NotColumbus: Covert Communication In The Cloud With Lambdas

Anil Yelam
UC San Diego

Shibani Subbareddy*
Salesforce 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 segment of the cloud services market. 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 are not well understood. In this work, we explore the feasibility of constructing a practical covert channel from lambdas. We establish that a fast and scalable co-residence detection for lambdas is key to enabling such a covert channel, and proceed to develop a generic, reliable, and scalable co-residence detector based on the memory bus hardware. Our technique enables dynamic neighbor discovery for co-resident lambdas and is incredibly fast, executing in a matter of seconds. We evaluate our approach for correctness and scalability, and perform a measurement study on lambda density in AWS cloud to demonstrate the practicality of establishing cloud covert channels using our co-residence detector for lambdas. Through this work, we show that efforts to secure co-residency detection on cloud platforms are not yet complete.

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. NotColumbus: 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, 12 pages. 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. [10]) 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 [7], Google's Cloud Functions [6] and Microsoft's Azure functions [4]). 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 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[1]?

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.)

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

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 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 [16], 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 for for the co-resident lambdas to communicate their IDs using memory bus contention and discover one another. A key enabler of our co-residence detection, the protocol also helps 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 capacity estimation** TODO ►◄
- **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 there is ample co-residence of lambdas and covert communication is practicable.

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

2 BACKGROUND & RELATED WORK

Anil ►*Did not like how this turned out, Ariana, reverting it. If you remember the first time around, this section was very piecemeal and contained lot of unrelated text, and we struggled to polish it, and it felt like a step back. I think its ok to just remove the related work section :). I may have lost some of your minor writing improvements by reverting the whole thing, I'll work on integrating them later.*◄

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 [7], and as cloud functions on GCP [6] and Azure [4], 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. 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 [14] 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 signalling bits by causing contention on shared hardware media on the server [8, 9, 15–17]. Past work has demonstrated covert channels in virtualized environments like the clouds using various harware such as caches [10, 17], memory bus [16], and even processor temperature [8].

Memory bus covert channel Of particular interest to this work is the covert channel based on memory bus hardware introduced by Wu et al. [16]. 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 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 to statistical multiplexing, and communication on the covert channel may be affected by 1) scheduler interruptions as 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 [16] to ensure reliable communication.

2.3 Co-residence Detection

In the cloud context, enabling covert communication comes with an additional challenge of placing sender and receiver on the same machine so that both can have access to the shared software or hardware-based covert channel. 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 [10], files in *procfs* or other environment variables [14, 16], and other such logical side-channels [12, 18].

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 the earlier section, which can bypass software isolation and are usually harder to fix. For example, Varadarajan et al. [13] 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 slow and not scalable.

3 MOTIVATION

We discuss the covert channel attack scenario that we are targetting, the challenges lambdas would pose in enabling such an attack and motivate the need for a co-residence detector for lambdas which is going to be our main focus in the rest of the paper.

3.0.1 Threat Model. Covert channel attacks require an "insider" to send the 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 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. Anil ▶*the abilities we assume on the part of the attacker to perform a succesful attack may sound far-fetched; was always afraid of this, but hey, the paper already got in! :D*◀

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 (comprimised) victim lambda(s) can then work together[2] to exchange the data over a covert channel like the memory bus hardware discussed in earlier section. However, as mentioned earlier, there are few unique challenges before we can use a traditional covert channel in the cloud. We need: 1) to colocate the sender and receiver on the same server which requires co-residence detection and 2) to handle the interruptions on such channel introduced by noisy neighbors and inconsistent access to the channel due to scheduling.

While these challenges have been handled for VMs (see related work), lambdas are inherently different from VMs 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 rendezvous points to allow for more covert communication. Additionally, lambdas are also densly packed packed than VMs (discussed later) exacerbating the noisy neighbor problem.

The ephemeral, numerous and dense nature of lambdas require a fast, scalable, and reliable co-residence detector. Such a robust co-residence detector will let the attacker to identify all the servers with two or more co-resident lambdas and establish a rendezvous on each such server. Moreover, the detector precisely identifies which lambdas are co-resident allowing the attacker to pick any two lambdas on a given machine, and use the covert channel without interference from neighbors.

4 CO-RESIDENCE DETECTOR FOR LAMBDAS

Anil ▶*An alternative framing for this section, lets see how you like it. Ariana, take a pass and do some sign-posting if you like; my brain was no longer concerned with polished text at this hour, but try not to make any big changes. If this is bad, we can always go back to earlier framing, content that we have at least tried. I will work on section 6 after I wake up, and we can send it to Stefan. Or you can just send it and say that we are still working on section 6.*◀

In this section, we present a novel co-residence detector for lambdas, previous solutions to this problem, and the unique challenges we faced with lambdas and how we address each of them.

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 was running on the same physical server at some point. Paraphrasing Varadarajan et al. [13], for any such mechanism to be useful across a wide range of launch strategies, it should have the following properties:

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

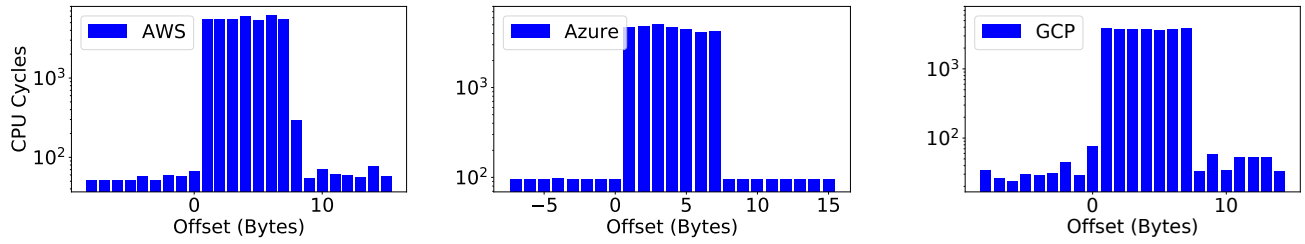

Figure 1: The plots show the latencies of atomic memory operations performed on an 8B memory region as we slide from one cache line across the boundary into another on AWS, Azure, and Google (GCP) respectively. The latencies are orders of magnitude higher when the 8B region falls across the two cache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

- **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).
- **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

4.2.1 Using Covert Channels for Co-residence Detection. As mentioned in section 2.3, co-residence detection was previouly done using unique software idenitifers that revealed underlying server to the tenants. Such software identifiers present the fastest and perhaps most reliable(Anil ►*reliable as in trustworthy information*◄) way for co-residence detection as all it takes is for each lambda to quickly "read" the identifier and communicate this information with all other lambdas (through the network or shared remote storage) to quickly identify the neighbors. However, such information can be easily obfuscated by platform providers; 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. They are also generally more difficult to remove, and more pervasive too, given that hardware is more homogenous across computing platforms than software.

4.2.2 Covert Channels Cannot Support Multiple Parties. As covert channels can send information, naturally we can expect to use the channel to communicate identity information (like unique ids) between co-resident lambdas all of which have access to the channel. However, covert channels generally presume that the sender and the receiver are already known and that there will be only two parties performing communication. But when multiple parties share a channel, we would need a channel access control, such as MAC protocols in Ethernet networks, to arbitrate access to the channel and handle collisions as they are in the same collision domain. However, covert channels are often binary channels (i.e., parties can either only send or receive a bit but not both) with no capability for collision detection and have very limited bandwidth (often only tens of bits per second) that such channel arbitration can be infeasible or complex enough to present significant overhead on channel bandwidth.

4.2.3 A fast ID broadcast designed for covert channels. For co-residence detection, lambdas only need to communicate their IDs with one another and as such, we don't need the channel to be very general and expressive. Thus, we assume that each lambda involved has a unique fixed-length (say N) bit-string corresponding to its ID that must be communicated. In regards to the covert channel, we only 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 1-bit, all the listeners would record a 1-bit. We also assume that lambdas can synchronize themselves to send or listen for each bit in time slots. We show both these assumptions to be reasonable in the next section. As such, we propose a communication protocol that efficiently broadcasts just these bit-strings on channel with such 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 co-resident 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 an 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

Algorithm 1 Neighbor discovery 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
```

the next highest 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 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.

4.2.4 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 $O(\log_2 N)$.

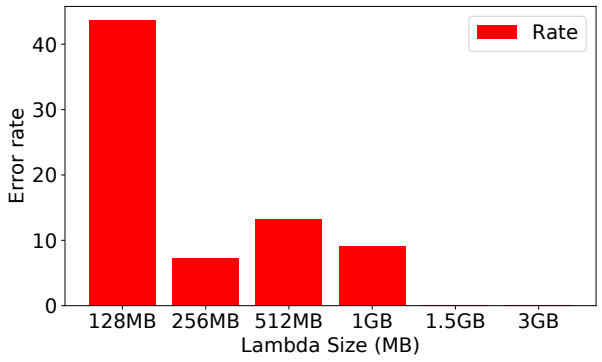

Figure 2: This figure presents the error rate (as a fraction of 1000 lambdas deployed) for different lambda sizes in the AWS Middle-East region. TODO ▶*make it print friendly*◀

4.3 Implementation

We implemented the above protocol using a covert channel based on memory bus hardware that is available for lambdas. We show how we used the hardware to reliably send and listen for bits in order to meet the requirements for the protocol.

4.3.1 Introducing Memory Bus Covert Channel. 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 [12, 19], and we find that they are still vulnerable today. To demonstrate the presence of the vulnerability, we measure the latency of atomic operations on a 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, demonstrating 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 as well.

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 **??**). 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
```

4.3.3 Listening for a bit: Mechanism. 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). There are two ways to listen for contetion. 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 [13]) 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 [13, 16] 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. Since memory probing involves regular (un-cached) memory accesses, it can be done 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.

4.3.4 Listening for a bit: 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 [13], 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

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
```

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-resident receivers and a few microsecond sampling time. TODO ▶*R1 needed clarification*◀

4.3.5 Listening for a bit: 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, 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 [7] 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.

4.3.6 Listening for a bit: 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

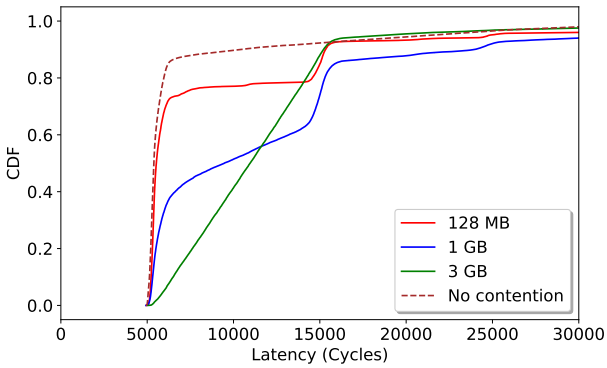

Figure 3: We present a CDF of latencies observed by 128 MB, 1 GB and 3 GB Lambdas during contention. The 128 MB lambda pair sees less contention due to more context switching, whereas the 1 GB and 3 GB lambdas see progressively more contention compared to the baseline, which we attribute to their relative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

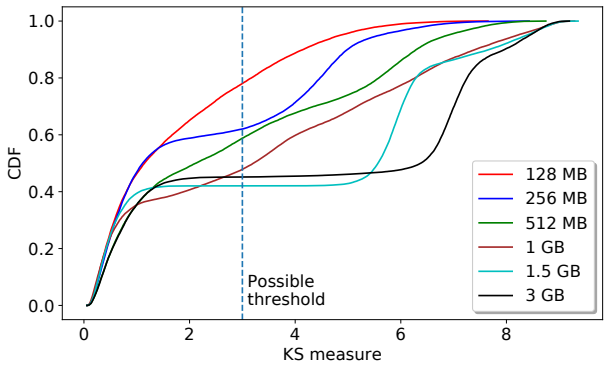

Figure 4: We present a CDF of KS values observed for various lambda sizes. A bimodal distribution with a longer step allows us to pick a KS-threshold that enables our technique to differentiate between 0-bit and 1-bit with high confidence.

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 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 4.3.5). 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.

4.3.7 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 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

Anil ▶*Making it a subsection looks right to me, Ariana, since we have both design and implementation now under section 5, this seems to follow naturally. lets ask Stefan's feedback perhaps*◀

We examine our co-residence detector with respect to reliability and scalability, 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 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. TODO ▶*add note about this fulfilling fast requirement?*◀

4.4.1 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 4.2.4, 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

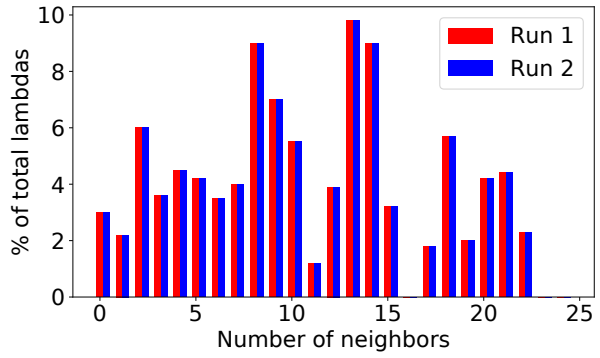

Figure 5: This figure shows the fraction of lambdas by the number of neighbors they identify for two independent runs that use same set of underlying AWS containers. The perfect correlation shows that both runs depict the co-residence status of those containers regardless of the lambdas that ran on them,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

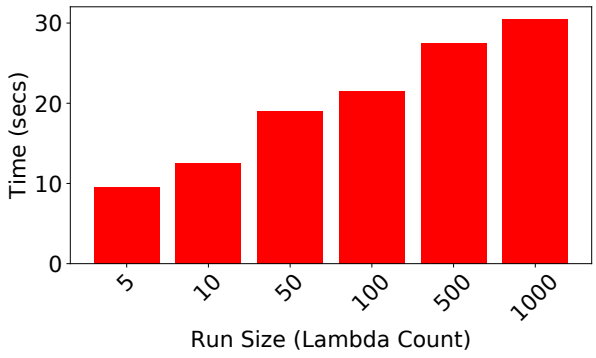

Figure 6: We present the average execution time of the lambdas for co-resident runs with a varying number of lambdas. The execution time increases logarithmically with the number of lambdas demonstrating the scalability of co-residence detection with our technique.

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. TODO ►*add cost here*◄

4.4.2 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-resident groups we see match the ground truth. For goal #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). Figure 2 indicates that smaller lambdas exhibit many more errors. This is expected because, as discussed in section 4.3.6, 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. TODO ►*rebuttal: clarify false postives and negatives*◄

Correctness TODO ►*promised clarity here in rebuttal*◄ To determine correctness, we require ground truth on which instances 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 lambdas runs in a dedicated container (sandbox). After execution, AWS caches these containers in order 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 executed in a particular container. This indicates that, for a given lambda, we can precisely note 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-residence is essentially co-residence of their containers, and given that containers persist across experiments that are executed within minutes of one another, lambda co-residence results must agree with the co-residence 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-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 continer colocation. Figure 5 shows that both experiments saw the same number of co-residing groups of different sizes. This proves the correctness of the results of our mechansim.

4.4.3 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. TODO ►*note here about sublinear time?*◄

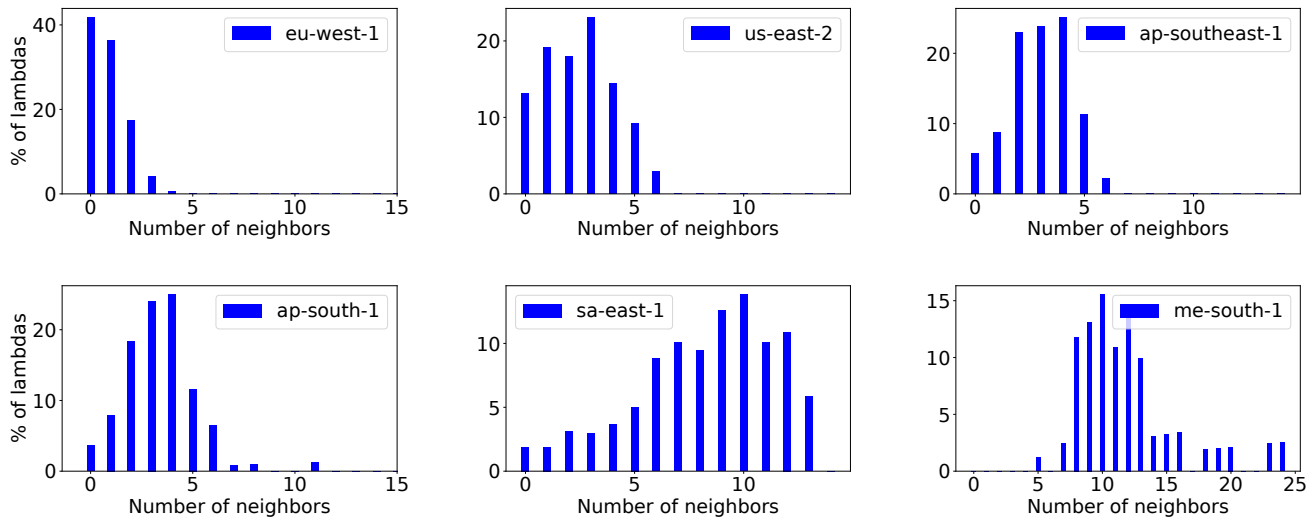

Figure 7: We present co-residence results from a 1000-lambda run in different AWS regions. Each bar shows the fraction of those 1000 lambdas (in %) that discovered a certain number of neighbors. The total amount and density of co-residence vary widely across regions, perhaps based on the size of those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

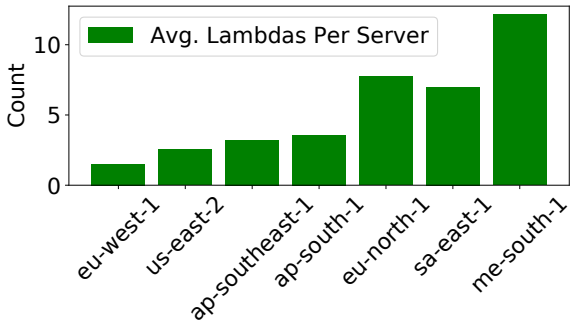

Figure 8: This figure shows the average number of lambdas per server i.e., the co-residence density seen in various AWS regions for the runs shown in Figure 7. The ample co-residence across regions demonstrates the practicality of establishing covert channels with lambdas in these regions.

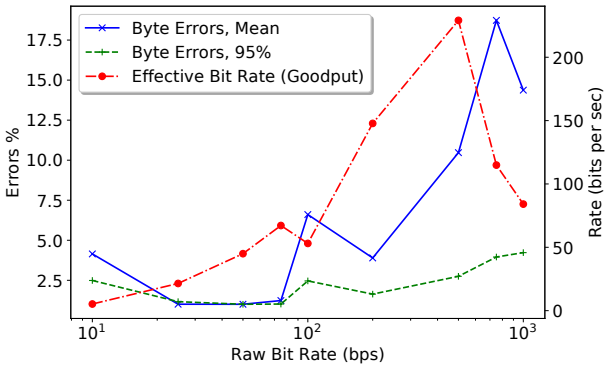

Figure 9: TODO ►*increase font*◄

5 PRACTICALITY OF COVERT COMMUNICATION

In this section, we present an evaluation on the practicality of a lambda covert channel that is discovered with our co-residence detector. The amount of information that can be transferred depends on two factors: 1) the channel capacity and 1) 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. [16], 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 estimae of the capacity of each covert channel.

Our protocol for data transfer uses the same mechanisms used for co-residence detection in section TODO ►*ref 5.2*◄ to send and receive bits and perform clock synchronization. However, since

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 concerened about noise from multiple receivers, and as such can allow the receiver to sample continuously (TODO ►*ref*◄) and sample at extremely small intervalls (milliseconds instead of seconds). While we want the sampling interval 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 9 shows the mean error ratio at 50% and 95% confidence intervals, both of which increase with the bitrate.

To correct these errors, we can use block-based error correction codes like Reed-Solomon, which uses byte-sized encoded symbols [16] as descheduling may result in burst errors Ariana ►*do we need to define this? is this important?*◄. However, error correction comes with an overhead; Reed-Solomon requires twice as many extra symbols Ariana ►*why dont we just use the term bytes here instead of symbols?*◄ as there are errors to correct. So, for each bitrate, we must compute effective bitrate by subtracting the overhead of error correction symbols. From Figure 9, we can see that effective bitrate rises to a maximum of over 200 bits per second (bps) (at 500bps 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 Rendezvous point density

TODO ►*Changes due to above subsection*◄ Next, we present measurements on serverless function density on AWS using our co-residence detector, and discuss the factors that may affect this density. As we discussed earlier, the key challenge in enabling traditional covert channels on the cloud is identifying which lambdas are on the same machine. 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 pairs 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 lambdas that end up co-resident on each servern in figure TODO ►.◄ 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. Ariana ►*does this make sense here as a sub-subsection?*◄ TODO ►*clarify the attacker model here, per R1 comments*◄ We execute our co-residence detector with 1000 1.5 GB Lambdas in various AWS regions. Figure 7 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 (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.

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 we 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 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 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 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 Figure 10.

6 DISCUSSION

Alternate use cases Our main motivation behind proposing a co-residence detector for lambdas is demonstrating the feasibility of a covert channel. 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-resident instances as necessary.
- Previous studies on performance aspects of lambdas (like performance isolation) [14] 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 [5] 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

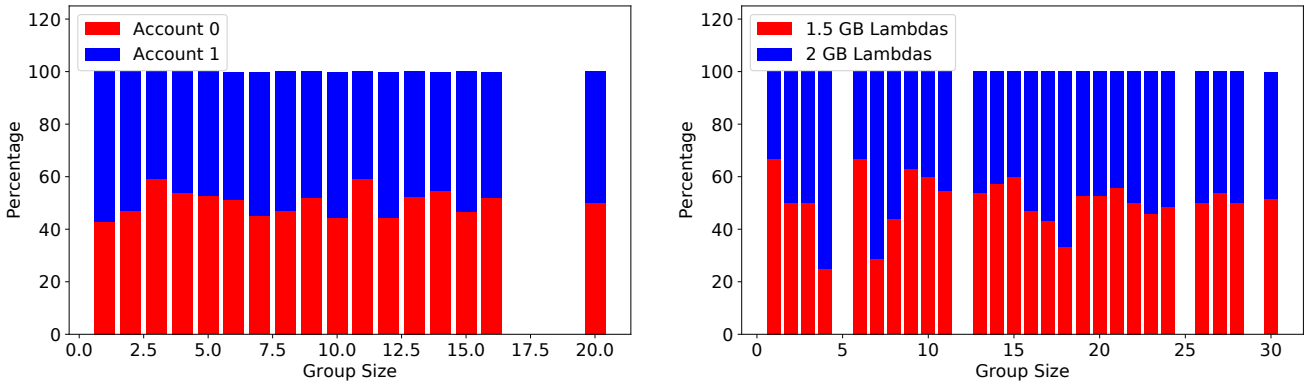

Figure 10: The left plot shows the breakdown of co-resident groups (of varying sizes) of lambdas by two different accounts in an experiment of 1000 lambdas, where 500 lambdas are launched from each account. The uniformity of the split indicates that the lambda scheduler might be invariant to the account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

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, cloud providers may look at more expensive solutions like BusMonitor [11] that isolate memory bus usage for different tenants by trapping the atomic operations to the hypervisor. We leave such exploration to future work. TODO ►*small note on limitations and future work?*◄

7 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 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.

8 CONCLUSION

In this paper, we have demonstrated techniques to build robust, scalable and efficient 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 channels by studying the co-residence density of lambdas on various AWS regions. TODO ►*add acknowledgements*◄

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] azure 2019. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[5] 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
[6] gcp 2019. Google Cloud Functions. https://cloud.google.com/functions/.
[7] Lambda 2019. AWS Lambda. https://aws.amazon.com/lambda/.
[8] 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
[9] 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
[10] 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
[11] Brendan Saltaformaggio, D. Xu, and X. Zhang. 2013. BusMonitor : A Hypervisor-Based Solution for Memory Bus Covert Channels.
[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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.