Dynamically Provisioning Isolation in Hierarchical Architectures[*]

Kevin Falzon and Eric Bodden

European Centre for Security and Privacy by Design (EC-SPRIDE)
`{kevin.falzon, eric.bodden}@ec-spride.de`

Abstract. Physical isolation provides tenants in a cloud with strong security guarantees, yet dedicating entire machines to tenants would go against cloud computing's tenet of consolidation. A fine-grained isolation model allowing tenants to request fractions of dedicated hardware can provide similar guarantees at a lower cost.
In this work, we investigate 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. We evaluate recent technological developments, including post-copy VM migration and OS containers, and show how they assist in improving reconfiguration times and utilisation. We incorporate these concepts into a unified framework, dubbed SafeHaven, and apply it to two case studies, showing its efficacy both in a reactive, as well as an anticipatory role. Specifically, we describe its use in detecting and foiling a *system-wide covert channel* in a matter of seconds, and in implementing a *multi-level moving target defence policy*.

Keywords: side channels · covert channels · migration · isolation

1 Introduction

The growing use of shared public computational infrastructures, most notably in the form of *cloud computing*, has raised concerns over side channel and covert channel attacks (collectively termed *illicit channels*). These are formed using unconventional and often discreet means that circumvent current security measures. This gives an attacker an edge over conventional attacks, which, while often effective, are well-characterised, conspicuous, and actively guarded against. To date, demonstrations of illicit channels have remained largely academic, with occasional influences on mainstream security practices. Nevertheless, the threat of such channels continues to grow as knowledge on the subject increases.

Hardware illicit channels are fundamentally the product of the unregulated sharing of locality, be it spatial or temporal. Side channels occur when a process inadvertently leaks its internal state, whereas covert channels are built by conspiring processes that actively leak state in an effort to transmit information.

[*] This work was supported by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG Collaborative Research Center CROSSING.

To break hardware locality, processes must be confined through what has been termed *soft* or *hard* isolation [38]. Hard isolation involves giving a process exclusive access to hardware, preventing illicit channels by removing their prerequisite of *co-location*. This approach is limited by the physical hardware available, yet it offers the strongest level of isolation. In contrast, soft isolation allows hardware to be shared but attempts to mask its characteristics.

Soft isolation often incurs an ongoing performance overhead, with some fraction of the machine's capacity committed to maintaining the isolation. Hard isolation does not typically incur a maintenance cost, but it can lead to underutilised hardware [26]. Nevertheless, underused capacity is not truly lost, and can potentially be used to perform functionally useful computations. Conversely, the maintenance costs of soft isolation consume resources.

The viability of hard isolation as a general mitigation technique depends on three factors, namely the availability of hardware, the degree of utilisation supported and the cost of reconfiguration. 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 cost of reconfiguration depends on the type of isolation being provisioned. 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

This work presents the following contributions:

- an investigation into the types of hard isolations present within modern hierarchical computer architectures, and the types of migration mechanisms available at each level, namely at the core, cache, and machine level, and their virtualised equivalents,
- the creation of a framework, dubbed SafeHaven, to orchestrate migration and distributed monitoring,
- an evaluation of the use of a series of maturing technologies, namely *post-copy live VM migration*, *OS-level containers* and *hardware counters*, and their application in improving a mitigation's agility and utilisation, and finally,
- an application of SafeHaven in mitigating a system-wide covert channel, in implementing a multi-level moving target defence, and in measuring the cost of migration at each level of the hierarchy.

2 Background and Related Work

The issue of isolating processes has been historically described as the *confinement problem* [25]. The following is an overview of the various ways in which confinements can be broken and upheld.

Attacks Confinements can be broken at different levels of a system architecture, such as the cache level (L1 [32], L2 [41] and L3 [42]), virtual machine level [33], system level [4,40], or network level [10], through various forms of attack. Attacks are characterised by type (side or covert), scope (socket, system or network-wide), bandwidth and feasibility. Illicit channels can be broadly categorised as being *time-driven*, *trace-driven* or *access-driven* [38]. Time-driven attacks rely on measuring variations in the aggregate execution time of operations. Trace-driven cache attacks are based on analysing an operation's evolution over time. Access-driven attacks allow an attacker to correlate effects of the underlying system's internal state to that of a co-located victim.

Covert channels are generally simpler to construct due to the involved parties cooperating. Fast channels have been shown at the L2 cache level [41], which in a virtualised environment would require VCPUs to share related cores, as well as across virtual machines [40]. Scheduling algorithms can also be leveraged to form a channel by modulating the time for which a VM [30] or process [20] is scheduled.

Defences Mitigations can broadly be categorised as being *passive*, *reactive* or *architectural*. Passive countermeasures attempt to preserve isolations through an indiscriminate process. For example, disabling hardware threads will eliminate a class of attacks [32] at the cost of performance. Alternatively, one can use a scheduling policy that only co-schedules entities belonging to the same process [24, 39] or *coalition* of virtual machines [34]. Policies can also be altered to limit their preemption rate, restricting the granularity of cache-level attacks [38]. Other countermeasures include periodically flushing caches [45], changing event release rates [6], and intercepting potentially dangerous operations [35].

Reactive countermeasures attempt to detect and mitigate attacks at runtime. Frameworks for distributed event monitoring [28] can be fed events generated via introspection [14], or can enforce a defined information flow policy [34].

Architectural mitigations are changes in hardware or to the way in which it is used. One example is Intel's introduction of specialised AES instructions, which insulate the operations' internal state from external caches [18]. Other solutions include randomly permuting memory placement [39], rewriting programs to remove timing variations [5, 13], reducing the precision of system clocks [19, 32] or normalising timings [26], cache colouring [24] and managing virtual machines entirely in hardware [23].

3 Isolation and Co-location

We briefly introduce the fundamental notions of co-location and migration using a simple graph model, with which the relationship between different forms of isolation can be represented.

3.1 Locality

A *confinement* delineates a boundary within which entities can potentially share state. Entities are themselves confinements, leading to a hierarchy.

Definition 1 (Locality). *A confinement (or locality) with a name N, a type $\boldsymbol{\Gamma}$, a set of capabilities* `C`*, and a set of sub-localities* $\mathrm{S}\textsc{b}$ *is denoted by* $\boldsymbol{\Gamma}{:}N(\mathtt{C})\,[\mathrm{S}\textsc{b}]$.

Capabilities regulate how confinements can modify each other, with operations on confinements only being allowed when they share a capability. We denote a locality X as being a sub-locality of D using $\mathrm{X} \in \mathrm{D}$. This is extended to the notion of transitive containment $\mathrm{X} \in^{+} \mathrm{D}$, where $\mathrm{X} \in^{+} \mathrm{D} \overset{\mathrm{def}}{=} \mathrm{X} \in \mathrm{D} \vee \exists \mathrm{X'} \in \mathrm{D}.\ \mathrm{X} \in^{+} \mathrm{X'}$.

Example 1 (Cache Hierarchy). Intel CPUs often implement *simultaneous multi-threading*, with two hardware threads (**C**) sharing an **L1** cache. A dual-core system with per-core **L2** caches and a common **L3** cache can be described as:

$$\mathbf{L3}{:}0()\,[\mathbf{L2}{:}0()\,[\mathbf{L1}{:}0()\,[\mathbf{C}{:}0()\,[]\,,\mathbf{C}{:}1()\,[]]]\,,\mathbf{L2}{:}1()\,[\mathbf{L1}{:}1()\,[\mathbf{C}{:}2()\,[]\,,\mathbf{C}{:}3()\,[]]]]$$

Definition 2 (Co-Location). *Two localities* X *and* Y *are co-located within* D *(denoted by* $\mathrm{X} \overset{\mathrm{D}}{\leftrightarrow} \mathrm{Y}$*) if* $\mathrm{X} \in \mathrm{D} \wedge \mathrm{Y} \in \mathrm{D}$*. The localities are transitively co-located in* D *(denoted by* $\mathrm{X} \overset{\mathrm{D}}{\Leftrightarrow} \mathrm{Y}$*) if* $\mathrm{X} \in^{+} \mathrm{D} \wedge \mathrm{Y} \in^{+} \mathrm{D}$.

We denote the movement of a locality X to a parent confinement D as $\mathrm{X} \curvearrowright \mathrm{D}$.

Example 2 (Cache Co-Location). For the hierarchy defined in Ex. 1, given that a process $\mathrm{P_i}$ executes on a hardware thread **C**:i, process $\mathrm{P_0}$ is transitively co-located with (i) $\mathrm{P_1}$ via **L1**:0, **L2**:0 and **L3**:0, and (ii) $\mathrm{P_2}$ via **L3**:0.

3.2 Confinements

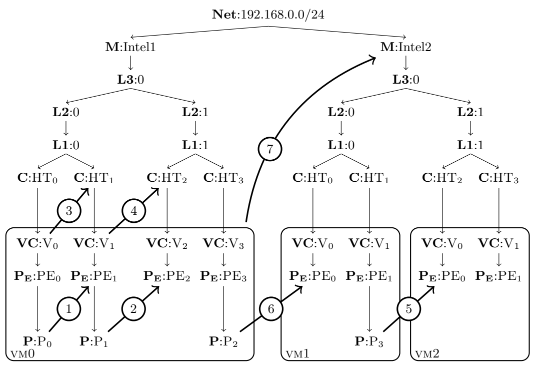
Fig. 1a lists the primary types of isolations with which this work is concerned, which are broadly categorised as being static or dynamic. The former are architectural elements such as caches and networks, which, while offering some degree of configuration, exist at fixed locations in relation to each other. The latter are isolations that can be created, destroyed or otherwise moved around. Fig. 1b is an example of a containment graph, with possible migration paths depicted through arrows 1-7, where paths denote how an isolation's parent can be changed. The mechanisms implementing each path will be detailed in Section 4.2.

An additional form of confinement is that produced by soft isolation [38], which attempts to decrease the amount of information that can be inferred from shared state, simulating a plurality of disjoint isolations. This often incurs an ongoing overhead, the severity of which varies depending on the technique being used [38]. For example, the `clflush` instruction, which flushes all cached versions of a given cache line, has been shown as an effective enabler of side-channel attacks [42, 44]. Disabling the instruction would impede attacks. While `clflush` is an unprivileged instruction that does not generate a hardware trap [44], closer inspection of its semantics shows that its execution depends upon a `clflush` flag within the machine's `cpuid` register being asserted [22]. This register is generally immutable, yet virtualisation can mask it [3]. Unfortunately, hardware-assisted virtualisation, such as that used by `KVM`, bypasses the virtualised `cpuid` register, limiting one to using an emulated VCPU such as `QEMU`. While we found this to be effective in disabling `clflush` (an invalid opcode exception was thrown on its invocation), a `QEMU` VCPU is substantially slower than its `KVM` equivalent, leading to a continuous overhead.

Static Confinements		
Type	Description	Can Contain
Net	Network	**Net**, **M**
M	Machine	**L3**, **OS**
L3	L3 Cache	**L2**
L2	L2 Cache	**L1**
L1	L1 Cache	**C**
C	Physical core	**VC**, $\mathbf{P_E}$, **Con**, **VM**
OS	Operating Sys.	$\mathbf{P_E}$, **Con**, **VM**

Dynamic Confinements		
Type	Description	Can Contain
VC	Virt. CPU	**VC**, $\mathbf{P_E}$, **Con**, **VM**
VM	Virt. machine	**VC**, **OS**
$\mathbf{P_E}$	Control group	**Con**, **P**
Con	Container	**P**
P	Process	-

(a) Confinement types



(b) Graph of $2 \times$ **M**, $3 \times$ **VM**. Edges denote containment. 1-7 denote migration paths.

Fig. 1: Example of a containment hierarchy, and various confinement types.

4 SafeHaven

With the basic terminology and notation required to model locality and colocation introduced, we now describe SafeHaven, a framework designed to facilitate the creation, deployment and evaluation of isolation properties.

4.1 Overview

SafeHaven is a framework that assists in creating and deploying a network 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 model 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 [16] 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 [31] 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* help to simplify the automatic generation and verification of policy enforcement code, paving the way for future formal analysis.

Probes and Agents A *probe* is an abstraction for an event source, typically implemented in SafeHaven as an Erlang server process. *Agents* are management probes that can modify one or more confinements.

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

Alg. 2: Agent partitioning processes between CPUs by owner via SafeHaven.

Capabilities An agent can create, destroy or migrate a locality if it owns its associated *capability*. Capabilities serve to describe the extent of an agent's influence. To exert influence on locations outside its scope, an agent must proxy its requests through an external agent that controls the target scope. For example, a probe within a tenant's virtual machine may ask an agent within the underlying cloud provider for an isolated **VC**, which then changes the **VC** to **C** mappings.

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* [16]) that serves as a communication capability.

Confinement Discovery 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.

To facilitate the creation of dynamic policies, SafeHaven provides a series of *reconnaissance* (or `recon`) functions that query the underlying system at runtime and build a partial model of the infrastructure, translating it into a graph of first-class Erlang objects. Alg. 2 demonstrates an agent's use of SafeHaven's `recon` functions. Handles to the system's running processes (Line 1) and available CPU cores (Line 2) are loaded into lists of `locality` structures that can be manipulated programmatically. This example describes a simple property that partitions processes to different **C**s based on their user ID (Lines 5-9). The procedure for pinning (or migrating) processes (Line 10) will be described in the next section.

4.2 Migrating Confinements

An agent's core isolation operator is *migration*. Agents perform both *objective* and *subjective moves* [11], as they can migrate confinements to which they be-long as well as external confinements. The following section describes methods with which one can migrate system structures, namely VCPUs, process groups, processes, containers and virtual machines.

Virtual CPUs (VC) Virtual CPUs in `KVM` [2] can be pinned to different sets of CPUs by means of a mask, set through libvirt [3]. **VC**s can only be migrated to cores to which the parent **VM** has been assigned.

Process/Control Groups ($\mathbf{P_E}$) Pinning processes to CPUs via affinities has a drawback in that unprivileged processes can change their own mappings at will, subverting their confinement. Instead, *control groups* (managed via `cpusets`) [27] are used to define a hierarchy of **C** partitions. Assigning processes to a partition confines their execution to that **C** group, which cannot be exited through `sched_setaffinity`. All processes are initially placed within a default *root* control group. Control groups can be created, remapped or destroyed dynamically. Destroying a group will not automatically kill its constituent processes, rather they will revert to that group's parent.

Processes and Containers (P, Con) Process migration moves a process from one $\mathbf{P_E}$ to another, using mechanisms that vary based on the level at which the control groups are co-located. Arbitrary processes can be moved directly amongst $\mathbf{P_E}$ groups within the same **OS** using `cpusets`, which is fast and can be performed in bulk. Conversely, if the target $\mathbf{P_E}$ exists within a different **OS**, additional mechanisms must be used to translate the process' data structures across system boundaries. In SafeHaven, this is handled using `criu` [1], which enables process checkpoint and restore from within user-space. Recent versions of the Linux kernel (3.11 onwards) have built-in support for the constructs required by `criu`. Migration preserves a process' $\mathbf{P_E}$ containment structure.

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 such as LXC or Docker [1]. Live migration for LXC containers is still under active development. An alternative stop-gap measure is to perform checkpoint and restore, transferring the frozen image in a separate step [37].

Virtual Machines (VM) SafeHaven uses `KVM` for virtualisation, managed via libvirt. 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`-like architectures, with hypervisor-level agents residing within dom0. The choice of hypervisor largely determines what type of instrumentation can be made available to probes.

Similarly to process migration, **VM**s can be migrated locally (changing **C** pinnings) using $\mathbf{P_E}$ groups, or at the global level (changing **OS**). The latter is performed using *live migration*, backed by a *Network File System* (NFS) server storing **VM** images. Recently, experimental patches have been released that enable *post-copy* migration through libvirt, which also requires patching the kernel and QEMU[1]. Using post-copy migration, a virtual machine is immediately migrated to its destination, and pages are retrieved from the original machine on demand. The drawback of post-copy migration is that a network failure can corrupt the **VM**, as its state is split across machines. *Hybrid migration* reduces this risk by initially using standard pre-copy and switching to post-copy migration if the system determines that the transfer will not converge, which would happen when memory pages are being modified faster than they can be transferred.

Other Operations In addition to being migrated, **VM**, **P** and **Con** isolations 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 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* [7], and a greedy allocation policy will not produce an optimal allocation. Nevertheless, our scheme 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.

5 Case Studies

The previous section detailed the architecture of SafeHaven and the migration techniques it employs. The following section describes the application and evaluation of these methods in the context of illicit-channel mitigation. All experiments were carried out on two Intel i7-4790 machines (4 cores $\times$ 2 hardware threads) with 8GB RAM. **VM**s were allocated 2 **VC**s and 2GB of RAM, and had 40GB images. A third computer acted as an NFS server hosting the virtual machines' images (average measured sequential speeds: 54MB/s read, 70 MB/s

[1] `https://git.cs.umu.se/cklein/libvirt`

write), and all machines were connected together via a consumer-grade gigabit switch. **VM**s were connected to the network through a bridged interface. All systems were running Ubuntu 14.04 LTS with the 3.19.0-rc2+ kernel and libvirtd version 1.2.11, patched to enable post-copy support (Section 4.2).

5.1 Case 1: System-Wide Covert Channel

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

Overview Wu et al. [40] 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 was then used to implement a cross-VM covert channel.

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* [21] 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 [15]. 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).

Policy Alg. 3 outlines the behaviour of the agents participating in the mitigation. 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 ϵ, 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 $\mathtt{isol}_{\mathrm{D}}(\mathrm{X})$ predicate, which checks whether X is isolated within D. Evaluating this accurately from

Require: An event rate threshold ϵ
Require: A_T set of tenant-owned agents, A_C set of cloud-owned agents

```
1:  agent TENANT(OS:X, A_T)
2:    for all P:P ∈⁺ X do
3:      if evs(P) ≥ ε ∧ ¬isol_X(P) then
4:        D ← ⊥
5:        if mobile(P) then
6:          D ∈ {D' | TENANT(D', ∗) ∈ A_T
                  ∧isol_D'(P)}
7:        if D ≠ ⊥ then
8:          P ↷ D
9:        else if X ∈ VM:V then
10:         request CLOUD(Y, ∗). V ∈⁺ Y
11:   TENANT(X, A_T)
12: end agent
```

(a) Tenant agent

```
1:  agent CLOUD(M:Y, A_C)
2:    receive isol request for VM:X ∈⁺ Y
3:    if ¬isol_Y(X) then
4:      D ∈ {D' | CLOUD(D', ∗) ∈ A_C
                ∧isol_D'(X)}
5:      if D ≠ ⊥ then
6:        X ↷ D
7:      else
8:        fallback strategies
9:    CLOUD(Y, A_C)
10: end agent
```

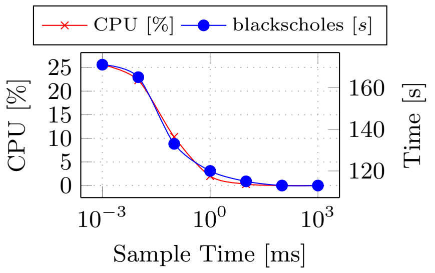
(b) Cloud agent

Alg. 3: Agents for mitigating a system-wide channel.

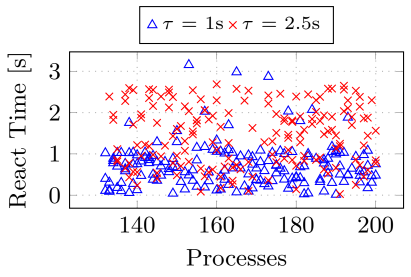
within the tenant's scope requires additional information from the cloud agent regarding its neighbours. The strictest interpretation of isolation would be to allocate a physical machine to each **VM** requesting isolation. Another approach is to stratify isolation into different classes determined by user access lists [12], 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 [35] 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.

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 τ within which the scan must be performed. Hardware counters were accessed using the *Performance Application Programming Interface* (PAPI) [29] library, with calls proxied through an Erlang module using *Native Implemented Functions* (NIF) [16]. The test machines exposed a native event type that counts misaligned atomic accesses (LOCK_CYCLES:



(a) Detector overhead against ϕ



(b) React time on varying $\vec{\mathbf{P}}$, τ = 1s, 2.5s

Fig. 4: Detector overhead and reaction times.

SPLIT_LOCK_UC_LOCK_DURATION [21]). Conversely, another machine to which we had access, namely an AMD Phenom II X6, 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 more false positives.

The procedure for measuring a process' event emission rate is to attach a counter to it, sleep for a sample time ϕ, 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 ϕ will affect the detector's duty cycle. Setting $\phi = \tau/|\vec{\mathbf{P}}|$ guarantees that each process will have been sampled once within each τ 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 ϕ produces an even CPU usage, but leads to an unbounded reaction time.

We tested our hypothesis regarding the infrequency of misaligned atomic accesses by sampling each process in a virtualised and non-virtualised environment 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 spikes on the order of a few hundreds per second during use. We then measured the emission rate of the attack's sender process using the reference implementation of Wu et al. [40], 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 ± 2 seconds.

Fig. 4a shows the detector's CPU usage (measured directly using `top`) against varying ϕ on shifting the detector's logic into a compiled C probe and enumerating processes directly from `/proc/`. To fully encompass the detector's overhead, we pinned the virtual machine to a single VCPU. At $\phi = 10$ms, overhead peaked at a measured 0.3%. This was confirmed by executing the CPU-intensive `blackscholes` computation from the PARSEC benchmark suite [8] in parallel with the detector, and observing a speed-up proportional to ϕ. Fig. 4b describes how reaction time varied against the number of processes being monitored, where reaction time 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.

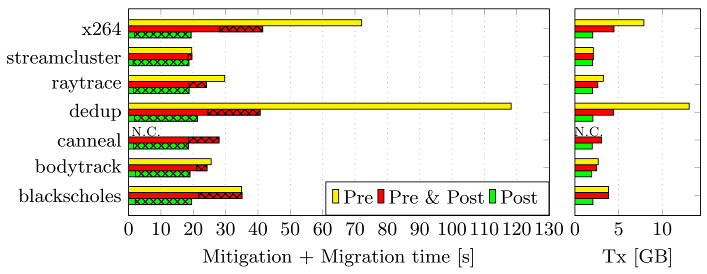


Fig. 5: Comparison of pre-copy, hybrid and post-copy migration.

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

Table 1: Summary of detection and mitigation times (s).

Mitigation Once a potential attack is detected, it must be isolated. The performance of process migration will be discussed in further detail in Section 5.2. For now, we will focus on the different modes of **VM** migration.

Fig. 5 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'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 traffic. Post-copy exhibited the most consistent behaviour, both in terms of migration time as well as generated traffic. During the course of our experiments, we found that attempting to start a migration 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. Nevertheless, **VMs** migrated using post-copy resumed execution at the target in at most 2.13 seconds, and 1.51 seconds on average, which includes the delay. Total migration time and data transferred were also consistently low, averaging 20 seconds and 2GB, respectively.

Table 1 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.

Conclusion We have shown how hardware event counters can be used to detect an attack efficiently, quickly and precisely, and how post-copy migration considerably narrows an attack'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.

5.2 Case 2: Moving Target Defence

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

Overview The moving target defence [46] 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.

Policy Alg. 6 describes the moving target defence as a generalisation of the formulation given by Zhang et al. [46]. 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 following section attempts to establish practical approximations for the aforementioned predicates.

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

Alg. 6: General form of the moving target defence.

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. We estimate the next value of $H()$ using an *exponential average* [36], 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. We take $h = 0.5$.

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, we 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.

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.

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. 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.

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.

		Con $\curvearrowright$ VC		**VC $\curvearrowright$ C**		**Con $\curvearrowright$ OS**			**Con $\curvearrowright$ OS**			**VM $\curvearrowright$ 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 2: Migration times for different isolation types and paths (ms).

Table 2 lists the migration times measured when migrating containers and **VM**s through each migration path (paths 1-7 in Fig. 1b) whilst executing various benchmarks from PARSEC, with the hosts being otherwise idle. Given its consistent behaviour, we only considered post-copy migration when moving **VM**s. The timings for **Con** migration were broken down into its phases. To keep **Con**

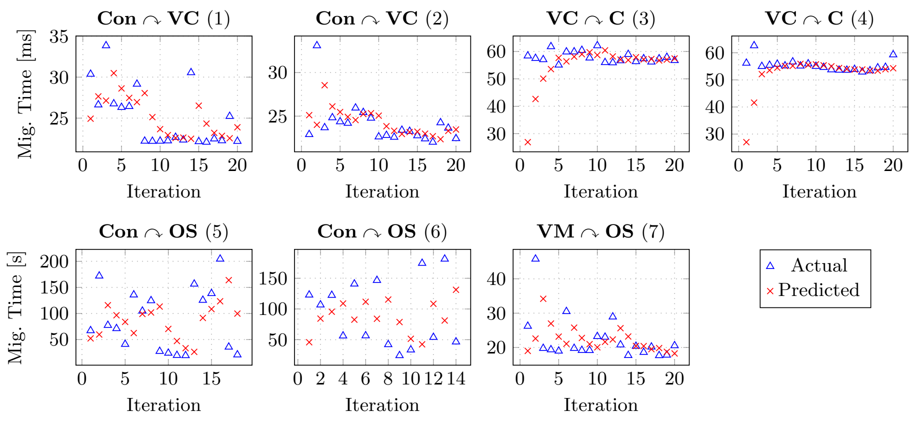


Fig. 7: Predictions of $H()$ against measured migration times.

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

We evaluated the relationship between performance and migration frequency on the system 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 2. A fifth **VM** served as a destination for cross-**VM** process migration, and was kept idle. The second machine ws configured with three tenants running benchmarks and two idle **VM**s. Table 3 in Appendix A 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. Fig. 7 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, we found that migration operations generally had no discernible effect on the neighbouring tenants, although we posit 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).

Conclusion We have investigated 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. 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.

5.3 Other Policies

HomeAlone *HomeAlone* [43] uses a Prime-Probe attack to monitor cache utilisation, and a trained classifier to recognize patterns indicative of shared locality. This can be used to implement a hypervisor-independent version of the `isol`() predicate described in Section 5.1, or to detect adversarial behaviour.

Network Isolation Networks can harbour illicit channels [9, 10]. Isolation at this level can be achieved via a combination of soft and hard isolation, with trusted machines sharing network segments and traffic normalisers [17] monitoring communication at the edges.

6 Conclusion

In this work, we examined the use of migration, in its many forms, to dynamically reconfigure a system at runtime. Through the SafeHaven framework, we described and evaluated the use of migration to implement an efficient and timely mitigation against a system-wide covert-channel attack. We also demonstrated how a moving target defence can be enhanced by considering multiple levels and granularities of isolation, examining the costs associated with migrating entities at each level, and showing how performance and granularity are correlated.

References

1. CRIU project page (Apr 2015), `http://criu.org/Main_Page`
2. KVM project page (Apr 2015), `http://www.linux-kvm.org/`
3. Libvirt project page (Apr 2015), `http://libvirt.org/`
4. Aciiçmez, O., Koç, c.K., Seifert, J.P.: On the power of simple branch prediction analysis. pp. 312–320. ASIACCS'07, ACM, New York, NY, USA (2007)
5. Agat, J.: Transforming out timing leaks. In: Proc. of the 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 40–53. POPL'00, ACM, New York, NY, USA (2000)
6. Askarov, A., Zhang, D., Myers, A.C.: Predictive black-box mitigation of timing channels. pp. 297–307. CCS'10, ACM, New York, NY, USA (2010)
7. Azar, Y., Kamara, S., Menache, I., Raykova, M., Shepard, B.: Co-location-resistant clouds. pp. 9–20. CCSW'14, ACM, New York, NY, USA (2014)
8. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Proc. of the 17th Int. Conf. on Parallel Architectures & Compilation Techniques (Oct 2008)

9. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: ESORICS. pp. 355–371 (2011)
10. Cabuk, S., Brodley, C.E., Shields, C.: Ip covert timing channels: design and detection. CCS'04, ACM, New York, NY, USA (2004)
11. Cardelli, L., Gordon, A.D.: Mobile ambients. POPL'98, ACM Press (1998)
12. Caron, E., Desprez, F., Rouzaud-Cornabas, J.: Smart resource allocation to improve cloud security. In: Sec., Priv. & Trust in Cloud Sys. Springer (2014)
13. Coppens, B., Verbauwhede, I., Bosschere, K.D., Sutter, B.D.: Practical mitigations for timing-based side-channel attacks on modern x86 processors. pp. 45–60. S&P'09, IEEE Computer Society, Washington, DC, USA (2009)
14. Dolan-Gavitt, B., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: Mining memory accesses for introspection. CCS'13, ACM, New York, NY, USA (2013)
15. Du, J., Sehrawat, N., Zwaenepoel, W.: Performance profiling in a virtualized environment. In: 2nd USENIX Workshop on Hot Topics in Cloud Computing (2010)
16. Ericsson AB: Erlang Reference Manual User's Guide, 6.2 edn. (Sep 2014), `http://www.erlang.org/doc/reference_manual/users_guide.html`
17. Gorantla, S., Kadloor, S., Kiyavash, N., Coleman, T., Moskowitz, I., Kang, M.: Characterizing the efficacy of the nrl network pump in mitigating covert timing channels. IEEE Transactions on Info. Forensics & Sec. 7(1), 64–75 (Feb 2012)
18. Gueron, S.: Intel advanced encryption standard (aes) new instructions set. `http://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf` (May 2010)
19. Hu, W.M.: Reducing timing channels with fuzzy time. pp. 8–20. S&P'91, IEEE Computer Society (May 1991)
20. Hu, W.M.: Lattice scheduling and covert channels. p. 52. S&P'92, IEEE Computer Society, Washington, DC, USA (1992)
21. Intel: System Programming Guide, Intel® 64 & IA-32 Architectures Software Developer's Manual, vol. 3B. Intel (May 2011)
22. Intel: Instruction Set Reference, Intel® 64 & IA-32 Architectures Software Developer's Manual, vol. 2. Intel (Jan 2015)
23. Keller, E., Szefer, J., Rexford, J., Lee, R.B.: Nohype: virtualized cloud infrastructure without the virtualization. In: 37th annual Int. Symp. on Computer architecture. pp. 350–361. ISCA'10, ACM, New York, NY, USA (2010)
24. Kim, T., Peinado, M., Mainar-Ruiz, G.: Stealthmem: system-level protection against cache-based side channel attacks in the cloud. Security'12, USENIX Association, Berkeley, CA, USA (2012)
25. Lampson, B.W.: A note on the confinement problem. CACM 16(10) (Oct 1973)
26. Li, P., Gao, D., Reiter, M.: Mitigating access-driven timing channels in clouds using stopwatch. In: 43rd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN). pp. 1–12 (Jun 2013)
27. Linux: cpuset(7) - Linux manual page (Aug 2014), `http://man7.org/linux/man-pages/man7/cpuset.7.html`
28. Mdhaffar, A., Ben Halima, R., Jmaiel, M., Freisleben, B.: A dynamic complex event processing architecture for cloud monitoring and analysis. In: 2013 IEEE 5th Int. Conf. on Cloud Comp. Tech. & Science. CloudCom, vol. 2, pp. 270–275 (Dec 2013)
29. Mucci, P.J., Browne, S., Deane, C., Ho, G.: Papi: A portable interface to hardware performance counters. In: Proc. of the DoD HPCMP Users Group Conf. (1999)
30. Okamura, K., Oyama, Y.: Load-based covert channels between xen virtual machines. In: 2010 ACM Symp. on Applied Computing. pp. 173–180. SAC'10, ACM, New York, NY, USA (2010)

31. OpenStack Foundation: OpenStack Documentation (Feb 2015), `http://docs.openstack.org/`
32. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of aes. In: RSA Conf. on Topics in Cryptology. CT-RSA'06, Springer-Verlag (2006)
33. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: CCS'09. pp. 199–212. ACM, New York, NY, USA (2009)
34. Sailer, R., Jaeger, T., Valdez, E., Cáceres, R., Perez, R., Berger, S., Linwood, J., Doorn, G.L.: Building a mac-based security architecture for the xen opensource hypervisor. In: 21st Annual Comp. Sec. Appl. Conf. ACSAC'05 (2005)
35. Saltaformaggio, B., Xu, D., Zhang, X.: Busmonitor: A hypervisor-based solution for memory bus covert channels. EuroSec'13, ACM (2013)
36. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, chap. 5, p. 161. Wiley Publishing, 7th edn. (2005)
37. Tycho: Live migration of linux containers. `http://tycho.ws/blog/2014/09/container-migration.html` (Oct 2014)
38. Varadarajan, V., Ristenpart, T., Swift, M.: Scheduler-based defenses against cross-vm side-channels. Security'14, USENIX Association, San Diego, CA (Aug 2014)
39. Wang, Z., Lee, R.B.: Covert and side channels due to processor architecture. In: 22nd Annual Comp. Sec. Appl. Conf. pp. 473–482. ACSAC'06, IEEE Computer Society, Washington, DC, USA (2006)
40. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: High-speed covert channel attacks in the cloud. Security'12, USENIX Association, Berkeley, CA, USA (2012)
41. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of l2 cache covert channels in virtualized environments. pp. 29–40. CCSW'11, ACM, New York, NY, USA (2011)
42. Yarom, Y., Falkner, K.E.: Flush+reload: a high resolution, low noise, l3 cache side-channel attack. IACR Cryptology ePrint Archive 2013, 448 (2013)
43. Zhang, Y., Juels, A., Oprea, A., Reiter, M.K.: Homealone: Co-residency detection in the cloud via side-channel analysis. pp. 313–328. S&P'11, IEEE Computer Society, Washington, DC, USA (2011)
44. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-tenant side-channel attacks in paas clouds. pp. 990–1003. CCS'14, ACM, New York, NY, USA (2014)
45. Zhang, Y., Reiter, M.K.: Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. pp. 827–838. CCS'13, ACM, New York, NY, USA (2013)
46. Zhang, Y., Li, M., Bai, K., Yu, M., Zang, W.: Incentive compatible moving target defense against vm-colocation attacks in clouds. In: Info. Sec. & Priv. Research, IFIP Adv. in Info. & Comm. Tech., vol. 376. Springer Berlin Heidelberg (2012)

A Appendix: Migration Frequency and Performance

			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)	210000	14	2,886	124	167	380	119	153	369
	180000	18	3,565	124	165	380	118	152	369
Con $\curvearrowright$ **OS** (6)	210000	14	2,780	122	164	375	119	155	373
VM $\curvearrowright$ **OS** (7)	120000	17	2,028	120	179	392	121	176	375
	90000	23	2,025	122	170	384	120	162	392
	60000	39	2,282	121	162	389	122	173	390
	30000	125	3,770	121	169	384	124	177	394

Table 3: Effect of migration frequency on performance when running at capacity.