Columbus: Fast, Reliable Co-residence Detection for Lambdas

Anonymous Author(s)

ABSTRACT

Cloud computing has seen explosive growth and popularity, especially "serverless" service modalities, such as AWS lambdas. These services are lighter-weight and provide more flexibility in scheduling and cost, which contributes to their popularity, however the security issues associated with serverless computing is not well understood. In this work, we construct a covert channel, a means of transmitting data outside of tranditional channels, that is practical for AWS lambdas. We extend the work in Wu et. al to utilize the memory-bus to develop a co-residence detector that is generic, reliable, and scalable. This technique performs dynamic neighbor discovery and is incredibly fast, executing in a matter of seconds. In addition to this protocol for lambda co-residence detection, we also perform and evaluate a series of measurements that exemplify the practicality of this approach in a cloud setting. Through this work, we show that efforts to secure co-residency detection in cloud providers is not yet complete.

CCS CONCEPTS

• **Security and privacy** → *Virtualization and security.*

KEYWORDS

cloud, cartography, serverless, coresidency, covert channels

ACM Reference Format:

Anonymous Author(s). 2020. Columbus: Fast, Reliable Co-residence Detection for Lambdas. In *The Web Conference 2021, April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Over the last decade, organizations have increasingly offloaded their data processing and storage needs to third-party "cloud" platforms. However, the economics of cloud platforms is predicated on high levels of statistical multiplexing and thus *co-tenancy* — the contemporaneous execution of computation from disparate customers on the same physical hardware – is the norm. The risks associated with this arrangement, both data leakage and interference, are well-appreciated and have generated both a vast research literature (starting with Ristenpart et al. [19]) as well a wide-array of technical isolation countermeasures employed by cloud platform providers. Most of this work has focused squarely on the risks of information channels between long-lived, heavy-weight virtual machines ("instances" in Amazon parlance) used to virtualize the traditional notion of dedicated network-connected servers.

However, over the last six years, most of the largest cloud providers have introduced a *new* "serverless" service modality that executes short-lived, lightweight computations on demand (e.g., Amazon's Lambda [14], Google's Cloud Functions [10] and Microsoft's Azure functions [5]). These services, by design, use lighter-weight tenant isolation mechanisms (so-called "micro-VMs" or containers) as well as a fixed system environment to provide low-latency startup and a reduced memory footprint. In return, serverless systems can support even higher levels of statistical multiplexing and thus can offer significant cost savings to customers whose needs are able to match this model (e.g., event-driven computations with embedded state). However, the security issues associated with serverless computing are far less well understood than their heavier weight brethren. While the transient and dynamic nature of service computing pose inherent challenges for attackers, their low-cost and light-weight isolation potentially present offer new points of purchase as well.

In our work, we explore these issues through the lens of a singular question: can a practical covert channel be constructed entirely from existing "serverless" cloud services?

Covert channels, as a general matter, provide a means of transmitting data that bypasses traditional monitoring or auditing – typically by encoding data into some resource access that is not normally deemed a communications medium but is externally visible. In virtualized environments, covert channels typically involve some shared resource (e.g. a cache) for which contention provides a means of signaling. In the serverless context, the threat model is that an adversary is able to launch, or inject code into, lambdas from inside a target organization and wishes to communicate information to parties outside the organization (i.e., to their own lambdas) without offering any clear evidence of such (e.g., opening network connections, etc.)

However, the serverless context presents a number of unique challenges for implementing such a channel. First, the scheduling and placement of lambdas is managed by the cloud service provider. Thus, there is no way to arrange that a sending lambda and a receiving lambda will execute on the same physical hardware, *let alone at the same time.* Second, given this reality, any serverless covert communications protocol must repeatedly launch lambdas in the hope that at least two sending and receiving lambdas are co-resident on the same hardware at the same time. The extent to which this is practicable, on existing cloud platforms and reasonable cost, is unknown. Third, it is not enough to simply *achieve* co-residency, but any lambdas lucky enough to be co-resident must be able to quickly determine this fact, and then use

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

The Web Conference 2021, April 19–23, 2021, Ljubljana, Slovenia

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the balance of their limited lifetimes to effect communications. Finally, since rendezvous in a serverless system is inherently statistical, any such protocol must anticipate the potential for interference (i.e., when multiple sending lambdas happen to be co-resident with a single receiving lambda).

In this paper we address each of these issues in turn and demonstrate the feasibility of covert communication entirely in the context of the Amazon serverless cloud platform. In particular, we make three key technical contributions:

- **Fast co-residence detection.** Leveraging the memory-bus contention work of Wu et. al [26], we develop and implement a lambda co-residence detector that is generic, reliable, scalable and, most importantly, fast, executing in a matter of seconds for thousands of concurrent lambdas.
- **Dynamic neighbor discovery.** We extend our co-residence detector to enumerate the set of co-resident neighbors, a requirement to avoid unwanted communication interference between neighbors.
- **Serverless density measurement.** We empirically establish measures of serverless function density – that is, for a given number of short-lifetime lambdas launched at a point in time, how many would be expected to become co-resident? We conduct these measurements across a range of Amazon data centers to establish that such an approach is practicable.

The remainder of this paper is organized as follows. In section 2, we present some background and motivate the need for co-residence detection for lambdas. Sections 3 and 4 present our co-residence detection mechanism in detail. We evaluate the mechanism in section 5 and perform a study of AWS lambda placement density using our technique to demonstrate the practicality of lambda covert channels in section 6. Finally, we conclude with a discussion on TODO ▶*what*◀ in section 7.

2 BACKGROUND & MOTIVATION

We begin with a brief background on relevant topics as we motivate the need for fast and scalable co-residence detection in enabling covert channels with lambdas.

2.1 Lambdas/Serverless Functions

We focus on serverless functions in this paper, as they are one of the fastest-growing cloud services and are less well-studied from a security standpoint. Offered as lambdas on AWS [14], and cloud functions on GCP [10] and Azure [5], these functions are of interest because they do not require the developer to provision, maintain, or administer servers. In addition to this low overhead, lambdas are much more cost-efficient than virtual machines (VMs) as they allow more efficient packing of functions on servers. Lambdas execute as much smaller units than containers and virtual machines, and are more ephemeral. For example, on AWS, the memory of lambdas is capped at 3 GB, with a maximum execution limit of 15 minutes. As with other cloud services, the user has no control over the physical location of the server(s) on which their lambdas are spawned.

While lambdas are limited in the computations they can execute (typically written in high-level languages like Python, C#, etc), they are conversely incredibly lightweight and can be initiated and deleted in a very short amount of time. Cloud providers run lambdas in dedicated containers with limited resources (e.g., Firecracker [1]), which are usually cached and re-used for future lambdas to mitigate cold-start latencies [2]. The ephemeral nature of serverless functions and their limited flexibility increases the difficulty in detecting co-residency, as we will discuss later. Previous studies that profiled lambdas [24] focused on the performance aspects like cold start latencies, function instance lifetime, CPU usage, etc. across various clouds, while the security aspects remain relatively understudied.

2.2 Covert Channels in the Cloud

In our attempt to shed light on the security aspects of lambdas, we focus particularly on the feasiblity of establishing a reliable covert channel in the cloud using lambdas. Covert channels enable a means of transmitting information beween entities that bypasses traditional monitoring or auditing. Typically, this is acheived by communicating data across unintended channels such as signalling bits by causing contention on shared hardware media on the server [16, 18, 25–27]. Past work has demonstrated covert channels in virtualized environments like the clouds using various harware such as caches [19, 27], memory bus [26], and even processor temperature [16].

Memory bus covert channel Of particular interest to this work is the covert channel based on memory bus hardware introduced by Wu et al. [26]. In x86 systems, atomic memory instructions designed to facilitate multi-processor synchronization are supported by cache coherence protocols as long as the operands stay within a cache line (generally the case as language compilers make sure that operands are aligned). However, if the operand is spread across two cache lines (referred to as ”exotic” memory operations), x86 hardware achieves atomicity by locking the memory bus to prevent any other memory access operations until the current operation finishes. This results in significantly higher latencies for such locking operations compared to traditional memory accesses. As a result, a few consecutive locking operations could cause contention on the memory bus that could be exploited for covert communication. Wu et al. acheived a data rate of 700 bps on the memory bus channel in an ideal laboratory setup.

Cloud environments, however, pose challenges to both establishing such covert channels and using them at their ideal performance. First, in the cloud setting, the sender and receiver entities are not known to be on the same server as this (coresidency) information is hidden from the tenants (even if the entities belong to the same tenant). Second, clouds present a virtualized platform that affects the communication on the covert channel as it introduces scheduling uncertainties between the participating entities and interference from other non-partiticpating workloads. While studies generally deal with the latter problem by employing traditional error correction techniques [26] to overcome errors, the former challenge of establishing the covert channel (by detecting coresidency) in the first place requires a co-residence detection mechansim which the attacker needs to employ to get the sender and receiver on the same machine.

2.3 Co-residence Detection

Past research has used various strategies to achieve co-residency for demonstrating various covert channel attacks in the cloud. Typically, the attacker launches a large number of cloud instances (VMs, Containers, etc.) following a certain launch strategy and employs a co-residence detection mechanism for detecting if any pair of those instances are running on the same machine. Traditionally, the co-residence detection was done based on software runtime information like public/internal IP addresses [19], files in *procfs* or other environment variables [24, 26] and other such logical side-channels [21, 29] that two instances running on a same server might share.

As virtualization platforms move towards stronger isolation between instances (e.g. AWS' Firecracker VM [1]), these logical covert-channels have become less effective or infeasible. Furthermore, some of these channels were only effective on container-based platforms that shared the underlying OS image and were thus less suitable for hypervisor-based platforms. This prompted a move towards using hardware-based covert channels, such as the ones discussed in the earlier section, which can bypass software isolation and are usually harder to fix. For example, Varadarajan et al. [22] uses the memory bus covert channel to detect co-residency for EC2 instances. However, as we will show, traditional approaches do not extend well to lambdas as they are too slow or are not scalable. Thus, we set forth to determine a fast and scalable co-residence detection for lambdas.

3 METHODOLOGY

Our goal is to determine a (co-operative) co-residence detection mechanism for lambdas. In other words, given a series of spawned lambdas in a certain region of a cloud service, how can we determine the lambdas that are co-located on the same machines? In this section, we discuss the details of such a mechanism, previous solutions to this problem, and the unique challenges we faced with lambda co-residence.

Given a set of cloud instances (VMs, Containers, Functions, etc) deployed to a public cloud, a co-residence detection mechanism would identify, for each pair of instances in the set, whether the pair was running on the same physical server at some point. Paraphrasing Varadarajan et al.[22], for any such mechanism to be useful across a wide range of launch strategies, we observe that it should have the following desirable properties:

- **Generic** The technique should be applicable across a wide range of server architectures and software runtimes. In practice, the technique would work across most third-party clouds and even among different platforms within a cloud.
- **Reliable** The technique should have a reasonable detection success with minimal false negatives (co-resident instances not detected) and even less false positives (non co-resident instances categorized as co-resident).
- **Scalable** A launch strategy may require hundreds or even thousands of instances to be deployed, and must scalable such that the technique will take less time to detect all co-resided pairs at a reasonable cost.

We add another property to that list which is relevant to lambdas:

- **Fast** The technique should be fast, preferably finishing in the order of seconds. As lambdas are ephemeral (with some clouds restricting their execution times to as low as a minute), the technique should leave ample time for other activities that make use of the resulting co-resident information.

Given these properties, we decide to investigate harware-based covert channels. Hardware-based covert-channels are more difficult to remove and obfuscate than software-based covert channels, and are also ubiquitous, given that hardware is more homogenous across computing platforms than software.

3.0.1 Memory bus channel. We chose the memory bus covert channel described in section 2.2 as it exploits a fundamental hardware vulnerability that is present across all generations of x86 hardware. Historically, multiple public cloud services have been vulnerable to this channel [21, 32], and we found that they are still vulnerable today. To demonstrate the presence of the vulnerability, we measure the latency of atomic operations on a 4B memory region as we slide the region from one cacheline into another across the cacheline boundary. We perform this experiment on three major clouds (AWS, Google and Microsoft Azure) and show the latencies observed in Figure 1. From the figure, we can see that all three clouds still exhibit significant difference in latencies for the "exotic" memory locking operations (where the memory region falls across cacheline boundary) when compared to regular memory accesses, demonstrating the presence of this covert channel on all of them. Moreover, we were able to run these experiments on serverless function instances. Since lambdas have runtimes that are generally restricted to high-level languages (that prevent the pointer arithmetic required to perform these exotic operations), we used the unsafe environments on these clouds — C++ on AWS, Unsafe Go on GCP, Unsafe C# On Azure. This shows the applicability of using the covert channel across different kinds of cloud instances as well.

3.0.2 Previous approaches. Previous works that used the memory bus for co-residence detection divide the deployed instances into sender and receiver roles, and attempt to colocate the sender role with a receiver role. The sender instances continually lock the memory bus (locking process) for a certain duration (~10 seconds) while the receiver samples the memory for any spike in access latencies (probing process). If all the deployed instances try the detection i.e., locking and probing at once, (some of) the receivers may see locking effects, but there would be no way of knowing which or how many senders co-resided with a particular receiver and caused the locking. This provides no information about the number of physical servers that ran these instances or the amount of co-location. The only information we can deduce is that receivers were probably co-located with just a single sender.

An alternative method is to try pair-wise detection where only one sender instance locks and one receiver instance probes at a time revealing co-residence of this pair, and repeating this serially for each pair. However, this technique is too slow and scales quadratically with the number of instances e.g., a hundred instances take more than 10 hours assuming 10 secs for each pair. Varadarajan et al.[21] speeds up this process significantly by performing detection for mutually-exclusive subsets in parallel, allowing for

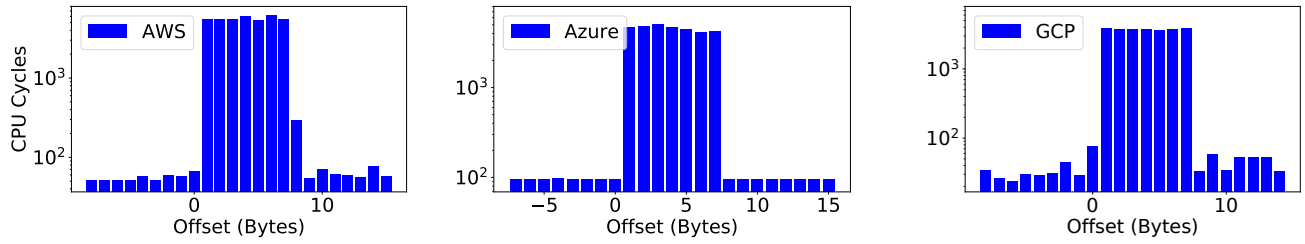

Figure 1: From left to right, the plots show the latencies of atomic memory operations performed on an 8B memory region as we slide it from one cache line across the boundary into another, on AWS, Azure and Google (GCP) clouds respectively. The latencies are orders of magnitude higher when the 8B region falls across the two cache lines (offsets 0-7B) demonstrating the presence of the memory bus covert channel on all these clouds.

false-positives and later eliminating the false-positives sequentially Ariana ►*might want to elaborate on this; on its own may be a bit confusing*◄. This would only scale linearly in the best case, which is still expensive; with a thousand instances, for example, the whole detection process takes well over 2 hours to finish, which is infeasible for lambdas that are, by nature, ephemeral. Thus, one challenge in this work is creating a faster neighbor detection algorithm.

3.0.3 The Path to Scalability. One method to quicken the co-location process is by decreasing the time a single sender-receiver pair requires to determine co-residence i.e., improving upon probing time and accuracy of the receiver. However, this method only affects total time by a constant factor. To improve scalability, we need to be able to run the detection for different sender-receiver pairs in parallel without sacrificing the certainty of information we get when they are run serially. For example, when two pairs observe co-residence, we must be certain that each receiver experienced co-residence because of its own paired sender instance, which is not possible if co-residence is ascertained based a simple yes/no signal from the sender instances.

Previous work utilized the memory bus channel to exchange more complex information like keys [26]. At first sight, it appears that the memory bus can be used to exchange information, such as unique IDs, between the co-resided instances, which would allow us to ascertain receiver/sender pairs. However, the previous work assumes that there is only one sender and one receiver that know exactly how and when to communicate. As we will see in the next section, this model is not sustainable when there exist many parties that have no knowledge of each other but try to communicate on the same channel.

To solve some of the challenges mentioned previously, we propose a protocol in which we use the memory bus covert channel to exchange information between the instances that have access to the channel. Using the protocol, co-resided instances can reliably exchange their unique IDs with each other to discover their neighbors. The protocol takes time on the order of number of instances involved, which is limited by the maximum number of co-located instances on a single server (tens), a number that is orders of magnitude less than total number of instances deployed to the cloud (hundreds to thousands). Removing this dependency on total number of instances deployed lets us scale our co-residence detection significantly, as we will see the the next section.

Algorithm 1 Writing 1-bit from the sender

```
now ← time.now()
end ← now + sampling_duration
address ← cache_line_boundary − 2
while now < end do
    __ATOMIC_FETCH_ADD(address)
    now ← time.now()
end while
```

4 NEIGHBOR DISCOVERY PROTOCOL

Co-residence detection for lambdas must be quick, as lambdas are ephemeral by nature. As noted earlier, co-residence detection is faster when co-residing instances on each server communicate among themselves and discover each other. Assuming that the instances have unique IDs, co-resided instances can exchange integer IDs with each other using the memory bus channel as a transmission medium, which allows us to scale co-residency detection. In this section, we present a communication protocol that the co-resided instances can use to achieve communication in a fast and reliable way. We first discuss the challenges we faced in making the channel reliable before examining the protocol itself.

4.1 Reliable Transmission

First, we must determine how to reliably transmit information between the sender and receiver. Senders and receivers can accurately communicate 0-bits and 1-bits by causing contention on the memory bus. Consider the simple scenario where there is one sender and one receiver instance on a machine, and the sender has a set of bits that it needs to communicate with the receiver via the memory bus covert channel. To communicate a 1-bit, the sender instance

causes contention on the memory bus by locking it using the special memory locking operations (discussed in section **??**). The receiver would then sample the memory bus for contention, inferring whether the communication is a 1-bit (when contention is observed) or a 0-bit (when contention is not observed). Pseudo-code for the sender instance is shown in Algorithm 1.

4.1.1 Sensing contention. Next, we determine how best for the receivers to sense contention on the memory bus. There are two ways to acheive this. When the memory bus is locked, any non-cached memory accesses will queue and therefore see higher latencies. The receiver can then continually make un-cached memory accesses (referred to as the *memory probing* receiver in previous literature [22]) and observe a spike in their latencies to detect contention. On the other hand, the receiver can also detect memory bus contention by using the same memory locking operations as the sender (referred to as *memory locking* receiver) to probe the memory bus. Since only one processor core can lock the memory bus at a given time, any other concurrent locking operation will see higher latency.

Of these two methods, we decide to use the memory locking receiver for our experiments. Previous studies [22, 26] have established that both memory probing and memory locking receivers experience significant latency overhead during memory bus contention, making them both viable avenues for sensing the covert-channel. Memory probing involves regular (un-cached) memory accesses, which is universal, unlike the locking operations which are rarely used, if at all, by standard applications. This makes memory probing the only viable option for **non-cooperative** co-residence detection, where receivers are not under the senders's control and cannot be assumed to perform locking operations. Furthermore, memory probing can be done on multiple receivers constantly without affecting each other (due to the high memory bandwidth), which prevents noise in measurements. This is an important attribute, as memory locking receivers must contend with this noise. However, bypassing multi-levels of caches in today's servers to perform memory accesses with reliable consistency is a challenging task. Even with a reliable cache-bypassing technique, the variety of cache architectures and sizes that we encounter on different clouds would make tuning the technique to suit these architectures an arduous task while reducing the applicability of our overall co-residence detection mechanism.

4.1.2 Sampling frequency. Another challenge for our protocol is determining an adequate sampling frequency. Ideally, a memory locking receiver would loop locking operations and determine contention in real-time by identifying a decrease in the moving average of the number of operations. Note that, in this case, there is essentially no difference between the sender and receiver (i.e., both continually issue locking operations) except that the receiver is taking measurements. This is adequate when there is a single sender and receiver [22], but when there are multiple receivers, the mere act of sensing the channel by one receiver causes contention and other receivers cannot differentiate between a silent (0-bit) and a locking (1-bit) sender. To avoid this, we space the sampling of memory bus such that no two receivers would sample the bus at the same time, with high probability. We achieve this by using large intervals between successive samples and a poisson-sampling to prevent time-locking of receivers. We determined that a millisecond poisson gap between samples is reasonable to minimize noise due to collisions in receiver sampling 1, assuming ten co-resided receivers and a few microsecond sampling time.

Algorithm 2 Reading a bit in the receiver

```
1:  now ← time.now()
2:  end ← now + sampling_duration
3:  sampling_rate ← num_samples/sampling_duration
4:  address ← cache_line_boundary − 2
5:  samples ← {}
6:  while now < end do
7:      before ← RDTSC()
8:      __ATOMIC_FETCH_ADD(address)
9:      after ← RDTSC()
10:     samples ← samples ∪ {(after − before)}
11:     wait until NEXT_POISSON(sampling_rate)
12:     now ← time.now()
13: end while
14: ks_val ← KOLMOGOROV_SMIRINOV(samples, baseline)
15: return ks_val < ksvalue_threshold
```

4.1.3 Sample Size. In addition to adequate sampling frequency, we must also determine sample size. A receiver can confirm contention with high confidence with only a few samples, assuming that the sender is actively causing contention on the memory bus and the receiver is constantly sampling the memory bus throughout the sampling duration. However, in practice, the time-sharing of processors produces difficulties. The sender is not continually causing contention, and neither is the receiver sensing it, as they are context-switched by the scheduler to run other processes. Assuming that the sender and receiver are running on different cores, the amount of time they are actively communicating depends on the proportion of time they are allocated on each core and how they are scheduled.

To illustrate such behavior, we run a sender-receiver pair using lambdas[14] of various sizes on AWS, and compare the distribution of latencies seen by the receiver during the contention in each case. Figure 2 shows that the much smaller 128 MB lambdas (which probably share a CPU core and are thus context-switched) exhibit less active communication than the bigger 3 GB lambdas (which may run on dedicated cores). This means that smaller instances that tend to share processor cores with many other instances may need to pause for more time and collect more samples to make up for lost communication due to scheduling.

4.1.4 Overcoming noise. Along with context switching and sensing noise, there are other imperfections in the measurement apparatus that may cause noise. For example, we use the difference in readings from the timestamp counter of the processor (RDTSC) before and after the locking operation to measure the latency of the operation in cycles. If the receiver process is context-switched in between the timer readings (e.g., at line 8 in Algorithm 2), the latency measured from their difference will be orders of magnitude higher as it includes the waiting time of the receiver process in the scheduler queue - which we believe is what contributes to the long

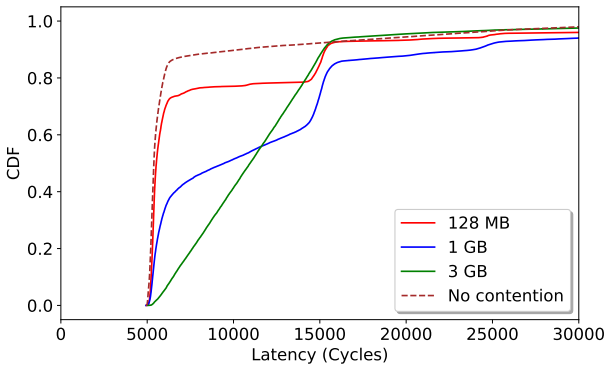

Figure 2: Shows CDF of latencies observed by 128 MB, 1 GB and 3 GB Lambdas during contention. The 128 MB lambda pair sees less contention due to more context switching, whereas the 1 GB and 3 GB lambdas see progressively more contention compared to baseline which we attribute to their relative stability on the underlying physical cores.

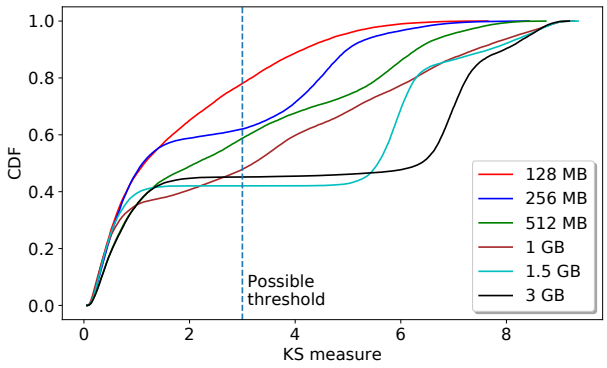

Figure 3: Shows CDF of KS values observed for various lambda sizes. A bimodal distribution with longer step lets us pick a KS-threshold that enables our technique to differentiate between 0-bit and 1-bit with high confidence.

tail in Figure 2. To overcome missed samples and noise, we record hundreds of samples and compare it to the baseline distribution of latencies sampled without contention. We then need to compare and differentiate the observed sample of latencies from the baseline to establish contention. To do this, we use a variant of the two-sample Kolomogorov-Smirinov (KS) test that takes the maximum of absolute difference between empirical CDFs of the samples as measure of comparison (In our variant, we take the *mean* of absolute difference instead of the maximum). Using this measure, we can categorize a KS-value above a certain threshold as a 1-bit (contention) and a value below the threshold as 0-bit (baseline).

To determine the KS-threshold, we deploy a large number of lambdas across AWS regions. Some of these lambdas cause contention (aka senders) while others observe contention by collecting samples of latencies (aka receivers). Each of the samples may or may not have observed contention depending on whether the receiver was co-resided with a sender lambda (an unknown at this point). We then calculate the KS-value for each sample against the baseline and plot a CDF of these values for lambdas of different sizes in Figure 3. Ideally, we expect a bi-modal distribution (stepped CDF) with the upper and lower peaks corresponding to samples that have and have not seen contention, and a big gap between the two (long step). Fortunately, we observe this differentiation with larger lambda sizes (which allows us to choose a clear threshold), but we do not observe a clear differentiation with smaller lambdas, where scheduling instability causes lossy communication (discussed in 4.1.3). This trend also reflects in the reliability of our technique across various lambda sizes, as we will show in our evaluation. Based on the plot, we picked a KS-threshold at 3.0 which seems to be constant across AWS regions, suggesting that this threshold is a platform constant.

We present the pseudo-code of a receiver lambda in Algorithm 2, which includes all the challenges and subsequent solutions discussed thus far.

4.1.5 Clock synchronization. Since communicating each bit of information takes time (i.e., receiver sampling duration), our algorithm requires synchronizing sender and receiver at the start of each bit. In traditional analog channels, this is achieved either using a separate clock signal or a self-clocking signal encoding. For example, prior work [**?**] uses differential Manchester encoding for clock synchronization for the memory bus covert channel. Using self-clocking encodings becomes more challenging when there are multiple senders and receivers. In this work, we use the system clock for synchronizing communication. All the instances involved in the communication are executing on the same physical server and share the server's clock. On AWS, for example, we observe that the system clock on lambdas is precise up to nanoseconds. While we have no evidence of it, we assume that the clocks between different lambdas drift only in the order of microseconds. This gives us enough space for synchronization mismatch as we take measurements only on millisecond scales due to noise constraints. We believe this to be a reasonable assumption as we did not observe any synchronization-related noise in our results.

4.1.6 Handling Collisions. In the preceeding section, we discussed using a communication channel with synchronized time slots. In each time slot, an instance can reliably send (broadcast) or receive (listen) a bit by causing or sensing for contention. Given that there are multiple instances that may want to broadcast information on the channel, we next must determine which instance broadcasts first, to avoid collisions. Traditional channels like Ethernet or Wireless detect and avoid collisions by employing a random exponential backoff mechanism. Such a mechanism will be challenging to implement in case of our channel for two reasons. First, lambda instances do not have the capability of sensing the channel while sending a bit, which is required for detecting collisions; instances can either cause contention or sense it, but not both. Note that senders do experience a higher latency for locking operations when other senders are simultaneously causing contention. However, reliably judging this higher latency requires each sender to already have calculated a baseline of latencies without collisions, which

Algorithm 3 ID exchange protocol TODO ▶*Improve pseudo-code*◀

```
1: sync_point ← Start time for all instances
2: ID ← Instance ID
3: N ← Number of bits in ID
4: advertising ← TRUE
5: instances ← {}
6: WAIT_TILL(sync_point)
7: while id_read do
8:    slots ← 0
9:    id_read ← 0
10:   participating ← advertising
11:   while slots < N do
12:      bit ← slots^th most significant bit of ID
13:      if participating and bit then
14:         WRITE_BIT() (Alg. 1)
15:         bit_read ← 1
16:      else
17:         bit_read ← READ_BIT() (Alg. 2)
18:         if bit_read then
19:            participating ← FALSE
20:         end if
21:      end if
22:      id_read ← 2 * id_read + bit_read
23:      slots ← slots + 1
24:   end while
25:   if id_read = ID then
26:      advertising ← FALSE
27:   end if
28:   instances ← instances ∪ {id_read}
29: end while
30: return instances
```

takes us back to the original problem of avoiding collisions. Second, even implementing a random or exponential backoff mechanism will introduce significant overhead before any meaningful communication occurs. This overhead will also increase as the number of instances involved increases. Since each time slot could take up to 1 second, this additional overhead can make the whole communication very slow. We address this by designing a specialized communication protocol that allows for collisions but pays a price in terms of expressiveness, as we will see in the next section.

4.2 Protocol

Ariana ▶*is it worth "recapping" and listing out all the challenges we faced, to really drive the point home to the reader?*◀

When considering a communication channel for lambda co-residence detection, we note that the channel need not be general and expressive as lambdas only need to communicate their IDs with one another. Thus, we assume that each instance involved has a unique fixed-length (say n) bit-string corresponding to its ID that must be communicated. As such, we propose a communication protocol that exchanges these bit-strings while allowing for collisions. We divide the running time of the protocol into phases, with each phase executing for an interval of n bit-slots. Each phase has a set of participating instances, which in the first phase would be all of

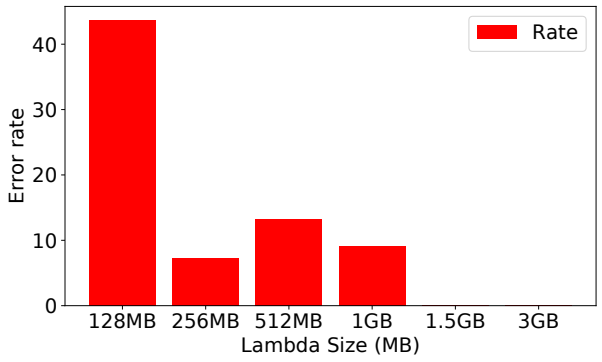

Figure 4: Shows the error rate (as a fraction of 1000 lambdas deployed) for different lambda sizes in AWS Middle-East region.

the co-resided instances. In each bit-slot k of n slots in a phase, every participating instance broadcasts a bit if the k^{th} bit of its bit-string (ID) is 1, otherwise it listens for a 0 or 1. If an instance senses a 1 while listening, it stops participating, and listens for the rest of the phase. Thus, only the instances with the highest ID among the initial set of participating lambdas continues broadcasting until the end of the protocol, effectively advertising its full ID to the rest of the (now listening) lambdas. In the next cycle, the lambda with the previously highest ID now only listens, allowing the next highest instance to advertise its ID, and so on. Since the IDs are unique, there will always be only one instance that broadcasts in every phase. The protocol ends after x phases (where x is number of co-resided instances), when none of the instances broadcast for n consecutive bit-slots. A pseudo-code of the protocol is provided in Algorithm 3. Note that the protocol itself is channel-agnostic and can be extended for other (future) covert channels with similar channel properties.

4.2.1 Time Complexity. Assuming N total deployed instances to the cloud, the bit-string needs to be $\log_2 N$ bits to uniquely identify each instance. If a maximum K of those instances are launched on the same server, the protocol executes for K phases of $\log_2 N$ bit-slots each, taking $(K+1)*\log_2 N$ bit-slots for the whole operation. For example, assuming 10,000 deployed lambdas and a maximum of 10 co-resided instances on each server, the entire co-residence detection requires around four minutes to fully execute (with 1-second time slots). In fact, it is not necessary to run the protocol for all K phases. After the first phase, all the co-resided instances would know one of their neighbors (as each phase reveals the ID of the biggest participating instance to others). If we use IDs that are globally unique, all (and only) the co-resided instances see the same ID. The instances can then exchange these IDs offline (e.g., through the network) to infer the rest of their neighbors. This simplification removes the dependency on number of co-resided instances (K) and decreases the complexity to $O(\log_2 N)$, allowing the entire protocol to finish within a minute instead of four.

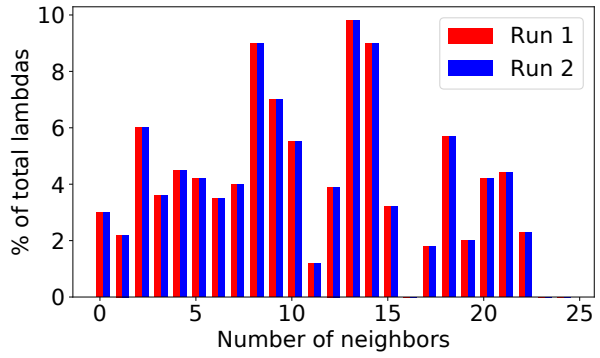

Figure 5: Shows the fraction of lambdas by the number of neighbors they saw for two independent runs that use same set of underlying AWS containers. The perfect correlation shows that both runs depict the colocation status of those containers regardless of the lambdas that ran on them, providing an evidence for the correctness of our approach.

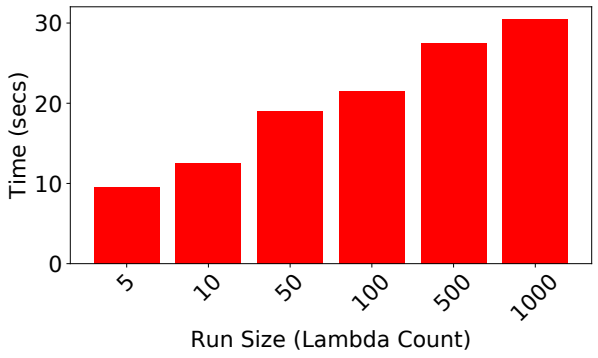

Figure 6: Shows the average execution time of the lambdas for co-location runs with varying number of lambdas. The execution time increases logarithmically with the number of lambdas demonstrating the scalability of co-residence detection with our technique.

5 EVALUATION

In this section, we evaluate the effectiveness of our co-residence detection technique with respect to reliability and scalability, the desirable detection properties mentioned in section 3. We run all of our experiments with AWS lambdas [3]. Though we decide to focus on one of the cloud providers, we have previously shown in section 3 that this covert channel exists on the other clouds, and thus these experiements can be replicated on their serverless functions as well. We use C++ runtime in AWS lambdas as it allows pointer arthmetic that is required to access the covert channel.

5.1 Setup

For each experiment, we deploy a series of instances from an AWS lambda account. Once deployed, each instance participates in the first phase of the protocol as noted in section 4.2.1, thereby learning the largest ID of their neighbors. As bit-flip errors are possible, we repeat the same phase for two more (independent) "rounds" and take the majority result to record the ID seen by this instance. If all three rounds result in different IDs, we classify this instance as erroneous and report it in the error rate. We group all the instances that saw the same ID as successful and neighbors. We repeat the experiments for different lambda sizes and in various cloud regions.

5.2 Reliability

We consider the results of the technique reliable when 1) most of the deployed instances successfully see the same result in majority of the independent rounds (indicating lesser bit-flip errors) and 2) the resulting co-located groups we see match the ground truth. For 1, we ran an experiment with 1000 AWS lambdas and compared the error rate across different lambda sizes (the error rate indicates the fraction of these 1000 lambdas that did not have a majority result). From Figure 4, we can see that smaller lambdas see lot more errors. This is expected because, as discussed in section 4.1.4, these lambdas experience lossy communication making it harder for our technique to sense contention. The lambdas above 1.5 GB, though, see a 100% success rate.

Correctness To determine correctness, we require ground truth on which instances are co-located with one another. While such information is not available, we are able to ascertain correctness of our approach by utilizing an AWS caching mechanism. On AWS, each lambdas runs in a dedicated container (sandbox). After execution, AWS caches these containers to reuse them[2] for repeat lambdas and mitigate "cold start" latencies. For C++ lambdas, we found that the data structures declared in the global namespace are tied to containers and are not cleared on each lambda invocation, so we can use a global array to record all the lambdas that were ever run in a particular container. This means, for a given lambda, we can precisely tell all the lambdas that previously ran in the same container (aka predecessors). Using this, we are able to validate that identical experiments repeated within minutes of one another will use the same set of underlying containers for running the deployed lambdas. Since lambda co-location is essentially co-location of their containers, and given that containers persist across experiments that are executed within minutes of one another, lambda co-location results must agree with the co-location of their underlying containers for true correctness.

To demonstrate the correctness of our technique using this insight, we run an experiment with 1000 1.5GB cold-started lambdas (ID'ed 1 to 1000) in one of densest AWS regions (AWS MiddleEast), which resulted in many co-located groups. We repeat the experiment within a few seconds, thereby ensuring that all 1000 lambdas are warm-started on the second trial (i.e., they use the same set of containers from the previous experiment). For each co-located group of lambdas in the latter experiment, we observed that their predecessor lambdas (that used the same set of containers) in the former experiment formed a co-located group as well. That is, while the lambdas to the underlying container mapping is different across both experiments, the results of the experiments agree perfectly on the continer colocation. Figure 5 shows that

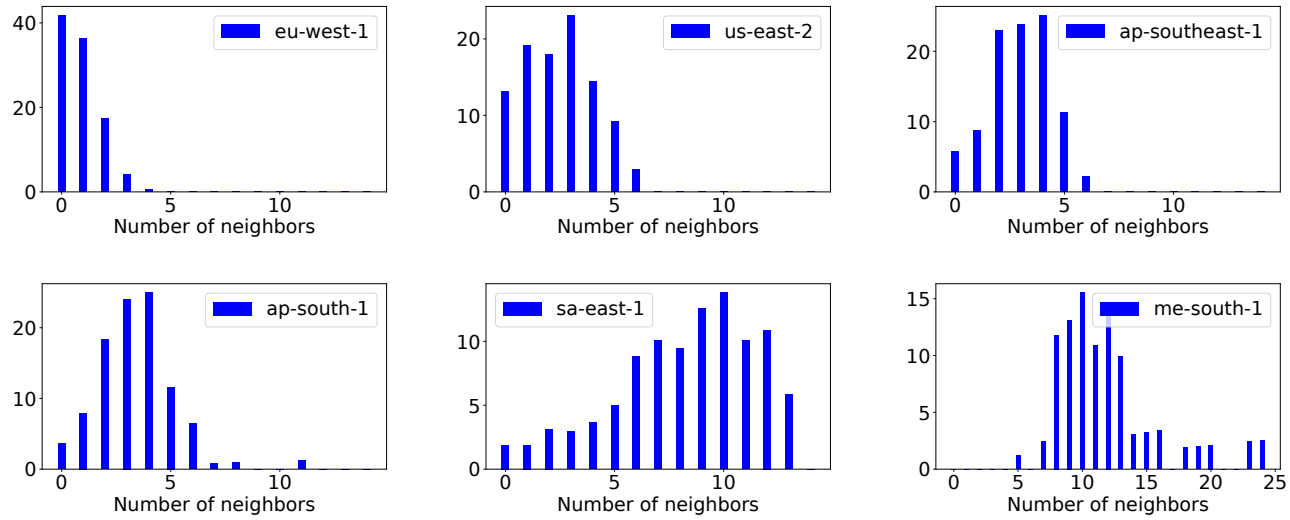

Figure 7: Shows co-residence results from a 1000-lambda run in different AWS regions. Each bar shows the fraction of those 1000 lambdas (in %) that saw a certain number of neighbors. The total amount and density of co-residence vary widely across regions, perhaps based on the size of those regions and the lambda activity within them.

both experiments saw the same number of co-residing groups of different sizes. This proves the correctness of the results of our mechansim.

5.3 Scalability

One of the key properties of this technique is its execution speed. Since communicating each binary bit of the ID takes one second, we are able to scale the technique logarithmically with the number of lambdas involved. Figure 6 shows this result with experiments involving different number of lambdas. For example, in an experiment with 1000 lambdas, each lambda can find its neighbors within a minute of its invocation, leaving ample time for the attacker to then establish the covert channel and use it to send information. The logarithmic scale of our method also indicates that the cost per lambda scales logarithmically, making neighbor detection cost-effective.

6 MEASUREMENT STUDY

In this section, we present a variety of measurements on serverless function density on AWS using our co-residence detector, and the factors that may affect this density. As we discussed earlier, the key challenge in enabling traditional covert channels on the cloud is getting the sender and receiver on the same machine. So we attempt to answer the following question: Assuming that the attacker launches a number of the (sender and receiver) lambdas at a point in time in the expectation that a set of them would end up co-resident, what is the expected number of such co-resident pairs that she might see? We deploy a large number of lambdas on various AWS regions and observe the co-residence density – that is, the average number of lambdas that end up co-resident on each server. The higher the co-residence density, the easier it is for the attacker to ultimately establish covert channels with lambdas. Unless specified otherwise, all the experiments are performed with 1.5 GB lambdas and executed successfully with **zero error** in co-residence detection.

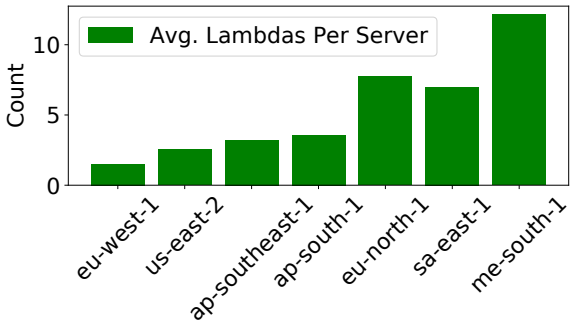

Figure 8: Shows the average number of lambdas per server i.e., the co-residence density seen in various AWS regions for the runs shown in Figure 7. The ample co-residence across regions demonstrates the practicality of establishing covert channels with lambdas in these regions.

6.1 Across AWS regions

We ran our co-residence detector with 1000 1.5 GB Lambdas in various AWS regions. Figure 7 comprises of multiple plots depicting the co-resided groups per region, with each bar indicating the fraction of lambdas that detected a certain number of neighbors (i.e., that belong to a co-resided group of a certain size). Plots that

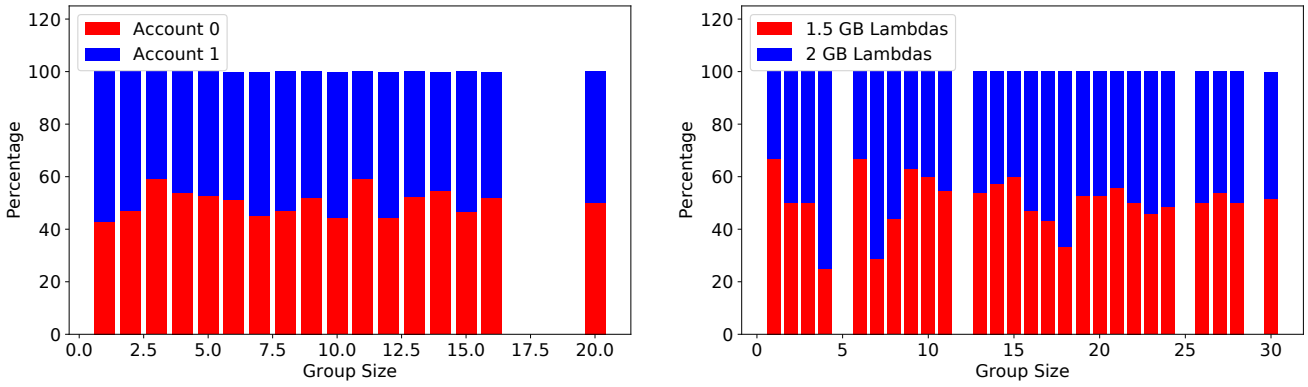

Figure 9: The left plot shows the breakdown of co-resided groups (of varying sizes) of lambdas by two different accounts in an experiment of 1000 lambdas where 500 lambdas are launched from each account. The uniformity of the split shows that lambda scheduler might be invariant to the account the lambdas are launched from. Similar results are shown for different lambda sizes in the right plot.

skew to the right indicate a higher co-residence density when compared to the plots skewed to the left (also illustrated in Figure 8). We note that, in most regions, almost all lambdas recognize at least one neighbor (indicated by smaller or non-existent first bar in each plot). We hypothesize that the co-residence density is (inversely) dependent on the total number of servers and the lambda activity in the region, both of which can be assumed to be lower in newer AWS regions, hence the higher co-residence density in those regions as we can see in Figure 8. The ample co-residence in general across all the regions shows that lambdas provide a fertile ground for covert channel attacks.

6.2 Other factors

We examine how the co-residence is affected by various launch strategies that the attacker may use like deploying lambdas from multiple AWS accounts and different lambda sizes. For example, does it matter if the attacker uses one account for sender lambdas and another for the receivers (which is normally the case with covert channels)? Or one lambda size for senders and another for receivers? Do they still get colocated on the same server? To answer these questions, we run an experiment with 1000 lambdas of which we launch 500 lambdas from one account (senders) and 500 from other deployed in a random order. The co-residence observed was comparable to the case where all the lambdas were launched from one account. In the left subfigure of Figure 9, we show the breakdown of co-resided group of lambdas of each size among the two accounts. We can see that among the co-resided groups of all sizes, roughly half of lambdas came from either account. This show that lambda scheduler is agnostic to the accounts the lambdas were launched from. We see similar results for different lambda sizes, as shown in the right subfigure of igure 9

From our limited experiments, we also observe that co-residence density in a region barely changes during course of the day or the week (data not shown in any figure). This gives the attacker the freedom to attempt the attack at any time and expect similar results.

7 DISCUSSION

Other use cases Our main motivation behind proposing the co-residence detector for lambdas is demonstrating the feasibility of covert channel attacks. However, there are other scenarios where such tool can be (ab)used, of which we provide some examples.

- While our co-residence detector does not directly help attackers locate their victims in the cloud, it can aid them in performing devastating DDOS attacks once by concentrating a number of attack instances on the victim machine. Also, the attacker could try to gain a wider surface area for targeted attacks in a cost-effective way by turning on/off her co-resided instances as necessary.
- Previous studies on performance aspects of lambdas (like performance isolation) [24] generally need a way to co-locate some lambdas. As software-level logical channels begin to disappear, our tool might provide a reliable alternative?
- Burst parallel frameworks like gg [8] that orchestrate lambdas can use our co-residence detector as a locality indicator to take advantage of server locality.

Mitigation In previous section, we showed that our co-residence detector makes the covert channels practical with lambdas, so it is important that clouds address this issue. One way to disable our co-residence detector is to fix the underlying memory bus channel that it employs. However, this only works for newer generation of servers and is not practical for existing infrastructure. An easier solution, one that is only practical with lambdas, is to disable the lambda support for low-level languages (or unsafe versions of high-level languages) by the cloud providers. This will prevent pointer arithmetic that is required to activate this channel. If that is not an option, clouds may look at more expensive solutions like BusMonitor [20] that isolate memory bus usage for different tenants. We leave such exploration to future work.

8 RELATED WORK

Cloud Attacks Co-residency is possible because of covert channels, so we begin our related work with an investigation into cloud attacks. Initial papers in co-residency detection utilized host information and network addresses arising due to imperfect virtualization [19]. However, these channels are now obsolete, as cloud provides have strengthened virtualization and introduced Virtual Private Clouds [4]. Later work used cache-based channels in various levels of the cache [12, 15, 28, 35] and hardware based channels like thermal covert channels [17], RNG module [7] and memory bus [26] have also been explored in the recent past. Moreover, studies have found that VM performance can be significantly degraded using memory DDoS attacks [31], while containers are susceptible to power attacks from adjacent containers [9].

Our work focuses on using the memory bus as a covert channel for determining cooperative co-residency. Covert channels using memory bus were first introduced by Wu et. al [26], and subsequently has been used for co-residency detection on VMs and containers [21**?**] Wu et. al [26] introduced a new technique to lock the memory bus by using atomic memory operations on addresses that fall on multiple cache lines, a technique we rely on in our own work.

Co-residency One of the first pieces of literature in detecting VM co-residency was introduced by Ristenpart et al., who demonstrated that VM co-residency detection was possible and that these techniques could be used to gather information about the victim machine (such as keystrokes and network usage) [19]. This initial work was further expanded in subsequent years to examine co-residency using memory bus locking [30] and active traffic analysis [6], as well as determining placement vulnerabilities in multi-tenant Public Platform-as-a-Service systems [22, 33]. Finally, Zhang et al. demonstrated a technique to detect VM co-residency detection via side-channel analyses [34]. Our work expands on these previous works by investigating co-residency for lambdas.

Lambdas While lambdas are a much newer technology than VMs, there still exists lterature on the subject. Recent studies examined cost comparisons of running web applications in the cloud on lambdas versus other architectures [23], and also examined the lambdas have been studied in the context of cost-effectivness of batching and data processing with lambdas [13]. Further research has shown how lambdas perform with scalability and hardware isolation, indicating some flaws in the lambda architecture [24]. From a security perspective, Izhikevich et. al examined lambda co-residency using RNG and memory bus techniques (similar to techniques utilized in VM co-residency) [11]. However, our work differs from this study in that our technique informs the user of which lambdas are on the same machine, not only that the lambdas experience co-residency.

9 ETHICAL CONSIDERATIONS

As with any large scale measurement project, there are ethical considerations to take into account. First, there are security and privacy concerns of using this technique to uncover other consumer's lambdas. However, since we focus on co-operative co-residence detection, we only determine co-residence for the lambdas we launched, and do not gain insight into other consumer's lambdas. Second, there is concern that our experiments may cause performance issues with other lambdas, as we may block their access to the memory bus. We believe this concern is small, for a number of reasons. Memory accesses are infrequent due to the multiple levels of caches; we would only be affecting a small number of operations. Memory accesses and locking operations are FIFO, which prevents starvation of any one of the lambdas sharing a machine. Moreover, lambdas are generally not recommended for latency-sensitive workloads, due to their cold-start latencies. Thus, the small amount of lambdas that we might affect should not, in practice, be affected in their longterm computational goals.

10 CONCLUSION & FUTURE WORK

TODO ▶*Copied abstract*◀ Cloud computing has seen explosive growth in the past decade. This is made possible by efficient sharing of infrastructure among tenants, which unfortunately also raises security challenges like preventing side-channel attacks. Providers, like AWS and Azure, have traditionally relied on hiding the co-residency information to prevent targeted attacks in their clouds. But recent works have repeatedly found co-residence detection techniques that break this encapsulation, prompting the providers to address them and harden isolation on their platforms. In this work, we find yet another such technique based on a memory bus covert channel that is more pervasive, reliable and harder to fix. We show that we can use this technique to reliably perform co-operative co-residence detection for thousands of AWS lambdas within a few seconds, which opens a way for attackers to perform DDoS attacks or learn cloud's internal mechanisms. We present this technique in detail, evaluate it and use it to perform a small study on lambda activity across a few AWS regions. Through this work, we hope to motivate the need to address this covert channel in the cloud.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] aws 2018. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[3] aws 2019. Amazon EC2. https://aws.amazon.com/ec2/.

[4] aws 2019. Amazon VPC. https://aws.amazon.com/vpc/.

[5] azure 2019. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[6] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. 2012. Detecting co-residency with active traffic analysis techniques. https://doi.org/10.1145/2381913.2381915

[7] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 843–857. https://doi.org/10.1145/2976749.2978374

[8] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[9] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2017. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *Dependable Systems and Networks (DSN), 2017 47th*

Annual IEEE/IFIP International Conference on. IEEE, IEEE, 237–248.

[10] gcp 2019. Google Cloud Functions. https://cloud.google.com/functions/.

[11] Elizabeth Izhikevich. 2018. *Building and Breaking Burst-Parallel Systems*. Master's thesis. University of California, San Diego.

[12] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. 1–6. https://doi.org/10.1145/2897937.2897962

[13] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. 2015. Lambda architecture for cost-effective batch and speed big data processing. *2015 IEEE International Conference on Big Data (Big Data)* (2015), 2785–2792.

[14] Lambda 2019. AWS Lambda. https://aws.amazon.com/lambda/.

[15] Fangfei Liu, Yuval Yarom, Qian ge, Gernot Heiser, and Ruby Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical, Vol. 2015. 605–622. https://doi.org/10.1109/SP.2015.43

[16] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-core Platforms. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 865–880. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti

[17] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-core Platforms. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, Berkeley, CA, USA, 865–880. http://dl.acm.org/citation.cfm?id=2831143.2831198

[18] Clementine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. https://doi.org/10.14722/ndss.2017.23294

[19] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. ACM, New York, NY, USA, 199–212. https://doi.org/10.1145/1653662.1653687

[20] Brendan Saltaformaggio, D. Xu, and X. Zhang. 2013. BusMonitor : A Hypervisor-Based Solution for Memory Bus Covert Channels.

[21] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 913–928. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan

[22] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2015. A Placement Vulnerability Study in Multi-tenant Public Clouds. *CoRR* abs/1507.03114. arXiv:1507.03114 http://arxiv.org/abs/1507.03114

[23] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano Merino, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. 2016. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. https://doi.org/10.1109/CCGrid.2016.37

[24] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[25] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, USA, 473–482. https://doi.org/10.1109/ACSAC.2006.20

[26] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 159–173. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu

[27] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (Chicago, Illinois, USA) *(CCSW '11)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/2046660.2046670

[28] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (Chicago, Illinois, USA) *(CCSW '11)*. ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/2046660.2046670

[29] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co-residence Threat inside the Cloud. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 929–944. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu

[30] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co-residence Threat inside the Cloud. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 929–944. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu

[31] Tianwei Zhang, Yinqian Zhang, and Ruby Lee. 2016. Memory DoS Attacks in Multi-tenant Clouds: Severity and Mitigation.

[32] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. 2016. A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds. In *Information and Communications Security*, Kwok-Yan Lam, Chi-Hung Chi, and Sihan Qing (Eds.). Springer International Publishing, Cham, 361–375.

[33] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. 2016. A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds, Vol. 9977. 361–375. https://doi.org/10.1007/978-3-319-50011-9_28

[34] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael Reiter. 2011. HomeAlone: Co-Residency Detection in the Cloud via Side-Channel Analysis. 313 – 328. https://doi.org/10.1109/SP.2011.31

[35] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA, 990–1003. https://doi.org/10.1145/2660267.2660356