

# 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/nnnnnnnnnnnnnn>

## 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

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

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

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnnnnnnnnn>

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

117 to achieve with lambdas can be easily replicated with other traditional  
 118 containers, as they are less, not more, restricted environments.  
 119

120 In this paper, we only focus on *cooperative* coresidence detection,  
 121 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  
 122 target a victim, it can be used to perform targeted DDoS attacks  
 123 **TODO** ► cite power attacks?◀ or learn cloud's internal mechanisms  
 124 that may aid this. We propose a technique which, for given a set of  
 125 lambdas deployed onto the cloud, can figure out which of them ran  
 126 on the same server in the cloud. We show that we can do this with  
 127 100% success rate for bigger lambdas and furthermore, we can do  
 128 this within a minute for thousands of lambdas. We then perform a  
 129 minor study on colocation patterns in various AWS regions with  
 130 some insights on lambda activity. **TODO** ► any other takeaways?◀  
 131 Through this, we hope to motivate the need to address the memory  
 132 bus covert channel in all the three clouds.  
 133

134 The remainder of this paper is organized as follows. Section 2  
 135 presents some background from the literature. Sections 3 and 4  
 136 present our co-residence detection mechanism in detail. We eval-  
 137 uate the mechanism in section 5 and conclude with a placement  
 138 study of AWS lambdas using our technique in section 6.  
 139

## 140 2 BACKGROUND

141 We begin with a brief background on relevant topics.  
 142

### 143 2.1 Lambdas/Serverless Functions

144 **TODO** ► needs sprucing◀ One of the fast-growing cloud services in  
 145 recent years are serverless functions such as lambdas on AWS [14]  
 146 and cloud functions on GCP [9]. This interest stems from the fact  
 147 that serverless architecture does not require the developer to worry  
 148 about provisioning, maintaining, and administering servers. Addi-  
 149 tionally, lambdas are much more cost-efficient than VMs as they  
 150 allow more efficient packing of the servers. However, these func-  
 151 tions are more ephemeral than containers and VMs, in many cases  
 152 only few minutes. While this attribute provides more flexibility in  
 153 cost and functionality, the nature of serverless functions also in-  
 154 creases the difficulty in detecting co-residency and launching suc-  
 155 cessful attacks.  
 156

157 We focus on AWS Lambdas in this paper but we show that our  
 158 study is applicable to other clouds as well. Lambdas execute as  
 159 much smaller units than containers and virtual machines. Lambda  
 160 sizes range from 128 MB to 3 GB, and their maximum timeout value  
 161 is 15 minutes. While lambdas are limited in the computations they  
 162 can execute, they are conversely incredibly lightweight and can be  
 163 initiated and deleted in a very short amount of time. Since lamb-  
 164 das are short-lived and lightweight, the user has no control over  
 165 the physical location of the server(s) on which their lambdas are  
 166 spawned.  
 167

### 168 2.2 Co-residence Detection

169 To perform side-channel attacks against other tenants in a cloud  
 170 setting, attackers need to co-locate their applications on the same  
 171 servers as their victims. Past research has used various strategies  
 172 to achieve co-residency for demonstrating such attacks. Typically,  
 173 achieving co-residency includes a (VM/Container) launch strategy  
 174

175 (varying number of instances, time of the day, etc) combined with  
 176 a co-residence detection mechanism for detecting if two instances  
 177 are running on the same machine. Traditionally, this was done  
 178 based on software runtime information like public/internal IP addresses[18],  
 179 files in *procfs* or other environment variables[22] and other such  
 180 logical side-channels[19, 26] that two instances running on a same  
 181 server might share.  
 182

183 As virtualization platforms move towards stronger isolation be-  
 184 tween instances (e.g. AWS' Firecracker VM [1]), these logical side-  
 185 channels have become less effective or infeasible. Furthermore, some  
 186 of these side-channels were only effective on container-based plat-  
 187 forms that share the underlying OS image and were less suitable  
 188 for hypervisor-based platforms. This prompted a move towards  
 189 hardware-based covert channels which can bypass software isolat-  
 190 ion and are usually harder to fix. Typically, these covert channels  
 191 involve sending/receiving information by causing contention on  
 192 a shared hardware that results in observable performance fluctua-  
 193 tions across applications. A number of such side-/covert channels  
 194 based on shared hardware like last-level caches **TODO** ► references◀,  
 195 memory bus [20, 23, 29] and storage devices **TODO** ► references◀  
 196 have been explored in the past, some of which have already been  
 197 addressed and are no longer even feasible in most clouds (e.g., last-  
 198 level cache-based channels[25]).  
 199

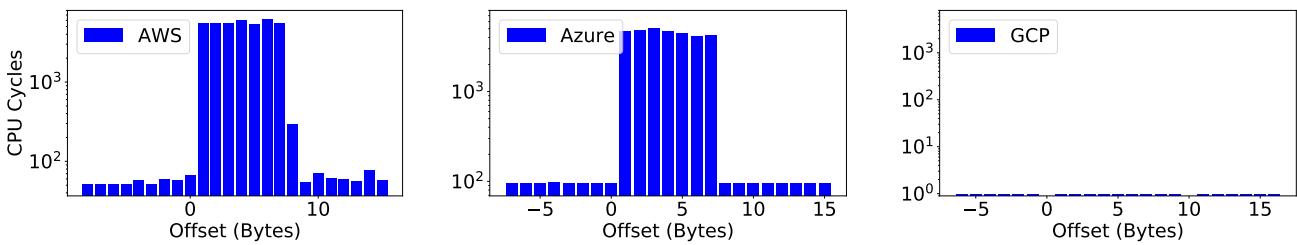
### 200 2.3 Memory Bus Covert Channel

201 One shared hardware that we examine in our study is the memory  
 202 bus. The memory bus is the piece of hardware that connects the  
 203 memory controller to main memory. Memory bus contention can  
 204 be caused by initiating repeated memory accesses to saturate the  
 205 bus bandwidth and cause observable latency spikes. However, this  
 206 turns out to be very challenging given the multiple levels of caches  
 207 on today's servers, which prevent repeated memory accesses, and  
 208 the high memory bandwidth that cannot be saturated by few CPU  
 209 cores.  
 210

211 In x86 systems, atomic memory instructions designed to facil-  
 212 itate multi-processor synchronization are supported by cache co-  
 213 herence protocols as long as the operands stay within a cache line  
 214 (which is generally the case as language compilers make sure that  
 215 operands are aligned). However, if the operand is spread across  
 216 two cache lines (referred to as "exotic" memory operations), x86  
 217 hardware achieves atomicity by locking the memory bus to pre-  
 218 vent any other memory access operations until the current opera-  
 219 tion finishes. This results in significantly higher latencies for the  
 220 other operations which cannot use the ample memory bandwidth  
 221 due to the lock[23]. Furthermore, this behaviour persists even in  
 222 the presence of multiple processor sockets, making the locking ef-  
 223 fects visible to all the cores on the machine. We exploit this prop-  
 224 erty of x86 hardware to cause contention on the memory bus and  
 225 use the resulting observable variations in performance as a covert  
 226 channel for detecting co-residency.  
 227

## 228 3 METHODOLOGY

229 **Ariana** ► make sure that all these terms are defined ahead of time: co-location,  
 230 co-operative, co-residence, serverless, lambdas◀  
 231



**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, GCP and Azure clouds respectively.** TODO ► Get plot for GCP◀ 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 hardware-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 homogeneous 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 we 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, whose runtimes are generally restricted to high-level languages that prevent the pointer arithmetic required to perform these exotic operations. (We used the unsafe environments—C++ on AWS, Unsafe Go on GCP, Unsafe C# On Azure—that these clouds allowed for their serverless functions). 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 method to quicken the co-location process is by cutting down on the time it takes for single attack-victim pair to determine co-residence i.e., improving upon probing time and accuracy of the victim. However, this 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 see co-residence, we must be certain that each victim in the pair experienced co-residence because of its own attack instance, which is not possible if co-residence is ascertained based a simple yes/no signal from the attack instances.

The memory bus covert channel was used to exchange more complex information like keys in previous work[23], and at first sight, can be used to exchange information such as unique IDs between the co-resided instances to solve our problem. However, the original work assumes that there is only one sender and one receiver who 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, the co-resided instances can reliably exchange their unique IDs with each other to discover their neighbors. **Ariana** ► *this sentence is a bit confusing* ◀ 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) – something 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 in the next section.

## 4 NEIGHBOR DISCOVERY PROTOCOL

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, this requires the co-resided instances to exchange these (integer) IDs with each other using the memory bus channel as a transmission medium. In this section, we present a communication protocol that the co-resided instances can use to achieve communication in a fast and reliable way. We first discuss the challenges we faced in making the channel reliable before examining the protocol itself.

### 4.1 Reliable Transmission

Senders and receivers can reliably 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 on the memory bus covert channel. To

---

#### 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

```

---

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.** There are two ways in which the receivers could detect memory bus contention. When the memory bus is locked, any non-cached memory accesses will queue and therefore see higher latencies. The receiver can then continually make un-cached memory accesses (referred to as the *memory probing* receiver in previous literature [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.

Though two methods exist for detecting memory bus contention, 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 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 suit these architectures an arduous task while reducing the applicability of our overall co-residence detection mechanism. Thus, we decide to use the memory locking receiver.

**4.1.2 Sampling frequency.** Ideally, a memory locking receiver would loop locking operations and determine contention in real-time by

**Algorithm 2** Reading a bit in the receiver

```

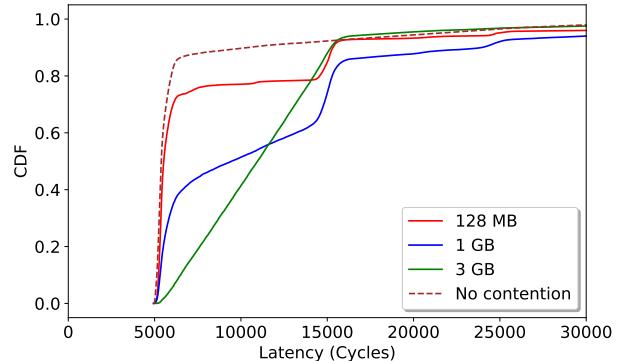
465
466 1: now ← time.now()
467 2: end ← now + sampling_duration
468 3: sampling_rate ← num_samples/sampling_duration
469 4: address ← cache_line_boundary - 2
470 5: samples ← {}
471 6: while now < end do
472 7:   before ← RDTSC()
473 8:   __ATOMIC_FETCH_ADD(address)
474 9:   after ← RDTSC()
475 10:  samples ← samples ∪ {(after - before)}
476 11:  wait until NEXT_POISSON(sampling_rate)
477 12:  now ← time.now()
478 13: end while
479 14: ks_val ← KOLMOGOROV_SMIRINOV(samples, baseline)
480 15: return ks_val < ksvalue_threshold
481
482
483

```

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 sampling takes a few microseconds each time.

**4.1.3 Sampling duration.** 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.

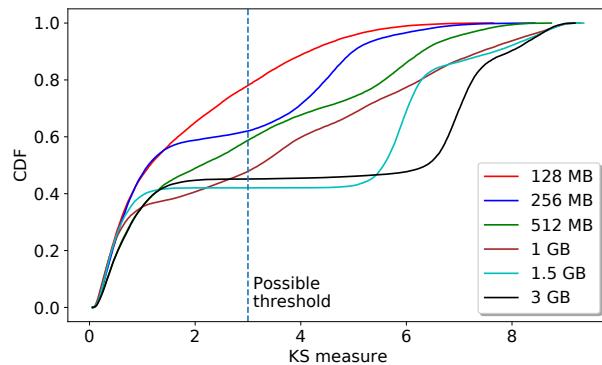
To illustrate such behavior, we run a sender-receiver pair using the Lambdas[14] of various sizes on AWS, and compare the distribution of latencies seen by the receiver during the contention in each case. Figure 2 shows that the much smaller 128 MB lambdas (which probably share a CPU core and are thus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



**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.

**4.1.4 Overcoming noise.** Along with context switching and sensing noise, there are other imperfections in measurement apparatus that cause (minor) 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. Since we effectively need to compare the empirical CDFs of the baseline measurements to the contention measurements, we decide to use a variant of the two-sample Kolmogorov-Smirnov (KS) test. By taking the mean of the absolute differences between the empirical CDFs, we can categorize a KS-value above a certain threshold (KS-threshold) as a 1-bit and a value below the threshold as 0-bit.

To determine the 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-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 lower and upper peaks corresponding to samples that have not and have seen contention respectively, and a big gap between the two (long step). Fortunately, we examine this differentiation with larger lambda sizes (which allows us to choose a



**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.

clear threshold), but we do not examine 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 is a platform constant.

We present the pseudo-code of a receiver lambda in Algorithm 2, which includes all the limitations 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, [?] 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 would be running on the same physical server and so they 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 as we only work in the millisecond regime due to sampling noise constraints **Ariana** ► is there some sort of citation for this?◀

## 4.2 Protocol

In the preceding section, we discussed 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

---

**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 ← slotsth 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

```

---

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 **Ariana** ► chicken and egg is a bit too colloquial...need to figure out how to communicate this thought better◀. 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 takes up to 1 second, the additional overhead can be detrimental to the efficacy of the communication channel.

**Ariana** ► I think in general we need a bit more sign-posting to guide the reader. I will think about this a bit more◀. However, a communication channel for lambda co-residence detection need not be general and

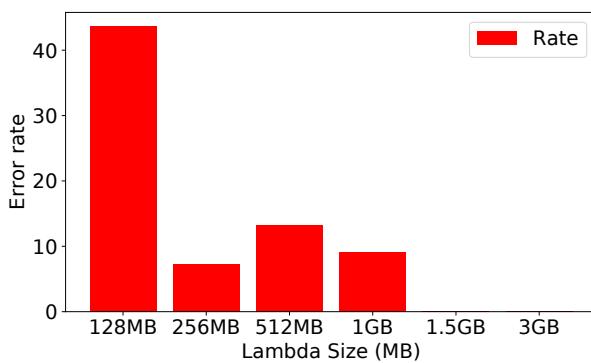


Figure 4: Shows the error rate (as a fraction of 1000 lambdas deployed) for different lambda sizes in AWS Middle-East region.

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.

**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 4 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

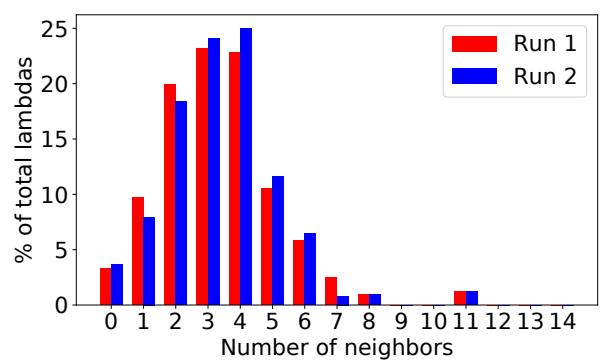


Figure 5: Shows the fraction of lambdas by the number of neighbors they saw for two independent runs that use same set of underlying AWS containers. The perfect correlation shows that both runs depict the colocation status of those containers regardless of the lambdas that ran on them, providing an evidence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

**TODO** ▶ this doesn't show perfect correlation because only 99% of lambdas warm started. Get another perfect run?◀

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.

## 5 EVALUATION

In this section, we evaluate the effectiveness of our co-residence detection technique with respect to the desirable properties mentioned in section 3 i.e., reliability and scalability.

### 5.1 Setup

We run all our experiments with AWS[4] lambdas. (We have showed that this covert channel exists on other clouds and so can be easily replicated with their serverless functions). Activating the memory bus covert channel requires address manipulation using pointers and is not supported by most serverless platform runtimes that only allow high-level languages like Python. However, we were able to find at least one exception to this on all clouds: AWS allows C++ programs while Azure and GCP allow unsafe versions of C# and Golang respectively. Once deployed, each instance participates in the first phase of the protocol as noted in section 4.2.1, thereby learning the ID of their biggest neighbor. 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 resulted in different IDs, we classify this instance as erroneous and report it in the error rate. We group all the successful instances that saw the same ID as neighbors. We repeat the experiments for different lambda sizes and in various cloud regions.

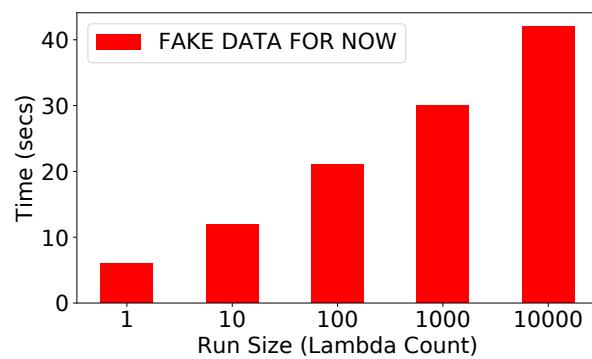


Figure 6: Shows the average runtime of a lambda for co-location runs of different sizes. The run time increases logarithmically with the number of lambdas as it is proportional to the number of bits required to uniquely identify all the lambdas. TODO ►Get real data.◀

## 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** Obtaining the ground truth on which instances were "actually" co-located is not possible, considering that that is the purpose of our technique. However, we found a way to validate our results with high confidence. AWS caches the containers used to run lambdas for a while (TODO ►how long?◀) to reuse them[3] for later lambdas and mitigate cold start latencies. For C++ lambdas, we found that the data structures declared in 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 validated that identical experiments repeated within a few minutes of each other will use the same set of underlying containers for running the deployed lambdas. Since the lambda co-location is essentially co-location of their containers and given that these containers persist across experiments (run within few minutes of each other), co-location results from such experiments must agree on the co-location of their underlying containers.

To demonstrate that this is the case, we run an experiment with 1000 1.5GB cold-started lambdas (ID'ed 1 to 1000) in the one of densest AWS regions (AWS MiddleEast), which resulted in many

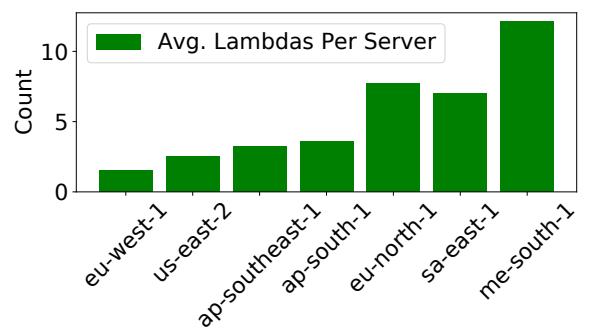


Figure 7: Shows the average number of lambdas per server i.e., colocation density seen in various AWS regions for a 1000-lambda run.

co-located groups. We repeat the experiment within few seconds, making sure that all 1000 lambdas are warm-started this time (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 in the former one formed a co-located group as well. We note that while different lambdas used the containers across the experiments, their co-located groups are perfectly correlated. 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 that it's really fast. It takes only a second to communicate each binary bit of the ID, enabling it to scale logarithmically with the number of lambdas involved. Figure 6 shows this for experiments involving different number of lambdas. For a run with 10000 lambdas for example, each lambda can find its neighbors within a minute of its invocation, leaving ample time to perform other things using this information. This also means the cost per lambda also scales logarithmically, making it very 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, 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◀.

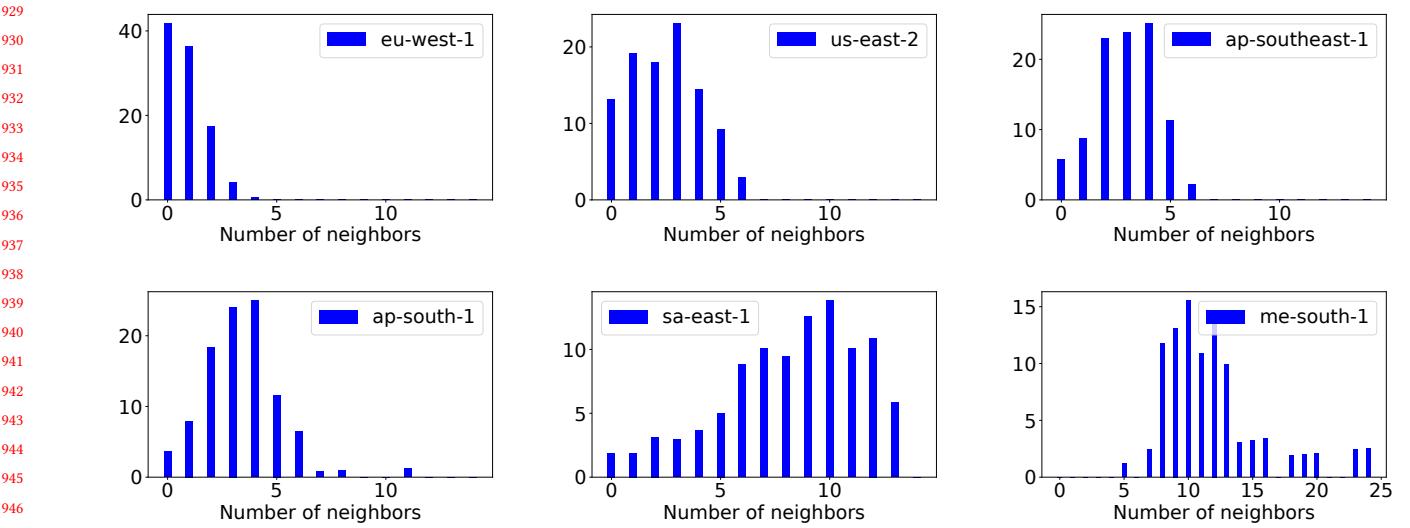


Figure 8: Shows co-location results for a 1000-lambda run in different AWS regions. Each bar shows the fraction of those 1000 lambdas (in %) that saw a certain number of neighbors. The total amount and density of co-location vary widely across regions, perhaps based on the size and lambda activity within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

## 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 hypothesize 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 et 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

1045 that fall on multiple cache lines..

1046  
 1047 **Co-residency** One of the first pieces of literature in detecting  
 1048 VM co-residency was introduced by Ristenpart et al., who demon-  
 1049 strated that VM co-residency detection was possible and that these  
 1050 techniques could be used to gather information about the victim  
 1051 machine (such as keystrokes and network usage) [18]. This initial  
 1052 work was further expanded in subsequent years to examine co-  
 1053 residency using memory bus locking [27] and active traffic anal-  
 1054 ysis [6], as well as determining placement vulnerabilities in multi-  
 1055 tenant Public Platform-as-a-Service systems [20, 30]. Finally, Zhang  
 1056 et al. demonstrated a technique to detect VM co-residency detec-  
 1057 tion via side-channel analyses [31]. Our work expands on these  
 1058 previous works by investigating co-residency for lambdas.  
 1059

1060 **Lambdas** TODO ► *pretty rough paragraph...needs a rewrite*◀ While  
 1061 lambdas are a newer technology than VMs, there still exists a vari-  
 1062 ety of literature. For example, recent studies examine cost compar-  
 1063 isons of running web applications in the cloud on lambdas versus  
 1064 other architectures [21]. Moreover, lambdas have been studied in  
 1065 the context of cost-effective batching and data processing [13]. Fur-  
 1066 ther research has shown how lambdas perform with scalability and  
 1067 hardware isolation, indicating some flaws in the lambda architec-  
 1068 ture [22]. From a security perspective, Izhikevich et. al examined  
 1069 lambda co-residency using RNG and memory bus techniques (simi-  
 1070 lar to techniques used looking at VM co-residency) [11].. However,  
 1071 our work differs from this study in that our technique informs the  
 1072 user of which lambdas are on the same machine, not only that the  
 1073 lambdas experience co-residency.  
 1074

## 1075 9 ETHICAL CONSIDERATIONS

1076 As with any large scale measurement project, there are ethical con-  
 1077 siderations to take into account. First, there are security and pri-  
 1078 vacy concerns of using this technique to uncover other consumer's  
 1079 lambdas. However, since we focus on co-operative co-residence de-  
 1080 tection, we only determine co-location for the lambdas we launched,  
 1081 and do not gain insight into other consumer's lambdas. Secondly,  
 1082 there is concern that our experiments may cause performance is-  
 1083 sues with other lambdas, as we may block their access to the mem-  
 1084 ory bus. We believe this concern is small, for a number of rea-  
 1085 sons. Memory accesses are infrequent due to the multiple levels  
 1086 of caches; we would only be affecting a small number of opera-  
 1087 tions. Moreover, memory accesses and locking operations are FIFO,  
 1088 which prevents starvation to the multiple processes sharing a ma-  
 1089 chine. Moreover, lambdas are generally not recommended for latency-  
 1090 sensitive workloads, due to their cold-start latencies. Thus, the small  
 1091 amount of lambdas that we might affect should not, by definition,  
 1092 be affected in their longterm computational goals. Ariana ► *is lambda*  
 1093 *cost by second? is it possible that by causing a longer lambda runtime, we are*  
 1094 *costing someone more money?*◀  
 1095

## 1096 10 CONCLUSION & FUTURE WORK

1097 TODO ► *Copied abstract*◀ Cloud computing has seen explosive growth  
 1098 in the past decade. This is made possible by efficient sharing of  
 1099 infrastructure among tenants, which unfortunately also raises se-  
 1100 curity challenges like preventing side-channel attacks. Providers,  
 1102

1103 like AWS and Azure, have traditionally relied on hiding the co-  
 1104 residency information to prevent targeted attacks in their clouds.  
 1105 But recent works have repeatedly found co-residence detection  
 1106 techniques that break this encapsulation, prompting the providers  
 1107 to address them and harden isolation on their platforms. In this  
 1108 work, we find yet another such technique based on a memory bus  
 1109 covert channel that is more pervasive, reliable and harder to fix.  
 1110 We show that we can use this technique to reliably perform co-  
 1111 operative co-residence detection for thousands of AWS lambdas  
 1112 within a few seconds, which opens a way for attackers to perform  
 1113 DDoS attacks or learn cloud's internal mechanisms. We present  
 1114 this technique in detail, evaluate it and use it to perform a small  
 1115 study on lambda activity across a few AWS regions. Through this  
 1116 work, we hope to motivate the need to address this covert channel  
 1117 in the cloud.  
 1118

## 1119 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 Evtushkin 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 Leaks 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, Aanjan 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>

1103  
 1104  
 1105  
 1106  
 1107  
 1108  
 1109  
 1110  
 1111  
 1112  
 1113  
 1114  
 1115  
 1116  
 1117  
 1118  
 1119  
 1120  
 1121  
 1122  
 1123  
 1124  
 1125  
 1126  
 1127  
 1128  
 1129  
 1130  
 1131  
 1132  
 1133  
 1134  
 1135  
 1136  
 1137  
 1138  
 1139  
 1140  
 1141  
 1142  
 1143  
 1144  
 1145  
 1146  
 1147  
 1148  
 1149  
 1150  
 1151  
 1152  
 1153  
 1154  
 1155  
 1156  
 1157  
 1158  
 1159  
 1160

- 1161 [18] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, 1219  
 1162 You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute 1220  
 1163 Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications 1221  
 1164 Security (Chicago, Illinois, USA) (CCS '09)*. ACM, New York, NY, USA, 199–212. 1222  
<https://doi.org/10.1145/1653662.1653687>
- 1165 [19] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael 1223  
 1166 Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In 1224  
 1167 *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 1225  
 Washington, D.C., 913–928. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan>
- 1168 [20] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. 1226  
 1169 Swift. 2015. A Placement Vulnerability Study in Multi-tenant Public Clouds. *CoRR* 1227  
 abs/1507.03114. arXiv:1507.03114 <http://arxiv.org/abs/1507.03114>
- 1170 [21] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, 1228  
 Mauricio Verano Merino, Ruby Casallas, Santiago Gil, Carlos Valencia, Angee 1229  
 Zambrano, and Mery Lang. 2016. Infrastructure Cost Comparison of Running 1230  
 Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice 1231  
 Architectures. <https://doi.org/10.1109/CCGrid.2016.37>
- 1171 [22] Liang Wang, Mengyan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. 1232  
 Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX 1233  
 Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. <https://www.usenix.org/conference/atc18/presentation/wang-liang>
- 1172 [23] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: 1234  
 High-speed Covert Channel Attacks in the Cloud. In *Presented as part of the 21st 1235  
 USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 159– 1236  
 173. <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu>
- 1173 [24] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, 1237  
 and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels 1238  
 in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud 1239  
 Computing Security Workshop (Chicago, Illinois, USA) (CCSW '11)*. ACM, New York, NY, USA, 29–40. <https://doi.org/10.1145/204660.2046670>
- 1174 [25] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, 1240  
 and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels 1241  
 in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud 1242  
 Computing Security Workshop (Chicago, Illinois, USA) (CCSW '11)*. Association for 1243  
 Computing Machinery, New York, NY, USA, 29–40. <https://doi.org/10.1145/204660.2046670>
- 1175 [26] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co- 1244  
 residence Threat inside the Cloud. In *24th USENIX Security Symposium (USENIX 1245  
 Security 15)*. USENIX Association, Washington, D.C., 929–944. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu>
- 1176 [27] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co- 1246  
 residence Threat inside the Cloud. In *24th USENIX Security Symposium (USENIX 1247  
 Security 15)*. USENIX Association, Washington, D.C., 929–944. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu>
- 1177 [28] Tianwei Zhang, Yinqian Zhang, and Ruby Lee. 2016. Memory DoS Attacks in 1248  
 Multi-tenant Clouds: Severity and Mitigation. <https://doi.org/10.1109/SP.2016.75>
- 1178 [29] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Ming- 1249  
 sheng Wang, and Peng Liu. 2016. A Comprehensive Study of Co-residence 1250  
 Threat in Multi-tenant Public PaaS Clouds. In *Information and Communications 1251  
 Security*, Kwok-Yan Lam, Chi-Hung Chi, and Sihan Qing (Eds.). Springer International Publishing, Cham, 361–375.
- 1179 [30] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Ming- 1252  
 sheng Wang, and Peng Liu. 2016. A Comprehensive Study of Co-residence 1253  
 Threat in Multi-tenant Public PaaS Clouds, Vol. 9977. 361–375. [https://doi.org/10.1007/978-3-319-50011-9\\_28](https://doi.org/10.1007/978-3-319-50011-9_28)
- 1180 [31] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael Reiter. 2011. HomeAlone: 1254  
 Co-Residency Detection in the Cloud via Side-Channel Analysis. 313 – 328. <https://doi.org/10.1109/SP.2011.31>
- 1181 [32] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross- 1255  
 Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM 1256  
 SIGSAC Conference on Computer and Communications Security (Scottsdale, Arizona, 1257  
 USA) (CCS '14)*. ACM, New York, NY, USA, 990–1003. <https://doi.org/10.1145/2660267.2660356>
- 1182  
 1183  
 1184  
 1185  
 1186  
 1187  
 1188  
 1189  
 1190  
 1191  
 1192  
 1193  
 1194  
 1195  
 1196  
 1197  
 1198  
 1199  
 1200  
 1201  
 1202  
 1203  
 1204  
 1205  
 1206  
 1207  
 1208  
 1209  
 1210  
 1211  
 1212  
 1213  
 1214  
 1215  
 1216  
 1217  
 1218  
 1219  
 1220  
 1221  
 1222  
 1223  
 1224  
 1225  
 1226  
 1227  
 1228  
 1229  
 1230  
 1231  
 1232  
 1233  
 1234  
 1235  
 1236  
 1237  
 1238  
 1239  
 1240  
 1241  
 1242  
 1243  
 1244  
 1245  
 1246  
 1247  
 1248  
 1249  
 1250  
 1251  
 1252  
 1253  
 1254  
 1255  
 1256  
 1257  
 1258  
 1259  
 1260  
 1261  
 1262  
 1263  
 1264  
 1265  
 1266  
 1267  
 1268  
 1269  
 1270  
 1271  
 1272  
 1273  
 1274  
 1275  
 1276  
 1277  
 1278  
 1279  
 1280  
 1281  
 1282  
 1283  
 1284  
 1285  
 1286  
 1287  
 1288  
 1289  
 1290  
 1291  
 1292  
 1293  
 1294  
 1295  
 1296  
 1297  
 1298  
 1299  
 1300  
 1301  
 1302  
 1303  
 1304  
 1305  
 1306  
 1307  
 1308  
 1309  
 1310  
 1311  
 1312  
 1313  
 1314  
 1315  
 1316  
 1317  
 1318  
 1319  
 1320  
 1321  
 1322  
 1323  
 1324  
 1325  
 1326  
 1327  
 1328  
 1329  
 1330  
 1331  
 1332  
 1333  
 1334  
 1335  
 1336  
 1337  
 1338  
 1339  
 1340  
 1341  
 1342  
 1343  
 1344  
 1345  
 1346  
 1347  
 1348  
 1349  
 1350  
 1351  
 1352  
 1353  
 1354  
 1355  
 1356  
 1357  
 1358  
 1359  
 1360  
 1361  
 1362  
 1363  
 1364  
 1365  
 1366  
 1367  
 1368  
 1369  
 1370  
 1371  
 1372  
 1373  
 1374  
 1375  
 1376  
 1377  
 1378  
 1379  
 1380  
 1381  
 1382  
 1383  
 1384  
 1385  
 1386  
 1387  
 1388  
 1389  
 1390  
 1391  
 1392  
 1393  
 1394  
 1395  
 1396  
 1397  
 1398  
 1399  
 1400  
 1401  
 1402  
 1403  
 1404  
 1405  
 1406  
 1407  
 1408  
 1409  
 1410  
 1411  
 1412  
 1413  
 1414  
 1415  
 1416  
 1417  
 1418  
 1419  
 1420  
 1421  
 1422  
 1423  
 1424  
 1425  
 1426  
 1427  
 1428  
 1429  
 1430  
 1431  
 1432  
 1433  
 1434  
 1435  
 1436  
 1437  
 1438  
 1439  
 1440  
 1441  
 1442  
 1443  
 1444  
 1445  
 1446  
 1447  
 1448  
 1449  
 1450  
 1451  
 1452  
 1453  
 1454  
 1455  
 1456  
 1457  
 1458  
 1459  
 1460  
 1461  
 1462  
 1463  
 1464  
 1465  
 1466  
 1467  
 1468  
 1469  
 1470  
 1471  
 1472  
 1473  
 1474  
 1475  
 1476  
 1477  
 1478  
 1479  
 1480  
 1481  
 1482  
 1483  
 1484  
 1485  
 1486  
 1487  
 1488  
 1489  
 1490  
 1491  
 1492  
 1493  
 1494  
 1495  
 1496  
 1497  
 1498  
 1499  
 1500  
 1501  
 1502  
 1503  
 1504  
 1505  
 1506  
 1507  
 1508  
 1509  
 1510  
 1511  
 1512  
 1513  
 1514  
 1515  
 1516  
 1517  
 1518  
 1519  
 1520  
 1521  
 1522  
 1523  
 1524  
 1525  
 1526  
 1527  
 1528  
 1529  
 1530  
 1531  
 1532  
 1533  
 1534  
 1535  
 1536  
 1537  
 1538  
 1539  
 1540  
 1541  
 1542  
 1543  
 1544  
 1545  
 1546  
 1547  
 1548  
 1549  
 1550  
 1551  
 1552  
 1553  
 1554  
 1555  
 1556  
 1557  
 1558  
 1559  
 1560  
 1561  
 1562  
 1563  
 1564  
 1565  
 1566  
 1567  
 1568  
 1569  
 1570  
 1571  
 1572  
 1573  
 1574  
 1575  
 1576  
 1577  
 1578  
 1579  
 1580  
 1581  
 1582  
 1583  
 1584  
 1585  
 1586  
 1587  
 1588  
 1589  
 1590  
 1591  
 1592  
 1593  
 1594  
 1595  
 1596  
 1597  
 1598  
 1599  
 1600  
 1601  
 1602  
 1603  
 1604  
 1605  
 1606  
 1607  
 1608  
 1609  
 1610  
 1611  
 1612  
 1613  
 1614  
 1615  
 1616  
 1617  
 1618  
 1619  
 1620  
 1621  
 1622  
 1623  
 1624  
 1625  
 1626  
 1627  
 1628  
 1629  
 1630  
 1631  
 1632  
 1633  
 1634  
 1635  
 1636  
 1637  
 1638  
 1639  
 1640  
 1641  
 1642  
 1643  
 1644  
 1645  
 1646  
 1647  
 1648  
 1649  
 1650  
 1651  
 1652  
 1653  
 1654  
 1655  
 1656  
 1657  
 1658  
 1659  
 1660  
 1661  
 1662  
 1663  
 1664  
 1665  
 1666  
 1667  
 1668  
 1669  
 1670  
 1671  
 1672  
 1673  
 1674  
 1675  
 1676  
 1677  
 1678  
 1679  
 1680  
 1681  
 1682  
 1683  
 1684  
 1685  
 1686  
 1687  
 1688  
 1689  
 1690  
 1691  
 1692  
 1693  
 1694  
 1695  
 1696  
 1697  
 1698  
 1699  
 1700  
 1701  
 1702  
 1703  
 1704  
 1705  
 1706  
 1707  
 1708  
 1709  
 1710  
 1711  
 1712  
 1713  
 1714  
 1715  
 1716  
 1717  
 1718  
 1719  
 1720  
 1721  
 1722  
 1723  
 1724  
 1725  
 1726  
 1727  
 1728  
 1729  
 1730  
 1731  
 1732  
 1733  
 1734  
 1735  
 1736  
 1737  
 1738  
 1739  
 1740  
 1741  
 1742  
 1743  
 1744  
 1745  
 1746  
 1747  
 1748  
 1749  
 1750  
 1751  
 1752  
 1753  
 1754  
 1755  
 1756  
 1757  
 1758  
 1759  
 1760  
 1761  
 1762  
 1763  
 1764  
 1765  
 1766  
 1767  
 1768  
 1769  
 1770  
 1771  
 1772  
 1773  
 1774  
 1775  
 1776  
 1777  
 1778  
 1779  
 1780  
 1781  
 1782  
 1783  
 1784  
 1785  
 1786  
 1787  
 1788  
 1789  
 1790  
 1791  
 1792  
 1793  
 1794  
 1795  
 1796  
 1797  
 1798  
 1799  
 1800  
 1801  
 1802  
 1803  
 1804  
 1805  
 1806  
 1807  
 1808  
 1809  
 1810  
 1811  
 1812  
 1813  
 1814  
 1815  
 1816  
 1817  
 1818  
 1819  
 1820  
 1821  
 1822  
 1823  
 1824  
 1825  
 1826  
 1827  
 1828  
 1829  
 1830  
 1831  
 1832  
 1833  
 1834  
 1835  
 1836  
 1837  
 1838  
 1839  
 1840  
 1841  
 1842  
 1843  
 1844  
 1845  
 1846  
 1847  
 1848  
 1849  
 1850  
 1851  
 1852  
 1853  
 1854  
 1855  
 1856  
 1857  
 1858  
 1859  
 1860  
 1861  
 1862  
 1863  
 1864  
 1865  
 1866  
 1867  
 1868  
 1869  
 1870  
 1871  
 1872  
 1873  
 1874  
 1875  
 1876  
 1877  
 1878  
 1879  
 1880  
 1881  
 1882  
 1883  
 1884  
 1885  
 1886  
 1887  
 1888  
 1889  
 1890  
 1891  
 1892  
 1893  
 1894  
 1895  
 1896  
 1897  
 1898  
 1899  
 1900  
 1901  
 1902  
 1903  
 1904  
 1905  
 1906  
 1907  
 1908  
 1909  
 1910  
 1911  
 1912  
 1913  
 1914  
 1915  
 1916  
 1917  
 1918  
 1919  
 1920  
 1921  
 1922  
 1923  
 1924  
 1925  
 1926  
 1927  
 1928  
 1929  
 1930  
 1931  
 1932  
 1933  
 1934  
 1935  
 1936  
 1937  
 1938  
 1939  
 1940  
 1941  
 1942  
 1943  
 1944  
 1945  
 1946  
 1947  
 1948  
 1949  
 1950  
 1951  
 1952  
 1953  
 1954  
 1955  
 1956  
 1957  
 1958  
 1959  
 1960  
 1961  
 1962  
 1963  
 1964  
 1965  
 1966  
 1967  
 1968  
 1969  
 1970  
 1971  
 1972  
 1973  
 1974  
 1975  
 1976  
 1977  
 1978  
 1979  
 1980  
 1981  
 1982  
 1983  
 1984  
 1985  
 1986  
 1987  
 1988  
 1989  
 1990  
 1991  
 1992  
 1993  
 1994  
 1995  
 1996  
 1997  
 1998  
 1999  
 2000  
 2001  
 2002  
 2003  
 2004  
 2005  
 2006  
 2007  
 2008  
 2009  
 2010  
 2011  
 2012  
 2013  
 2014  
 2015  
 2016  
 2017  
 2018  
 2019  
 2020  
 2021  
 2022  
 2023  
 2024  
 2025  
 2026  
 2027  
 2028  
 2029  
 2030  
 2031  
 2032  
 2033  
 2034  
 2035  
 2036  
 2037  
 2038  
 2039  
 2040  
 2041  
 2042  
 2043  
 2044  
 2045  
 2046  
 2047  
 2048  
 2049  
 2050  
 2051  
 2052  
 2053  
 2054  
 2055  
 2056  
 2057  
 2058  
 2059  
 2060  
 2061  
 2062  
 2063  
 2064  
 2065  
 2066  
 2067  
 2068  
 2069  
 2070  
 2071  
 2072  
 2073  
 2074  
 2075  
 2076  
 2077  
 2078  
 2079  
 2080  
 2081  
 2082  
 2083  
 2084  
 2085  
 2086  
 2087  
 2088  
 2089  
 2090  
 2091  
 2092  
 2093  
 2094  
 2095  
 2096  
 2097  
 2098  
 2099  
 2100  
 2101  
 2102  
 2103  
 2104  
 2105  
 2106  
 2107  
 2108  
 2109  
 2110  
 2111  
 2112  
 2113  
 2114  
 2115  
 2116  
 2117  
 2118  
 2119  
 2120  
 2121  
 2122  
 2123  
 2124  
 2125  
 2126  
 2127  
 2128  
 2129  
 2130  
 2131  
 2132  
 2133  
 2134  
 2135  
 2136  
 2137  
 2138  
 2139  
 2140  
 2141  
 2142  
 2143  
 2144  
 2145  
 2146  
 2147  
 2148  
 2149  
 2150  
 2151  
 2152  
 2153  
 2154  
 2155  
 2156  
 2157  
 2158  
 2159  
 2160  
 2161  
 2162  
 2163  
 2164  
 2165  
 2166  
 2167  
 2168  
 2169  
 2170  
 2171  
 2172  
 2173  
 2174  
 2175  
 2176  
 2177  
 2178  
 2179  
 2180  
 2181  
 2182  
 2183  
 2184  
 2185  
 2186  
 2187  
 2188  
 2189  
 2190  
 2191  
 2192  
 2193  
 2194  
 2195  
 2196  
 2197  
 2198  
 2199  
 2200  
 2201  
 2202  
 2203  
 2204  
 2205  
 2206  
 2207  
 2208  
 2209  
 2210  
 2211  
 2212  
 2213  
 2214  
 2215  
 2216  
 2217  
 2218  
 2219  
 2220  
 2221  
 2222  
 2223  
 2224  
 2225  
 2226  
 2227  
 2228  
 2229  
 2230  
 2231  
 2232  
 2233  
 2234  
 2235  
 2236  
 2237  
 2238  
 2239  
 2240  
 2241  
 2242  
 2243  
 2244  
 2245  
 2246  
 2247  
 2248  
 2249  
 2250  
 2251  
 2252  
 2253  
 2254  
 2255  
 2256  
 2257  
 2258  
 2259  
 2260  
 2261  
 2262  
 2263  
 2264  
 2265  
 2266  
 2267  
 2268  
 2269  
 2270  
 2