

# Columbus: Fast, Reliable Co-residence Detection for Lambdas

1  
2  
3  
4  
5

## ABSTRACT

“Serverless” cloud services, such as AWS lambdas, are one of the fastest growing segment of the cloud services market. These services are lighter-weight and provide more flexibility in scheduling and cost, which contributes to their popularity, however the security issues associated with serverless computing are not well understood. In this work, we explore the feasibility of constructing a practical covert channel from lambdas. We establish that a fast and scalable co-residence detection for lambdas is key to enabling such a covert channel, and proceed to develop a generic, reliable, and scalable co-residence detector based on the memory bus hardware. Our technique enables dynamic neighbor discovery for co-resident lambdas and is incredibly fast, executing in a matter of seconds. We evaluate our approach for correctness and scalability, and perform a measurement study on lambda density in AWS cloud to demonstrate the practicality of establishing cloud covert channels using our co-residence detector for lambdas. Through this work, we show that efforts to secure co-residency detection on cloud platforms are not yet complete.

25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37

## CCS CONCEPTS

- Security and privacy → Virtualization and security.

KEYWORDS

cloud, cartography, serverless, coresidency, covert channels

ACM Reference Format:

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

1 INTRODUCTION

Over the last decade, organizations have increasingly offloaded their data processing and storage needs to third-party “cloud” platforms. However, the economics of cloud platforms is predicated on high levels of statistical multiplexing and thus *co-tenancy* – the contemporaneous execution of computation from disparate customers on the same physical hardware – is the norm. The risks associated with this arrangement, both data leakage and interference, are well-appreciated and have generated both a vast research literature (starting with Ristenpart et al. [19]) as well a wide-array of technical isolation countermeasures employed by cloud platform

providers. Most of this work has focused squarely on the risks of information channels between long-lived, heavy-weight virtual machines (“instances” in Amazon parlance) used to virtualize the traditional notion of dedicated network-connected servers.

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

In our work, we explore these issues through the lens of a singular question: can a practical covert channel be constructed entirely from existing “serverless” cloud services<sup>1</sup>?

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

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

Permission to make digital or hard copies of all or part of this work for personal or educational use is granted. Copying for general distribution or 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 redistribute, 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).

Under review for 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/nnnnnnnn.nnnnnnn>

2020-10-20 00:17. Page 1 of 12.

<sup>1</sup>We will use the term lambdas to stand for all such services going forward.

59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115

117 the balance of their limited lifetimes to effect communications. Fi-  
 118 nally, since rendezvous in a serverless system is inherently statis-  
 119 tical, any such protocol must anticipate the potential for interfer-  
 120 ence (i.e., when multiple sending lambdas happen to be co-resident  
 121 with one or more receiving lambdas).

122 In this paper we address each of these issues in turn and demon-  
 123 strate the feasibility of covert communication entirely in the con-  
 124 text of the Amazon’s serverless cloud platform. In particular, we  
 125 make three key technical contributions:

- 126 • **Fast co-residence detection.** Leveraging the memory-bus  
 127 contention work of Wu et. al [26], we develop and imple-  
 128 ment a lambda co-residence detector that is generic, reliable,  
 129 scalable and, most importantly, fast, executing in a matter  
 130 of seconds for thousands of concurrent lambdas.
- 131 • **Dynamic neighbor discovery.** We present a novel proto-  
 132 col for the co-resident lambdas to communicate their IDs  
 133 using memory bus contention and discover one another. A  
 134 key enabler of our co-residence detection, the protocol also  
 135 helps a sender or receiver lambda to enumerate *all* its co-  
 136 resident neighbors, a requirement to avoid unwanted com-  
 137 munication interference while performing covert communi-  
 138 cation.
- 139 • **Serverless density measurement.** We empirically estab-  
 140 lish measures of serverless function density – that is, for a  
 141 given number of short-lifetime lambdas launched at a point  
 142 in time, how many would be expected to become co-resident?  
 143 We conduct these measurements across a range of Amazon  
 144 data centers to establish that there is ample co-residence of  
 145 lambdas and covert communication is practicable.

146 The remainder of this paper is organized as follows. In section 2,  
 147 we present some background and motivate the need for co-residence  
 148 detection for lambdas. Sections 3 and 4 present our co-residence  
 149 detection mechanism in detail. We evaluate the mechanism in sec-  
 150 tion 5 and perform a study of AWS lambda placement density us-  
 151 ing our technique to demonstrate the practicality of lambda covert  
 152 channels in section 6. We conclude with a discussion on other use  
 153 cases for the co-residence detector and potential mitigation strate-  
 154 gies in section 7.

## 157 2 BACKGROUND & MOTIVATION

158 We begin with a brief background on relevant topics as we moti-  
 159 vate the need for fast and scalable co-residence detection in lamb-  
 160 das.

### 163 2.1 Lambdas/Serverless Functions

164 We focus on serverless functions in this paper, as they are one of  
 165 the fastest-growing cloud services and are less well-studied from a  
 166 security standpoint. Offered as lambdas on AWS [14], and as cloud  
 167 functions on GCP [10] and Azure [5], these functions are of interest  
 168 because they do not require the developer to provision, maintain,  
 169 or administer servers. In addition to this low overhead, lambdas  
 170 are much more cost-efficient than virtual machines (VMs) as they  
 171 allow more efficient packing of functions on servers. Moreover,  
 172 lambdas execute as much smaller units and are more ephemeral

173 than virtual machines. For example, on AWS, the memory of lamb-  
 174 das is capped at 3 GB, with a maximum execution limit of 15 min-  
 175 utes. As with other cloud services, the user has no control over  
 176 the physical location of the server(s) on which their lambdas are  
 177 spawned.

178 While lambdas are limited in the computations they can exe-  
 179 cute (typically written in high-level languages like Python, C#, etc),  
 180 they are conversely incredibly lightweight and can be initiated  
 181 and deleted in a very short amount of time. Cloud providers run  
 182 lambdas in dedicated containers with limited resources (e.g., Fire-  
 183 cracker [1]), which are usually cached and re-used for future lamb-  
 184 das to mitigate cold-start latencies [2]. The ephemeral nature of  
 185 serverless functions and their limited flexibility increases the dif-  
 186 ficulty in detecting co-residency, as we will discuss later. While  
 187 previous studies that profiled lambdas [24] focused on the perfor-  
 188 mance aspects like cold start latencies, function instance lifetime,  
 189 and CPU usage across various clouds, the security aspects remain  
 190 relatively understudied.

## 194 2.2 Covert Channels in the Cloud

195 In our attempt to shed light on the security aspects of lambdas,  
 196 we focus particularly on the feasibility of establishing a reliable  
 197 covert channel in the cloud using lambdas. Covert channels enable  
 198 a means of transmitting information between entities that bypasses  
 199 traditional monitoring or auditing. Typically, this is achieved by  
 200 communicating data across unintended channels such as signalling  
 201 bits by causing contention on shared hardware media on the server [16,  
 202 18, 25–27]. Past work has demonstrated covert channels in virtu-  
 203 alized environments like the clouds using various hardware such  
 204 as caches [19, 27], memory bus [26], and even processor tempera-  
 205 ture [16].

206 **Memory bus covert channel** Of particular interest to this work  
 207 is the covert channel based on memory bus hardware introduced  
 208 by Wu et al. [26]. In x86 systems, atomic memory instructions de-  
 209 signed to facilitate multi-processor synchronization are supported  
 210 by cache coherence protocols as long as the operands remain within  
 211 a cache line (generally the case as language compilers make sure  
 212 that operands are aligned). However, if the operand is spread across  
 213 two cache lines (referred to as “exotic” memory operations), x86  
 214 hardware achieves atomicity by locking the memory bus to pre-  
 215 vent any other memory access operations until the current opera-  
 216 tion finishes. This results in significantly higher latencies for such  
 217 locking operations compared to traditional memory accesses. As  
 218 a result, a few consecutive locking operations could cause con-  
 219 tention on the memory bus that could be exploited for covert com-  
 220 munication. Wu et al. achieved a data rate of 700 bps on the mem-  
 221 ory bus channel in an ideal laboratory setup.

222 Achieving such ideal performance, however, is generally not  
 223 possible in cloud environments, as they pose challenges to both es-  
 224 tablishing such covert channels and using them at their ideal per-  
 225 formance. First, in a cloud setting, co-residency information is hid-  
 226 den, even if the entities belong to the same tenant, so it is unclear  
 227 whether a sender and receiver are on the same server. Second, com-  
 228 munication on the covert channel may be noisy and affected by  
 229 scheduling uncertainties and interference from other non-participating

workloads, due to the virtualized platform utilized by cloud providers. Studies generally deal with the latter problem by employing traditional error correction techniques [26] to overcome errors. However, the former challenge of establishing the covert channel requires a co-residence detection mechanism which the attacker needs to employ to ensure the sender and receiver are on the same machine.

### 2.3 Co-residence Detection

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

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

## 3 METHODOLOGY

Our goal is to determine a co-residence detection mechanism for lambdas. In other words, given a series of spawned lambdas in a certain region of a cloud service, how can we determine the lambdas that are co-resident 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 should identify, for each pair of instances in the set, whether the pair was running on the same physical server at some point. Paraphrasing Varadarajan et al. [22], for any such mechanism to be useful across a wide range of launch strategies, it should have the following properties:

- **Generic** The technique should be applicable across a wide range of server architectures and software runtimes. In practice, the technique would work across most third-party cloud platforms and even among different platforms within a cloud.
- **Reliable** The technique should have a reasonable detection success with minimal false negatives (co-resident instances

not detected) and even less false positives (non co-resident instances categorized as co-resident).

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

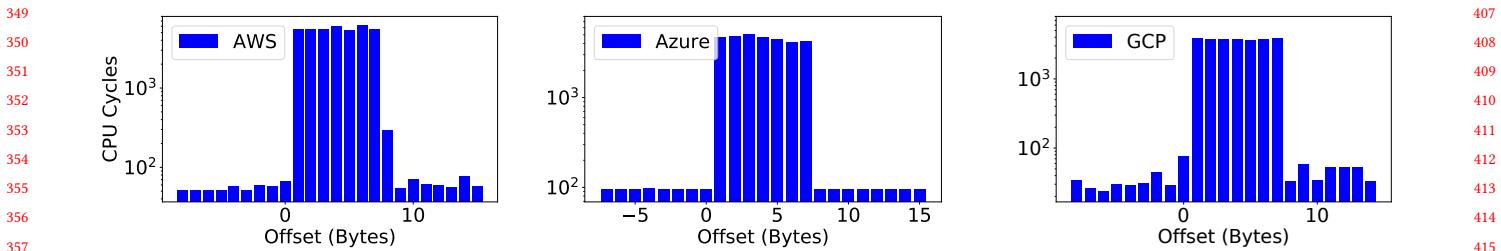
We add another property to that list which is relevant to lambdas:

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

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 more pervasive, given that hardware is more homogenous across computing platforms than software.

**3.0.1 Memory bus channel.** We utilize the memory bus covert channel described in section 2.2 as it exploits a fundamental hardware vulnerability that is present across all generations of x86 hardware. Historically, multiple public cloud services have been vulnerable to this channel [21, 32], and we find that they are still vulnerable today. To demonstrate the presence of the vulnerability, we measure the latency of atomic operations on a 8B memory region as we slide the region from one cacheline into another across the cacheline boundary. We perform this experiment on three major cloud platforms (AWS, Google and Microsoft Azure) and show the latencies observed in Figure 1. From the figure, we can see that all three cloud platforms still exhibit a significant difference in latencies for the “exotic” memory locking operations (where the memory region falls across cacheline boundary) when compared to regular memory accesses, demonstrating the presence of this covert channel on all of them. Moreover, we were able to execute these experiments on serverless function instances. Since lambdas have runtimes that are generally restricted to high-level languages (that prevent the pointer arithmetic required to perform these exotic operations), we used the unsafe environments on these clouds – C++ on AWS, Unsafe Go on GCP, Unsafe C# On Azure. This shows the applicability of using the covert channel across different kinds of cloud instances as well.

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



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

An alternative method is to try pair-wise detection where one sender instance locks and one receiver instance probes at a time, revealing co-residence of this pair, and repeating this serially for each pair. However, this technique is too slow and scales quadratically with the number of instances e.g., a hundred instances would take more than 10 hours assuming 10 secs for each pair. Varadarajan et al. [21] speed up this process significantly by performing detection for mutually-exclusive subsets in parallel, allowing for false-positives and later eliminating the false-positives sequentially. This would only scale linearly in the best case, which is still expensive; with a thousand instances, for example, the whole detection process takes well over two hours to finish, which is infeasible for lambdas that are, by nature, ephemeral. Thus, one challenge in this work is creating a faster co-residence algorithm.

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

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

To solve some of the challenges mentioned previously, we propose a protocol in which we use the memory bus covert channel to exchange information between the instances that have access to the channel. Using the protocol, co-resident instances can reliably

---

#### 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

```

---

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 the total number of instances deployed to the cloud (hundreds to thousands). Removing this dependency on total number of instances deployed allows us scale our co-residence detection significantly, as we will see in the next section.

## 4 NEIGHBOR DISCOVERY PROTOCOL

Co-residence detection for lambdas must be quick, as lambdas are ephemeral by nature. As noted earlier, co-residence detection is faster when co-residing instances on each server communicate among themselves and discover each other. Assuming that the instances have unique IDs, co-resident 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-resident instances can use to achieve communication in a fast and reliable way. We first discuss the challenges we faced in making the channel reliable before examining the protocol itself.

### 4.1 Reliable Transmission

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

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

---

```

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

```

---

481 causes contention on the memory bus by locking it using the spe-  
 482 cial memory locking operations (discussed in section ??). The re-  
 483 ceiver would then sample the memory bus for contention, infer-  
 484 ring whether the communication is a 1-bit (when contention is ob-  
 485 served) or a 0-bit (when contention is not observed). Pseudo-code  
 486 for the sender instance is shown in Algorithm 1.

491 **4.1.1 Sensing contention.** Next, we determine how best for the re-  
 492 ceivers to sense contention on the memory bus. There are two  
 493 ways to achieve this. When the memory bus is locked, any non-  
 494 cached memory accesses will queue and therefore see higher laten-  
 495 cies. The receiver can then continually make un-cached memory  
 496 accesses (referred to as the *memory probing* receiver in previous  
 497 literature [22]) and observe a spike in their latencies to detect con-  
 498 tention. On the other hand, the receiver can also detect memory  
 499 bus contention by using the same memory locking operations as  
 500 the sender (referred to as *memory locking* receiver) to probe the  
 501 memory bus. Since only one processor core can lock the memory  
 502 bus at a given time, any other concurrent locking operation will  
 503 see higher latency.

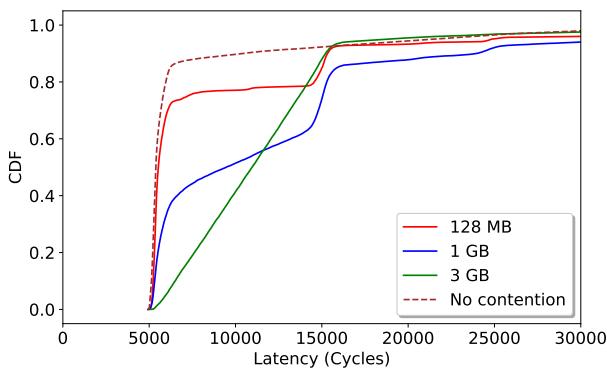
504 Of these two methods, we decide to use the memory locking  
 505 receiver for our experiments. Previous studies [22, 26] have estab-  
 506 lished that both memory probing and memory locking receivers  
 507 experience significant latency overhead during memory bus con-  
 508 tention, making them both viable avenues for sensing the covert-  
 509 channel. Since memory probing involves regular (un-cached) mem-  
 510 ory accesses, it can be done on multiple receivers concurrently  
 511 without affecting each other (due to the high memory bandwidth),  
 512 which prevents noise in measurements. This is an important at-  
 513 tribute, as memory locking receivers must contend with this noise.  
 514 However, bypassing multi-levels of caches in today’s servers to  
 515 perform memory accesses with reliable consistency is a challeng-  
 516 ing task. Even with a reliable cache-bypassing technique, the vari-  
 517 ety of cache architectures and sizes that we encounter on different  
 518 clouds would make tuning the technique to suit these architectures  
 519 an arduous task while reducing the applicability of our overall co-  
 520 residence detection mechanism.

523 **4.1.2 Sampling frequency.** Another challenge for our protocol is  
 524 determining an adequate sampling frequency. Ideally, a memory  
 525 locking receiver would loop locking operations and determine con-  
 526 tention in real-time by identifying a decrease in the moving aver-  
 527 age of the number of operations. Note that, in this case, there is es-  
 528 sentially no difference between the sender and receiver (i.e., both  
 529 continually issue locking operations) except that the receiver is tak-  
 530 ing measurements. This is adequate when there is a single sender  
 531 and receiver [22], but when there are multiple receivers, the mere  
 532 act of sensing the channel by one receiver causes contention and  
 533 other receivers cannot differentiate between a silent (0-bit) and a  
 534 locking (1-bit) sender. To avoid this, we space the sampling of mem-  
 535 ory bus such that no two receivers would sample the bus at the  
 536 same time, with high probability. We achieve this by using large  
 537 intervals between successive samples and a poisson-sampling to  
 538 prevent time-locking of receivers. We determined that a millisecond  
 539 poisson gap between samples is reasonable to minimize noise  
 540 due to collisions in receiver sampling 1, assuming ten co-resident  
 541 receivers and a few microsecond sampling time.

542 **4.1.3 Sample Size.** In addition to adequate sampling frequency,  
 543 we must also determine sample size. A receiver can confirm con-  
 544 tention with high confidence with only a few samples, assuming  
 545 that the sender is actively causing contention on the memory bus  
 546 and the receiver is constantly sampling the memory bus through-  
 547 out the sampling duration. However, in practice, the time-sharing  
 548 of processors produces difficulties. The sender is not continually  
 549 causing contention, and neither is the receiver sensing it, as they  
 550 are context-switched by the scheduler, which runs other processes.  
 551 Assuming that the sender and receiver are running on different  
 552 cores, the amount of time they are actively communicating de-  
 553 pends on the proportion of time they are allocated on each core  
 554 and how they are scheduled.

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

565 **4.1.4 Overcoming noise.** Along with context switching and sens-  
 566 ing noise, there are other imperfections in the measurement ap-  
 567 paratus that may cause noise. For example, we use the difference  
 568 in readings from the timestamp counter of the processor (RDTSC)  
 569 before and after the locking operation to measure the latency of  
 570 the operation in cycles. If the receiver process is context-switched  
 571 in between the timer readings (e.g., at line eight in Algorithm 2),  
 572 the latency measured from their difference will be orders of magni-  
 573 tude higher as it includes the waiting time of the receiver process  
 574 in the scheduler queue - which we believe is what contributes to  
 575 the long tail in Figure 2. To overcome missed samples and noise,  
 576 we record hundreds of samples and compare it to the baseline dis-  
 577 tribution of latencies sampled without contention. We then need  
 578 to compare and differentiate the observed sample of latencies from

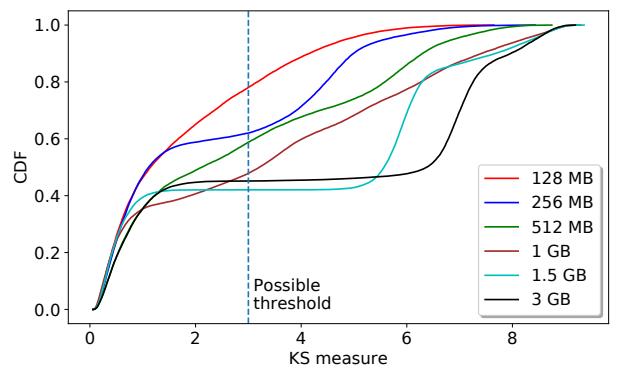


**Figure 2:** We present a CDF of latencies observed by 128 MB, 1 GB and 3 GB Lambdas during contention. The 128 MB lambda pair sees less contention due to more context switching, whereas the 1 GB and 3 GB lambdas see progressively more contention compared to the baseline, which we attribute to their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

the baseline to establish contention. To do this, we use a variant of the two-sample Kolmogorov-Smirnov (KS) test, which typically compares the maximum of the absolute difference between empirical CDFs of samples (in our variant, we take the *mean* of the absolute difference instead of the maximum to reduce sensitivity to outliers). Using this measure, we can categorize a KS-value above a certain threshold as a 1-bit (contention) and a value below the threshold as 0-bit (baseline).

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



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

**4.1.5 Clock synchronization.** Since communicating each bit of information takes time (i.e., receiver sampling duration), our algorithm requires synchronizing the 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 [26] 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. Assuming that clocks between different lambdas only exhibit a drift in the order of microseconds, sampling at a millisecond scale should provide us a margin for synchronization mismatch. Since we do not observe any synchronization-related noise in our results, we believe this is a reasonable assumption.

**4.1.6 Handling Collisions.** In the preceding 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 must next 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 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 collisionsSecond, even

**Algorithm 3** Neighbor discovery protocol

---

```

697 1: sync_point  $\leftarrow$  Start time for all instances
698 2: ID  $\leftarrow$  Instance ID
699 3: N  $\leftarrow$  Number of bits in ID
700 4: advertising  $\leftarrow$  TRUE
701 5: instances  $\leftarrow$  {}
702 6: WAIT_TILL(sync_point)
703 7: while id_read do
704 8:   slots  $\leftarrow$  0
705 9:   id_read  $\leftarrow$  0
706 10:  participating  $\leftarrow$  advertising
707 11:  while slots  $<$  N do
708 12:    bit  $\leftarrow$  slotsth most significant bit of ID
709 13:    if participating and bit then
710 14:      WRITE_BIT() (Alg. 1)
711 15:      bit_read  $\leftarrow$  1
712 16:    else
713 17:      bit_read  $\leftarrow$  READ_BIT() (Alg. 2)
714 18:      if bit_read then
715 19:        participating  $\leftarrow$  FALSE
716 20:      end if
717 21:    end if
718 22:    id_read  $\leftarrow$  2 * id_read + bit_read
719 23:    slots  $\leftarrow$  slots + 1
720 24:  end while
721 25:  if id_read = ID then
722 26:    advertising  $\leftarrow$  FALSE
723 27:  end if
724 28:  instances  $\leftarrow$  instances  $\cup$  {id_read}
725 29: end while
726 30: return instances
727
728
729
730

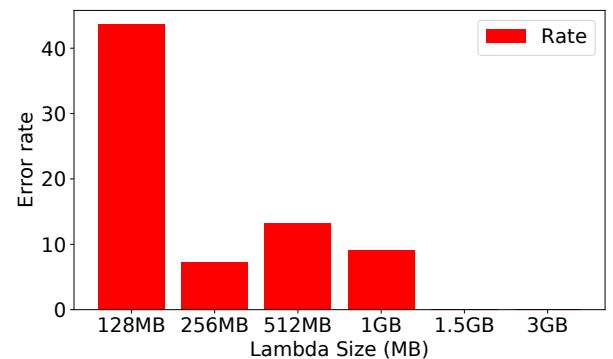
```

---

implementing a random or exponential backoff mechanism will introduce significant overhead before any meaningful communication occurs. This overhead will also increase as the number of instances involved increases. Since each time slot could take up to 1 second, this additional overhead can make the whole communication very slow. We address this by designing a specialized communication protocol that allows for collisions but conversely is less expressive, as we will see in the next section.

## 4.2 Protocol

When considering a communication channel for lambda co-residence detection, we note that the channel need not be general and expressive as lambdas only need to communicate their IDs with one another. Thus, we assume that each instance involved has a unique fixed-length (say *N*) bit-string corresponding to its ID that must be communicated. As such, we propose a communication protocol that exchanges these bit-strings while allowing for collisions. We divide the running time of the protocol into phases, with each phase executing for an interval of *N* bit-slots. Each phase has a set of participating instances, which in the first phase would be all of the co-resident instances. In each bit-slot *K* of *N* slots in a phase, every participating instance broadcasts a bit if the *K*<sup>th</sup> bit of its bit-string (ID) is 1, otherwise it listens for a 0 or 1. If an instance senses



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

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-resident instances), when none of the instances broadcast for *N* consecutive bit-slots. The pseudo-code of the protocol is provided in Algorithm 3. Note that the protocol itself is channel-agnostic and can be extended for other (future) covert channels with similar channel properties.

**4.2.1 Time Complexity.** Assuming *N* total deployed instances to the cloud, the bit-string needs to be  $\log_2 N$  bits to uniquely identify each instance. If a maximum *K* of those instances are launched on the same server, the protocol executes for *K* phases of  $\log_2 N$  bit-slots each, taking  $(K+1) * \log_2 N$  bit-slots for the whole operation. For example, assuming 10,000 deployed lambdas and a maximum of 10 co-resident instances on each server, the entire co-residence detection requires around four minutes to fully execute (with 1-second time slots). In fact, it is not necessary to run the protocol for all *K* phases. After the first phase, all the co-resident instances would know one of their neighbors (as each phase reveals the ID of the biggest participating instance to others). If we use IDs that are globally unique, all the co-resident instances will see the same ID. The instances can then exchange these IDs offline (e.g., through the network) to infer the rest of their neighbors. This simplification removes the dependency on number of co-resident 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 reliability and scalability, the desirable detection properties mentioned in section 3. We run all

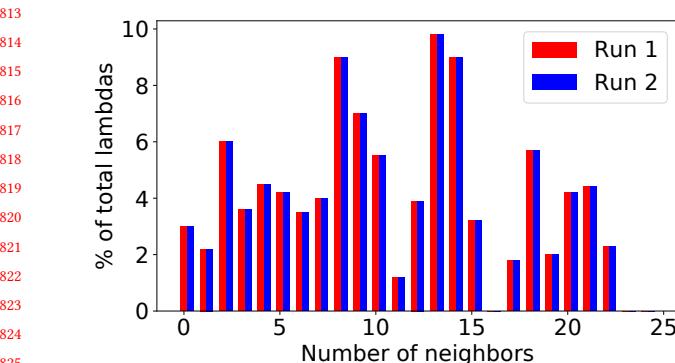


Figure 5: This figure shows the fraction of lambdas by the number of neighbors they identify for two independent runs that use same set of underlying AWS containers. The perfect correlation shows that both runs depict the co-residence status of those containers regardless of the lambdas that ran on them,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

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

## 5.1 Setup

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

## 5.2 Reliability

We consider the results of the technique reliable when 1) most of the deployed instances successfully see the same result in majority of the independent rounds (indicating lesser bit-flip errors) and 2) the resulting co-resident groups we see match the ground truth. For goal #1, we ran an experiment with 1000 AWS lambdas and compared the error rate across different lambda sizes (the error rate indicates the fraction of these 1000 lambdas that did not have a majority result). Figure 4 indicates that smaller lambdas exhibit many 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. Lambdas that are 1.5 GB and larger, though, exhibit a 100% success rate.

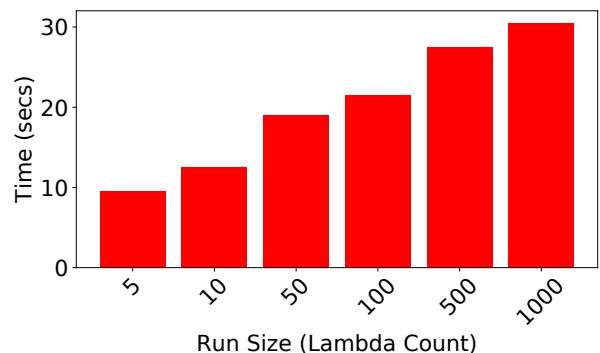


Figure 6: We present the average execution time of the lambdas for co-resident runs with a varying number of lambdas. The execution time increases logarithmically with the number of lambdas demonstrating the scalability of co-residence detection with our technique.

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

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

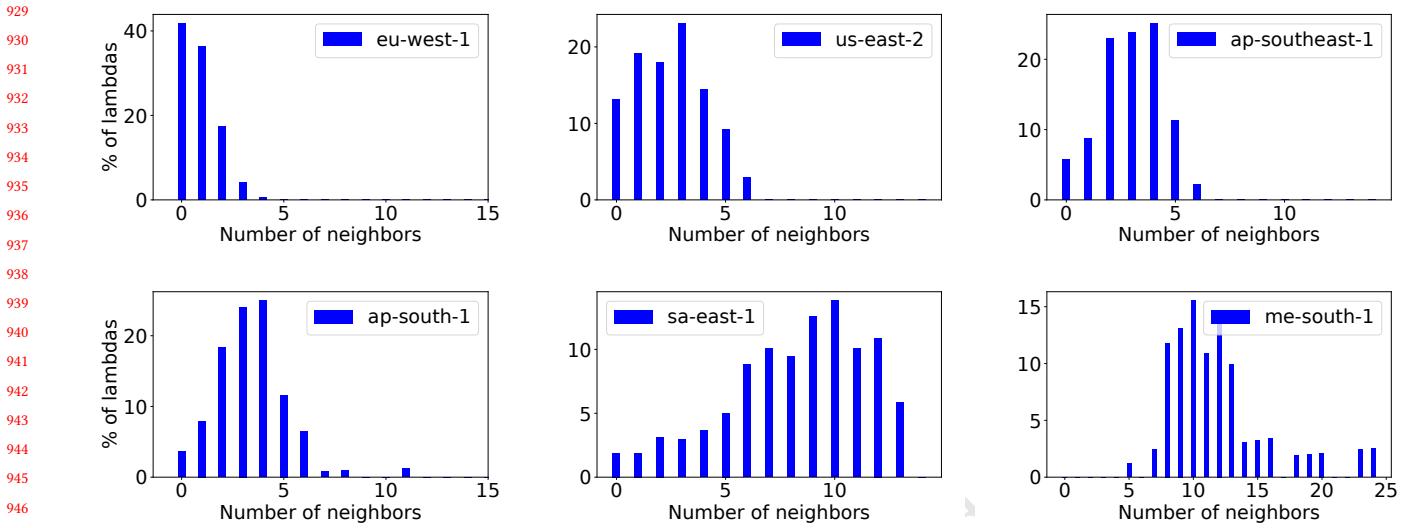


Figure 7: We present co-residence results from a 1000-lambda run in different AWS regions. Each bar shows the fraction of those 1000 lambdas (in %) that discovered a certain number of neighbors. The total amount and density of co-residence vary widely across regions, perhaps based on the size of those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

### 5.3 Scalability

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

## 6 MEASUREMENT STUDY

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

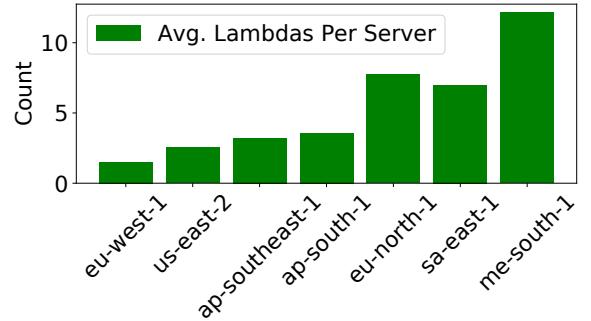


Figure 8: This figure shows the average number of lambdas per server i.e., the co-residence density seen in various AWS regions for the runs shown in Figure 7. The ample co-residence across regions demonstrates the practicality of establishing covert channels with lambdas in these regions.

### 6.1 Across AWS regions

We execute our co-residence detector with 1000 1.5 GB Lambdas in various AWS regions. Figure 7 comprises multiple plots depicting the co-resident groups per region, with each bar indicating the fraction of lambdas that detected a certain number of neighbors (i.e., that belong to a co-resident group of a certain size). Plots that skew to the right indicate a higher co-residence density when compared to the plots skewed to the left (also illustrated in Figure 8). We note that, in most regions, almost all lambdas recognize at least one neighbor (indicated by smaller or non-existent first bar in each plot). We hypothesize that the co-residence density is (inversely)

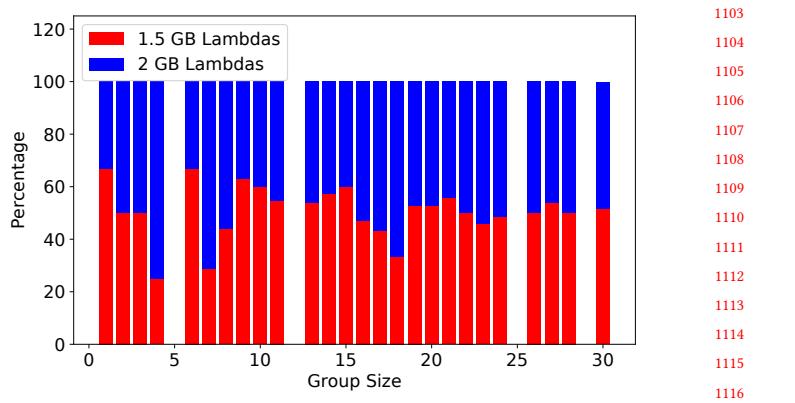
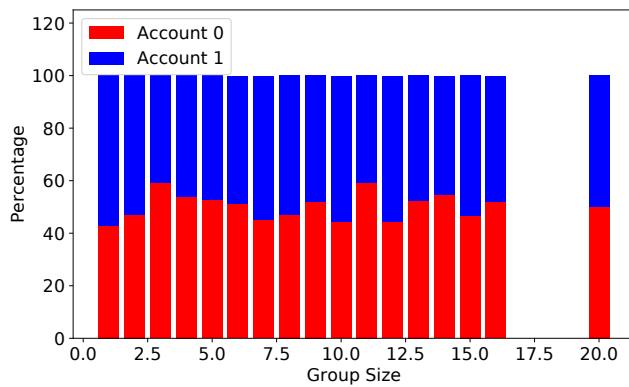
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059

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

dependent on the total number of servers and the lambda activity in the region, both of which can be assumed to be lower in newer AWS regions, hence the higher co-residence density in those regions as we can see in Figure 8. The ample co-residence in general across all the regions shows that lambdas provide a fertile ground for covert channel attacks.

## 6.2 Other factors

We also examine how co-residence is affected by various launch strategies that the user may use, like deploying lambdas from multiple AWS accounts and different lambda sizes. In particular, we wish to determine if our mechanism exhibits different results when: 1) the user deploys sender lambdas and receiver lambdas on two separate accounts (normally the case with covert channels) and 2) the senders and receivers are created with different lambdas sizes. To answer these questions, we run an experiment with 1000 lambdas of which we launch 500 lambdas from one account (senders) and 500 from other deployed in a random order. The co-residence observed was comparable to the case where all the lambdas were launched from one account. In the left subfigure of Figure 9, we show the breakdown of co-resident group of lambdas of each size among the two accounts. We can see that among the co-resident groups of all sizes, roughly half of lambdas came from either account. This shows that lambda scheduler is agnostic to the accounts the lambdas were launched from. We see similar results for different lambda sizes, as shown in the right subfigure of Figure 9.

## 7 DISCUSSION

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

- While our co-residence detector does not directly help attackers locate their victims in the cloud, it can aid them

in performing devastating DDOS attacks once by concentrating a number of attack instances on the victim machine. Also, the attacker could try to gain a wider surface area for targeted attacks in a cost-effective way by turning on/off her co-resident instances as necessary.

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

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

## 8 RELATED WORK

**Cloud Attacks** Co-residency is possible because of covert channels, so we begin our related work with an investigation into cloud attacks. Initial papers in co-residency detection utilized host information and network addresses arising due to imperfect virtualization [19]. However, these channels are now obsolete, as cloud providers have strengthened virtualization and introduced Virtual Private Clouds [4]. Later work used cache-based channels in various

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

levels of the cache [12, 15, 28, 35] and hardware based channels like thermal covert channels [17], RNG module [7] and memory bus [26] have also been explored in the recent past. Moreover, studies have found that VM performance can be significantly degraded using memory DDoS attacks [31], while containers are susceptible to power attacks from adjacent containers [9].

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

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

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

## 9 ETHICAL CONSIDERATIONS

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

are generally not recommended for latency-sensitive workloads, due to their cold-start latencies. Thus, the small amount of lambdas that we might affect should not, in practice, be affected in their longterm computational goals.

## 10 CONCLUSION

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

## REFERENCES

- [1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. <https://www.usenix.org/conference/nsdi20/presentation/agache>
- [2] aws 2018. Understanding Container Reuse in AWS Lambda. <https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/>.
- [3] aws 2019. Amazon EC2. <https://aws.amazon.com/ec2/>.
- [4] aws 2019. Amazon VPC. <https://aws.amazon.com/vpc/>.
- [5] azure 2019. Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>.
- [6] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. 2012. Detecting co-residency with active traffic analysis techniques. <https://doi.org/10.1145/2381913.2381915>
- [7] Dmitry 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] Sajjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. <https://www.usenix.org/conference/atc19/presentation/fouladi>
- [9] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2017. ContainerLeaks: Emerging Security Threats of Information Leaks in Container Clouds. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, IEEE, 237–248.
- [10] gcp 2019. Google Cloud Functions. <https://cloud.google.com/functions/>.
- [11] Elizabeth Izhikevich. 2018. *Building and Breaking Burst-Parallel Systems*. Master's thesis, University of California, San Diego.
- [12] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. 1–6. <https://doi.org/10.1145/2897937.2897962>
- [13] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. 2015. Lambda architecture for cost-effective batch and speed big data processing. *2015 IEEE International Conference on Big Data (Big Data) (2015)*, 2785–2792.
- [14] Lambda 2019. AWS Lambda. <https://aws.amazon.com/lambda/>.
- [15] Fangfei Liu, Yuval Yarom, Qian ge, Gernot Heiser, and Ruby Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical, Vol. 2015. 605–622. <https://doi.org/10.1109/SP.2015.43>
- [16] Ramya Jayaram Masti, Devendra Rai, Aanjan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-core Platforms. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 865–880. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti>
- [17] Ramya Jayaram Masti, Devendra Rai, 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>
- [18] Clementine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. <https://doi.org/10.14722/ndss.2017.23294>

- 1277 [19] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, 1335  
 1278 You, Get off of My Cloud: Exploring Information Leakage in Third-party 1336  
 1279 Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications 1337*  
 1280 Security (Chicago, Illinois, USA) (CCS '09). ACM, New York, NY, USA, 199–212. 1338  
<https://doi.org/10.1145/1653662.1653687> 1339  
 1281 [20] Brendan Saltaformaggio, D. Xu, and X. Zhang. 2013. BusMonitor : A Hypervisor- 1340  
 1282 Based Solution for Memory Bus Covert Channels. 1341  
 1283 [21] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael 1342  
 1284 Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. 1343  
 1285 In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 1344  
 1286 Washington, D.C., 913–928. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan> 1345  
 1287 [22] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. 1346  
 1288 Swift. 2015. A Placement Vulnerability Study in Multi-tenant Public Clouds. 1347  
*CoRR* abs/1507.03114. arXiv:1507.03114 <http://arxiv.org/abs/1507.03114> 1348  
 1289 [23] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, 1349  
 Mauricio Verano Merino, Ruby Casallas, Santiago Gil, Carlos Valencia, Angee 1350  
 Zambrano, and Mery Lang. 2016. Infrastructure Cost Comparison of Running 1351  
 Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice 1352  
 Architectures. <https://doi.org/10.1109/CCGrid.2016.37> 1353  
 1290 [24] Liang Wang, Mengyuan Li, Yingqian Zhang, Thomas Ristenpart, and Michael 1354  
 Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX 1355*  
 Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA, 1356  
 133–146. <https://www.usenix.org/conference/atc18/presentation/wang-liang> 1357  
 1291 [25] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to 1358  
 Processor Architecture. In *Proceedings of the 22nd Annual Computer Security 1359*  
 Applications Conference (ACSAC '06). IEEE Computer Society, USA, 473–482. 1360  
<https://doi.org/10.1109/ACSAC.2006.20> 1361  
 1292 [26] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: 1362  
 High-speed Covert Channel Attacks in the Cloud. In *Presented as part of the 21st 1363*  
 USENIX Security Symposium (USENIX Security 12). USENIX, Bellevue, WA, 159– 1364  
 1300 173. <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu> 1365  
 1301 [27] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, 1366  
 and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels 1367  
 in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud 1368*  
 Computing Security Workshop (Chicago, Illinois, USA) (CCSW '11). Association 1369  
 for Computing Machinery, New York, NY, USA, 29–40. <https://doi.org/10.1145/2046660.2046670> 1370  
 1302 [28] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, 1371  
 and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels 1372  
 in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud 1373*  
 Computing Security Workshop (Chicago, Illinois, USA) (CCSW '11). ACM, New 1374  
 York, NY, USA, 29–40. <https://doi.org/10.1145/2046660.2046670> 1375  
 1303 [29] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co- 1376  
 residence Threat inside the Cloud. In *24th USENIX Security Symposium (USENIX 1377*  
 Security 15). USENIX Association, Washington, D.C., 929–944. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu> 1378  
 1304 [30] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co- 1379  
 residence Threat inside the Cloud. In *24th USENIX Security Symposium (USENIX 1380*  
 Security 15). USENIX Association, Washington, D.C., 929–944. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu> 1381  
 1305 [31] Tianwei Zhang, Yinqian Zhang, and Ruby Lee. 2016. Memory DoS Attacks in 1382  
 Multi-tenant Clouds: Severity and Mitigation. 1383  
 1306 [32] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Ming- 1384  
 sheng Wang, and Peng Liu. 2016. A Comprehensive Study of Co-residence 1385  
 Threat in Multi-tenant Public PaaS Clouds. In *Information and Communications 1386*  
 Security, Kwok-Yan Lam, Chi-Hung Chi, and Sihan Qing (Eds.). Springer International Publishing, Cham, 361–375. 1387  
 1307 [33] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Ming- 1388  
 sheng Wang, and Peng Liu. 2016. A Comprehensive Study of Co-residence 1389  
 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) 1390  
 1308 [34] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael Reiter. 2011. HomeAlone: Co- 1391  
 Residency Detection in the Cloud via Side-Channel Analysis. 313 – 328. <https://doi.org/10.1109/SP.2011.31> 1392  
 1309 [35] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross- 1393  
 Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM 1394*  
 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> 1395  
 1310 1396  
 1311 1397  
 1312 1398  
 1313 1399  
 1314 1400  
 1315 1401  
 1316 1402  
 1317 1403  
 1318 1404  
 1319 1405  
 1320 1406  
 1321 1407  
 1322 1408  
 1323 1409  
 1324 1410  
 1325 1411  
 1326 1412  
 1327 1413  
 1328 1414  
 1329 1415  
 1330 1416  
 1331 1417  
 1332 1418  
 1333 1419  
 1334 1420