Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration

Soo-Jin Moon
Carnegie Mellon University
Pittsburgh, PA USA
soojinm@andrew.cmu.edu

Vyas Sekar
Carnegie Mellon University
Pittsburgh, PA USA
vsekar@andrew.cmu.edu

Michael K. Reiter
University of North Carolina
Chapel Hill, NC USA
reiter@cs.unc.edu

ABSTRACT

Recent studies have shown a range of co-residency side channels that can be used to extract private information from cloud clients. Unfortunately, addressing these side channels often requires detailed attack-specific fixes that require significant modifications to hardware, client virtual machines (VM), or hypervisors. Furthermore, these solutions cannot be generalized to future side channels. Barring extreme solutions such as single tenancy which sacrifices the multiplexing benefits of cloud computing, such side channels will continue to affect critical services. In this work, we present *Nomad*, a system that offers vector-agnostic defense against known and future side channels. *Nomad* envisions a provider-assisted VM migration service, applying the moving target defense philosophy to bound the information leakage due to side channels. In designing *Nomad*, we make four key contributions: (1) a formal model to capture information leakage via side channels in shared cloud deployments; (2) identifying provider-assisted VM migration as a robust defense for arbitrary side channels; (3) a scalable online VM migration heuristic that can handle large datacenter workloads; and (4) a practical implementation in `OpenStack`. We show that *Nomad* is scalable to large cloud deployments, achieves near-optimal information leakage subject to constraints on migration overhead, and imposes minimal performance degradation for typical cloud applications such as web services and Hadoop MapReduce.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection – *Information flow controls*

General Terms

Security

Keywords

Cloud computing; Cross-VM side-channel attacks; VM migration

1 Introduction

Several recent efforts have demonstrated the possibility of information leakage via *co-residency side channels* in shared cloud environments, where VMs from different clients are multiplexed on the same hardware. Following the early work of Ristenpart et al., [33], a variety of side channels have already been demonstrated. These exploit different levels of hardware caches [42, 44], main memory [36], and OS/hypervisor scheduling effects [42] to extract private information from unsuspecting co-resident clients. The set of known side channels and the efficiency of information leakage continue to grow over time.

Unfortunately, mitigating side-channel attacks forces cloud providers and clients into undesirable situations. Each such side channel requires detailed fixes to hardware, guest VM OS configurations, and hypervisors (e.g., [32, 37, 34, 23, 45]). This has two practical problems. First, these require significant changes to existing deployments and applications. Second, given that the set of side channels is unknown (and growing), this puts us on an untenable trajectory of constant hardware/software changes to tackle future attacks.

Ideally, we want defenses that are (a) *general* across a broad spectrum of side-channel attacks and (b) *immediately deployable* with minimal or no modifications to existing cloud hardware and software. At first glance, these goals seem fundamentally at odds with the multiplexing benefits of cloud computing since the ultimate way to avoid side channels is to eliminate co-residency altogether; i.e., by creating "private" single-client deployments. However, this reduces the cost savings via statistical multiplexing, which has been a key driver for cloud adoption.

In this paper, we present *Nomad*, a system that demonstrates that it is possible to achieve a *general* and *immediately deployable* defense against side-channel attacks without resorting to single tenancy. The high-level idea behind *Nomad* is simple. Rather than eliminate co-residency altogether, we aim to limit the information leakage due to co-residency by carefully coordinating the placement and migration of VMs. To this end, we envision cloud providers offering a migration-as-a-service to their clients to mitigate co-residency side channels. In this respect, *Nomad* can be viewed as an application of the moving target defense philosophy to mitigate side channels [13].

This approach has several natural advantages. First, by focusing on the *root cause* of side channels (i.e., co-residency), *Nomad* is agnostic to the specific side-channel vector used, and is robust against unforeseen side channels that meet certain conditions (§4). Second, it requires no changes to the cloud provider's hardware, client applications, and hypervisors and can be deployed "out of the box" as it requires only changing the VM placement/scheduling algorithms deployed by the cloud provider.

The key challenges in realizing this vision in practice are (1) scalability of the placement and migration scheduler and (2) impact on application performance. For (1), we develop scalable heuristics

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813706.

that can handle large-scale cloud deployments. For (2), we use a proof of concept implementation of *Nomad* atop `OpenStack` [5] and evaluate application performance for Wikibench [6] and Hadoop MapReduce [2]. We observe that the performance impact on applications is minimal with *Nomad*'s default configurations which can handle several classes of attacks. However, for emerging attacks that rely on fast side channels (i.e., capable of extracting a key in few minutes), we acknowledge the need for out-of-band defense.[1] Furthermore, many cloud applications have in-built resilience which further minimizes impact of *Nomad*-induced migrations; e.g., web servers run replicas with elastic load balancing and Big-Data workloads have mechanisms to deal with stragglers (e.g., [14]).

Contributions and Roadmap: In summary, this paper makes four contributions:

- We formalize the problem space and characterize different models of information leakage across two key dimensions: collaboration across adversary VMs and information replication across a client's own VMs (§3).
- We identify provider-assisted live migration as a robust defense against a broad spectrum of co-residency side channels (§4).
- We develop a practical and scalable migration strategy that can handle large datacenter workloads, which is several orders of magnitude faster than strawman solutions (§5).
- We develop a practical implementation of *Nomad* by extending `OpenStack` (§7).

In the rest of the paper, we begin with background and related work in §2. We evaluate *Nomad*'s scalability and information leakage resilience in §8. We discuss potential attacks against *Nomad* in §6, and open issues in §9, before concluding in §10.

2 Background and Related Work

In this section, we review recent work on side channel threats in public clouds and argue why known defenses are not practical. We also provide a brief overview on prior work on VM live migration.

Side-channel attacks in cloud: Cloud services (e.g., Infrastructure-as-a-Service) place VMs of different clients on the same physical machine. This relies on virtual machine monitors (VMM) to provide *isolation* between co-resident VMs. Unfortunately, this is insufficient, and recent works (e.g., [33, 36, 29, 41, 37, 42, 27, 21]) have demonstrated the feasibility of adversaries performing cross-VM side-channel attacks. One of the first use cases of cross-VM side-channel attacks in cloud was to demonstrate that an attacker can identify where the target VM is likely to reside by measuring activity burst time of a VM [33]. Other attacks have identified pages that a VM shares with its co-resident client VMs, revealing information about the victim's applications [36] and OS [29]. More fine-grained attacks have exfiltrated cryptographic keys via *Prime+Probe* attacks on the square-and-multiply implementation of GnuPG [42] and sensitive application data on Platform-as-a-Service (PaaS) clouds [44]. Furthermore, some side channels have extracted keys by exploiting the memory sharing (i.e., memory deduplication) on LLC (last-level caches) [41, 20]. Recently, researchers have demonstrated fast side channels. Liu et al., [27] have shown a fast *Prime+Probe* attack on LLC by probing only one cache set, and Irazoqui et al., [21] have recovered the AES key in 2–3 min by exploiting the use of huge size pages.

[1] Adjusting the configurations to handle fast side channels (i.e., [27, 21]) comes at a cost of performance degradation. Thus, we recommend the use of other side-channel defenses in conjunction with *Nomad* to strengthen the defense (§4).

Proposed defenses against side channels: Given the spate of attacks, several countermeasures have been proposed at different levels: hypervisor, guest OS, hardware and application-layer approaches.

Hypervisor-based approaches include hiding the program execution time [38] and altering the timing exposed to an external observer [26]. To address the attack of Zhang et al., [42] which extracted a key by frequently preempting a target VM, Varadarajan et al., [37] proposed modifying the Xen scheduler to limit the frequency in which an attacker can preempt the victim. Hypervisor-based defenses can also use statistical multiplexing of shared resources to prevent eavesdropping [32, 23]. In particular, Kim et al., [23] proposed locking cache lines to prevent preemption by an attacker and multiplexing the cache lines among VMs such that each has an access to its own.

Defenses have been suggested inside the client guest OS (e.g. injecting noise into protected processes on L1 and L2 caches [45]), or at the application level (e.g., partitioning a cryptographic key across multiple VMs [31]). Other than software-based defenses, defenses can also be incorporated in hardware designs by applying access randomization (e.g., [40, 28]) and resource partitioning (e.g., [30]).

At a high level, these proposed approaches suffer from two fundamental limitations as (1) they cannot be generalized to different types of side channels; and (2) these require significant changes to the hypervisor, OS, hardware, and applications.

VM migration and placement: A key enabler for our work is VM live migration that has become an invaluable management tool. These general trends with the advancement of VM live migration bode well for the adoption of *Nomad* (e.g., [16, 35, 22, 46]).

VM placement as side-channel defenses: Concurrent to our work on *Nomad*, recent efforts also formulate theoretical problem of VM placement to limit cross-VM side-channel attacks (e.g., [9, 43, 25, 19]). The work closest to *Nomad* is by Li et al., [25]. In comparison to these efforts, *Nomad* is a) more scalable (e.g., we can handle tens of thousands machines whereas most of these efforts do not consider scalability); b) more general in terms of threat model (e.g., we consider collaboration and replication); and c) makes no assumption on which clients or VMs are likely threats. Furthermore, these efforts fall short of providing a real implementation; we demonstrate a practical implementation in `OpenStack` with minimal code changes.

3 Problem Overview

In this section, we describe a general model of information leakage in public clouds that (a) is independent of the specific types of side channels; (b) can capture powerful adversaries whose VMs may collaborate; and (c) incorporates the information replication characteristics of clients.

3.1 Threat Model

We begin by scoping the adversary goals and capabilities.

Adversary goals and capabilities: We assume that each cloud client has some private information (e.g., secret keys or private database records). The goal of the adversary is to extract as much information as possible. We consider a powerful adversary model with the following characteristics:

- *Arbitrary side channels:* The adversary is capable of launching a wide spectrum (of possibly unknown) side-channel attacks against other co-resident VMs. We are agnostic to the specific algorithms or system resources used by these side channels (e.g., CPU, memory, network, power).

- *Target identification:* We assume that the adversary can determine if/when the target client of interest is co-resident with a VM it owns; e.g., inspecting the pattern of behavior or using external probing [33]. As such, we assume detecting the target incurs zero cost to the adversary.
- *Arbitrary client workload:* We consider an *open* system where the adversary can control its own workload and launch VMs and terminate them as it chooses.
- *Efficient information collation:* We assume that the adversary can accumulate private bits across time (epochs) when it is co-resident with the target and has some intelligent techniques for information aggregation. For example, if a client C and adversary A are co-resident at time T1 and T3, but not T2, then we assume that A can combine the information it has gathered during T1 and T3. Note that by assuming such efficient collation, we consider a stronger adversary model; i.e., in practice a real adversary may get duplicate/redundant bits across epochs but we conservatively assume that the adversary gets unique bits per epoch.
- *Unknown adversary:* Finally, we assume that the client or the cloud provider cannot pinpoint a specific client who could be malicious.

We do, however, impose two constraints on adversaries' capabilities. First, we assume that the adversary does not have explicit control over the placement of VMs in the cloud environment. Second, we assume that while the VMs for the same client may collaborate in some deployment models, there is *no collusion* across clients. With respect to collusion, we assume that there is some non-trivial cost to creating a new client identifier (e.g., a verified credit card) so that it is not possible to launch Sybil attacks for collusion [12]. Moreover, in order to formulate arbitrary side-channel attacks, we abstract away the details of individual attacks, and consider an attack that has a constant leakage rate of K bits per epoch. We acknowledge that different attacks may have different rates or different temporal properties (e.g., K may decrease or increase with time). Modeling the temporal efficiency of attacks is outside the scope of this paper.

3.2 Components of Information Leakage Model

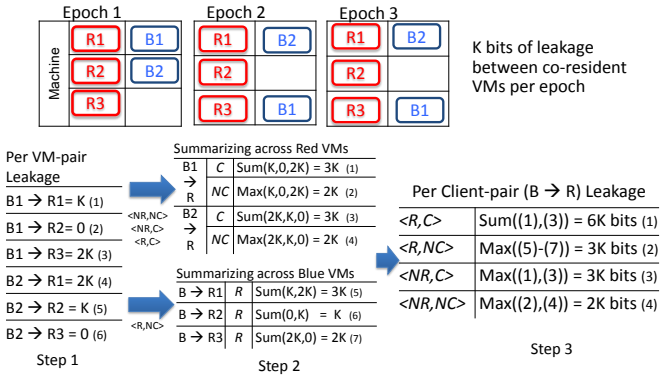


Figure 1: Implication of having Replicated (R) vs. NonReplicated (NR) clients and Collaborating (C) vs. NonCollaborating (NC) adversaries

We identify three key dimensions that affect information leakage (*InfoLeakage*) model that we discuss below. We explain these using the example scenario in Figure 1 where we have two clients: Blue and Red with 2 and 3 VMs respectively. The figure shows the placement of their VMs across three epochs across 3 machines with 2 VM slots per machine.

1. *Across time:* The total amount of private information leaked from a client c to c' is naturally proportional to the total temporal co-residency between their VMs. For instance, in the *Prime+Probe* attack by Zhang et al., [42], information leakage happens at a rate of few bits per minute and thus the total leakage will be more if they are co-resident longer. In Figure 1 if we assume a constant leakage rate of K bits per epoch via some side channel, then B2 will leak $2 \times K$ bits to R1 as they are co-resident for two epochs (Row 4 of Step 1).
2. *Information sharing across adversary's VMs:* Our threat model allows adversarial client's VMs to *collaborate* to increase the rate of information extraction. For instance, in the context of *Prime+Probe* attack [42], adversarial VMs may decide to work on different parts of the cryptographic key and combine these to reconstruct the key. Let us revisit the example in Figure 1. As shown in Step 2, if the Red VMs collaborate then they can extract more information (marked as C in Step 2 which extracted $3 \times K$ bits in contrast to $2 \times K$ bits for NC model). Assuming the same leakage rate of K bits per epoch between a pair of VMs, we now see that the Red client can extract a total $3 \times K$ bits over the three epochs from each Blue VM (Row 1 and 3 of Step 2).
3. *Information replication of the target:* Depending on a client's workload, different VMs belonging to a client may carry the same bits of private information. For instance, consider a *replicated* web server deployment with all the replicas having the same private database records. Intuitively, having replicated client workloads poses higher threat as they can lead to higher information leakage. Revisiting our example in Figure 1, we see that the Red VM R1 can potentially extract $3 \times K$ bits over the three epochs in case the Blue VMs are replicas because the Blue VMs B1 and B2 carry the same information. In contrast, if B1 and B2 were not replicas, then R1 will have K bits from B1 and $2 \times K$ bits from B2 but these could not be combined as information are distinct. Note that this can be combined with the collaboration described above. That is, if R1–3 were also collaborating, then in aggregate the adversary Red will have gathered $6 \times K$ bits of private information from the tenant Blue over the three epochs (Row 1 of Step 3).

3.3 Formalizing Information Leakage Model

The above discussion provided an intuitive overview of the different dimensions in modeling information leakage. Next, we formally define the information leakage so that it can be used to guide the placement and migration decisions of the *Nomad* system.

Preliminaries: Let $VM_{c,i}$ denote VM i belonging to the client c and $VM_{c',i'}$ denote the VM i' of a different client c' (i.e., a potential adversary). Let $CoRes_{c,i,c',i'}(t)$ be a binary indicator variable that captures if $VM_{c,i}$ and $VM_{c',i'}$ are co-resident at an epoch, t; i.e., there exists some machine m on which they both reside at t. Then, having defined per-epoch co-residency between VM pairs, we need to summarize this value across time. In order to aggregate information leakage as a function of co-residency across time, we consider a *sliding window* model over the most recent Δ epochs. For example, if cryptographic keys are refreshed periodically (say every few hours), then any bits of information an adversary VM has gathered the previous day will have no value since the key has been modified.

Let $InfoLeak_{c,i \to c',i'}(t, \Delta)$ denote the information leakage from $VM_{c,i}$ to $VM_{c',i'}$ measured over the sliding window of time $[t -$

$\Delta, t]$. Formally,

$$InfoLeak_{c,i\rightarrow c',i'}(t,\Delta) = \sum_{t\in[\text{t-}\Delta,\text{t}]} CoRes_{c,i,c',i'}(t) \quad (1)$$

Then, we can use information leakage at a *VM granularity* to construct information leakage at a *client granularity* over time, $InfoLeak_{c\rightarrow c'}(t,\Delta)$. This quantity defines information leakage from a given client c to a different c' measured at epoch t over the sliding window of time $[t-\Delta, t]$, and is also a function of client replication and adversary collaboration.[2]

Modeling different leakage scenarios: Given these preliminaries, we can model four possible cases.

1. *NonReplicated client; NonCollaborating adversary ($\langle NR, NC\rangle$):* If there is no replication and no collaboration, then the information leakage for a client will be the maximum per-VM-pair information leakage across all pairs of clients. Formally,

$$InfoLeak^{\langle NR,NC\rangle}_{c\rightarrow c'}(t,\Delta) = \underset{i}{Max}\ \underset{i'}{Max}\ InfoLeak_{c,i\rightarrow c',i'}(t,\Delta) \quad (2)$$

   Under the $\langle NR, NC\rangle$ scenario (Figure 1), the Blue client leaked a total of $2\times K$ bits to the Red adversary because the maximum information leakage between any VM pair was $2\times K$ bits (Row 4 of Step 3).

2. *NonReplicated client; Collaborating adversary ($\langle NR, C\rangle$):* In this case, for each client VM, there will be a cumulative effect across the adversary VMs since they can collaborate. However, the inter-client leakage will be determined by the client VM that has leaked the most. Formally,

$$InfoLeak^{\langle NR,C\rangle}_{c\rightarrow c'}(t,\Delta) = \underset{i}{Max} \sum_{i'} InfoLeak_{c,i\rightarrow c',i'}(t,\Delta) \quad (3)$$

   In our example, under the $\langle NR, C\rangle$ scenario, each Blue client's VMs leaked $3\times K$ bits across all Red VMs. Thus, the leakage from Blue to Red will be $3\times K$ bits (Row 3 of Step 3).

3. *Replicated client; NonCollaborating adversary ($\langle R, NC\rangle$):* In this case, there will be a cumulative effect across the client VMs since they have the same information but the inter-client leakage will be determined by the adversary VM that has extracted the most information. Formally,

$$InfoLeak^{\langle R,NC\rangle}_{c\rightarrow c'}(t,\Delta) = \underset{i'}{Max} \sum_{i} InfoLeak_{c,i\rightarrow c',i'}(t,\Delta) \quad (4)$$

   Revisiting our example, we see that each Red VM (i.e., VM_{R1}, VM_{R2}, and VM_{R3}) has extracted $3\times K$, K, and $2\times K$ bits, respectively, from all Blue client's VMs. Therefore, the Blue client, under the non-collaborating scenario of the Red VMs, has leaked a total of $3\times K$ bits (Row 2 of Step 3).

4. *Replicated client; Collaborating adversary ($\langle R, C\rangle$):* Finally, when the client is replicated and the adversary can collaborate, we see cumulative effects across both client and adversary VMs. Formally,

$$InfoLeak^{\langle R,C\rangle}_{c\rightarrow c'}(t,\Delta) = \sum_{i}\sum_{i'} InfoLeak_{c,i\rightarrow c',i'}(t,\Delta) \quad (5)$$

   Revisiting our example, we see that the Blue client leaks a total of $6\times K$ bits to the Red client (Row 1 of Step 3).

[2]Note that the information leakage is asymmetric and $InfoLeak_{c\rightarrow c'}$ need not be equal to $InfoLeak_{c'\rightarrow c}$.

These equations naturally capture our intuitive explanations from earlier; the leakage is highest when we have replication and collaboration (i.e., $\langle R, C\rangle$) and least when neither occurs (i.e.,$\langle NR, NC\rangle$). When we have either replication or collaboration but not both, the value will be in between these two extremes.

4 System Overview

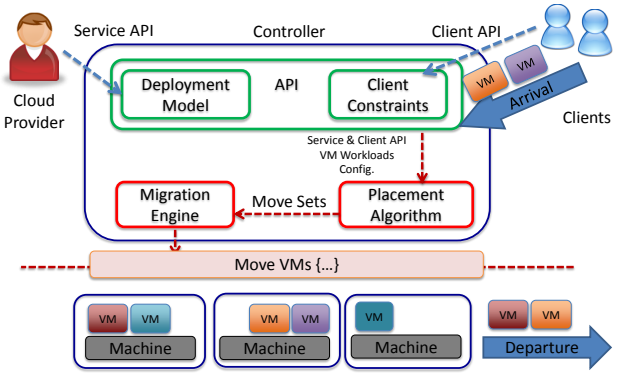


Figure 2: System overview

In this section, we provide a high-level overview of the *Nomad* system which provides a *side-channel agnostic* mechanism to defend against the information leakage attacks discussed in the previous sections. Figure 2 shows the overall system architecture of *Nomad*.

High-level idea: Recall that we consider a strong adversary model capable of (a) launching arbitrary (and unforeseen) side channels and (b) precisely targeting potential victims. Moreover, the client does not know which other clients might be potential threats. Thus, every other client is a potential side-channel threat. Our goal is to provide a mitigation mechanism against this strong threat model and without *any modification* to client guest OS, hypervisors, or the cloud provider's hardware platforms.

One extreme solution might be for clients to request "single tenancy" solutions (i.e., dedicated hardware). While this may be an option, it sacrifices the statistical multiplexing gains that are key for the low costs of cloud computing. That said, this extreme solution does provide some intuition on how we can defend against arbitrary side channels from arbitrary tenants in the cloud, namely *minimizing co-residency*.

Building on this insight, we envision a *provider-assisted* approach where cloud clients leverage the provider as a trusted ally via an *opt-in* "migration-as-a-service" solution. Our specific contribution here is in identifying a new *security-specific* use case for migration beyond the typical applications for planned maintenance [11].

Having described the high-level idea of *Nomad*, we now describe the APIs (i.e., Service API and Client API) of *Nomad* before describing the end-to-end workflow.

Service API: As we saw in the previous section (§3), the information leakage between a pair of clients depends on the information sharing capabilities of the adversarial client's VMs and the information replication across the client's VMs.

Thus, the cloud provider needs to make a decision at deployment time regarding the type of adversarial and client model it wants to offer; i.e., decide if it wants $\langle NR, NC\rangle$, $\langle NR, C\rangle$, $\langle R, NC\rangle$, or $\langle R, C\rangle$ model. We assume this decision is made public. Different cloud providers may choose one or the other depending on their cost-performance considerations or customer needs or the same

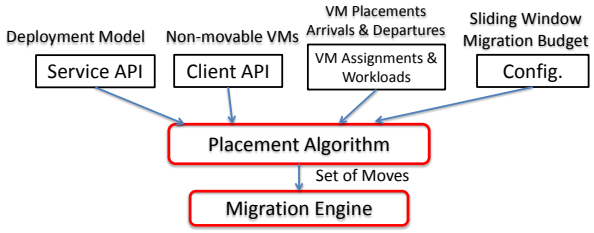


Figure 3: High-level view of the *Placement Algorithm*

provider may offer varied offerings. Clients who wish to be protected against a specific model of information leakage based on their workloads and preferences can choose a cloud provider with the desired service offering (i.e., R client who wants a guarantee against NC adversary chooses a provider with $\langle R, NC \rangle$ offering).

Client API: *Nomad* allows clients to specify their workload constraints to help minimize the impact of migrations on client applications. This API allows clients to specify non-migration constraints on some VM instances. For example, in the web server case the front-end load balancer has to have zero down time and similarly in the Hadoop case the master node has to always be up.

End-to-end workflow: Now, we describe the end-to-end workflow of *Nomad*. Client VMs arrive and depart based on their workload requirements. The provider runs a *Placement Algorithm* with the goal of minimizing the information leakage across arbitrary pairs of clients (e.g., ensuring that no specific pair of clients are co-resident for a significant chunk of time) while also minimizing disruption to the client applications. To achieve this goal, the *Placement Algorithm* takes a few key inputs to decide a placement policy (Figure 3).

1. *API:* The *Placement Algorithm* needs to know 1) Service API (i.e., the deployment model) to correctly compute the *InfoLeakage* between client-pairs, and 2) Client API to know which VM instances are "non-movable" when computing VM placements.
2. *VM placements and workloads:* The current and past VM assignments for past Δ epochs are used to decide next placement assignments. The *Placement Algorithm* also takes an input of VM arrivals/departures since the last epoch. These are used to update VM-pair co-residency states internally tracked by the algorithm.
3. *Configurations:* Configurations such as Δ (*sliding window*) and migration budget are also used in the algorithm.

Using these inputs, the *Placement Algorithm* computes the VM assignments for the next epoch. The provider runs a *Migration Engine* which takes the logical output of the *Placement Algorithm* and periodically re-balances the placement of client VMs across its provisioned hardware. The period (i.e., duration of an *epoch*, D) and configuration parameters such as Δ determine the security implications of our system and should be adjusted based on the state-of-the-art side-channel attacks.

Security implications of *Nomad*: Having presented the overview of *Nomad*, we now explain the security semantics and implications of the system.

The security implications depend on three parameters: 1) K, leakage rate; 2) Δ, number of epochs in a sliding window; and 3) D, epoch duration. Note that D and Δ are configured by providers and K summarizes the capability of the side channels.

In essence, *Nomad* is resilient against any side channel where $K \times \Delta \times D \leq P$ where P is the Min(time to refresh the secret, time it takes to extract the secret). Intuitively, this condition means that an adversary should not be able to recover the secret *even if* an adversarial client is co-resident for the entire duration of the sliding window. Note that K, in reality, refers to abstract rate of leakage, which is the basic leakage rate for $\langle NR, NC \rangle$ (i.e., single VM case) and some function of replication and collaboration otherwise. For clarity, we base our discussion on the $\langle NR, NC \rangle$ case.

These parameters can be configured on knowing the state-of-the-art leakage characteristics (i.e., K). To make our discussion more concrete, we explain using the work of Zhang et al., [42] as an example, which took 6 hours to extract a 457-bit key giving a leakage rate of 1.27 bits per min. Thus, we suggest $D = 30$ min and $\Delta = 10$ epochs to ensure that $K \times \Delta \times D \leq P = 6$ hours.

We do acknowledge that *Nomad* is not resilient against fast side-channel attacks that extract the secret within an epoch (i.e., [21, 27]). Fast side-channel attacks, in principle, could be addressed by reducing D accordingly (i.e., for side-channel attacks capable of extracting the key in 2–3 min, we suggest $D = 30$ sec). However, decreasing D comes at a cost of performance degradation. Furthermore, the cluster size has to decrease accordingly to handle small D to ensure that the time to compute the placement is smaller than D (Figure 5). In this case, we suggest using other side-channel defenses in conjunction with *Nomad* (i.e., a general side-channel solution) to strengthen defenses against evolving side channels. For instance, such approaches can be used to 1) reduce K (e.g., injecting cache noise [45] or divide the private keys among several client VMs [31]); and 2) reduce P by refreshing the key (i.e., secret) frequently.

Challenges: Having described this high-level view of the *Nomad* system, we highlight key practical challenges we need to address to turn this vision into practice:

- *Efficient algorithm:* Given the four different models of information leakage, we need efficient algorithms that can work in all four models. For instance, a seemingly natural solution might be to simply randomize the VM assignments across epochs. However, as we will see, such naive solutions can actually be counterproductive (Figure 4 in §8). Finally, we need to ensure that the provider and the clients do not incur significant performance penalty due to VM migrations.
- *Scalability:* Large cloud deployments have roughly tens of thousands servers. Therefore, the *Nomad Placement Algorithm* must be capable of scaling to such large deployments. While the problem can be theoretically formulated as a large constrained optimization problem, even state-of-art solvers cannot solve a problem instance with more than 40 machines even after a day (Table 1 in §8).
- *Deployability: Nomad* must be incrementally deployable with minimal changes to the existing production stack and control platforms and without modifications to client applications.

In the following section, we show how we design efficient and scalable greedy heuristics that can apply generally to all four deployment options. Then, in §7 we describe how *Nomad* can be seamlessly added to a production cloud management system and discuss our specific experiences with `OpenStack`. Finally, we show empirically in §8 with simulated workloads that we can achieve good bounds on information leakage using a small number of migrations and that the impact on typical cloud workloads (e.g., replicated web services and Hadoop MapReduce) is small.

5 Nomad Placement Algorithm

In this section, we describe the design of the *Placement Algorithm* in *Nomad*. We begin by describing the high-level problem that we

Algorithm 1 Baseline Greedy Algorithm

```
1: function GREEDYALGORITHM(CurPlace, Budget, TypesofMoves)
2:     NumMig = 0, ChosenMove={}
3:     MoveSet = InitializeMoves(TypesofMoves,CurPlace)
4:                                  ▷ Return a set of ⟨move,cost⟩
5:     while NumMig ≤ Budget do
6:         MoveBenefits=
       InfoLeakReduction(MoveSet, curPlace,clientConstraints)
7:                                  ▷ Return a set of ⟨move,cost,benefit⟩
8:         move = PickBestMove(MoveBenefits)
9:         ChosenMove.insert(move)
10:        NumMig += move.cost()
11:        CurPlace = UpdatePlacement(CurPlace,move)
12:        MoveSet = UpdateMoves(MoveSet,move)
13:    end while
14:    Return ChosenMove
15: end function
```

need to solve and then highlight the scaling limitations of strawman solutions. Then, we present the main ideas underlying the *Nomad* approach that enable a scalable realization.

5.1 Problem Formulation

We begin by describing the abstract problem that the *Placement Algorithm* needs to solve to provide the context for why this problem is intractable in practice.

The *Placement Algorithm* needs to compute the VM placements every epoch with the goal of minimizing the information leakage between arbitrary pairs of cloud clients, while ensuring that the overall cost of doing so (i.e., number of migrations) is low. Specifically, we want to minimize the total information leakage function subject to some *budget* on the migration overhead measured in terms of total number of migrations. We acknowledge that there are other ways to capture the trade-off between migration overhead vs. information leakage.

Meeting the security concern should not come at the cost of scalability. Our problem target size (i.e., large public cloud deployment) is tens of thousands of servers with roughly 4-5 VM slots per server.[3] We envision the *Nomad Placement Algorithm* running at the beginning of each epoch, with each epoch lasting several minutes up to tens of minutes. A reasonable target to compute the placement assignments for one epoch would be roughly under 1 min. The choice of 1 min for the computation time allows epoch duration to be as small as few minutes (§4).

Since the cloud provider cannot predict the VM arrivals and departures into the future, we consider a *myopic* formulation that determines placement for the next epoch given the historical placements over the previous few epochs. We can model this placement problem subject to migration budget constraints as an *integer linear program* (ILP). For completeness, we present the full ILP in Appendix A. Unfortunately, solving this ILP is intractable and it takes more than a day to even solve a small problem instance with just 40 machines (Table 1 in §8). Thus, while the ILP approach is an exact optimal solution in terms of the leakage subject to fixed migration budget, it is far from viable in terms of the scalability requirements. This motivates the need for heuristic approximations as we describe next.

[3]While public numbers are hard to get, tens of thousands of servers seems roughly in the ball park of public deployment instances.

5.2 Baseline Greedy

Given that we are solving a budgeted optimization problem, we resort to a natural greedy algorithm. Algorithm 1 shows our baseline greedy algorithm.

In the baseline greedy design, we enumerate a set of *moves* involving VMs. For instance, we can consider all possible n-way swaps between VMs or consider pair-wise swaps (i.e., TypesofMoves = {free-insert, pair-wise swap, . . . , n-way swaps})

Each move has both a cost incurred in terms of number of migrations required to execute the move and the benefit it yields in terms of the reduction in information leakage. Then, in each iteration of the greedy algorithm, we pick the best move (Line 8 in Algorithm 1) within the migration budget that gives us the maximum benefit in terms of reduction in information leakage.

Note that each move conceptually changes the state of the system and thus the benefit of future moves may decrease or increase depending on the moves we have already made; e.g., moving $VM_{c,i}$ may mean that all previously considered swaps involving this instance may no longer provide any value. Thus, we explicitly recompute the set of allowed moves and the benefit that they yield (Line 6 and 12 in Algorithm 1).

Unfortunately, even using this greedy algorithm instead of the ILP solver does not provide the desired scalability; e.g., even running this on a small 50 node cluster does not meet our 1 min goal.

5.3 Scalable Greedy Algorithm

Next, we describe key ideas of our scalable greedy algorithm to improve scalability. Using a careful run-time analysis, we identified three key bottlenecks in this baseline greedy algorithm:

1. Calculating the benefit of each move (Line 6): Recomputing benefit is computationally expensive as the *InfoLeakage* across all VM pairs and client pairs have to be computed.
2. Large search space (Line 3): A large search space results from having many machines and many types of moves (i.e., free-inserting a VM into an empty slot, pair-wise swaps, etc.,).
3. Updating move after each state change (Line 12): Updating move sets requires generating all possible moves which leads to a large input size for the benefit computation (Line 6).

Incremental benefit computation: Recomputing the benefit (Line 6 in Algorithm 1) is a large contributor to high run time of the baseline greedy. Thus, we use an *incremental benefit computation* which computes the *delta* in the current value of the objective function by only updating information leakage for set of dependent client pairs whose *InfoLeakage* are affected by the *move*. This eliminates the need to compute the entire co-residency pairs across all VMs when 1) a potential move has been tried to evaluate the benefit of a move or 2) a move is made.

However, to enable the use of this approach, we need to make approximations to non-$\langle R, C\rangle$ *InfoLeakage* models which consist of Max operations. Finding the *delta* with Max operation requires the algorithm to iterate over all other inputs to the Max. This is in contrast to the Sum whose *delta* only depends on one input's value before and after an update. Therefore, we introduce the concept of "Soft-Max" for $\langle NR, NC\rangle$, $\langle NR, C\rangle$, and $\langle R, NC\rangle$ models to benefit from the scalability gain by using an *incremental benefit computation* as shown below.

1. $\langle NR, NC\rangle$:

$$InfoLeak_{c\to c'}^{\langle NR,NC\rangle}(t,\Delta) \approx e^{\alpha * InfoLeak_{c,i\to c',i'}(t,\Delta)}, \alpha > 0 \quad (6)$$

2. $\langle NR, C\rangle$:

$$InfoLeak_{c\to c'}^{\langle NR,C\rangle}(t,\Delta) \approx e^{\alpha * \sum_{i'} InfoLeak_{c,i\to c',i'}(t,\Delta)},\ \alpha > 0 \quad (7)$$

3. $\langle R, NC\rangle$:

$$InfoLeak_{c\to c'}^{\langle R,NC\rangle}(t,\Delta) \approx e^{\alpha * \sum_{i} InfoLeak_{c,i\to c',i'}(t,\Delta)},\ \alpha > 0 \quad (8)$$

The intuition behind "Soft-Max" is that an exponential function is a good approximation of the Max operation as the function gives more weights to larger values. Suppose we consider a sliding window Δ of size 5. Then, for the $\langle NR, NC\rangle$ model, the inputs to $InfoLeak_{c,i\to c',i'}(t,\Delta)$ ranges from 0 to 5. Using Equation 6 (for $\alpha = 0.8$), the values $[0, 1, 2, 3, 4, 5]$ would map to $[0, 2.23, 4.95, 11.02, 24.53, 54.60]$. Our scalable greedy algorithm then will naturally see the most benefit in reducing the larger co-residency values; reducing $InfoLeak_{c,i\to c',i'}(t,\Delta)$ from 5 to 4 gives a larger *InfoLeakage* reduction than reducing $InfoLeak_{c,i\to c',i'}(t,\Delta)$ from 4 to 3.

Search space: We present two key ideas to tackle two causes (many types of moves and many machines) for a large search space (Line 3 in Algorithm 1).

1. Hierarchical placement: The high-level idea is to group machines into clusters (with each cluster consisting of approximately 1,500 machines) and each client assigned to a cluster. This design choice builds on the following insight. A move can only affect clients whose VMs reside in the affected machines. Suppose a move involves moving a $VM_{c,i}$ from machine 1 to machine 2. Then, the only *InfoLeakage* client pairs affected are clients whose VMs reside in machine 1 and 2 (i.e., only the co-residency between $VM_{c,i}$ and VMs in machine 1 and 2 are affected by this move). Therefore, we consider it *inefficient* to try the moves across all machines when the number of affected co-residency pair is bounded.
2. Pruning the move sets: We could potentially consider a move from a free-insert up to n-way swaps where n is the number of VMs. However, we limit the types of moves to a free-insert and pair-wise swaps. Our evaluation on the effect of pruning the move sets shows significant gain in scalability with little-to-no loss optimality (Figure 6 in §8).

Lazy evaluation: Third, we identified that the need to recompute the move sets after each move affects scalability (Line 12 in Algorithm 1). Consider a 1,500 machine per cluster with 4 VM slots per machine with an expected occupancy rate of 50% (i.e., 3,000 VMs). Entire move sets then would contain approximately $3,000 \times 1,500 + \binom{3000}{2}$ entries. Therefore, recomputing the entire move set is *inefficient* when only a few are affected. To tackle this, we use lazy evaluation [24]. First, we populate the entire move table at the beginning of an epoch. Second, the algorithm traverses the move set starting from the move that gives the most benefit. If the claimed benefit from the time that benefit was computed lies within 95% of the current benefit and a move is feasible, then that move is made. If not, the move is re-inserted with an updated benefit. We show that lazy evaluation brings little-to-no loss in optimality even for a cluster size of 50 (§8). Note that as a side effect of providing a client-agnostic defense and considering global co-residency across all client-pairs, *Nomad* can utilize the same algorithm for $\langle R, NC\rangle$ and $\langle NR, C\rangle$.

Dealing with heterogeneous resource constraints: In a real cloud setting, servers are not identical and the resource requirements of each VM may also vary. In our design of the algorithm, we have abstracted the server resource constraints as VM slots but this can easily be extended to consider heterogeneous VM resource requirements and server constraints (i.e., vCPU, RAM, Disk, etc.) Resource constraints are handled in the *Nomad Placement Algorithm* when the scheduler looks for free VM slots. For clarity, we evaluated the algorithm using a homogeneous server configuration (Figure 6 in §8).

6 Security Analysis

In this section, we describe how *Nomad* deals with strategic adversaries and the potential threats that could arise with the deployment of *Nomad*.

Strategic adversary: By construction, *Nomad* defends against legacy side-channel adversaries. Here, we focus on strategies of advanced adversaries who are aware of the *Nomad*'s algorithms. Specifically, we identify three possible attack vectors to obtain high information leakage in spite of *Nomad*: (1) launch many VMs; (2) exploit the non-migration constraints of the client-facing API; and (3) induce a lot of churn. Here, we qualitatively analyze these scenarios and defer quantitative results to §8. For brevity, we only discuss the $\langle R, C\rangle$ model since that has the highest possible leakage surface (i.e., it subsumes other models) and argue why *Nomad* either renders these vectors ineffective or induces high costs for the adversary.

- *Launch many VMs:* An adversarial client can launch a large number of long-lasting VMs hoping to be co-resident with target clients. First, we observe that this comes at a high cost for the adversary; e.g., public clouds such as Amazon EC2 which charges based on CPU hours [1]. Moreover, *Nomad*'s goal of minimizing *InfoLeakage* will naturally tend to localize VMs of clients with many VMs (even without explicitly identifying the adversary client).
- *Exploit non-migration constraints of the Client API (§4):* To help legitimate applications with strong dependencies on bottleneck VMs, *Nomad*'s Client API allows a client to specify non-migration constraints. An adversary may try to exploit this feature and request a large percentage of its workloads to fall under non-migration constraints to avoid the eventual clustering mentioned above. Note that this is a serious threat as this is legitimate behavior allowed by the API and thus is *non-detectable*. Second, an adversary incurs no additional cost in specifying non-migration constraints. We observe, however, that *Nomad* is resilient to this strategy. For "non-movable" instances, *Nomad* runs the *Placement Algorithm* to determine the initial placement of VM to cause minimal increase to the overall *InfoLeakage*. The algorithm will then naturally localize "non-movable" VMs of a specific client upon "non-movable" instances' arrivals.
- *Frequent churn:* An adversary can induce frequent churn with VMs arrival/departure. This can exhaust the migration budget of the *Placement Algorithm* and poses a higher threat than statically launching the same number of VMs, as it does not give *Nomad Placement Algorithm* enough epochs to localize the adversary VMs. Suppose that such anomalous behavior is *detectable*. Then, after the detection, the cloud provider can dedicate a set of machines which are assigned to the particular client creating frequent churn. Note that this will not impact legitimate clients who may also exhibit high churn (even though this is unlikely). However, designing algorithms for detecting churn is outside the scope of this work and can be done via well-known anomaly detection techniques [10].

Potential new threats: We acknowledge the potential new threats that could arise with the deployment of *Nomad*.

- *New side-channel threats:* We acknowledge that deploying *Nomad* may have *indirect* consequences that may strengthen some side channels. For instance, with *Nomad* which incurs periodic migration, memory deduplication(e.g., [36]), if enabled, will lose its benefits. Therefore, if more aggressive memory deduplication is deployed to preserve memory saving benefits, existing side channels that rely on memory deduplication (e.g., [41, 20]) may become even stronger. We acknowledge that this is a limitation of *Nomad* and studying the implication of migration on side channels is an interesting direction for future work.
- *Other cloud threats:* Cross-VM side channels are not the only risks in cloud environments. For instance, a separate risk in cloud deployments is a compromised hypervisor. Constantly migrating VMs may increase the risk of a VM being placed on a machine with a compromised hypervisor. However, existing defenses for compromised hypervisors could also be used in conjunction with *Nomad* (i.e., [8, 39]). Addressing this threat is an interesting direction for future work.

7 System Implementation

In this section, we describe our *Nomad* prototype (using `OpenStack` [5]) following the structure in Figure 2. We begin with a high-level overview of `OpenStack` before describing how we modified it to implement *Nomad*.

OpenStack overview: `OpenStack` is a popular open-source cloud computing software platform that is used to deploy Infrastructure-as-a-Service (IaaS) cloud solutions [5]. At a high level, `OpenStack` controls Compute, Storage, and Network resources. The key component of interest to us is `OpenStack` Compute, known as `Nova`, which consists of the cloud controller representing a global state and other compute nodes (i.e., machines). Each compute node runs a hypervisor, in our case, `KVM`, which is responsible for managing VMs and executing the migration of VM invoked via the `OpenStack` API calls.

Migration choices: `OpenStack` supports different modes of VM migration [3]: 1) Non-live migration which shuts down instance for a period of time to migrate to another hypervisor; and 2) Live migration which has almost zero instance downtime. To allow for minimal impact on client application, the natural choice was to use live migration. Within live migration, there are several implementation options [3]: shared storage-based live migration, block live migration, and volume-backed live migration. In general, shared storage and block live migration have better performance that volume-backed live migration as they avoid copying the entire VM image from the source compute nodes to the destination. For implementation convenience in our testbed, we choose the shared storage live migration option.

Migration Engine: Recall the role of the *Nomad Migration Engine*, which executes the migration of VMs as dictated by the *Placement Algorithm*. We implement the engine in the `Nova-Scheduler` at the Controller node, which was a natural implementation choice as *Nomad* requires having a global view of all the machines and VM states. We extended the code to implement *Migration Engine* as part of the controller services.

The high-level workflow is as follows. First, when VMs are launched, the *Migration Engine* saves the VM ID and the client ID to its internal client-to-VM mapping. At every epoch, *Migration Engine* queries `OpenStack`'s database to get the VM-to-host mappings. Then, *Migration Engine* offloads the job to the *Placement Algorithm* to compute the VM assignments. Once the algorithm finishes computing the VM placements, the *Migration Engine* executes migrations as dictated by the algorithm.

Recall that one of the goals of *Nomad* is the minimal modification to an existing cloud platform. Our implementation consists of roughly 200 lines of *Python* code in the `Nova-Scheduler` code and achieves our objectives of minimal modification.

Placement Algorithm: Our implementation of the *Placement Algorithm* consists of 2,000 lines of custom `C++` code. This module is invoked every epoch by an API call from the *Migration Engine*. Upon the call from the *Migration Engine*, which sends the high-level inputs described in §4, the algorithm computes the VM assignments and also internally stores co-residency history to be used in subsequent epochs. All the optimizations described in §5 are implemented as part of the *Placement Algorithm*.

Finally, we note that the modular design of *Nomad* with a standardized interface between the placement algorithm and the *Migration Engine* allows us to easily decouple the scheduling logic from the implementation of the *Migration Engine*. In our own development experience, this "plug-and-play" capability proved quite useful.

8 Evaluation

In this section, we address the following questions:
(1) How does the information leakage resilience of *Nomad*'s algorithm compare with strawman solutions?
(2) Does *Nomad* algorithm scale to large deployments? What are the benefits of the optimizations from §5?
(3) What is the impact of migrations on real applications in a realistic cloud workload?
(4) How resilient is *Nomad* to smarter adversaries ?

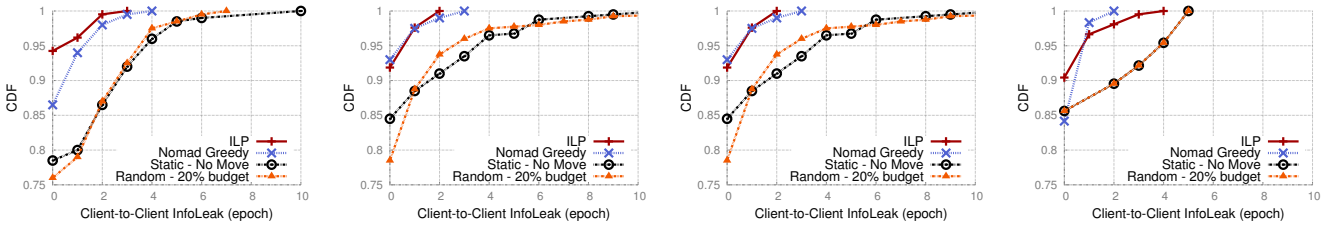
Setup: For (3), we use a local *OpenStack Icehouse* deployment on our testbed equipped with 2.50 GHz 64-bit Intel Xeon CPU L5420 processor with 8-cores, 16 GB RAM, 500 to 1000GB disks, and two network interfaces with 100Mbps and 1Gbps network speed. Each machine runs KVM on Ubuntu 14.04 (Linux kernel v3.13.0). For (1), (2), and (4), we evaluated *Nomad Placement Algorithm* and other placement strategies using synthetic workloads. The evaluation of *Nomad* placement algorithm was conducted using varying cluster sizes with 4 slots each. For our simulation workloads, the number of customers was the same as the cluster size and the initial setup consisted of 2 VMs per clients.[4] Every epoch a 15% of new VMs would arrive and 15% of an existing VMs would depart, creating constant churn every epoch. The migration budget was set to 15% for testing our solution and an ILP solution. In testing the end-to-end application performance with *Nomad*, we used epoch duration (i.e., D) of 4 min for web-service and 1 min for Hadoop MapReduce.[5]

8.1 Information Leakage

We compare the per-client leakage achieved by *Nomad* vs. three strawman solutions: (1) Integer Linear Programming, (2) Random Scheduler, and (3) Static Scheduler. The random scheduler picks a VM at random and a random slot. The VM picked at random is inserted to the slot if the slot is empty. If the slot is occupied, the

[4]Note that the use of 4 slots per server and 2 VMs per client was bottlenecked by the run-time of the ILP and is not fundamental limitation of the algorithm.

[5]Different Ds are used because the job completion time for each experiment (i.e., Wikibench and Hadoop) differs. Therefore, D was scaled such that the number of epochs, hence the number of migrations, in each experiment is roughly the same for both experiments (i.e., $D = 4$ min for 20 min completion time and $D = 1$ min for 3–4 min completion time).



(a) $\langle R, C \rangle$ - CDF (b) $\langle NR, C \rangle$ - CDF (c) $\langle R, NC \rangle$ - CDF (d) $\langle NR, NC \rangle$ - CDF

Figure 4: CDF of client-to-client *InfoLeakage* for different VM placement strategies

	Cluster Size			
	10	20	40	50
ILP	0.48s	7.83	>1 day	>1 day
Nomad scheduler	0.00005s	0.002s	0.015s	0.02s

Table 1: Scalability of ILP vs. *Nomad*

chosen and occupying VMs are swapped. This process of swapping is run until all the migration budget is exhausted. The static scheduler runs an initial randomized placement when VMs arrive but runs no migration. The ILP (Appendix A) runs the exact optimization algorithm using `CPLEX` [4]. We compare these approaches on a simulated cluster size of 20 with 4 slots per machine, 20 clients, with an expected occupancy rate of 50% and 15% arrival and 15% departure rates per epoch. We chose a small setup since the ILP does not scale to larger setups.

Figure 4 shows the CDF of inter-client *InfoLeakage* measured over a sliding window of 5 epochs for the 4 different *InfoLeakage* models. We make two key observations. First, naive random migration or static placement can result in substantially higher leakage. Second, the *Nomad Placement Algorithm* achieves close-to-optimal performance w.r.t. the ILP solution.[6] Finally, there is one subtle observation regarding fairness across different clients; i.e., do some clients incur more leakage or migration relative to others? By intent, the ILP or the *Nomad* algorithm does not explicitly take fairness into account. But we find that both ILP and *Nomad* achieve good fairness in practice as even the 95th percentile of the distribution is low.

8.2 Scalability

***Nomad* vs. ILP:** Recall that we chose a greedy heuristic because of the scaling limitations of the optimal ILP formulation. First, we compare the scaling properties of *Nomad* vs. ILP in Table 1. The result shows that the ILP is fundamentally intractable even for a small cluster size of 40 machines and that *Nomad* is several orders of magnitude faster. Note that this scalability benefit does not compromise optimality as we saw in Figure 4 where the optimality loss of *Nomad* is negligible.

Scaling to large deployments: Our target computation time for *Nomad* was roughly 1 min. Next, we analyze the scalability of the scheduler to determine the dimensions of the cluster that can meet this target. Figure 5 shows the scaling properties for different datacenter sizes. Based on the result, we can determine that a reasonable size of our cluster is 1,500 machines across the different models.

[6]Note that both the ILP and *Nomad* Greedy optimize the total *InfoLeakage*. Therefore, it is possible for *Nomad* Greedy to outperform the ILP solution w.r.t. the inter-client *InfoLeakage*.

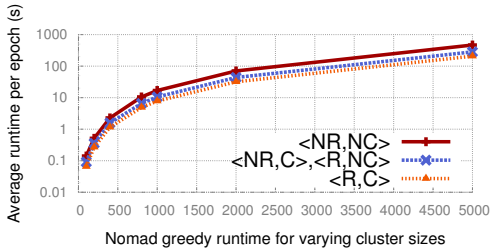
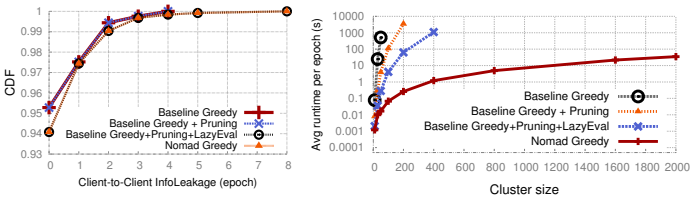


Figure 5: Scalability for different *InfoLeakage* models

Effect of design choices: The scaling properties of *Nomad* stem from three key design decisions in Appendix §5): (1) pruning, (2) lazy evaluation, and (3) incremental computation. Next, we evaluate the relative benefits of each of these ideas and also confirm that these do not sacrifice optimality.

To tease apart the relative contributions, we evaluate the impact of applying these ideas progressively as follows: (1) We begin with a *Baseline Greedy* which is a naive implementation of Algorithm 1 that considers three types of moves: free-insert, pair-wise swaps, and 3-way swaps;[7] and (2) We *prune* the search space to only consider free inserts and pair-wise swaps; (3) We enable *lazy evaluation* to eliminate the need to re-compute the move table after every state changes (every move); (4) Last, we enable the *incremental computation* to efficiently compute the benefit of each move. Note that (4) also entails the use of "Soft-Max" of *InfoLeakage* calculations for non-$\langle R, C \rangle$ models.

Figure 6a shows that these design choices result in negligible drop in optimality. We could only show the optimality results for a 50-node cluster because the basic greedy takes several hours to complete for larger sizes. Figure 6b also shows the increased scalability with each idea enabled and shows that each idea is critical to provide an order-of-magnitude reduction in compute time. We also see that the largest decrease comes from the use of *incremental benefit computation*.



(a) Optimality loss (b) Compute time

Figure 6: Impact of key design ideas in scaling the *Nomad Placement Algorithm* and justifying that these do not sacrifice optimality

[7]Even adding 3-way moves dramatically increases the run time and thus we do not consider more complex moves.

8.3 System and Application Performance

Migration microbenchmark: First, we start with microbenchmarking the time it takes to migrate three instances using shared-NFS storage live migration (Table 2) [3]. Note that the migration occurs via 1Gbps network and we envision faster migration in faster networks.

	Ubuntu-Cloud (512MB RAM, 1.5GB)	Cirros (512MB RAM, 132MB)	Ubuntu (2048MB RAM, 7 GB)
Total Migration Time (s)	0.89	0.97	1.47

Table 2: Total migration time for different images (in s)

As an end-to-end experiment, we also experimented with an application benchmark to see the performance impact at the application level which combines all CPU, disk I/O and network I/O of application. We chose two representative workloads: web-server and MapReduce workloads.

Wikibench evaluation: For web-service application, we choose *Wikibench* because it uses a real application (Mediawiki) and website is populated with real data dumps [6]. We took the trace file from Sept. 2007 and post-processed it such that the request rate is approximately 10 to 15 HTTP requests per sec.

We conduct an experiment in our 20-node setup. Initial setup includes launching 4 benign clients and 2 additional clients whom for the purpose of illustration play a role of an adversary.[8] Each of 4 client has the following setup: 1) 3 replicated Wikibench backends 2) 1 proxy to load balance between 3 servers, and 3) a worker instance sending HTTP GET requests using the Wikibench trace file. At each epoch, adversarial clients create 15% arrival and 15% departure churn. In our setup, each benign client requests that the

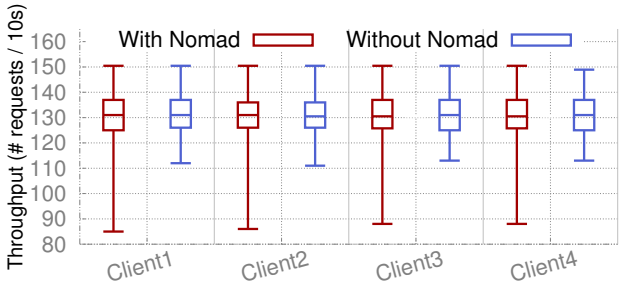


Figure 7: Distribution of throughput for Wikibench workload (with and without *Nomad*)

client worker and a proxy to be "non-movable." The experiment was conducted for 20 min (4 min per epoch, 5 as a Δ) with and without our system. We present the distribution of the throughput (i.e., number of completed requests per 10s bin) over the entire run for each client (Figure 7).

For each client running the Wikibench workload, Figure 7 shows the distribution of throughput using a box-and-whiskers plot with the 25th, 50th, and 75th percentiles, and the minimum and 98th percentile value. We observe relative resilience of our system to migration as the distribution is largely identical with and without *Nomad*. We only observe the decrease in throughput for the lower tail of the distribution. This plot also shows the fairness across each client as no particular client is being penalized by incurring high performance degradation.

[8]For the purpose of illustration, we introduced the concept of benign and adversarial clients. However, the algorithm makes no assumption on which client is adversarial.

Hadoop evaluation: The second representative workload is Hadoop Terasort sorting 800MB data. The VM arrival and departure workloads are identical with that of Wikibench except that the churn was introduced every minute and the epoch size was set to 1 min. Each Hadoop client consists of 5 VMs (i.e.,1 master VM and 4 slave VMs). Each client, via the client API, requests that the master node to be "non-movable". The results are shown in Figure 8. For this experiment, we report the distribution of the job completion time from 100 runs. We consider two types of initial placements (i.e., random vs. clustered). Clustered initial placement refers to the setup in which each client is clustered on 2 machines. Thus, for each client, we report three categories: 1) with *Nomad*- random initial placement; 2) without *Nomad*- random initial placement; and 3) without *Nomad*- optimal (i.e., clustered) initial placement. The results show that *Nomad* does not impact the job performance. Both Wikibench and Hadoop experiments demonstrate that: (1) our system prototype can handle real workloads in an open system; and (2) cloud-representative applications are resilient to migration.

Figure 8: Distribution of job completion time for Hadoop workload (with and without *Nomad*)

8.4 Resilience to advanced adversaries

For brevity, we only focus on the adversary that exploits the non-migration constraints in the Client API, as the other attacks (i.e., launch many VMs or induce churn) are largely disincentivized because they induce high cost or are detectable. Figure 9 presents the effectiveness of our system with a strategic adversary that launched 30 non-movable VMs at an epoch 10. The base case (i.e., simple adversary) refers to an adversary with only 2 VMs, and the without *Nomad* system refers to the system that does random initial placements for all arriving VMs. This result confirms our intuition that *Nomad* is resilient to strategic adversaries exploiting "non-migration" constraints of the Client API (§5).

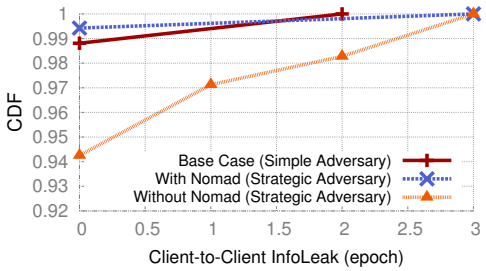


Figure 9: CDF of client-to-adversary *InfoLeakage* at an epoch 19 for a sliding window of 5 epochs (cluster size: 200, number of clients: 200)

9 Discussion

Before we conclude, we discuss four outstanding issues.

Network impact: Overall network impact of *Nomad* due to migration could be a concern. However, we note that modern datacenters have well-provisioned networks (e.g., 10Gbps full bisection bandwidth [17]). Furthermore, *Nomad*'s migrations are predictable and amenable to traffic engineering [7]. Furthermore, with techniques like incremental diffs, the transfer size will be much less than the base VM memory image with 50% reduction (e.g., [18]). Thus, we expect minimal network impact from *Nomad*, especially given that datacenters handle much larger (tens of GB) flows [17].

Fairness across clients: There are two issues with fairness. The first is w.r.t. the leakage guarantees that each client achieves and the second is w.r.t. the migration costs that each client faces. While our current algorithm does not explicitly take into account fairness objectives across clients, our analysis and empirical evaluations show that the greedy algorithm naturally achieves a reasonably fair outcome. That said, extending the basic algorithm to support fairness goals is an interesting direction for future work. In particular, defining suitable fairness objectives in a heterogeneous environment where different clients have varying number of VMs, degrees of replication, and sensitivity to migration is an interesting direction for future work. We posit that some concepts like dominant resource fairness might be useful here [15].

Handling heterogeneous client workloads: Our current *Placement Algorithm* utilizes a different algorithm for each deployment model. An interesting extension is designing the *Placement Algorithm* using a hybrid approach that enables a cloud provider to handle heterogeneous client workloads given an adversarial model (i.e., client specifying to be R or NR under C or NC scenario).

Incentives for adoption: We argue that cloud clients who are security conscious have a natural incentive to opt-in to the *Nomad* service to minimize the impact of side channels. Moreover, the impact of *Nomad* migrations on the applications for reasonable epoch duration will likely be small and thus the cost is quite low. Providers too have a natural incentive to enable *Nomad* as a service as it might introduce new monetization avenues; e.g., *Nomad* can be offered as an value-added security service for a slight, additional fee. Furthermore, we believe that this vision of adding provider-assisted services is naturally aligned with the real-world economics of cloud computing.

10 Conclusions

Co-residency side channels in public cloud deployments have been demonstrated to be a real and growing threat. Unfortunately, existing solutions require detailed changes to hardware/software and client applications, and/or sacrificing the multiplexing benefits that clouds offer. *Nomad* offers a practical vector-agnostic and robust defense against arbitrary (and future) side channels. Moreover, it is effective against strong adversary models where client VMs can collaborate and where we do not even need to pinpoint who the adversary is. The key insight is in leveraging provider-assisted VM migration as a way to bound co-residency and hence limit information leakage across all client pairs. We demonstrated that *Nomad* can scale to large cloud deployments and imposes low overhead on client applications. While there are open questions (e.g., very fast side channels, heterogeneous guarantees), we believe that the core idea of *Nomad* is quite powerful and can complement attack-specific side channel defenses. Seen in a broader context, *Nomad* is a proof point demonstrating a novel cloud provider-assisted security solution and we believe that this paradigm can more broadly enable novel and robust defenses against other security problems.

11 Acknowledgments

This work was supported in part by NSF awards CNS-1440065 and CNS-1330599. We thank Anupam Gupta, Ravishankar Krishnaswamy, Kyle Soska, the anonymous reviewers, and our shepherd Cristiano Giuffrida for their helpful suggestions.

12 References

[1] *Amazon EC2 Pricing*. `http://aws.amazon.com/ec2/pricing/`.
[2] *Apache Hadoop*. `http://hadoop.apache.org/`.
[3] *Configure migrations - OpenStack Cloud Administrator Guide*. `http://docs.openstack.org/admin-guide-cloud/content/section_configuring-compute-migrations.html`.
[4] *IBM CPEX Optimizer*. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`.
[5] *OpenStack OpenSource Cloud Computing Software*. `https://www.openstack.org/`.
[6] *Wikibench*. `http://www.wikibench.eu/`.
[7] M. Al-Fares et al. Hedera: Dynamic flow scheduling for data center networks. In *Proc. USENIX NSDI*, 2010.
[8] A. M. Azab et al. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proc. ACM CCS*, 2010.
[9] Y. Azar et al. Co-location-resistant clouds. In *Proc. ACM CCSW*, 2014.
[10] V. Chandola et al. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.
[11] C. Clark et al. Live migration of virtual machines. In *Proc. USENIX NSDI*, 2005.
[12] J. R. Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.
[13] D. Evans et al. Effectiveness of moving target defenses. In *Moving Target Defense*, pages 29–48. Springer, 2011.
[14] R. C. Fernandez et al. Making state explicit for imperative big data processing. In *Proc. USENIX ATC*, 2014.
[15] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. USENIX NSDI*, 2011.
[16] S. Ghorbani et al. Transparent, live migration of a software-defined network. In *Proc. ACM SOCC*, 2014.
[17] A. Greenberg et al. Vl2: A scalable and flexible data center network. In *Proc. ACM SIGCOMM*, 2009.
[18] D. Gupta et al. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. USENIX OSDI*, 2008.
[19] Y. Han et al. Security games for virtual machine allocation in cloud computing. In *Decision and Game Theory for Security*. 2013.
[20] G. Irazoqui et al. Wait a minute! a fast, cross-vm attack on aes. In *Cryptology ePrint Archive,Report 2014/435*. 2014.
[21] G. Irazoqui et al. S$a: A shared cache attack that works across cores and defies vm sandboxing-and its application to aes. In *Proc. IEEE S&P*, 2015.
[22] C. Jo et al. Efficient live migration of virtual machines using shared storage. In *Proc. ACM VEE*, 2013.
[23] T. Kim et al. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Proc. USENIX Security*, 2012.
[24] A. Krause and D. Golovin. Submodular function maximization. *Tractability: Practical Approaches to Hard Problems*, 3:19, 2012.
[25] M. Li et al. Improving cloud survivability through dependency based virtual machine placement. In *SECRYPT*, pages 321–326, 2012.
[26] P. Li et al. Stopwatch: A cloud architecture for timing channel mitigation. *ACM TISSEC*, 17(2), Nov. 2014.
[27] F. Liu et al. Last-level cache side-channel attacks are practical. In *Proc. IEEE S&P*, 2015.
[28] F. Liu and R. B. Lee. Random fill cache architecture. In *Proc. Micro*, 2014.
[29] R. Owens and W. Wang. Non-interactive os fingerprinting through memory de-duplication technique in virtual machines. In *Proc. IPCCC*, 2011.
[30] D. Page. Partitioned cache architecture as a side-channel defence mechanism. In *Cryptology ePrint Archive,Report 2005/280*. 2005.
[31] E. Pattuk et al. Preventing cryptographic key leakage in cloud virtual machines. In *Proc. USENIX Security*, 2014.
[32] H. Raj et al. Resource management for isolation enhanced cloud services. In *Proc. ACM CCSW*, 2009.
[33] T. Ristenpart et al. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proc. ACM CCS*, 2009.
[34] J. Shi et al. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proc. DSN*, 2011.
[35] V. Shrivastava et al. Application-aware virtual machine migration in data centers. In *Proc. INFOCOM*, 2011.
[36] K. Suzaki et al. Memory deduplication as a threat to the guest os. In *Proc. EUROSEC*, 2011.

[37] V. Varadarajan et al. Scheduler-based defenses against cross-vm side-channels. In *Proc. USENIX Security*, 2014.

[38] B. C. Vattikonda et al. Eliminating fine grained timers in xen. In *Proc. ACM CCSW*, 2011.

[39] J. Wang et al. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, 2010.

[40] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proc. Micro*, 2008.

[41] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proc. USENIX Security*, 2014.

[42] Y. Zhang et al. Cross-vm side channels and their use to extract private keys. In *Proc. ACM CCS*, 2012.

[43] Y. Zhang et al. Incentive compatible moving target defense against vm-colocation attacks in clouds. In *Information Security and Privacy Research*, pages 388–399. Springer, 2012.

[44] Y. Zhang et al. Cross-tenant side-channel attacks in paas clouds. In *Proc. ACM CCS*, 2014.

[45] Y. Zhang and M. K. Reiter. Duppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proc. ACM CCS*, 2013.

[46] J. Zheng et al. Workload-aware live storage migration for clouds. In *Proc. ACM VEE*, 2011.

APPENDIX

A ILP Formulation

In this section, we describe the ILP formulation for the optimization problem that the *Nomad Placement Algorithm* needs to solve every epoch. As discussed earlier, the algorithm takes as input the VM placement over the last sliding window of epochs and other parameters that determine the information leakage. The objective function is to minimize the total information leakage across all client pairs.

We use binary indicator variables, $d_{c,i,k}(\mathrm{t})$, to denote the placement assignment of a specific VM instance i of a client c at machine k in the epoch t. These are the key control variables that we need to set to determine the optimal placements. Then, it naturally follows that the inputs to the ILP formulation are these control variables for the past Δ epochs to be used in summarizing co-residency over time. The output would be the placement assignments for the next epoch.

Equation 9 in Figure 10 states that the total number of VMs assigned to a machine should not exceed machine's capacity. Furthermore, each VM should only have one machine assignment (Equation 10). Equation 11 models a per-epoch binary co-residence relationship that indicates if 2 VMs are co-resident in a machine k at epoch t. Intuitively, this is like modeling an AND operation; i.e., if both VMs have the same machine assignment k, this binary indicator is set to 1. If there exists any machine in which two VMs are co-resident, then we flag that these two VMs ($VM_{c,i}$ and $VM_{c',i'}$) are co-resident at an epoch t (Equation 12).

Having determined whether a given pair of VMs are co-resident, we can summarize the information leakage across the three dimensions, namely over time, over adversary's VMs, and over client's VMs. This models the *InfoLeakage* between a client pair c and c'. For brevity, the figure only shows the model for the $\langle R, C\rangle$ case, the summarization involves summation over adversary's and client's VMs.

Minimize $\sum_{c,c'} InfoLeak^{\langle R,C\rangle}_{c\to c'}(t,\Delta)$ such that

$$\forall k,t,\ \sum_{c}\sum_{i} d_{c,i,k}(\mathrm{t}) \le\ Cap_k \tag{9}$$

$$\forall c,i,k,t \in lifetime_{c,i},\ \sum_{k} d_{c,i,k}(\mathrm{t}) =\ 1 \tag{10}$$

$$\begin{aligned}&\forall c,i,c',i',c\neq c',k,t\\ &CoRes_{c,i,c',i',k}(\mathrm{t}) \ge d_{c,i,k}(\mathrm{t}) + d_{c',i',k}(\mathrm{t}) \ -\ 1\\ &CoRes_{c,i,c',i',k}(\mathrm{t}) \le d_{c,i,k}(\mathrm{t})\\ &CoRes_{c,i,c',i',k}(\mathrm{t}) \le\ d_{c',i',k}(\mathrm{t})\end{aligned} \tag{11}$$

$$\begin{aligned}&\forall c,i,c',i',c\neq c',t,\\ &CoRes_{c,i,c',i'}(\mathrm{t}) \le \sum_{k} CoRes_{c,i,c',i',k}(\mathrm{t})\end{aligned}$$

$$CoRes_{c,i,c',i'}(\mathrm{t}) \ge CoRes_{c,i,c',i',k}(\mathrm{t}) \tag{12}$$

$$\forall c,c',c\neq c',t,\ InfoLeak^{\langle R,C\rangle}_{c\to c'}(t,\Delta) = \sum_{i}\sum_{i'}\sum_{t'\in[\mathrm{t}\text{-}\Delta,\mathrm{t}]} CoRes_{c,i,c',i'}(\mathrm{t}') \tag{13}$$

$$\begin{aligned}&\forall c,i,k,t,\ mv_{c,i,m}(\mathrm{t}) = d_{c,i,k}(\mathrm{t}) \oplus d_{c,i,k}(\mathrm{t}-1)\\ &mv_{c,i,m}(\mathrm{t}) \le\ d_{c,i,k}(\mathrm{t}) + d_{c,i,k}(\mathrm{t}-1)\\ &mv_{c,i,m}(\mathrm{t}) \ge d_{c,i,k}(\mathrm{t}) - d_{c,i,k}(\mathrm{t}-1)\\ &mv_{c,i,m}(\mathrm{t}) \le 2 - d_{c,i,k}(\mathrm{t}) - d_{c,i,k}(\mathrm{t}-1)\end{aligned}$$

$$\forall c,i,t,\ mv_{c,i}(\mathrm{t}) = 0.5\ \sum_{k} mv_{c,i,m}(\mathrm{t}) \tag{14}$$

$$\forall c,t,\ GlobalMigCost(\mathrm{t}) = \sum_{c,i} mv_{c,i}(\mathrm{t}') \tag{15}$$

$$\forall c,t,\ GlobalMigCost(\mathrm{t}) \le PercentBudget \sum_{c,i,k} d_{c,i,k}(\mathrm{t}) \tag{16}$$

Figure 10: $\langle R, C\rangle$: ILP formulation for *Nomad Placement Algorithm*

The ILP formulation is also applicable for three other deployment models; Equation 13 needs to be changed to reflect the correct *InfoLeakage* model.

Now that we have modeled the co-residency across client pairs, the only remaining factor is the migration cost. To this end, we introduce another binary indicator that indicates whether a VM has migrated from a previous epoch $t-1$. $mv_{c,i,m}(\mathrm{t})$ is a binary variable that indicates whether $VM_{c,i}$ was either in machine k in the previous epoch and is no longer in machine k or vice versa. This gives an indicator that tells whether a VM has moved away or moved into this machine k. Intuitively, this is like modeling a XOR operation (Equation 14). Summing over this variable for each VM instance givesi the total number of migrations from the previous epoch (Equation 15). Lastly, Equation 16 models the migration budget such that the total number of migrations should not exceed *PercentBudget* of the total workloads (i.e., total number of VMs that currently have place assignments).