL: http://doi.acm.org/10.1145/1461928.1461949.

[Cop+09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere and Bjorn De Sutter. 'Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. S&P '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 45–60. ISBN: 978-0-

7695-3633-0. DOI: 10.1109/SP.2009.19. URL: http://dx.doi.org/10.1109/SP.2009.19.

[Cpu] *cpuset(7) - Linux manual page*. Linux. Aug. 2014. URL: http://man7.org/linux/man-pages/man7/cpuset.7.html.

[CQM14] Qi Alfred Chen, Zhiyun Qian and Z. Morley Mao. 'Peeking into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks'. In: *Proceedings of the 23rd USENIX Security Symposium*. (San Diego, CA). USENIX Security '14. Berkeley, CA, USA: USENIX Association, Aug. 2014, pp. 1037–1052. ISBN: 978-1-931971-15-7. URL: http://dl.acm.org/citation.cfm?id=2671225.2671291.

[CRC14] Eddy Caron and Jonathan Rouzaud-Cornabas. *Improving users' isolation in IaaS: Virtual Machine Placement with Security Constraints*. Research Report RR-8444. INRIA, Jan. 2014. URL: https://hal.inria.fr/hal-00924296.

[Cri] *CRIU project page*. Jan. 2016. URL: http://criu.org/Main_Page.

[DG+13] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh and Wenke Lee. 'Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection'. In: *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*. (Berlin, Germany). CCS '13. New York, NY, USA: ACM, 2013, pp. 839–850. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516697. URL: http://doi.acm.org/10.1145/2508859.2516697.

[Doy+13] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne and Jan Reineke. 'CacheAudit: A Tool for the Static Analysis of Cache Side Channels'. In: *Proceedings of the 22nd USENIX Security Symposium*. (Washington, D.C.). USENIX Security '13. Berkeley, CA, USA: USENIX Association, 2013, pp. 431–446. ISBN: 978-1-931971-03-4. URL: http://dl.acm.org/citation.cfm?id=2534766.2534804.

[DSZ10] Jiaqing Du, Nipun Sehrawat and Willy Zwaenepoel. 'Performance Profiling in a Virtualized Environment'. In: *2nd USENIX Workshop on Hot Topics in Cloud Computing*. (Boston, Massachusetts, USA). 2010.

[Erl] *Erlang Reference Manual User's Guide*. 6.2. Ericsson AB. Sept. 2014. URL: http://www.erlang.org/doc/reference_manual/users_guide.html.

[Erl07] Martin Abadi Úlfar Erlingsson. *Operating system protection against side-channel attacks that exploit memory latency*. Tech. rep. Aug. 2007, p. 7. URL: https://www.microsoft.com/en-us/research/publication/operating-system-

protection-against-side-channel-attacks-that-exploit-memory-latency/.

[FB15] Kevin Falzon and Eric Bodden. 'Dynamically Provisioning Isolation in Hierarchical Architectures'. In: *18th International Conference on Information Security (ISC 2015)*. Ed. by Javier Lopez and Chris J. Mitchell. Vol. 9290. Lecture Notes in Computer Science. Springer International Publishing, Sept. 2015, pp. 83–101. ISBN: 978-3-319-23317-8. DOI: 10.1007/978-3-319-23318-5_5.

[FB16a] Kevin Falzon and Eric Bodden. *There and Back Again: Temporary Virtual Machine Migration via Aborted Post-Copy*. Under submission. 2016.

[FB16b] Kevin Falzon and Eric Bodden. 'Towards a Comprehensive Model of Isolation for Mitigating Illicit Channels'. In: *5th International Conference on Principles of Security and Trust (POST 2016)*. Ed. by Frank Piessens and Luca Viganò. Vol. 9635. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, Apr. 2016, pp. 116–138. ISBN: 978-3-662-49635-0. DOI: 10.1007/978-3-662-49635-0_7.

[FL12] Yangchun Fu and Zhiqiang Lin. 'Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection'. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. May 2012, pp. 586–600. DOI: 10.1109/SP.2012.40.

[Gao+10] Xinbo Gao, Bing Xiao, Dacheng Tao and Xuelong Li. 'A survey of graph edit distance'. English. In: *Pattern Analysis and Applications* 13.1 (2010), pp. 113–129. ISSN: 1433-7541. DOI: 10.1007/s10044-008-0141-y. URL: http://dx.doi.org/10.1007/s10044-008-0141-y.

[Gli93] Virgil Gligor. *A Guide to Understanding Covert Channel Analysis of Trusted Systems (Light Pink Book)*. Rainbow Series. NAVAL COASTAL SYSTEMS CENTER. Nov. 1993.

[GO96] Oded Goldreich and Rafail Ostrovsky. 'Software protection and simulation on oblivious RAMs'. In: *J. ACM* 43.3 (May 1996), pp. 431–473. ISSN: 0004-5411. DOI: 10.1145/233551.233553. URL: http://doi.acm.org/10.1145/233551.233553.

[Gor+12] S.K. Gorantla, S. Kadloor, N. Kiyavash, T.P. Coleman, IS. Moskowitz and M.H. Kang. 'Characterizing the Efficacy of the NRL Network Pump in Mitigating Covert Timing Channels'. In: *Information Forensics and Security, IEEE Transactions on* 7.1 (Feb. 2012), pp. 64–75. ISSN: 1556-6013. DOI: 10.1109/TIFS.2011.2163398.

[Gue10] Shay Gueron. *Intel Advanced Encryption Standard (AES) New Instructions Set*. May 2010. URL: http://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[Gur+14] Mordechai Guri, Gabi Kedma, Assaf Kachlon and Yuval Elovici. 'AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies'. In: *9th International Conference on Malicious and Unwanted Software: The Americas MALWARE 2014, Fajardo, PR, USA, October 28-30, 2014*. 2014, pp. 58–67.

[Gur+15] Mordechai Guri, Assaf Kachlon, Ofer Hasson, Gabi Kedma, Yisroel Mirsky and Yuval Elovici. 'GSMem: Data Exfiltration from Air-Gapped Computers over GSM Frequencies'. In: *Proceedings of the 24th USENIX Security Symposium*. USENIX Security '15. Washington, D.C.: USENIX Association, Aug. 2015, pp. 849–864. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/guri.

[HDG09] Michael R. Hines, Umesh Deshpande and Kartik Gopalan. 'Post-copy Live Migration of Virtual Machines'. In: *SIGOPS Operating Systems Review* 43.3 (July 2009), pp. 14–26. ISSN: 0163-5980. DOI: 10.1145/1618525.1618528. URL: http://doi.acm.org/10.1145/1618525.1618528.

[Her+13] Amir Herzberg, Haya Shulman, Johanna Ullrich and Edgar Weippl. 'Cloudoscopy: Services Discovery and Topology Mapping'. In: *Proceedings of the 5th ACM Cloud Computing Security Workshop*. (Berlin, Germany). CCSW '13. New York, NY, USA: ACM, 2013, pp. 113–122. ISBN: 978-1-4503-2490-8. DOI: 10.1145/2517488.2517491. URL: http://doi.acm.org/10.1145/2517488.2517491.

[HPSP10] Danny Harnik, Benny Pinkas and Alexandra Shulman-Peleg. 'Side Channels in Cloud Services: Deduplication in Cloud Storage'. In: S&P '10 8.6 (2010), pp. 40–47. ISSN: 1540-7993. DOI: http://doi.ieeecomputersociety.org/10.1109/MSP.2010.187.

[HS96] Theodore G. Handel and Maxwell T. Sandford II. 'Hiding Data in the OSI Network Model'. In: *First International Workshop on Information Hiding*. London, UK, UK: Springer-Verlag, 1996, pp. 23–38. ISBN: 3-540-61996-8. URL: http://dl.acm.org/citation.cfm?id=647594.731523.

[HSS15] Kai-Yuan Hou, Kang G. Shin and Jan-Lung Sung. 'Application-assisted Live Migration of Virtual Machines with Java Applications'. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. New York, NY, USA: ACM, 2015, 15:1–15:15. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741950. URL: http://doi.acm.org/10.1145/2741948.2741950.

[Hu91] Wei-Ming Hu. 'Reducing timing channels with fuzzy time'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. S&P '91. IEEE Computer Society, May 1991, pp. 8–20. DOI: 10.1109/RISP.1991.130768.

[Hu92] Wei-Ming Hu. 'Lattice Scheduling and Covert Channels'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. S&P '92. Washington, DC, USA: IEEE Computer Society, 1992, p. 52. ISBN: 0-8186-2825-1. URL: http://dl.acm.org/citation.cfm?id=882488.884165.

[Int11] Intel. *System Programming Guide*. Vol. 3B. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel, May 2011.

[Int15a] Intel. *Instruction Set Reference*. Vol. 2. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel, Jan. 2015.

[Int15b] Intel. *Introduction to Intel Memory Protection Extensions*. July 2015. URL: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[Int16a] Intel. *Intel AES-NI Sample Library*. May 2016. URL: https://software.intel.com/sites/default/files/article/181731/intel-aesni-sample-library-v1.2.zip.

[Int16b] Intel. *System Programming Guide*. Vol. 3A. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel, June 2016.

[Ira+14] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth and Berk Sunar. 'Fine Grain Cross-VM Attacks on Xen and VMware'. In: *Proceedings of the 2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. BDCLOUD '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 737–744. ISBN: 978-1-4799-6719-3. DOI: 10.1109/BDCloud.2014.102. URL: http://dx.doi.org/10.1109/BDCloud.2014.102.

[Jar+12] Yosr Jarraya, Arash Eghtesadi, Mourad Debbabi, Ying Zhang and Makan Pourzandi. 'Cloud calculus: Security verification in elastic cloud computing platform'. In: *CTS*. Ed. by Waleed W. Smari and Geoffrey Charles Fox. IEEE, 2012, pp. 447–454. ISBN: 978-1-4673-1381-0. URL: http://dblp.uni-trier.de/db/conf/cts/cts2012.html#JarrayaEDZP12.

[Jo+13] Changyeon Jo, Erik Gustafsson, Jeongseok Son and Bernhard Egger. 'Efficient Live Migration of Virtual Machines Using Shared Storage'. In: *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. (Houston, Texas, USA). VEE '13. New York, NY, USA: ACM, 2013, pp. 41–50. ISBN: 978-1-4503-1266-0. DOI: 10.1145/2451512.2451524.

[JSS07] Trent Jaeger, Reiner Sailer and Yogesh Sreenivasan. 'Managing the risk of covert information flows in virtual machine systems'. In: *12th ACM symposium on Access control models and technologies*. (Sophia Antipolis, France). SACMAT '07. New York, NY, USA: ACM, 2007, pp. 81–90. ISBN: 978-1-59593-745-2. DOI: 10.1145/1266840.1266853. URL: http://doi.acm.org/10.1145/1266840.1266853.

[Kel+10] Eric Keller, Jakub Szefer, Jennifer Rexford and Ruby B. Lee. 'NoHype: virtualized cloud infrastructure without the virtualization'. In: *37th annual international symposium on Computer architecture*. (Saint-Malo, France). ISCA '10. New York, NY, USA: ACM, 2010, pp. 350–361. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1816010. URL: http://doi.acm.org/10.1145/1815961.1816010.

[Ken86] Christopher Angel Kent. 'Cache coherence in distributed systems'. PhD thesis. Purdue University, 1986.

[Kle16] Cristian Klein. *libvirt support for QEMU post-copy*. Aug. 2016. URL: https://git.cs.umu.se/cklein/libvirt.

[Klu+11] Tobias Klug, Michael Ott, Josef Weidendorfer and Carsten Trinitis. 'Transactions on high-performance embedded architectures and compilers III'. In: ed. by Per Stenström. Berlin, Heidelberg: Springer-Verlag, 2011. Chap. autopin: automated optimization of thread-to-core pinning on multicore systems, pp. 219–235. ISBN: 978-3-642-19447-4. URL: http://dl.acm.org/citation.cfm?id=1980776.1980792.

[KPMR12] Taesoo Kim, Marcus Peinado and Gloria Mainar-Ruiz. 'STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud'. In: *Proceedings of the 21st USENIX Security Symposium*. USENIX Security '12. Berkeley, CA, USA: USENIX Association, 2012, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=2362793.2362804.

[Kra+10] Valentin Kravtsov, Pavel Bar, David Carmeli, Assaf Schuster and Martin Swain. 'A scheduling framework for large-scale, parallel, and topology-aware applications'. In: *Journal of Parallel and Distributed Computing* 70.9 (2010), pp. 983–992. ISSN: 0743-7315.

[Kvma] *KVM project page*. Sept. 2015. URL: http://www.linux-kvm.org/.

[Kvmb] *Virtio-Serial API*. Sept. 2015. URL: http://www.linux-kvm.org/page/Virtio-serial_API.

[Lam73] Butler W. Lampson. 'A note on the confinement problem'. In: *Communications of the ACM* 16.10 (Oct. 1973), pp. 613–615. ISSN: 0001-0782. DOI: 10.1145/362375.362389. URL: http://doi.acm.org/10.1145/362375.362389.

[LGR13] Peng Li, Debin Gao and M.K. Reiter. 'Mitigating access-driven timing channels in clouds using StopWatch'. In: *43rd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. June 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575299.

[Lib] *Libvirt project page*. Jan. 2016. URL: http://libvirt.org/.

[Liu+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser and Ruby B. Lee. 'Last-Level Cache Side-Channel Attacks are Practical'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. (San Jose, CA, USA). S&P '15. Washington, DC, USA: IEEE Computer Society, May 2015, pp. 605–622. DOI: 10.1109/SP.2015.43. URL: http://dx.doi.org/10.1109/SP.2015.43.

[LMS14] Bartosz Lipinski, Wojciech Mazurczyk and Krzysztof Szczypiorski. 'Improving Hard Disk Contention-Based Covert Channel in Cloud Computing'. In: *Proceedings of the 2014 IEEE Security and Privacy Workshops*. SPW '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 100–107. ISBN: 978-1-4799-5103-1. DOI: 10.1109/SPW.2014.24. URL: http://dx.doi.org/10.1109/SPW.2014.24.

[Lxc] *Linux Containers*. Aug. 2016. URL: https://linuxcontainers.org/.

[Mar+12] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon and Srdjan Capkun. 'Analysis of the Communication Between Colluding Applications on Modern Smartphones'. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. (Orlando, Florida, USA). ACSAC '12. New York, NY, USA: ACM, 2012, pp. 51–60. ISBN: 978-1-4503-1312-4. DOI: 10.1145/2420950.2420958.

[Mdh+13] A. Mdhaffar, R. Ben Halima, M. Jmaiel and B. Freisleben. 'A Dynamic Complex Event Processing Architecture for Cloud Monitoring and Analysis'. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 2. CloudCom. Dec. 2013, pp. 270–275. DOI: 10.1109/CloudCom.2013.146.

[Mil+00] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler and Songnian Zhou. 'Process Migration'. In: *ACM Computing Surveys* 32.3 (Sept. 2000), pp. 241–299. ISSN: 0360-0300. DOI: 10.1145/367701.367728. URL: http://doi.acm.org/10.1145/367701.367728.

[MKS12] Keaton Mowery, Sriram Keelveedhi and Hovav Shacham. 'Are AES x86 Cache Timing Attacks Still Feasible?' In: *Proceedings of the 4th ACM Cloud Computing Security Workshop*. (Raleigh, North Carolina, USA). CCSW '12. New York,

NY, USA: ACM, 2012, pp. 19–24. ISBN: 978-1-4503-1665-1. DOI: 10.1145/2381913.2381917. URL: http://doi.acm.org/10.1145/2381913.2381917.

[Mol+06] David Molnar, Matt Piotrowski, David Schultz and David Wagner. 'The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks'. In: *Proceedings of the 8th International Conference on Information Security and Cryptology*. (Seoul, Korea). ICISC '05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 156–168. ISBN: 3-540-33354-1, 978-3-540-33354-8. DOI: 10.1007/11734727_14. URL: http://dx.doi.org/10.1007/11734727_14.

[MSR15] Soo-Jin Moon, Vyas Sekar and Michael K. Reiter. 'Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration'. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. (Denver, Colorado, USA). CCS '15. New York, NY, USA: ACM, 2015, pp. 1595–1606. DOI: 10.1145/2810103.2813706. URL: http://doi.acm.org/10.1145/2810103.2813706.

[Muc+99] Philip J. Mucci, Shirley Browne, Christine Deane and George Ho. 'PAPI: A Portable Interface to Hardware Performance Counters'. In: *Proceedings of the Department of Defense HPCMP Users Group Conference*. 1999, pp. 7–10.

[NS07] Michael Neve and Jean-Pierre Seifert. 'Advances on Access-Driven Cache Attacks on AES'. In: *Selected Areas in Cryptography: 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*. Ed. by Eli Biham and Amr M. Youssef. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 147–162. ISBN: 978-3-540-74462-7. DOI: 10.1007/978-3-540-74462-7_11.

[Nsa] *NSA ANT Catalog*. Aug. 2016. URL: https://www.eff.org/files/2014/01/06/20131230-appelbaum-nsa_ant_catalog.pdf.

[OO10] Keisuke Okamura and Yoshihiro Oyama. 'Load-based covert channels between Xen virtual machines'. In: *2010 ACM Symposium on Applied Computing*. (Sierre, Switzerland). SAC '10. New York, NY, USA: ACM, 2010, pp. 173–180. ISBN: 978-1-60558-639-7. DOI: 10.1145/1774088.1774125. URL: http://doi.acm.org/10.1145/1774088.1774125.

[Ope] *OpenStack Documentation*. OpenStack Foundation. Feb. 2015. URL: http://docs.openstack.org/.

[Ore+15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan and Angelos D. Keromytis. 'The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications'. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. (Denver, Colorado, USA). CCS '15. New York, NY,

USA: ACM, 2015, pp. 1406–1418. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813708. URL: http://doi.acm.org/10.1145/2810103.2813708.

[OST06] Dag Arne Osvik, Adi Shamir and Eran Tromer. 'Cache attacks and countermeasures: the case of AES'. In: *The Cryptographers' Track at the RSA Conference on Topics in Cryptology*. (San Jose, CA). CT-RSA '06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1–20. ISBN: 3-540-31033-9, 978-3-540-31033-4. DOI: 10.1007/11605805_1. URL: http://dx.doi.org/10.1007/11605805_1.

[Pag02] D. Page. *Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel*. Tech. rep. CSTR-02-003. Department of Computer Science, University of Bristol, June 2002. URL: http://www.cs.bris.ac.uk/Publications/Papers/1000625.pdf.

[Pat+14] Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin and Huseyin Ulusoy. 'Preventing Cryptographic Key Leakage in Cloud Virtual Machines'. In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security '14. San Diego, CA: USENIX Association, Aug. 2014, pp. 703–718. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pattuk.

[PB09] Frédéric Peschanski and Joël-Alexis Bialkiewicz. 'Modelling and Verifying Mobile Systems Using $\pi$-Graphs'. In: *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*. (Špindlerův Mlýn, Czech Republic). SOFSEM 2009: Theory and Practice of Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 2009, pp. 437–448. ISBN: 978-3-540-95891-8. DOI: 10.1007/978-3-540-95891-8_40. URL: http://dx.doi.org/10.1007/978-3-540-95891-8_40.

[Per05] Colin Percival. 'Cache missing for fun and profit'. In: *Proceedings of BSDCan*. May 2005.

[Pri+14] Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, David Eyers, Brian Shand, Ruediger Kapitza and Peter Pietzuch. 'CloudSafetyNet: Detecting Data Leakage between Cloud Tenants'. In: *Proceedings of the 6th ACM Cloud Computing Security Workshop*. (Scottsdale, Arizona, USA). CCSW '14. New York, NY, USA: ACM, Nov. 2014.

[Qema] *QEMU Macine Protocol*. Feb. 2016. URL: http://wiki.qemu.org/QMP.

[Qemb] *QEMU Post-Copy Live Migration*. Feb. 2016. URL: http://wiki.qemu.org/Features/PostCopyLiveMigration.

[Qia+15] Weizhong Qiang, Kang Zhang, Weiqi Dai and Hai Jin. 'Secure cryptographic functions via virtualization-based outsourced computing'. In: *Concurrency and Computation: Practice and Experience* (2015). cpe.3706. ISSN: 1532-0634. DOI: 10.1002/cpe.3706. URL: http://dx.doi.org/10.1002/cpe.3706.

[Raj+09] Himanshu Raj, Ripal Nathuji, Abhishek Singh and Paul England. 'Resource Management for Isolation Enhanced Cloud Services'. In: *Proceedings of the 1st ACM Cloud Computing Security Workshop*. (Chicago, Illinois, USA). CCSW '09. New York, NY, USA: ACM, 2009, pp. 77–84. ISBN: 978-1-60558-784-4. DOI: 10.1145/1655008.1655019. URL: http://doi.acm.org/10.1145/1655008.1655019.

[Ris+09] Thomas Ristenpart, Eran Tromer, Hovav Shacham and Stefan Savage. 'Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds'. In: *Proceedings of the 16th ACM SIGSAC Conference on Computer and Communications Security*. (Chicago, Illinois, USA). CCS '09. New York, NY, USA: ACM, 2009, pp. 199–212. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653687. URL: http://doi.acm.org/10.1145/1653662.1653687.

[Sai+05] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramón Cáceres, Ronald Perez, Stefan Berger, John Linwood and Griffin Leendert Doorn. 'Building a MAC-based security architecture for the Xen opensource hypervisor'. In: *21st Annual Computer Security Applications Conference (ACSAC'05*. 2005.

[Sch] *sched_setaffinity(2) - Linux manual page*. Linux. Aug. 2014. URL: http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.

[SGG05] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. 'Operating System Concepts'. In: 7th. Wiley Publishing, 2005. Chap. 5, p. 161. ISBN: 0470128720.

[Sha79] Adi Shamir. 'How to share a secret'. In: *Communications of the ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: http://doi.acm.org/10.1145/359168.359176.

[Smi+06] Matthew Smith, Thomas Friese, Michael Engel and Bernd Freisleben. 'Countering Security Threats in Service-oriented On-demand Grid Computing Using Sandboxing and Trusted Computing Techniques'. In: *Journal of Parallel Distributed Computing* 66.9 (Sept. 2006), pp. 1189–1204. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2006.04.009. URL: http://dx.doi.org/10.1016/j.jpdc.2006.04.009.

[SRM00] Michael S. Schlansker, B. Ramakrishna Rau and Multitemplate. *EPIC: An architecture for instruction-level parallel processors*. Tech. rep. Palo Alto: HP Laboratories, Feb. 2000. URL: www.hpl.hp.com/techreports/1999/HPL-1999-111.pdf.

[SXZ13] B. Saltaformaggio, D. Xu and X. Zhang. 'BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels'. In: *6th European Workshop on Systems Security*. (Prague, Czech Republic). EuroSec '13. ACM, 2013.

[Tir07] Kris Tiri. 'Side-channel Attack Pitfalls'. In: *Proceedings of the 44th Annual Design Automation Conference*. (San Diego, California). DAC '07. New York, NY, USA: ACM, 2007, pp. 15–20. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278485. URL: http://doi.acm.org/10.1145/1278480.1278485.

[Tyc14] Tycho. *Live Migration of Linux Containers*. Oct. 2014. URL: http://tycho.ws/blog/2014/09/container-migration.html.

[URv03] Theo Ungerer, Borut Robič and Jurij Šilc. 'A survey of processors with explicit multithreading'. In: *ACM Comput. Surv.* 35.1 (Mar. 2003), pp. 29–63. ISSN: 0360-0300. DOI: 10.1145/641865.641867. URL: http://doi.acm.org/10.1145/641865.641867.

[Var+12] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart and Michael M. Swift. 'Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense)'. In: *Proceedings of the 19th ACM SIGSAC Conference on Computer and Communications Security*. (Raleigh, North Carolina, USA). CCS '12. New York, NY, USA: ACM, 2012, pp. 281–292. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382228. URL: http://doi.acm.org/10.1145/2382196.2382228.

[VRS14] Venkatanathan Varadarajan, Thomas Ristenpart and Michael Swift. 'Scheduler-based Defenses against Cross-VM Side-channels'. In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security '14. San Diego, CA: USENIX Association, Aug. 2014, pp. 687–702. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan.

[WL06] Zhenghong Wang and Ruby B. Lee. 'Covert and Side Channels Due to Processor Architecture'. In: *22nd Annual Computer Security Applications Conference*. ACSAC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 473–482. ISBN: 0-7695-2716-7. DOI: 10.1109/ACSAC.2006.20. URL: http://dx.doi.org/10.1109/ACSAC.2006.20.

[Woo+07] Timothy Wood, Prashant Shenoy, Arun Venkataramani and Mazin Yousif. 'Black-box and Gray-box Strategies for Virtual Machine Migration'. In: *4th USENIX Conference on Networked Systems Design and Implementation*. (Cambridge, MA). NSDI '07. Berkeley, CA, USA: USENIX Association, 2007, pp. 17–17. URL: http://dl.acm.org/citation.cfm?id=1973430.1973447.

[WXW12] Zhenyu Wu, Zhang Xu and Haining Wang. 'Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud'. In: *Proceedings of the 21st USENIX Security Symposium*. (Bellevue, WA). USENIX Security '12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=2362793.2362802.

[Xu+11] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen and Richard Schlichting. 'An exploration of L2 cache covert channels in virtualized environments'. In: *Proceedings of the 3rd ACM Cloud Computing Security Workshop*. (Chicago, Illinois, USA). CCSW '11. New York, NY, USA: ACM, 2011, pp. 29–40. ISBN: 978-1-4503-1004-8. DOI: 10.1145/2046660.2046670. URL: http://doi.acm.org/10.1145/2046660.2046670.

[YF14] Yuval Yarom and Katrina Falkner. 'FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack'. In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security '14. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[Zha+11] Yinqian Zhang, Ari Juels, Alina Oprea and Michael K. Reiter. 'HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. S&P '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 313–328. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.31. URL: http://dx.doi.org/10.1109/SP.2011.31.

[Zha+12a] Yinqian Zhang, Ari Juels, Michael K. Reiter and Thomas Ristenpart. 'Cross-VM side channels and their use to extract private keys'. In: *Proceedings of the 19th ACM SIGSAC Conference on Computer and Communications Security*. (Raleigh, North Carolina, USA). CCS '12. New York, NY, USA: ACM, 2012, pp. 305–316. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382230. URL: http://doi.acm.org/10.1145/2382196.2382230.

[Zha+12b] Yulong Zhang, Min Li, Kun Bai, Meng Yu and Wanyu Zang. 'Incentive Compatible Moving Target Defense against VM-Colocation Attacks in Clouds'. In: *Information Security and Privacy Research*. Ed. by Dimitris Gritzalis, Steven Furnell and Marianthi Theoharidou. Vol. 376. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2012, pp. 388–399. ISBN: 978-3-642-30435-4. DOI: 10.1007/978-3-642-30436-1_32. URL: http://dx.doi.org/10.1007/978-3-642-30436-1_32.

[Zha+14] Yinqian Zhang, Ari Juels, Michael K. Reiter and Thomas Ristenpart. 'Cross-Tenant Side-Channel Attacks in PaaS Clouds'. In: *Proceedings of the 21st ACM*

*SIGSAC Conference on Computer and Communications Security*. (Scottsdale, Arizona, USA). CCS '14. New York, NY, USA: ACM, 2014, pp. 990–1003. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660356. URL: http://doi.acm.org/10.1145/2660267.2660356.

[Zhe+03] Lantian Zheng, Stephen Chong, Andrew C. Myers and Steve Zdancewic. 'Using Replication and Partitioning to Build Secure Distributed Systems'. In: *Proceedings of the IEEE Symposium on Security & Privacy*. S&P '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 236–. ISBN: 0-7695-1940-7. URL: http://dl.acm.org/citation.cfm?id=829515.830549.

[Zho+13] Fangfei Zhou, Manish Goel, Peter Desnoyers and Ravi Sundaram. 'Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing'. In: *J. Comput. Secur.* 21.4 (July 2013), pp. 533–559. ISSN: 0926-227X. URL: http://dl.acm.org/citation.cfm?id=2590624.2590626.

[ZR13] Yinqian Zhang and Michael K. Reiter. 'Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud'. In: *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*. (Berlin, Germany). CCS '13. New York, NY, USA: ACM, 2013, pp. 827–838. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516741. URL: http://doi.acm.org/10.1145/2508859.2516741.

[Orb16] Orbit Project. *QEMU post-copy extensions*. May 2016. URL: https://github.com/orbitfp7/qemu/tree/wp3-postcopy.

# Biography

The author was born on December 27$^{th}$ 1987 in Pietà, Malta. He read for his Bachelor of Science with honours in Information Technology in 2009 at the University of Malta, with his thesis titled “Thread Scheduling Within Paravirtualised Domains”. In 2011, he read for a master’s degree in Computer Science and Artificial Intelligence at the same institution, with his thesis titled “Combining Runtime Verification and Testing Techniques”.

In 2012, he commenced his studies with the Secure Software Engineering (SSE) group, a constituent of the European Centre for Security and Privacy by Design (EC SPRIDE) and TU Darmstadt, under the supervision of Professor Eric Bodden.