Columbus: Fast, Reliable Co-residence Detection for Lambdas

Anonymous Author(s)

ABSTRACT

Cloud computing has seen explosive growth in the past decade. While efficient sharing of infrastructure among tenants has contributed to this growth, the same principles also open avenues for covert-channels, or the capability to share information between processes that should be hypothetically isolated. Providers like AWS and Azure have traditionally relied on obfuscation to prevent this leakage of information, but recent works have repeatedly found detection techniques that break this encapsulation, prompting the providers to harden isolation on their platforms. In this work, we find yet another such covert-channel for lambdas based on the memory bus, which is more pervasive, reliable, and harder to fix. We show that this channel can be used to reliably perform co-operative co-residence detection for thousands of AWS lambdas within a few seconds, which could aid attackers in performing DDoS attacks or learn cloud's internal mechanisms. In this paper, we present the technique in detail, evaluate it, and use it to perform a measurement study on lambda activity across AWS regions. Through this work, we hope to motivate the need to address this covert channel.

CCS CONCEPTS

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

KEYWORDS

cloud, cartography, serverless, coresidency, covert channels

ACM Reference Format:

Anonymous Author(s). 2020. Columbus: Fast, Reliable Co-residence Detection for Lambdas. In *The Web Conference 2021, April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cloud computing is a fast growing technology that is being widely used around the globe TODO ►*cite; can also throw in numbers to make this more convincing*◄. Major cloud providers like AWS [4], Microsoft Azure [5] and Google Cloud [10] provide a multitude of compute and storage services and have seen immense growth over the past decade TODO ►*cite*◄. While there are many benefits of using cloud services, like increased scalability and decreased IT costs [2], cloud also brings new security concerns to the table as multiple tenants share the same underlying physical infrastructure. For example, by multiplexing applications from multiple tenants on the same server, cloud computing opens up new avenues and opportunities for side-channel attacks through shared server hardware like caches [15, 24]. While the virtualization platforms used to share the infrastructure have been constantly improving to harden the isolation between the tenants, it's a never ending process and new attack surfaces keep showing up TODO ►*cite, HELP!*◄. Also, hardening isolation usually comes at the cost of performance which may offset the benefits of sharing the infrastructure that enabled cloud computing in the first place.

Traditionally, clouds have relied on hiding their placement mechanisms (i.e., how they pack tenants onto their servers) and coresidence information (i.e., which particular tenants are packed together in a server) as a first line of defence against targeted attacks. By reducing attacker's ability to get on the same server as the victim, attackers may have to go with trial-and-error or brute-force solutions that can be very expensive with low yield. However, the encapsulation mechanisms are not perfect. A plethora of previous work, kick-started by Ristenpart et al. [18] a decade ago, have exploited various covert channels to enable co-residency detection on clouds like AWS, and used them to demonstrate targeted attacks or shed light on cloud's internal placement and resource allocation mechanisms TODO ►*cite*◄ that further weaken the encapsulation. Clouds have since promptly fixed many of these covert channels and came up with containers that provide better isolation, like AWS Firecracker[1] for example. In this work, we set out to prove that the job is not done yet.

We found yet another way to detect coresidency among cloud instances (i.e., whether they are running on the same physical server), one which is more pervasive, reliable and harder to fix than with previous approaches. We use a covert channel based on memory bus hardware, first introduced by Wu et al.[23], which we show to be omni-present in all clouds. Unlike software-based covert channels that can be fixed with new releases, this one is inherent to x86 hardware and is harder to fix. Finally, while previous approaches have made simplistic use of this covert channel[20] due to noise and synchronization issues, we overcome these to communicate bits reliably over the channel that enabled us to achieve lightning-fast colocation detection for thousands of cloud instances.

In this work, we chose serverless functions as our cloud containers of choice. Serverless functions have seen increased interest in recent years with most clouds providing these services, such as lambdas on AWS [14] and cloud functions on GCP [9] (We use the terms lambdas, serverless functions and cloud instances interchangeably hereafter). They provide a more challenging environment for coresidence detection techniques based on covert channels as they usually have restricted runtimes that limit low-level code sometimes required to access such covert channels, and are more ephemeral (forcing the coresidence detection to be faster). They are also significantly cheaper, providing us a cost-effective way to demonstrate the technique. Note that whatever we are able

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

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

to achieve with lambdas can be easily replicated with other traditional containers, as they are less, not more, restricted environments.

In this paper, we only focus on *cooperative* coresidence detection, to start with. That is, all the lambdas are assumed to be in our/attacker's control. While this doesn't by itself help attacker target a victim, it can be used to perform targeted DDoS attacks TODO ▶*cite power attacks?*◀ or learn cloud's internal mechanisms that may aid this. We propose a technique which, for given a set of lambdas deployed onto the cloud, can figure out which of them ran on the same server in the cloud. We show that we can do this with 100% success rate for bigger lambdas and furthermore, we can do this within a minute for thousands of lambdas. We then perform a minor study on colocation patterns in various AWS regions with some insights on lambda activity. TODO ▶*any other takeaways?*◀ Through this, we hope to motivate the need to address the memory bus covert channel in all the three clouds.

The remainder of this paper is organized as follows. Section 2 presents some background from the literature. Sections 3 and 4 present our co-residence detection mechanism in detail. We evaluate the mechanism in section 5 and conclude with a placement study of AWS lambdas using our technique in section 6.

2 BACKGROUND

We begin with a brief background on relevant topics.

2.1 Lambdas/Serverless Functions

TODO ▶*needs sprucing*◀ One of the fast-growing cloud services in recent years are serverless functions such as lambdas on AWS [14] and cloud functions on GCP [9]. This interest stems from the fact that serverless architecture does not require the developer to worry about provisioning, maintaining, and administering servers. Additionally, lambdas are much more cost-efficient than VMs as they allow more efficient packing of the servers. However, these functions are more ephemeral than containers and VMs, in many cases only few minutes. While this attribute provides more flexibility in cost and functionality, the nature of serverless functions also increases the difficulty in detecting co-residency and launching successful attacks.

We focus on AWS Lambdas in this paper but we show that our study is applicable to other clouds as well. Lambdas execute as much smaller units than containers and virtual machines. Lambda sizes range from 128 MB to 3 GB, and their maximum timeout value is 15 minutes. While lambdas are limited in the computations they can execute, they are conversely incredibly lightweight and can be initiated and deleted in a very short amount of time. Since lambdas are short-lived and lightweight, the user has no control over the physical location of the server(s) on which their lambdas are spawned.

2.2 Co-residence Detection

To perform side-channel attacks against other tenants in a cloud setting, attackers need to co-locate their applications on the same servers as their victims. Past research has used various strategies to achieve co-residency for demonstrating such attacks. Typically, achieving co-residency includes a (VM/Container) launch strategy (varying number of instances, time of the day, etc) combined with a co-residence detection mechanism for detecting if two instances are running on the same machine. Traditionally, this was done based on software runtime information like public/internal IP addresses[18], files in *procfs* or other environment variables[22] and other such logical side-channels[19, 26] that two instances running on a same server might share.

As virtualization platforms move towards stronger isolation between instances (e.g. AWS' Firecracker VM [1]), these logical side-channels have become less effective or infeasible. Furthermore, some of these side-channels were only effective on container-based platforms that share the underlying OS image and were less suitable for hypervisor-based platforms. This prompted a move towards hardware-based covert channels which can bypass software isolation and are usually harder to fix. Typically, these covert channels involve sending/receiving information by causing contention on a shared hardware that results in observable performance fluctuations across applications. A number of such side-/covert channels based on shared hardware like last-level caches TODO ▶*references*◀, memory bus [20, 23, 29] and storage devices TODO ▶*references*◀ have been explored in the past, some of which have already been addressed and are no longer even feasible in most clouds (e.g., last-level cache-based channels[25]).

2.3 Memory Bus Covert Channel

One shared hardware that we examine in our study is the memory bus. The memory bus is the piece of hardware that connects the memory controller to main memory. Memory bus contention can be caused by initiating repeated memory accesses to saturate the bus bandwidth and cause observable latency spikes. However, this turns out to be very challenging given the multiple levels of caches on today's servers, which prevent repeated memory accesses, and the high memory bandwidth that cannot be saturated by few CPU cores.

In x86 systems, atomic memory instructions designed to facilitate multi-processor synchronization are supported by cache coherence protocols as long as the operands stay within a cache line (which is 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 the other operations which cannot use the ample memory bandwidth due to the lock[23]. Furthermore, this behaviour persists even in the presence of multiple processor sockets, making the locking effects visible to all the cores on the machine. We exploit this property of x86 hardware to cause contention on the memory bus and use the resulting observable variations in performance as a covert channel for detecting co-residency.

3 METHODOLOGY

Ariana ▶*make sure that all these terms are defined ahead of time: co-location, co-operative, co-residence, serverless, lambdas*◀

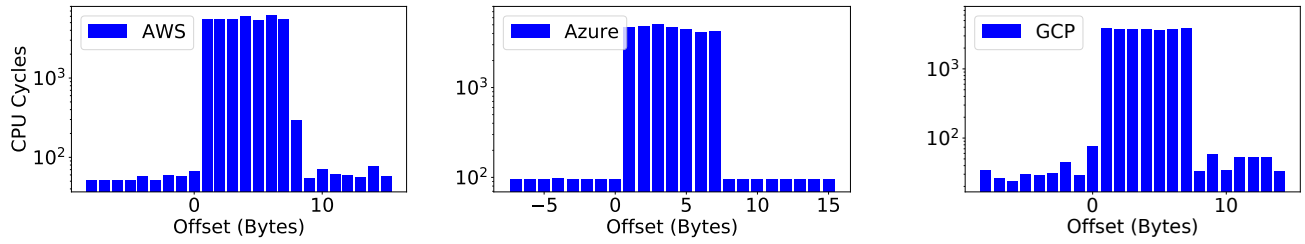

Figure 1: From left to right, the plots show the latencies of atomic memory operations performed on an 8B memory region as we slide it from one cache line across the boundary into another, on AWS, Azure and Google (GCP) clouds respectively. The latencies are orders of magnitude higher when the 8B region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

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

Given a set of cloud instances (VMs, Containers, Functions, etc) deployed to a public cloud, a co-residence detection mechanism would identify, for each pair of instances in the set, whether the pair was running on the same physical server at some point. Paraphrasing Varadarajan et al.[20], for a co-detection mechanism to be useful across a wide range of launch strategies, we observe that it should have the following desirable properties:

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

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

3.0.1 Memory bus channel. We chose the memory bus covert channel described in section 2.3 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 [19, 29], and we found that they are still vulnerable today. To demonstrate the presence of the vulnerability, we measure the latency of atomic operations on a 4B memory region as we slide the region from one cacheline into another across the cacheline boundary. We perform this experiment on three major clouds (AWS, Google and Microsoft Azure) and show the latencies observed in Figure 1. From the figure, we can see that all three clouds still exhibit significant difference in latencies for the "exotic" memory locking operations (where the memory region falls across cacheline boundary) when compared to regular memory accesses, demonstrating the presence of this covert channel on all of them. Moreover, we were able to run these experiments on serverless function instances. Since serverless function instances have runtimes that are generally restricted to high-level languages (that prevent the pointer arithmetic required to perform these exotic operations), we used the unsafe environments on these clouds — C++ on AWS, Unsafe Go on GCP, Unsafe C# On Azure. This shows the applicability of using the covert channel across different kinds of cloud instances as well.

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

An alternative method is to try pair-wise detection where only one attack instance locks and one victim instance probes at a time revealing co-residence of this pair, and repeating this serially for each pair. However, this technique is too slow and scales quadratically with the number of instances e.g., a hundred instances take more than 10 hours assuming 10 secs for each pair. Varadarajan et al.[19] speeds up this process significantly by performing detection for mutually-exclusive subsets in parallel, allowing for false-positives and later eliminating the false-positives sequentially Ariana ►*might want to elaborate on this; on its own may be a bit confusing*◄. This

would still only scale linearly in the best case, which is still expensive; with a thousand instances, for example, the whole detection process takes well over 2 hours to finish, which is infeasible for lambdas that are, by nature, ephemeral. Thus, one challenge in this work is creating a faster neighbor detection algorithm.

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

Previous work utilized the memory bus channel to exchange more complex information like keys [23]. At first sight, it appears that the memory bus can be used to exchange information, such as unique IDs, between the co-resided instances, which would allow us to ascertain victim/attacker pairs. However, the previous work assumes that there is only one sender and one receiver that know exactly how and when to communicate. As we will see in the next section, this model is not sustainable when there exist many parties that have no knowledge of each other but try to communicate on the same channel.

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

4 NEIGHBOR DISCOVERY PROTOCOL

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

Algorithm 1 Writing 1-bit from the sender

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

4.1 Reliable Transmission

First, we must determine how to reliably transmit information between the sender and receiver. Senders and receivers can accurately communicate 0-bits and 1-bits by causing contention on the memory bus. Consider the simple scenario where there is one sender and one receiver instance on a machine, and the sender has a set of bits that it needs to communicate with the receiver via the memory bus covert channel. To communicate a 1-bit, the sender instance causes contention on the memory bus by locking it using the special memory locking operations (discussed in section 2.3). The receiver would then sample the memory bus for contention, inferring whether the communication is a 1-bit (when contention is observed) or a 0-bit (when contention is not observed). Pseudo-code for the sender instance is shown in Algorithm 1.

4.1.1 Sensing contention. Next, we determine how best for the receivers to sense contention on the memory bus, given that there are two available methods. 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 [20]) and observe a spike in their latencies to detect contention. 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[20, 23] have established that both memory probing and memory locking receivers experience significant latency overhead during memory bus contention, making them both viable avenues for sensing the covert-channel. Memory probing involves regular (un-cached) memory accesses, which is universal, unlike the locking operations which are rarely used, if at all, by standard applications. This makes memory probing the only viable option for **non-cooperative** co-residence detection, where victims are not under attacker's control and cannot be assumed to perform locking operations. Furthermore, memory probing can be done on multiple receivers constantly without affecting each other (due to the high memory bandwidth), which prevents noise in measurements. This is an important attribute, as memory locking receivers must contend with this noise. However, bypassing multi-levels of caches in today's servers to perform memory accesses with reliable consistency is a challenging task TODO ►*cite some papers*◄. 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

Algorithm 2 Reading a bit in the receiver

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

suit these architectures an arduous task while reducing the applicability of our overall co-residence detection mechanism.

4.1.2 Sampling frequency. Another challenge for our protocol is determining an adequate sampling frequency. Ideally, a memory locking receiver would loop locking operations and determine contention in real-time by identifying a decrease in the moving average of the number of operations. Note that, in this case, there is essentially no difference between the sender and receiver (i.e., both continually issue locking operations) except that the receiver is taking measurements. This is adequate when there is a single sender and receiver [20], but when there are multiple receivers, the mere act of sensing the channel by one receiver causes contention and other receivers cannot differentiate between a silent (0-bit) and a locking (1-bit) sender. To avoid this, we space the sampling of memory bus such that no two receivers would sample the bus at the same time, with high probability. We achieve this by using large intervals between successive samples and a poisson-sampling to prevent time-locking of receivers. We determined that a millisecond poisson gap between samples is reasonable to minimize noise due to collisions in receiver sampling 1, assuming ten co-resided receivers and a few microsecond sampling time.

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

To illustrate such behavior, we run a sender-receiver pair using lambdas[14] of various sizes on AWS, and compare the distribution of latencies seen by the receiver during the contention in each

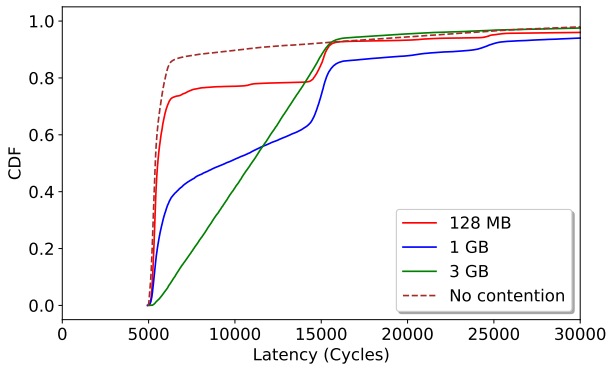

Figure 2: Shows CDF of latencies observed by 128 MB, 1 GB and 3 GB Lambdas during contention. The 128 MB lambda pair sees less contention due to more context switching, whereas the 1 GB and 3 GB lambdas see progressively more contention compared to baseline which we attribute to their relative stability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

case. Figure 2 shows that the much smaller 128 MB lambdas (which probably share a CPU core and are thus context-switched) exhibit less active communication than the bigger 3 GB lambdas (which may run on dedicated cores). This means that smaller instances that tend to share processor cores with many other instances may need to pause for more time and collect more samples to make up for lost communication due to scheduling.

4.1.4 Overcoming noise. Along with context switching and sensing noise, there are other imperfections in the measurement apparatus that may cause noise. For example, we use the difference in readings from the timestamp counter of the processor (RDTSC) before and after the locking operation to measure the latency of the operation in cycles. If the receiver process is context-switched in between the timer readings (e.g., at line 8 in Algorithm 2), the latency measured from their difference will be orders of magnitude higher as it includes the waiting time of the receiver process in the scheduler queue - which we believe is what contributes to the long tail in Figure 2. To overcome missed samples and noise, we record hundreds of samples and compare it to the baseline distribution of latencies sampled without contention. We then need to compare and differentiate the observed sample of latencies from the baseline to establish contention. To do this, we use a variant of the two-sample Kolomogorov-Smirinov (KS) test that takes the maximum of absolute difference between empirical CDFs of the samples as measure of comparison ((In our variant, we take the *mean* of absolute difference instead of the maximum)). Using this measure, we can categorize a KS-value above a certain threshold as a 1-bit (contention) and a value below the threshold as 0-bit (baseline).

To determine the KS-threshold, we deploy a large number of lambdas across AWS regions. Some of these lambdas cause contention (aka senders) while others observe contention by collecting samples of latencies (aka receivers). Each of the samples may or may not have observed contention depending on whether the

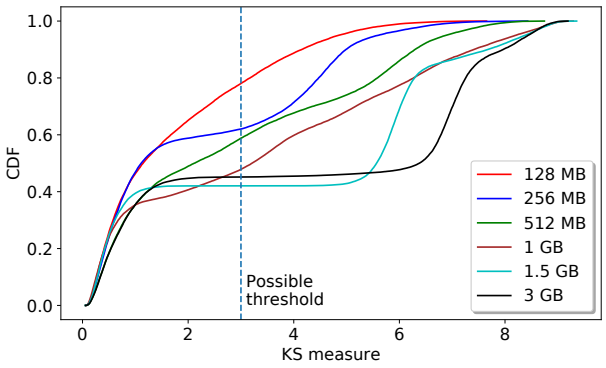

Figure 3: Shows CDF of KS values observed for various lambda sizes. A bimodal distribution with longer step lets us pick a KS-threshold that enables our technique to differentiate between 0-bit and 1-bit with high confidence.

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

We present the pseudo-code of a receiver lambda in Algorithm 2, which includes all the challenges and subsequent solutions discussed thus far. **Ariana** ►*I'm not a huge fan of this sentence...need to find a way to rework*◄

4.1.5 Clock synchronization. Since communicating each bit of information takes time (i.e., receiver sampling duration), our algorithm requires synchronizing sender and receiver at the start of each bit. In traditional analog channels, this is achieved either using a separate clock signal or a self-clocking signal encoding. For example, prior work [**?**] uses differential Manchester encoding for clock synchronization for the memory bus covert channel. Using self-clocking encodings becomes more challenging when there are multiple senders and receivers. In this work, we use the system clock for synchronizing communication. All the instances involved in the communication are executing on the same physical server and share the server's clock. On AWS, for example, we observe that the system clock on lambdas is precise up to nanoseconds with a sub-microsecond drift between different lambdas running on the same server, which is accurate enough for our purposes as the sampling noise constraints limit our measurements to milliseconds. **Ariana** ►*is there some sort of citation for this?*◄

4.2 Protocol

Ariana ►*still feels like we are talking about challenges...no good fix yet*◄ In the preceeding section, we discussed using a communication channel with synchronized time slots. In each time slot, an instance can reliably send (broadcast) or receive (listen) a bit by causing or sensing for contention. Given that there are multiple instances that may want to broadcast information on the channel, we next must determine which instance broadcasts first, to avoid collisions. Traditional channels like Ethernet or Wireless detect and avoid collisions by employing a random exponential backoff mechanism. Such a mechanism will be challenging to implement in case of our channel for two reasons. First, lambda instances do not have the capability of sensing the channel while sending a bit, which is required for detecting collisions; instances can either cause contention or sense it, but not both. Note that senders do experience a higher latency for locking operations when other senders are simultaneously causing contention. However, reliably judging this higher latency requires each sender to already have calculated a baseline of latencies without collisions, which takes us back to the original problem of avoiding collisions. Second, even implementing a random or exponential backoff mechanism will introduce significant overhead before any meaningful communication occurs. This overhead will also increase as the number of instances involved increases. Since each time slot could take up to 1 second, this additional overhead can make the whole communication very slow.

Ariana ►*I think in general we need a bit more sign-posting to guide the reader. I will think about this a bit more*◄. We make the insight that a communication channel for lambda co-residence detection need not be general and expressive as lambdas only need to communicate their IDs with one another. Thus, we assume that each instance involved has a unique fixed-length (say n) bit-string corresponding to its ID that must be communicated. As such, we propose a communication protocol that exchanges these bit-strings while allowing for collisions. We divide the running time of the protocol into phases, with each phase executing for an interval of n bit-slots. Each phase has a set of participating instances, which in the first phase would be all of the co-located instances. In each bit-slot k of n slots in a phase, every participating instance broadcasts a bit if the k^{th} bit of its bit-string (ID) is 1, otherwise it listens for a 0 or 1. If an instance senses a 1 while listening, it stops participating, and listens for the rest of the phase. Thus, only the instances with the highest ID among the initial set of participating lambdas continues broadcasting until the end of the protocol, effectively advertising its full ID to the rest of the (now listening) lambdas). In the next cycle, the lambda with the previously highest ID now only listens, allowing the next highest instance to advertise its ID, and so on. Since the IDs are unique, there will always be only one instance that broadcasts in every phase. The protocol ends after x phases (where x is number of co-located instances), when none of the instances broadcast for n consecutive bit-slots. A pseudo-code of the protocol is provided in Algorithm 3. Note that the protocol itself is channel-agnostic and can be extended for other (future) covert channels with similar channel properties.

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

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

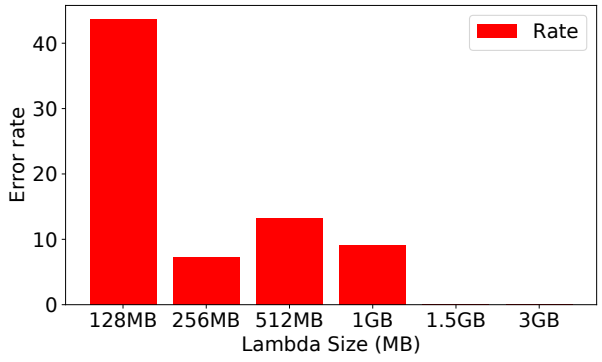

Figure 4: Shows the error rate (as a fraction of 1000 lambdas deployed) for different lambda sizes in AWS Middle-East region.

4.2.1 Complexity. Assuming N total deployed instances to the cloud, the bit-string needs to be $\log_2 N$ bits to uniquely identify each instance. If a maximum K of those instances are launched on the same server, the protocol executes for K phases of $\log_2 N$ bit-slots each, taking $(K+1) * \log_2 N$ bit-slots for the whole thing. For example, assuming 10,000 deployed lambdas and a maximum of 10 co-located instances on each server, the entire co-residence detection requires around four minutes to fully execute (with 1-second time slots). In fact, it is not necessary to run the protocol for all K phases. After the first round, all the co-located instances would know one of their neighbors. Ariana ▶*confused about this sentence*◀ Instances can exchange the globally unique IDs offline (through the network) to determine the rest of their neighbors. This simplification removes the dependency on number of co-located instances (K) and decreases the complexity to $O(\log_2 N)$, allowing the entire protocol to finish within a minute instead of four.

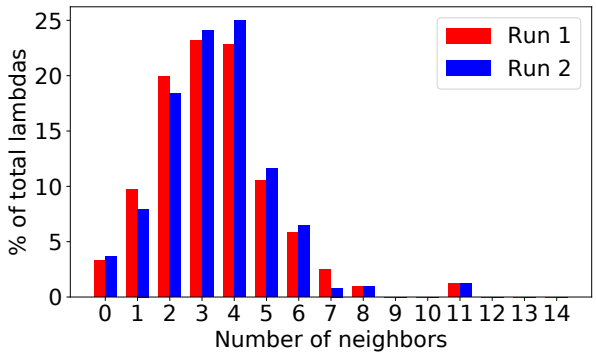

Figure 5: Shows the fraction of lambdas by the number of neighbors they saw for two independent runs that use same set of underlying AWS containers. The perfect correlation shows that both runs depict the colocation status of those containers regardless of the lambdas that ran on them, providing an evidence for the correctness of our technique. TODO ▶*this doesn't show perfect correlation because only 99% of lambdas warm started. Get another perfect run?*◀

5 EVALUATION

In this section, we evaluate the effectiveness of our co-residence detection technique with respect realiability and scability, the desirable detection properties mentioned in section 3. We run all of our experiments with AWS lambdas [4]. Though we decide to focus on one of the cloud providers, we have previously shown in section 3 that this covert channel exists on the other clouds, and thus these experiements can be replicated on their serverless functions as well. We use C++ runtime in AWS lambdas as it allows pointer arthmetic that is required to access the covert channel.

5.1 Setup

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

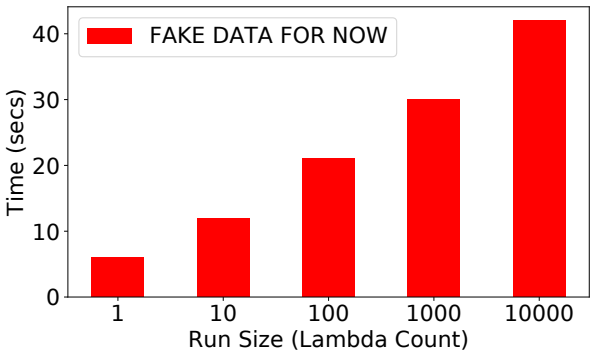

Figure 6: Shows the average runtime of a lambda for co-location runs of different sizes. The run time increases logarithmically with the number of lambdas as it is proportional to the number of bits required to uniquely identify all the lambdas. TODO ►*Get real data.*◄

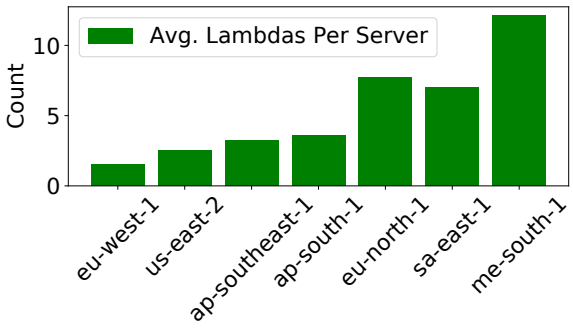

Figure 7: Shows the average number of lambdas per server i.e., colocation density seen in various AWS regions for a 1000-lambda run.

erroneous and report it in the error rate. We group all the instances that saw the same ID as successful and neighbors. We repeat the experiments for different lambda sizes and in various cloud regions.

5.2 Reliability

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

Correctness To determine correctness, we require ground truth on which instances are co-located with one another, which we are able to ascertain by utilizing an AWS caching mechanism. After execution, AWS caches the lambda containers to reuse them[3] for repeat lambdas and mitigate "cold start" latencies. For C++ lambdas, we found that the data structures declared in the global namespace are tied to containers and are not cleared on each lambda invocation, so we can use a global array to record all the lambdas that were ever run in a particular container. This means, for a given lambda, we can precisely tell all the lambdas that previously ran in the same container (aka predecessors). Using this, we are able to validate that identical experiments repeated within minutes of one another will use the same set of underlying containers for running the deployed lambdas. Since lambda co-location is essentially co-location of their containers, and given that containers persist across experiments that are executed within minutes of one another, lambda co-location results must agree with the co-location of their underlying containers for true correctness.

To demonstrate this principle, we run an experiment with 1000 1.5GB cold-started lambdas (ID'ed 1 to 1000) in one of densest AWS regions (AWS MiddleEast), which resulted in many co-located groups. We repeat the experiment within a few seconds, thereby ensuring that all 1000 lambdas are warm-started on the second trial (i.e., they use the same set of containers from the previous experiment). For each co-located group of lambdas in the latter experiment, we checked whether their predecessor lambdas (that used the same set of containers) in the former one formed a co-located group a well. Ariana ►*a bit lost here*◄ Anil ►*how about now?*◄ We observed that while the lambdas that used a set of containers differ across both experiments, the results of these experiments agree perfectly on the continer colocation. Figure 5 shows that both experiments saw the same number of groups of different sizes. This proves the correctness of our co-location results.

5.3 Scalability

One of the key properties of this technique is its execution speed. Since communicating each binary bit of the ID takes one second, we are able to scale the technique logarithmically with the number of lambdas involved. Figure 6 shows this principle for experiments involving different number of lambdas. For example, in an experiment with 10000 lambdas, each lambda can find its neighbors within a minute of its invocation, leaving ample time to use this information for nefarious purposes. The logarithmic scale of our method also indicates that the cost per lambda scales logarithmically, making neighbor detection cost-effective.

6 PLACEMENT STUDY

Our goal in developing this technique is to demonstrate how easy it is for attackers to exploit the pervasive memory bus covert channel and obtain co-residency information, thereby motivating the need to address it. This information can be used by attackers in aiding a lot of attack scenarios TODO ►*for example?*◄ or simply learn the internal mechanisms of a cloud. In the section, we explore some ways in which the tool can be used to gain some insights into lambda activity in some AWS regions. Unless specified otherwise,

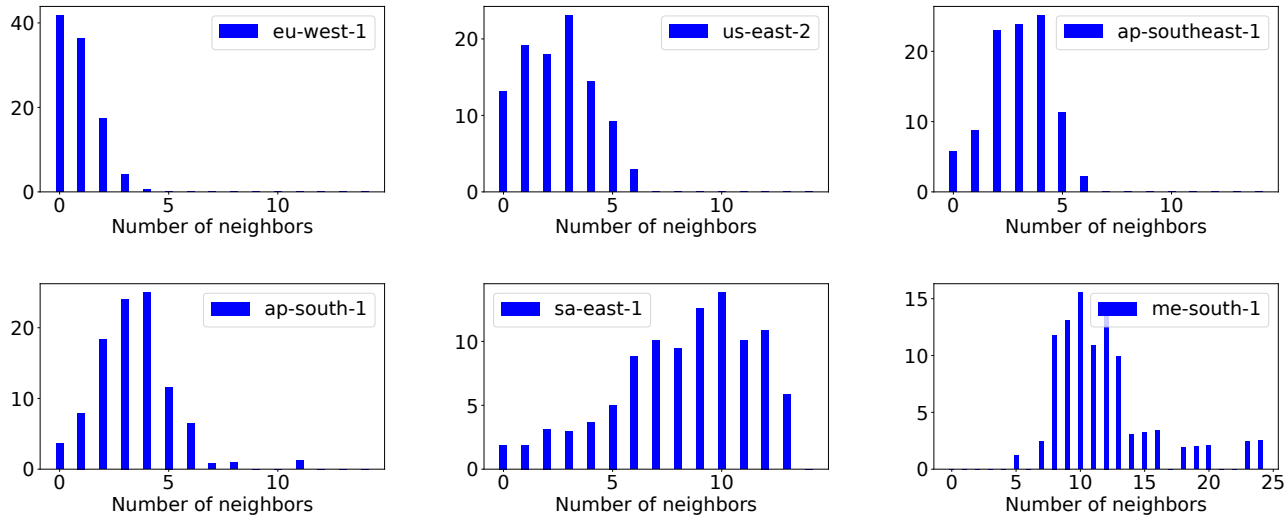

Figure 8: Shows co-location results for a 1000-lambda run in different AWS regions. Each bar shows the fraction of those 1000 lambdas (in %) that saw a certain number of neighbors. The total amount and density of co-location vary widely across regions, perhaps based on the size and lambda activity within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

all the experiments are performed with 1.5 GB lambdas and ran successfully with zero error rate. We find that TODO ►*summary of takeaways from the study*◄.

6.1 Co-residence across AWS regions

We ran co-residence detection in different AWS regions with 1000 1.5GB Lambdas. Figure 8 shows multiple plots showing the colocated groups, one per region, with each bar in the plot showing the fraction of lambdas that saw a certain number of neighbors (i.e., that belong to a colocated group of certain size). Plots that are right-heavy (towards bottom-right) indicate higher colocation density compared to the left-heavy (towards top-left) ones (which is also illustrated in figure 7). We can see that most regions have almost all lambdas see at least one neighbor (smaller or non-existent bar at 0). Assuming that the cloud placement scheduler tries to efficiently bin pack lambdas, we hypothsize that the colocation is 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. We note that most right-heavy plots correspond to the relatively newer AWS regions. The maximum size of a co-located group we ever saw was 25 (1.5G) lambdas on a single machine. TODO ►*any other insights I missed?*◄

6.2 Weekly & Daily Patterns

TODO ►*Run experiments in different times of the day or using different deployment strategies that may affect co-location.*◄

6.3 Different accounts

TODO ►*Run experiments with lambdas from different user accounts and see how the colocation is affected between single vs different accounts.*◄

7 DISCUSSION

TODO ►◄ What are the limitations of the technique? What can an attacker do with this colocation information? How can AWS prevent this? Something about firecracker? Helps performance isolation studies such as [22]. This technique can be used with other containers like VMs. While the co-operative coresidence detection itself does not directly help attackers locate their victims, it can help attackers perform highly sophisticated DDOS attacks once they did find the victim.

8 RELATED WORK

TODO ►*needs polishing*◄

Covert Channels & Cloud Attacks Co-residency is possible because of covert channels, so we begin our related work with an investigation into covert channels and cloud attacks. Initial papers in co-residency detection utilized host information and network addresses arising due to imperfect virtualization [18]. However, these covert channels are now obsolete, as cloud provides have strengthened virtualization and introduced Virtual Private Clouds TODO ►*cite*◄. Later work used cache-based channels in various levels of the cache [12, 16, 24, 32] and hardware based covert channels like thermal covert channels [17], RNG module [7] and memory bus [23] have also been explored in the recent past. Moreover, studies have found that VM performance can be significantly degraded using memory DDoS attacks [28], while containers are susceptible to power attacks from adjacent containers [8]. Our work focuses on using the memory bus as a covert channel for determining cooperative co-residency. . Ariana ►*work this in?*◄Covert channels using memory bus were first introduced by Wu etl. al [**?**], and subsequently has been used for co-residency detection on VMs and Containers [19**?**] Wu et. al [**?**] introduced a new technique to lock the memory bus by using atomic memory operations on addresses

that fall on multiple cache lines. .

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) [18]. This initial work was further expanded in subsequent years to examine co-residency using memory bus locking [27] and active traffic analysis [6], as well as determining placement vulnerabilities in multi-tenant Public Platform-as-a-Service systems [20, 30]. Finally, Zhang et al. demonstrated a technique to detect VM co-residency detection via side-channel analyses [31]. Our work expands on these previous works by investigating co-residency for lambdas.

Lambdas TODO ▶*pretty rough paragraph...needs a rewrite*◀ While lambdas are a newer technology than VMs, there still exists a variety of literature. For example, recent studies examine cost comparisons of running web applications in the cloud on lambdas versus other architectures [21]. Moreover, lambdas have been studied in the context of cost-effective batching and data processing [13]. Further research has shown how lambdas perform with scalability and hardware isolation, indicating some flaws in the lambda architecture [22]. From a security perspective, Izhikevich et. al examined lambda co-residency using RNG and memory bus techniques (similar to techniques used looking at VM co-residency) [11].. However, our work differs from this study in that our technique informs the user of which lambdas are on the same machine, not only that the lambdas experience co-residency.

9 ETHICAL CONSIDERATIONS

As with any large scale measurement project, there are ethical considerations to take into account. First, there are security and privacy concerns of using this technique to uncover other consumer's lambdas. However, since we focus on co-operative co-residence detection, we only determine co-location for the lambdas we launched, and do not gain insight into other consumer's lambdas. Secondly, 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. Moreover, memory accesses and locking operations are FIFO, which prevents starvation to the multiple processes 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, by definition, be affected in their longterm computational goals. Ariana ▶*is lambda cost by second? is it possible that by causing a longer lambda runtime, we are costing someone more money?*◀

10 CONCLUSION & FUTURE WORK

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

REFERENCES

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

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. https://doi.org/10.1145/1721654.1721672

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

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

[5] Azure 2019. Azure VMs. https://azure.microsoft.com/en-us/services/virtual-machines/.

[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] 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.

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

[10] GoogleCloud 2019. Google Compute. https://cloud.google.com/products/compute/.

[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] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[16] 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

[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] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

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

[29] 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.

[30] 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

[31] 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

[32] 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