CoResident Evil: Covert Communication In The Cloud With Lambdas

Anil Yelam
UC San Diego

Shibani Subbareddy*
Salesforce.com, Inc.

Keerthana Ganesan*
Facebook, Inc.

Stefan Savage
UC San Diego

Ariana Mirian
UC San Diego

ABSTRACT

"Serverless" cloud services, such as AWS lambdas, are one of the fastest growing segments of the cloud services market. These services are popular in part due to their light-weight nature and flexibility in scheduling and cost, however the security issues associated with serverless computing are not well understood. In this work, we explore the feasibility of constructing a practical covert channel from lambdas. Ariana ►*Suggested change: We explore the challenges of a co-residence detector for lambdas and proceed to develop a reliable and scalable co-residence detector using the memory bus hardware.*◄ Anil ►*prefer not, because that adds a jump to co-residece detector from covert channels, reader may not quickly connect the two*◄ We establish that a fast co-residence detection for lambdas is key to enabling such a covert channel, and proceed to develop a reliable and scalable co-residence detector based on the memory bus hardware. Our technique enables dynamic discovery for co-resident lambdas and is incredibly fast, executing in a matter of seconds. We evaluate our approach for correctness and scalability, and use it to establish covert channels and perform data transfer on AWS lambdas. We show that we can establish tens of individual covert channels for every 1000 lambdas, and each of those channels can send data at a rate of 200 bits per second, thus demonstrating the feasibility of covert communication via lambdas.

CCS CONCEPTS

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

KEYWORDS

cloud, cartography, serverless, coresidency, covert channels

ACM Reference Format:
Anil Yelam, Shibani Subbareddy, Keerthana Ganesan, Stefan Savage, and Ariana Mirian. 2021. CoResident Evil: Covert Communication In The Cloud With Lambdas. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3442381.3450100

*Work done while at UC San Diego

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3450100

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 as 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 serverless computing pose inherent challenges for attackers, their low-cost and light-weight isolation potentially present 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[1]?

Covert channels 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

[1]We will use the term lambdas to stand for all such services going forward.

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 covert channels. First, the physical location of a lambda is unknown, as 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 with 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 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 one or more receiving lambdas).

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's 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 present a novel protocol in which co-resident lambdas communicate their IDs using a hardware-based covert channel in order to identify one another. The protocol also allows a sender or receiver lambda to enumerate *all* its co-resident neighbors, a requirement to avoid unwanted communication interference while performing covert communication.
- **Covert channel demonstration** We use our co-residence detector to establish lambda covert channels on AWS. We then perform a study on the feasibility of covert communication by 1) estimating the capacity of each channel and 2) measuring co-residence density – that is, for a given number of lambdas launched at the same point in time, how many would become co-resident? We conduct these measurements across a range of Amazon data centers to establish that there is ample lambda co-residence and that covert communication is practicable.

Our implementation is publicly available at *https://github.com/anilkyelam/columbus*.

2 BACKGROUND

We begin with a brief background on related topics.

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 as 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 maintenance, lambdas are much more cost-efficient than virtual machines (VMs) as they allow more efficient packing of functions on servers. Moreover, lambdas execute as much smaller units and are more ephemeral than virtual machines. 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. While previous studies that profiled lambdas [24] focused on the performance aspects like cold start latencies, function instance lifetime, and CPU usage across various clouds, 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 signaling 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 hardware such as caches [19, 27], memory bus [26], and even processor temperature [16].

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 remain 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 bits per second (bps) on the memory bus channel in an ideal laboratory setup.

Achieving such ideal performance, however, is generally not possible in cloud environments. Cloud platforms employ virtualization to enable statistical multiplexing and as such, communication on the covert channel may be affected by: 1) scheduler interruptions as the sender or the receiver may only get intermittent access to the channel and 2) interference from other non-participating workloads. This may result in errors in the transmitted data and require additional mechanisms like error correction [26] to ensure reliable communication.

2.3 Co-residence Detection

In the cloud context, enabling communication over traditional covert channels comes with an additional challenge of placing sender and receiver on the same machine. However, such co-residency information is hidden, even if the entities belong to the same tenant. Past research has used various strategies to achieve co-residency in order to demonstrate various covert channel attacks in the cloud. Typically, the attacker launches a large number of cloud instances (VMs, Lambdas, etc.), following a certain launch pattern, and employs a co-residence detection mechanism for detecting if any pair of those instances are running on the same machine. Traditionally, such detection has been based on software runtime information that two instances running on the same server might share, like public/internal IP addresses [19], files in *procfs* or other environment variables [24, 26], and other such logical side-channels [21, 29].

As virtualization platforms moved 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 section 2.2, which can bypass software isolation and are thus usually harder to fix. For example, Varadarajan et al. [22] use the memory bus covert channel to detect co-residency for EC2 instances. However, as we will show, their approach does not extend well to lambdas as it is neither fast nor scalable.

3 MOTIVATION

In this section, we discuss our covert channel attack scenario, the challenges lambdas would pose in enabling such an attack and, the need for a co-residence detector for lambdas.

Threat Model Covert channel attacks generally require an "insider" who sends data over a covert medium for exfiltration. We assume that the attacker uses social engineering techniques or some other means (beyond the scope of this work) to introduce such insiders in the victim system. In the case of lambdas, this insider code could be in the lambda itself or in a system that controls lambda deployments for an organization and already possesses the sensitive data that needs to be exfiltrated. We further assume that the attacker has the knowledge of the cloud region where the victim is operating and can deploy lambdas under its own account.

In a typical attack, the attacker launches a set of lambdas (receivers) in the cloud region where the victim lambdas (senders) are expected to operate. The attacker and (compromised) victim lambdas can then work together[2] to exchange the data over a covert channel, like the memory bus hardware discussed earlier. However, as mentioned earlier, there are few unique challenges before we can use a traditional covert channel in the cloud. We need to: 1) co-locate the sender and receiver on the same server (via co-residence detection) and 2) handle interruptions on such a channel, such as noisy neighbors neighbors or scheduling.

While these challenges have been handled for other cloud platforms like VMs [21, 26], lambdas are inherently different in that they have very short lifetimes. A covert channel between two co-resident lambdas will not last very long. However, while lambdas are not persistent, it is trivial and cheap to launch lambdas in large numbers at once and establish multiple co-residence points to allow for more covert communication. Additionally, lambdas are also more densely packed than VMs, exacerbating the noisy neighbor problem.

The ephemeral, numerous, and dense nature of lambdas require a fast, scalable, and reliable co-residence detector. This detector should also allow the attacker identify all the servers with two or more co-resident lambdas and establish a covert channel on each such server Ariana ►*how is the rendezvous different that the following sentence? If we want to use this term we need to explain what it means. I dont think we need it but I wanted to make a note in case I'm missing something*◄ Anil ►*there*◄. Moreover, the detector should precisely identify how many and which lambdas are co-resident, allowing the attacker to pick any two lambdas on a given machine, and use the covert channel between these two lambdas without interference from the rest.

4 CO-RESIDENCE DETECTOR FOR LAMBDAS

In this section, we describe the necessary requirements for a lambda-based co-residence detector, how these requirements have driven our design choices, the detailed implementation that we have used to overcome the unique challenges in this setting and an evaluation of its effectiveness.

4.1 Specification

Given a set of cloud instances (VMs, containers, functions, etc) deployed to a public cloud, a co-residence detection mechanism should identify, for each pair of instances in the set, whether the pair is 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, it should have the following 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 cloud platforms 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).

[2]We do not explicitly differentiate attack and victim lambdas hereafter as they are all assumed to be in the control of the attacker

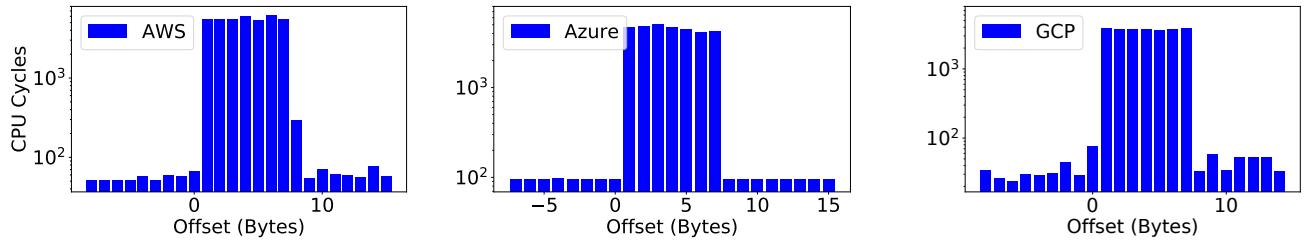

Figure 1: The plots show the latencies of atomic memory operations performed on an 8B memory region as we slide from one cache line across the boundary into another on AWS, Azure, and Google (GCP) respectively. The latencies are orders of magnitude higher when the 8B region falls across the two cache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

- **Scalable** A launch strategy may require hundreds or even thousands of instances to be deployed, and must scale such that the technique will detect all co-resident 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.

4.2 Design

Ariana ►*Alternate title: Design Considerations*◄ Ariana ►*each of these sections is basically setting up the problems so that we can address them later* Anil ►*well, the final subsubsection is the design itself..*◄◄

4.2.1 Generality. As mentioned in section 2.3, co-residence detection was previously accomplished using unique software identifers that revealed underlying the server structure to the tenants. Such software identifiers present the fastest and perhaps most reliable way (in terms of ground truth) for co-residence detection. In our scenario, each lambda could quickly "read" the identifier and communicate this information with all other lambdas (through the network or shared remote storage) to identify the co-resident lambdas. However, such information can be easily obfuscated by platform providers; for example, currently there is no such identifier available for AWS lambdas. So, we turn to hardware-based covert channels that are, by definition, also accessible to all the tenants on a single server. Hardware-based covert channels are also generally more difficult to remove, as well as more pervasive, given that the hardware is more homogenous across computing platforms than software.

4.2.2 Challenge with Covert Channels. Ariana ►*alternate subtitle: Multiple Party Support*◄ Ariana ►*this paragraph differs from the other two in that it doesnt have a solution for this design consideration*◄

Since covert channels can send information, we would naturally expect that we can use the same channel to communicate identity information (such as unique IDs) between co-resident lambdas. However, covert channels generally presume that the sender and the receiver are already known and that there will be only two parties performing communication. When multiple parties share a channel, we need stricter control over the channel, to arbitrate access to the channel and handle collisions. However, covert channels are often binary channels (i.e., parties can either only send or receive a bit, not both) with no capability for collision detection. Covert channels also typically have very limited bandwidth (often only tens of bits per second) that channel arbitration can be complex enough and present significant overhead to be considered infeasible. Ariana ►*needs the high level solution to match the flow of the others* Anil ►*updated name may render this moot*◄◄

4.2.3 Efficient ID Broadcasting. Ariana ►*alternate subtitle: Broadcasting IDs Quickly or Fast ID Broadcasting*◄

Ariana ►*I think the problem statement should be a bit clearer*◄ For co-residence detection, lambdas only need to communicate their IDs with one another. As such, we do not require the communicate channel be general or expressive, but that the communication be fast (to account for the ephemeral nature of lambdas). We assume that each lambda involved has a unique fixed-length (say N) bit-string corresponding to its ID that must be communicated. We also assume that any lambda with access to the channel can choose to either send or listen for a bit, and if at least one lambda chooses to send a bit, all the listeners would record a bit. Finally, we assume that lambdas can synchronize themselves between sending or listening a bit in each time slot. We show both these assumptions to be reasonable in section 4.3. As such, we propose a communication protocol that efficiently broadcasts just these bit-strings on channels with such restricted properties.

We divide the total 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* lambdas, which in the first phase would be all of the lambdas. In each bit-slot K of N slots in a phase, every participating lambda broadcasts a bit if the K^{th} bit of its bit-string (ID) is 1, otherwise it listens for a 0 or 1. If a lambda senses a 1 while listening, it stops participating, and listens for the rest of the phase. Thus, only the lambdas 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 phase, the lambda with the previously highest ID now only listens, allowing the next highest

Algorithm 1 ID Broadcast Protocol

```
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. 2)
15:       bit_read ← 1
16:     else
17:       bit_read ← READ_BIT() (Alg. 3)
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
```

lambda to advertise its ID, and so on. If the IDs are unique, there will always be only one lambda that broadcasts in every phase. The protocol ends after x phases (where x is the number of co-resident lambdas), when none of the lambdas broadcast for N consecutive bit-slots. The pseudo-code of the protocol is provided in Algorithm 1.

Time complexity Assuming N total deployed lambdas to the cloud, the bit-string needs to be $\log_2 N$ bits to uniquely identify each lambda. If a maximum K of those lambdas 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. In fact, it is not necessary to run the protocol for all K phases. After the first phase, all the co-resident lambdas would know one of their neighbors (as each phase reveals the ID of the biggest participating lambda to others). If we use IDs that are globally unique, all the co-resident lambdas will see the same ID. The lambdas 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-resident lambdas (K) and decreases the complexity to $O(\log_2 N)$, providing a sub-linear co-residence detection mechanism.

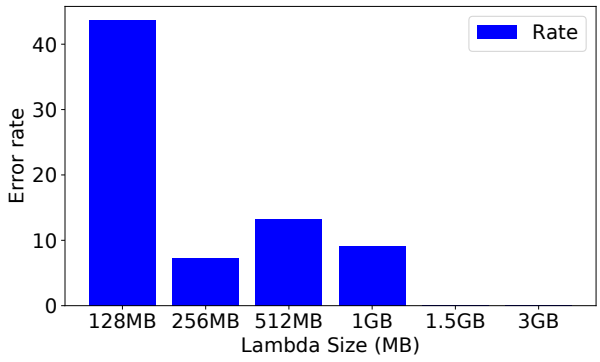

Figure 2: This figure presents the error rate (as a fraction of 1000 lambdas deployed) for different lambda sizes in the AWS Middle-East region.

4.3 Implementation

Using the design considerations just discussed, we implemented the above protocol using a covert channel based on memory bus hardware. We discuss how we used the hardware to send and listen for bits in order to meet the requirements for the protocol.

4.3.1 Memory Bus Covert Channel. Ariana ▶*alternate subtitle: Memory Bus as a Cover Channel* Anil ▶*removed "introducing"*◀◀ We utilize 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 find that they are still vulnerable today. To demonstrate the presence of the vulnerability, we measure the latency of atomic operations on an 8B memory region as we slide the region from one cacheline into another across the cacheline boundary. We perform this experiment on three major cloud platforms (AWS, Google and Microsoft Azure) and show the latencies observed in Figure 1. From the figure, we can see that all three cloud platforms still exhibit a significant difference in latencies for the "exotic" memory locking operations where the memory region falls across cacheline boundary. When compared to regular memory accesses, it demonstrates the presence of this covert channel on all of them. Moreover, we were able to execute 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, fulfilling the generic aspect of our mechanism.

4.3.2 Sending a bit. Senders and receivers can accurately communicate 0-bits and 1-bits by causing contention on the memory bus. 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 2.2). The pseudo-code for the sender instance is shown in Algorithm 2.

Algorithm 2 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
```

Algorithm 3 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
```

4.3.3 Listening for a bit. The receiver can simply 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). However, there are two ways to listen for contention. 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.

Mechanism Of these two methods, we decide to use the memory locking receiver for our experiments. Since memory probing involves regular (un-cached) memory accesses, it can be performed on multiple receivers concurrently 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.

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 the 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 can 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 the receiver sampling, assuming tens of co-resident receivers and a few microseconds sampling time.

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 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, which runs 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 3 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.

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 eight in Algorithm 3), 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 tail in Figure 3. 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

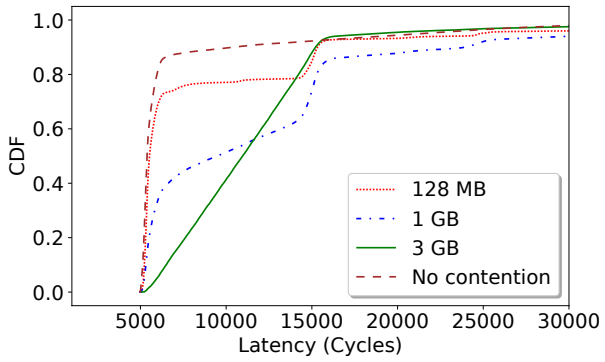

Figure 3: We present a CDF of latencies observed by 128 MB, 1 GB and 3 GB Lambdas during contention. The 128 MB lambda pair sees less contention due to more context switching, whereas the 1 GB and 3 GB lambdas see progressively more contention compared to the baseline, which we attribute to their relative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

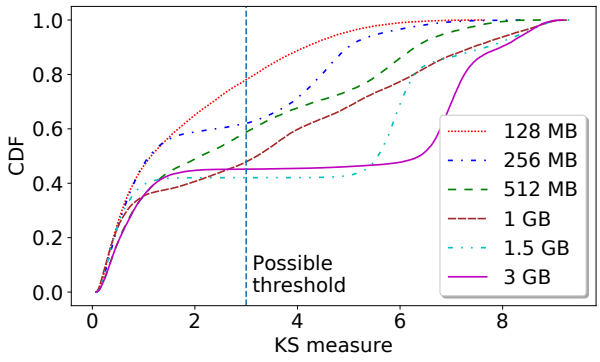

Figure 4: We present a CDF of KS values observed for various lambda sizes. A bimodal distribution with a longer step allows us to pick a KS-threshold that enables our technique to differentiate between 0-bit and 1-bit with high confidence.

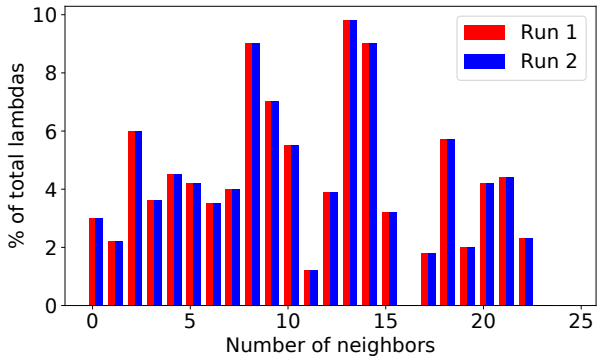

Figure 5: This figure shows the fraction of lambdas by the number of neighbors they identify for two independent runs that use same set of underlying AWS containers. The perfect correlation shows that both runs depict the co-residence status of those containers regardless of the lambdas that ran on them,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

the two-sample Kolomogorov-Smirinov (KS) test, which typically compares the maximum of the absolute difference between empirical CDFs of samples (in our variant, we take the *mean* of the absolute difference instead of the maximum to reduce sensitivity to outliers). 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-resident 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 4. Ideally, we expect a bimodal 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 4.3.3). 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 consistent across AWS regions, suggesting that this threshold is a platform constant.

We present the pseudo-code of a receiver lambda in Algorithm 3, which includes all the challenges and subsequent solutions discussed thus far.

4.3.4 Synchronization. A major enabler of our protocol in section 4.2.3 is the ability to synchronize all the co-resident lambdas when sending and receiving bits. As all these lambdas are running on the same physical server, they share the server's clock. On AWS, for example, we observe that the system clock on lambdas is precise up to nanoseconds. Assuming that the clocks between different lambdas only exhibit a drift in the order of microseconds, sampling at a millisecond scale should provide us a margin for synchronization mismatch. Since we do not observe any synchronization-related noise in our results, we believe that this is a reasonable assumption.

4.4 Evaluation

We next examine our co-residence detector with respect to reliability, scalability, and speed, the desirable detection properties mentioned in section 4. We run all of our experiments with AWS

lambdas [3]. Though we decide to focus on only one of the cloud providers as a case study, we have previously shown in section 4 that this covert channel exists on the other clouds, and thus we believe these experiments can be replicated on their serverless functions as well. We use the C++ runtime in AWS lambdas as it allows pointer arthmetic that is required to access the covert channel.

4.4.1 Setup. We start by deploying a series of lambdas from an AWS lambda account. Once deployed, each lambda participates in the first phase of the protocol as noted in section 4.2.3, 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 lambda. If all three rounds result in different IDs, we classify this lambda as erroneous and report it in the error rate. We group all the lambdas that saw the same ID as successful and neighbors. We repeat the experiments for different lambda sizes and in various cloud regions.

4.4.2 Reliability. We consider the results of the technique reliable when most of the deployed lambdas successfully see the same result in a majority of the independent rounds (indicating lesser bit-flip errors) and the resulting co-resident groups we see examine the ground truth. For the former requirement, 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 returned erroneous result). **Ariana** ▶*the phrasing of majority here confuses me a bit but im not sure how to fix it* **Anil** ▶*defined in above paragraph*◀◀ Figure 2 indicates that smaller lambdas exhibit more errors. This is expected because, as discussed in section 4.3.3, these lambdas experience lossy communication, making it harder for our technique to sense contention. Lambdas that are 1.5 GB and larger, though, exhibit a 100% success rate. Of these errors, false positives are rare as we run multiple independent rounds, and the chance of two rounds erring at the same bit(s) (to give the same wrong result across the majority of rounds) is low. False negatives, however, are common and contribute to most of the errors.

Correctness To determine correctness, we require ground truth on which lambdas are co-resident 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 lambda runs in a dedicated container (sandbox). After execution, AWS caches these containers in order to reuse them [2] and mitigate "cold start" latencies. We found that global objects like files persist across warm-starts, and can be used to track all the lambdas that were ever executed in a particular container. Using this insight, 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. This allows us to test the correctness of our technique as variance in the co-residence results between these experiments would suggest a lack of fidelity in our approach. (Note that the warm-start information cannot be used to detect co-residency itself nor can it be used to verify correctness in all scenarios.)

To demonstrate this correlation, we run an experiment with 1000 1.5GB cold-started lambdas (ID'ed 1 to 1000) in one of newer AWS regions (me-south-1), which resulted in many co-resident 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-resident 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-resident 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 container colocation. Figure 5 shows that both experiments saw the same number of co-resident groups of different sizes, showing the correctness of the results of our mechansim.

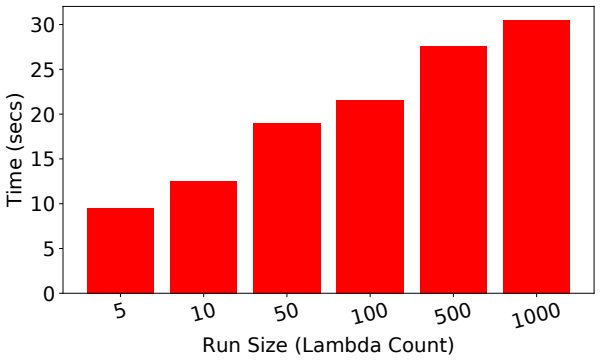

Figure 6: We present the average execution time of the lambdas for co-resident runs with a varying number of lambdas. The execution time increases logarithmically with the number of lambdas demonstrating the scalability of co-residence detection with our technique.

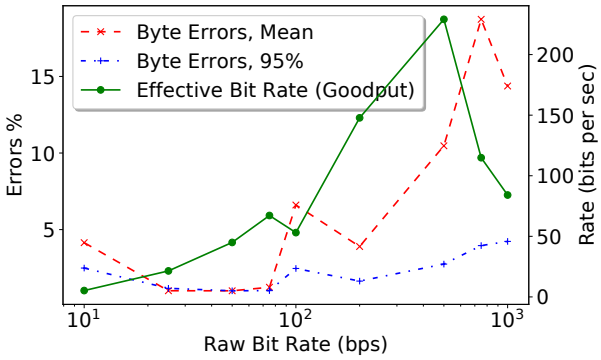

Figure 7: The dashed lines in the figure presents the mean error ratio at 50% and 95% confidence when sending bits at various rates over the covert channel. The solid line shows the effective channel rate after accounting for the overhead of error correction at the 95% error.

4.4.3 Scalability & Speed. One of the key properties of this technique is its short execution time and scalability. Since communicating each binary bit of the ID takes one second, we are able to

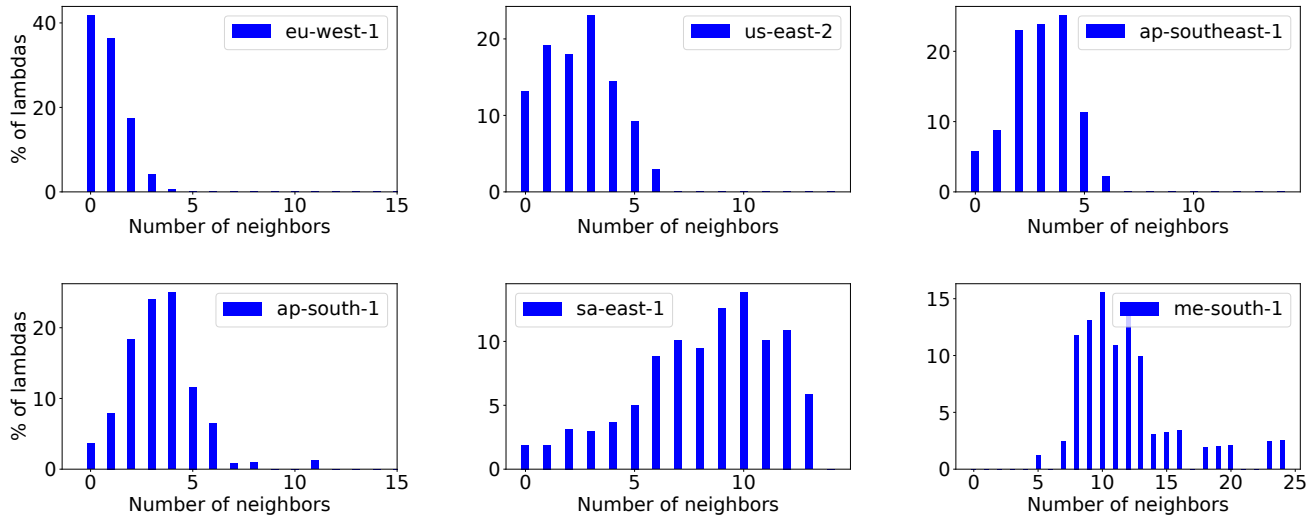

Figure 8: We present co-residence results from a 1000-lambda run in different AWS regions. Each bar shows the fraction of those 1000 lambdas (in %) that discovered a certain number of neighbors. The total amount and density of co-residence vary widely across regions, perhaps based on the size of those regions and the lambda activity within them.

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 (for reference, lambdas on AWS can only run for 15 minutes at most). The logarithmic scale of our method also indicates that the cost per lambda scales logarithmically, making neighbor detection cost-effective. For example, one 1000 lambda run with 1.5 GB lambdas cost around just 1.5 USD.

5 PRACTICALITY OF COVERT COMMUNICATION

In this section, we perform a study to demonstrate the practicality of data transfer using the lambda covert channels discovered with our co-residence detector. The amount of information that can be transferred depends on two factors: 1) the capacity of each channel and 2) the number of co-resident clusters of lambdas, or rendezvous points, that materialize during the attack. We first produce an estimate on the capacity of the covert channels established, and then examine the co-residence density in various AWS regions to understand the number of rendezvous points and factors that affect it.

5.1 Covert Channel Capacity

Once co-residence between any two lambas is established, the attacker can then use the same memory bus hardware to perform covert communication. Wu et al. [26], who first introduced covert channel based on this hardware channel, also presented an efficient and error-free communication protocol targeting cloud-based platforms like VMs. While such a protocol should theoretically work for lambdas, extending it is beyond the scope of this work. We do, however, use a much simpler (albeit more inefficient) protocol to report a conservative estimate of the capacity of each covert channel.

Our protocol for data transfer uses the bus contention in the same way as the co-residence detector in section 4.3 to send and receive bits and perform clock synchronization. However, now that we can use our co-residence detector to identify lambdas on a machine and target the two that we wish to label as the sender and receiver, we are not concerned about noise from multiple receivers, and as such can allow the receiver to sample continuously (section 4.3.3) and sample for extremely small duration (milliseconds instead of seconds). While we want the sampling duration to be as small as possible (in order to increase the rate of bits transferred), the chances of erasures or errors also increases as the sender and receiver may get descheduled during this time.

To demonstrate this, we launched hundreds of 3 GB lambdas on AWS and use our co-residence detector to establish tens of covert channels. We then send data over these channels at various bitrates and record the error ratio (for byte-sized data segments). Figure 7 shows the mean error ratio at 50% and 95% one-sided confidence intervals, both of which increase with the bitrate.

To correct these errors, we use Reed-Solomon coding, a block-based error correction code that is suitable for burst-errors caused by descheduling [26]. However, error correction comes with an overhead; with byte-sized symbols, Reed-Solomon requires twice as many parity bytes as there are errors to correct. So, for each bitrate, we must compute effective bitrate by subtracting the overhead of error correction bytes. From Figure 7, we can see that effective bitrate rises to a maximum of over 200 bits per second (bps)

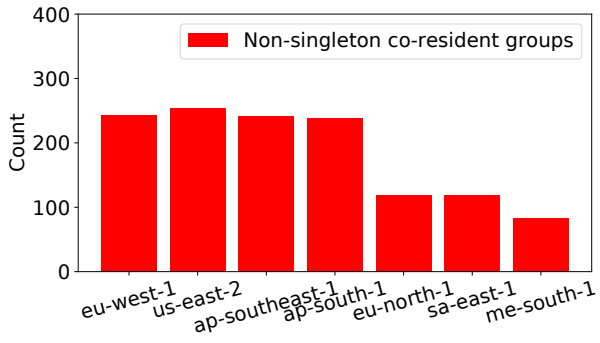

Figure 9: This figure shows the number of co-resident groups with more than two lambdas seen in various AWS regions for the runs shown in Figure 8. Each such co-resident group can host a covert channel, indicating that a 1000 lambda deployment can enable hundreds of covert channels.

(at 500 bps raw rate) before falling again due to high error rate. We confirmed this by sending Reed-Solomon encoded data over the covert channels at this rate and observed near-zero data corruption. Thus, we conclude that, by a conservative estimate, we can safely send data across each of these covert channels at a rate of 200 bps.

5.2 Covert Channel Density

Finally, we present measurements on covert channel density on AWS using our co-residence detector, and discuss the factors that may affect this density. As we discussed in section 3, each co-resident group of lambdas represent an individual server in the cloud and hence can enable an independent covert channel whereever the group has more than two lambdas. So we attempt to answer the following question: assuming that the user launches a number of (sender and receiver) lambdas at a specific point in time, what is the expected number of such co-resident groups (with two or more lambdas) that they might see? We deploy a large number of lambdas on various AWS regions and report the co-residence density, that is, the average number of such co-residence groups. The higher the co-residence density, the easier it is for the user to ultimately establish covert channels with lambdas, and the more information they can send. Unless specified otherwise, all the experiments discussed in this section are performed with 1.5 GB lambdas and executed successfully with **zero error** in co-residence detection.

5.2.1 Across AWS regions. We execute our co-residence detector with 1000 1.5 GB lambdas in various AWS regions. Figure 8 comprises multiple plots depicting the co-resident 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-resident group of a certain size). Plots that skew to the right indicate a higher co-residence density when compared to the plots skewed to the left. 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 resulting in higher co-residence density in those regions. Figure 9 shows the total number of co-resident groups with two or more lambdas, each of which can be an independent covert channel. The ample co-residence in general across all the regions shows that lambdas provide a fertile ground for covert channels.

5.2.2 Other factors. We also examine how co-residence is affected by various launch strategies that the user may use, like deploying lambdas from multiple AWS accounts and different lambda sizes. In particular, we wish to determine if our mechanism exhibits different results when: 1) the user deploys sender lambdas and receiver lambdas on two separate accounts (normally the case with covert channels) and 2) the senders and receivers are created with different lambdas sizes. To answer these questions, we run an experiment with 1000 lambdas, in which we launch 500 lambdas from one account (senders) and 500 from the 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 10, we show the breakdown of co-resident group of lambdas of each size among the two accounts. We can see that among the co-resident groups of all sizes, roughly half of the lambdas came from either account. This suggests that lambda scheduler could be agnostic to the accounts the lambdas were launched from. We see similar results for different lambda sizes, as shown in the right subfigure of Figure 10.

6 DISCUSSION

Alternate use cases Our main motivation behind proposing a co-residence detector for lambdas is to demonstrate the feasibility of covert channels. However, there are other scenarios where such tool can be (ab)used, of which we provide a couple of examples.

- Previous studies on performance aspects of lambdas (like performance isolation [24]) generally need a way to find co-resident lambdas. As software-level logical channels begin to disappear, our tool might provide a reliable alternative.
- Burst parallel frameworks [8] that orchestrate lambdas can use our co-residence detector as a locality indicator to take advantage of server locality.

Mitigation In the 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 operations that are required to activate this channel. However, in all three cloud platforms we examined, these unsafe or low level languages were introduced as options at a later point, indicating that there is a business use case. In that case, cloud providers may look at more expensive solutions like

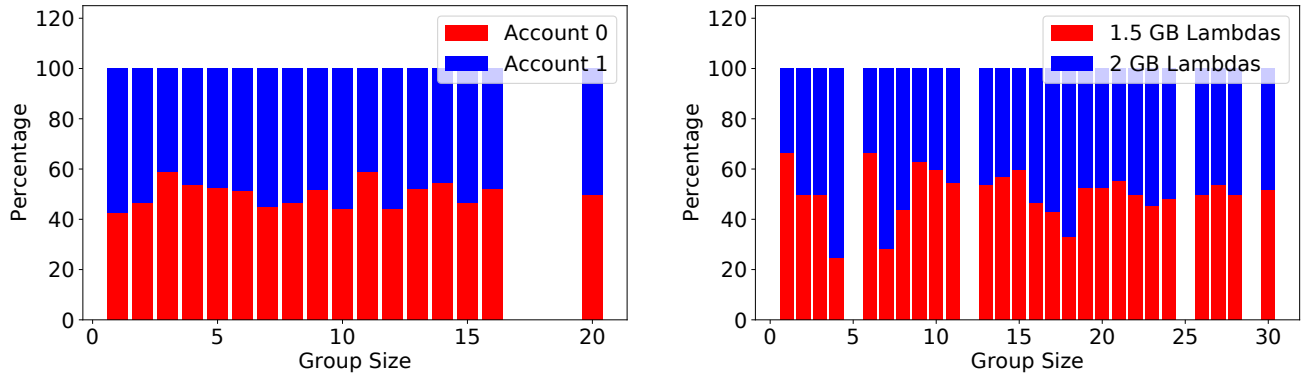

Figure 10: The left plot shows the breakdown of co-resident groups (of varying sizes) of lambdas by two different accounts in an experiment of 1000 lambdas, where 500 lambdas are launched from each account. The uniformity of the split suggests that the lambda scheduler might be invariant to the account the lambdas are launched from. Similar results are shown for different lambda sizes in the right plot.

BusMonitor [20] that isolates memory bus usage for different tenants by trapping the atomic operations to the hypervisor. We leave such exploration to future work.

7 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, 34] 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 literature 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 in the context of cost-effectiveness of batching and data processing with them [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.

8 ETHICAL CONSIDERATIONS

As with any large scale measurement project, we discuss the ethical considerations. 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 the 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.

9 CONCLUSION

In this paper, we have demonstrated a technique to build covert channels entirely using serverless cloud functions such as AWS lambdas. To achieve this goal, we developed a fast and reliable co-residence detector for lambdas, and evaluated it for correctness and scalability. Finally, we have empirically demonstrated the practicality of such covert communication by studying the covert channel capacity and co-residence density of lambdas on various AWS regions.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grant CNS1705050, generous gifts from Google and Facebook and the Irwin Mark and Joan Klein Jacobs Chair in Information and Computer Science. We are indebted to Stewart Grant, Alex Snoeren, Geoff Voelker, and the anonymous reviewers for feedback on earlier drafts of this manuscript. We also thank Sabareesh Ramachandran for helpful conversations on the algorithmic part of the work.

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