Peeking Behind the Curtains of Serverless Platforms

Liang Wang, ***UW-Madison;*** **Mengyuan Li and Yinqian Zhang,** ***The Ohio State University;***
Thomas Ristenpart, ***Cornell Tech;*** **Michael Swift,** ***UW-Madison***

https://www.usenix.org/conference/atc18/presentation/wang-liang

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Peeking Behind the Curtains of Serverless Platforms

Liang Wang [1], Mengyuan Li [2], Yinqian Zhang [2], Thomas Ristenpart[3], Michael Swift[1]

[1]UW-Madison, [2]Ohio State University, [3]Cornell Tech

Abstract

Serverless computing is an emerging paradigm in which an application's resource provisioning and scaling are managed by third-party services. Examples include AWS Lambda, Azure Functions, and Google Cloud Functions. Behind these services' easy-to-use APIs are opaque, complex infrastructure and management ecosystems. Taking on the viewpoint of a serverless customer, we conduct the largest measurement study to date, launching more than 50,000 function instances across these three services, in order to characterize their architectures, performance, and resource management efficiency. We explain how the platforms isolate the functions of different accounts, using either virtual machines or containers, which has important security implications. We characterize performance in terms of scalability, coldstart latency, and resource efficiency, with highlights including that AWS Lambda adopts a bin-packing-like strategy to maximize VM memory utilization, that severe contention between functions can arise in AWS and Azure, and that Google had bugs that allow customers to use resources for free.

1 Introduction

Cloud computing has allowed backend infrastructure maintenance to become increasingly decoupled from application development. Serverless computing (or function-as-a-service, FaaS) is an emerging application deployment architecture that completely hides server management from tenants (hence the name). Tenants receive minimal access to an application's runtime configuration. This allows tenants to focus on developing their functions — small applications dedicated to specific tasks. A function usually executes in a dedicated *function instance* (a container or other kind of sandbox) with restricted resources such as CPU time and memory. Unlike virtual machines (VMs) in more traditional infrastructure-as-a-service (IaaS) platforms, a function instance will be launched only when the function is invoked and is put to sleep immediately after handling a request. Tenants are charged on a per-invocation basis, without paying for unused and idle resources.

Serverless computing originated as a design pattern for handling low duty-cycle workloads, such as processing in response to infrequent changes to files stored on the cloud. Now it is used as a simple programming model for a variety of applications [14, 22, 42]. Hiding resource management from tenants enables this programming model, but the resulting opacity hinders adoption for many potential users, who have expressed concerns about: security in terms of the quality of isolation, DDoS resistance, and more [23, 35, 37, 40]; the need to understand resource management to improve application performance [4, 19, 24, 27, 28, 40]; and the ability of platforms to deliver on performance [10–12, 29–31]. While attempts have been made to shed light on platforms' resource management and security [33, 34], known measurement techniques, as we will show, fail to provide accurate results.

We therefore perform the most in-depth study of resource management and performance isolation to date in three popular serverless computing providers: AWS Lambda, Azure Functions, and Google Cloud Functions (GCF). We first use measurement-driven approaches to partially reverse-engineer the architectures of Lambda and Azure Functions, uncovering many undocumented details. Then, we systematically examine a series of issues related to resource management: how quickly function instances can be launched, function instance placement strategies, function instance reuse, and more. Several security issues are identified and discussed.[1] We further explore how CPU, I/O and network bandwidth are allocated among functions and the ensuing performance implications. Last but not least, we explore whether all resources are properly accounted for, and report on two resource accounting bugs that allow tenants to use extra resources for free. Some highlights of our results include:

- AWS Lambda achieved the best scalability and the lowest coldstart latency (the time to provision a new function instance), followed by GCF. But

[1]We responsibly disclosed our findings to related parties before this paper was made public.

the lack of performance isolation in AWS between function instances from the same account caused up to a 19x decrease in I/O, networking, or coldstart performance.

- Azure Functions used different types of VMs as hosts: 55% of the time a function instance runs on a VM with debased performance.
- Azure had exploitable placement vulnerabilities [36]: a tenant can arrange for function instances to run on the same VM as another tenant's, which is a stepping stone towards cross-function side-channel attacks.
- An accounting issue in GCF enabled one to use a function instance to achieve the same computing resources as a small VM instance at almost no cost.

Many more results are given in the body. We have repeated several measurements in May 2018 and highlight in the paper the improvements the providers have made. We noticed that serverless platforms are evolving quickly; nevertheless, our findings serve as a snapshot of the resource management mechanisms and efficiency of popular serverless platforms, provide performance baselines and design options for developers to build more reliable platforms, and help tenants improve their use of serverless platforms. More generally, our study provides new measurement techniques that are useful for other researchers. Towards facilitating this, we will make our measurement code public and open source.[2]

2 Background

Serverless computing platforms. In serverless computing, an application usually consists of one or more ***functions*** — standalone, small, stateless components dedicated to handle specific tasks. A function is most often specified by a small piece of code written in some scripting language. Serverless computing providers manage the execution environments and backend servers of functions, and allocate resources dynamically to ensure their scalability and availability.

In recent years, many serverless computing platforms have been developed and deployed by cloud providers, including Amazon, Azure, Google, and IBM. We focus on Amazon AWS Lambda, Azure Functions and Google Cloud Functions.[3] In these services, a function is executed in a dedicated container or other type of sandbox with limited resources. We use ***function instance*** to refer to the container/sandbox a function runs on. The resources advertised as available to a function instance varies across platforms, as shown in Table 1. When the function is invoked by requests, one or more function instances (depending on the request volume) will be launched to execute the function. After the function instance(s) have processed the requests and exited or reached the maximum execution time (see "Timeout" in Table 1), the function instance(s) becomes idle. They may be reused to handle subsequent requests to avoid the delay of launching new instances. However, idle function instances can also be suddenly terminated [32]. Each function instance is associated with a non-persistent local disk for temporarily storing data, which will be erased when the function instance is destroyed.

	AWS	Azure	Google
Memory (MB)	64 * k (k = 2, 3, ..., 24)	1536	128 * k (k = 1, 2, 4, 8, 16)
CPU	Proportional to Memory	Unknown	Proportional to Memory
Language	Python 2.7/3.6 Nodejs 4.3.2/6.10.3 Java 8, and others	Nodejs 6.11.5, Python 2.7, and others	Nodejs 6.5.0
Runtime OS	Amazon Linux	Windows 10	Debian 8*
Local disk (MB)	512	500	> 512
Run native code	Yes	Yes	Yes
Timeout (second)	300	600	540
Billing factor	Execution time Allocated memory	Execution time Consumed memory	Execution time Allocated memory Allocated CPU

Table 1: A comparison of function configuration and billing in three services. (*: We infer the OS version of GCF by checking the help information and version of several Linux tools such as `APT`.)

One benefit of using serverless services is that tenants do not pay for resources consumed when function instances are idle. Tenants are billed based on resource consumption only during execution.[4] In common across platforms is charging for aggregated function execution time across all invocations. Additionally, the price varies depending on the pre-configured function memory (AWS, Google) or the actual consumed memory during invocations (Azure). Google further charges different rates based on CPU speed.

Related work. Many serverless application developers have conducted their own experiments to measure coldstart latency, function instance lifetime, maximum idle time before shut down, and CPU usage in AWS Lambda [10–12, 19, 27, 28, 40]. Unfortunately, their experiments were ad-hoc, and the results may be misleading because they did not control for contention by other instances. A few research papers report on measured performance in AWS. Hendrickson et al. [18] measured request latency and found it had higher latency than AWS Elastic Beanstalk (a platform-as-a-service system). McGrath et al. [34] conducted preliminary measurements on four serverless platforms, and found

[2] `https://github.com/liangw89/faas_measure`

[3] We use AWS, Azure and Google to refer to these services.

[4] Azure Functions offers two types of function hosting plans. *Consumption Plan* manages resources in a serverless-like way while *App Service Plan* is more like "container-as-a-service". We only consider Consumption Plan in this paper.

that AWS achieved better scalability, coldstart latency, and throughput than Azure and Google.

A concurrent study from Lloyd et al. [33] investigated the factors that affect application performance in AWS and Azure. The authors developed a heuristic to identify the VM a function runs on in AWS based on the VM uptime in `/proc/stat`. Our experimental evaluation suggests that their heuristic is unreliable (see §4.5), and that the conclusions they made using it are mostly inaccurate.

In our work, we design a reliable method for identifying instance hosts, and use systematic experiments to inspect resource scheduling and utilization.

3 Methodology

We take the viewpoint of a serverless user to characterize serverless platforms' architectures, performance, and resource management efficiency. We set up vantage points in the same cloud provider region to manage and invoke functions from one or more accounts via official APIs, and leverage the information available to functions to determine important characteristics. We repeated the same experiment under various settings by adjusting function configuration and workloads to determine the key factors that could affect measurement results. In the rest of the paper, we only report on the relevant factors affecting the experiment results.

We integrate all the necessary functionalities and subroutines into a single function that we call a *measurement function*. A measurement function performs two tasks: (1) collect invocation timing and function instance runtime information, and (2) run specified subroutines (e.g., measuring local disk I/O throughput, network throughput) based on received messages. The measurement function collects runtime information via the proc filesystem on Linux (`procfs`), environment variables, and system commands. It also reports on execution start and end time, invocation ID (a random 16-byte ASCII string generated by the function that uniquely identify an invocation), and function configurations to facilitate further analysis.

The measurement function checks the existence of a file named *InstanceID* on the local disk, and if it does not exist, creates this file with a random 16-byte ASCII string that serves as the function instance ID. Since the local disk is non-persistent and has the same lifetime as the associated function instance, the *InstanceID* file will not exist for a fresh function instance, and will not be modified or deleted during the function instance lifetime once created.

The regions for functions were us-east-1, us-central-1, "EAST US" in AWS, Google and Azure (respectively). The vantage points were VMs with at least 4 GB RAM and 2 vCPUs. We used the software recommended by the

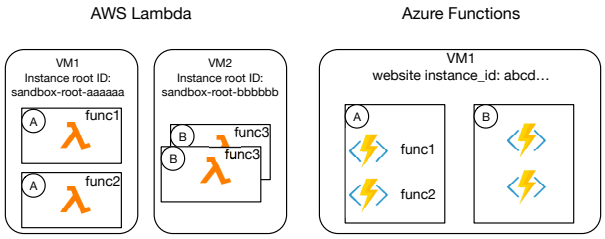


Figure 2: VM and function instance organization in AWS Lambda and Azure Functions. A rectangle represents a function instance. A or B indicates different tenants.

providers and follow the official instructions to configure the time synchronization service in the vantage points. [5]

We implemented the measurement function in various languages, but most experiments used Python 2.7 and Nodejs 6.* as the language runtime (the top 2 most popular languages in AWS according to Newrelic [25]). We invoked the functions via synchronous HTTP requests. Most of our measurements were done from July–Dec 2017.

Ethical considerations. We built our measurement functions in a way that should not cause undue burden on platforms or other tenants. In most experiments, the function did no more than collecting necessary information and sleeping for a certain amount of time. Once we discovered performance issues we limited our tests to not DoS other tenants. We only conducted small-scale tests to inspect the security issues but did not further exploit them.

4 Serverless Architectures Demystified

We combine two approaches to infer the architectures of AWS Lambda, Google Cloud Functions, and Azure Functions: (1) reviewing official documents, related online articles and discussions, and (2) measurements — analyzing the data collected from running our measurement functions many times ($>$ 50,000) under varying conditions. This data enables partially reverse engineering the architectures of AWS, Azure, and Google.

4.1 Overview

AWS. A function executes in a dedicated function instance. Our measurements suggest different versions of a function will be treated as distinct and executed in different function instances (we discuss outliers in §5.5). The `procfs` file system exposes global statistics of the underlying VM host, not just a function instance, and contains useful information for profiling runtime,

[5] AWS: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html`; Google: `https://developers.google.com/time/`; Azure does not offer instructions so we use the default NTP servers at `http://www.pool.ntp.org/en/use.html`

(1) Set up N distinct functions $f_1, ..., f_N$ that run the following task upon receiving a `RUN` message: record `/proc/diskstats`, write 20 K – 30 K times to a file (1 byte each time), and record `/proc/diskstats` again.

(2) Invoke each function once without `RUN` message to launch N function instances.

(3) Assuming the instances of $f_1, ..., f_k$ (k instances) share the same instance root ID, invoke $f_1, ..., f_k$ once each with the `RUN` message and examine I/O statistics of each function instance.

Figure 3: I/O-based coresidency test in **AWS**.

identifying host VMs, and more. From `procfs`, we found host VMs mostly have 2 vCPUs and 3.75 GB physical RAM (same as EC2 c4.large instances).

Azure. Azure Functions uses *Function Apps* to organize functions. A function app, corresponding to one function instance, is a container that contains the execution environments for individual functions [5]. The environment variables in the function instance contain some global information about the host VM. The environment variables collected suggest the host VMs can have 1, 2 or 4 vCPUs.

One can create multiple functions in a function app and run them concurrently. In our experiments, we assume that a function app has only one function.

Google. Google isolates and filters information that can be accessed from `procfs`. The files under `procfs` only report usage statistics of the current function instance. Also, many system files and syscalls are obscured or disabled so we cannot get much information about runtime. The `/proc/meminfo` and `/proc/cpuinfo` files suggest a function instance has 2 GB RAM and 8 vCPUs, which we suspect is the configuration for VMs.

4.2 VM identification

Associating function instances with VMs enables us to perform richer analysis. The heuristic for identifying VMs in AWS Lambda proposed by Lloyd et al., though theoretically possible, has never been evaluated experimentally [33]. Therefore, we looked for a more robust method.

AWS. The `/proc/self/cgroup` file has a special entry that we call *instance root ID*. It starts with "sandbox-root-" followed by a 6-byte random string. We found it can be used to reliably identity a host VM. Using the I/O-based coresidency tests (shown in Figure 3), we confirmed that the instances sharing the same instance root ID are on the same VM, as the difference in the total bytes written between two consecutive invocations, for f_i and f_{i+1} respectively, is almost the same as the number of bytes written by f_i. Moreover, we can get the same kernel uptime (or memory usage statistics) from the instances when reading `/proc/uptime` (`/proc/meminfo`) at the same time.

We call the IP obtained via querying IP address lookup tools from an instance *VM public IP*, and the IP obtained from running `uname` command *VM private IP*. Function instances that share the same instance root ID have the same VM public IP and VM private IP.

Azure. The `WEBSITE_INSTANCE_ID` environment variable serves as the VM identifier, according to official documents [6]. We refer to it as *Azure VM ID*. We used Flush-Reload via shared DLLs to verify coresidency of instances sharing the same Azure VM ID [43]. The results suggest Azure VM ID is a robust VM identifier.

Google. We could not find any information enabling us to identify a host. Using I/O-based coresidency did not work as `procfs` contains no global usage statistics. We tried to use performance as a covert-channel (e.g., performing patterned I/O operations in one function instance and detecting the pattern from I/O throughput variation in another) but found this is not reliable, as performance varied greatly (See §6.2).

4.3 Tenant isolation

Prior studies showed that co-located VMs in AWS EC2 allow attacks [36, 38, 41]. With the knowledge of instance-VM relationship, we examined how well tenants' primary resources — function instances — are isolated. We assume that one tenant corresponds to one user account, and only consider VM-level coresidency.

AWS. The functions created by the same tenant will share the same set of VMs, regardless of their configurations and code. The detailed instance placement algorithm will be discussed in §5.1. AWS assigns different VMs to each tenant, since we have never seen function instances from different tenants in the same VM. We conducted a cross-tenant coresidency test to confirm this assumption. The basic principle is similar to Figure 3: in each round, we create a new function under each of the two accounts at the same time, write a random number of bytes in one function, and check the disk usage statistics in another function. We ran this test for 1 week, but found no VM-coresidency of cross-tenant function instances.

Azure. Azure Functions are a part of the Azure App service, in which all tenants share the same set of VMs according to Azure [2]. Hence, tenants in Azure Functions should also share VM resources. A simple test confirmed this assumption: we invoked 500 functions in each of two accounts and found that 30% of function instances were coresident with a function instance from the other account, executing in a total of 120 VMs. Note that as of May 2018, different tenants no longer share the same VMs in Azure. See §5.1 for more details.

4.4 Heterogeneous infrastructure

We found the VMs in all the considered services had a variety of configurations. The variety, likely resulting from infrastructure upgrades, can cause inconsistent function performance. To estimate the fraction of different types of VM in a given service, we examined the configurations of the host VMs of 50,000 unique function instances in each service.

In AWS, we checked the *model_name* and the processor numbers in the `/proc/cpuinfo`, and the *MemTotal* in the `/proc/meminfo`, and found five types of VMs: two E5-2666 vCPUs (2.90 GHZ), two E5-2680 vCPUs (2.80 GHZ), two E5-2676 vCPUs (2.40 GHZ), two E5-2686 vCPUs (2.30 GHZ), and **one** E5-2676 vCPUs. These types account for 59.3%, 37.5%, 3.1%, 0.09% and 0.01% of 20,447 distinct VMs.

Azure shows a greater diversity of VM configurations. The instances in Azure report various vCPU counts: of 4,104 unique VMs, 54.1% use 1 vCPU, 24.6% use 2 vCPUs, and 21.3% use 4 vCPUs. For a given vCPU count, there are three CPU models: two Intel and one AMD. Thus, nine (at least) different types of VMs are being used in Azure. Performance may vary substantially based on what kind of host (more specifically, the number of vCPUs) runs the function. See §6 for more details.

In Google, the *model_name* is always "unknown", but there are 4 unique model versions (79, 85, 63, 45), corresponding to 47.1%, 44.7%, 4.2%, and 4.0% of selected function instances.

4.5 Discussion

Being able to identify VMs in AWS is essential for our measurements. It helps to reduce noise in experiments and get more accurate results. For the sake of comparison, we evaluated the heuristic designed by Lloyd et al. [33]. The heuristic assumes that different VMs have distinct boot times, which can be obtained from `/proc/stat`, and group function instances based on the boot time. We sent 10 – 50 concurrent requests at a time to 1536 MB functions for 100 rounds, used our methodology (instance root ID + IP) to label the VMs, and compared against the heuristic. The heuristic identified 940 VMs as 600 VMs, so 340 (36%) VMs were incorrectly labeled. So, we conclude this heuristic is not reliable.

None of these serverless providers completely hide runtime information from tenants. More knowledge of instance runtime and the backend infrastructure could make finding vulnerabilities in function instances easier for an adversary. In prior studies, `procfs` has been used as a side-channel [9, 21, 46]. In the serverless setting, one actually can use it to monitor the activity of coresident instances; while seemingly harmless, a dedicated adversary might use it as a steppingstone to more sophisticated attacks. Overall, accesses to runtime information, unless necessary, should be restricted for security purposes. Additionally, providers should expose such information in an auditable way, i.e., via API calls, so they are able to detect and block suspicious behaviors.

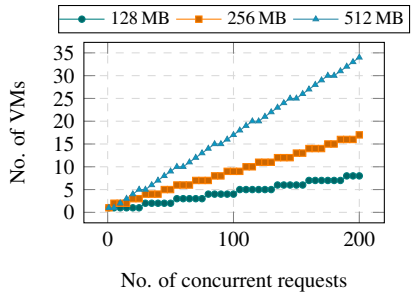


Figure 4: The total number of VMs being used after sending a given number of concurrent requests in **AWS**.

5 Resource Scheduling

We examine how instances and VMs are scheduled in the three serverless platforms in terms of instance coldstart latency, lifetime, scalability, and more.

5.1 Scalability and instance placement

Elastic, automatic scaling in response to changes in demand is a main advertised benefit of the serverless model. We measure how well platforms scale up.

We created 40 measurement functions of the same memory size $f_1, f_2, \ldots, f_{40}$ and invoked each f_i with $5i$ concurrent requests. We paused for 10 seconds between batches of invocations to cope with rate limits in the platforms. All measurement functions simply sleep for 15 seconds and then return. For each configuration we performed 50 rounds of measurements.

AWS. AWS is the best among the three services with regard to supporting concurrent execution. In our measurements, N concurrent invocations always produced N concurrently running function instances. AWS could easily scale up to 200 (the maximum measured concurrency level) fresh function instances.

We observed that **3,328 MB** was the maximum aggregate memory that can be allocated across all function instances on any VM in AWS Lambda. AWS Lambda appears to treat instance placement as a bin-packing problem, and tries to place a new function instance on an existing active VM to maximize *VM memory utilization rates*, i.e., the sum of instance memory sizes divided by 3,328. We invoked a single function with sets of concurrent requests, increasing from 5 to 200 with a step of 5, and recorded the total number of VMs being used after each number of requests. A few examples are shown in Figure 4.

#vCPU	Total	1	2	3	4	>4
1	61.3	16.6	24.6	13.7	4.9	1.5
2	19.5	7.3	7.1	3.3	1.4	0.4
4	19.2	7.6	6.2	3.9	1.3	0.2
All	100	31.5	37.9	20.9	7.6	2.1

Table 5: The average (over 10 runs) probabilities (as percentages) of getting N-way single-account coresidency (for $N \in \{1,2,3,4,\}$ and $N > 4$, when launching 1,000 function instances in **Azure**. Here $N = 1$ indicates no coresidency among the functions.

The number of active VMs are close to the "expected" number if AWS maximizes VM memory utilization. Quantitatively speaking, more than 89% of VMs we got in the test achieved 100% memory utilization. Sending concurrent requests to different functions resulted in the same pattern, indicating placement is agnostic to function code.

In a further test we sent 10 sets of random numbers of concurrent requests to randomly-chosen functions of varied memory sizes over 50 runs. AWS's placement still worked efficiently: the average VM memory utilization rate across VMs in the same run ranged from 84.6% to 100%, with a median of 96.2%.

Azure. Azure documentation states that it will automatically scale up to at most 200 instances for a single Nodejs-based function and at most one new function instance can be launched every 10 seconds [7]. However, in our tests of Nodejs-based functions, we saw at most 10 function instances running concurrently for a single function, no matter how we changed the interval between invocations. All the requests were handled by a small set of function instances. None of the concurrently running instances were on the same VM. So, it appears that Azure does not try to co-locate function instances of the same function on the same VMs.

We conducted a single-account coresidency test to examine how function instances are placed on VMs of different numbers of vCPUs. We invoked 100 different functions from one account at a time until we had 1,000 concurrent, distinct function instances running. We then checked for co-residency, and repeated the entire test 10 times.

We observed at most 8 instances on a single 1/2/4-vCPU VM. Co-resident instances tend to be on 1-vCPU VMs (presumably because there are more 1-vCPU VMs for Azure Functions). We show the breakdown of co-residency results in Table 5. In general, co-residency is undesirable for users wanting many function instances, as contention between instances on low-end VMs will exacerbate performance issues.

We further conducted a cross-account coresidency test in a more realistic scenario where an attacker wants to place her function instances on the same VM with the instances of a target victim. In each round of this test, we launched either 5 or 100 function instances from one account (the *victim*) and 500 simultaneous function instances from another account (the *attacker*). On average, 0.12% (3.82%) of the 500 attacker instances were coresident with the 5 (100) victim instances in each round (10 rounds in total). So, it was possible to achieve cross-tenant coresidency even for a few targets. In the test with 100 victim instances, we were able to obtain up to 5 attacker instances on the same VM. Security implications will be discussed in §5.6.

We repeated the coresidency tests in May 2018 but could not find any cross-tenant coresident instances, even in the test in which we tired 500 victim instances. Therefore, we believe that Azure has fixed the cross-tenant coresidency issue.

Google. Google failed to provide our desired scalability, even though Google claims HTTP-triggered functions will scale to the desired invocation rate quickly [13]. In general, only about half of the expected number of instances, even for a low concurrency level (e.g., 10), could be launched at the same time, while the remainder of the requests were queued.

5.2 Coldstart and VM provisioning

We use *coldstart* to refer to the process of launching a new function instance. For the platform, a coldstart may involve launching a new container, setting up the runtime environment, and deploying a function, which will take more time to handle a request than reusing an existing function instance (*warmstart*). Thus, coldstarts can significantly affect application responsiveness and, in turn, user experience.

For each platform, we created 1,000 distinct functions of the same memory and language and sequentially invoked each of them twice to collect its coldstart and warmstart latency. We use the difference of invocation send time (recorded by the vantage point) and function execution start time (recorded by the function) as an estimation of its coldstart/warmstart latency. As baselines, the median warmstart latency in AWS, Google, and Azure were about 25, 79 and 320 ms (respectively) across all invocations.

AWS. We examine two types of coldstart events: a function instance is launched (1) on a new VM that we have never seen before and (2) on an existing VM. Intuitively, case (1) should have significantly longer coldstart latency than (2) because case (1) may involve starting a new VM. However, we found case (1) was only slightly longer than (2) in general. The median coldstart latency in case (1) was only 39 ms longer than (2) (across all settings). Plus, the smallest VM kernel uptime (from `/proc/uptime`) we found was 132 seconds, indicating that the VM has been launched before the invocation.

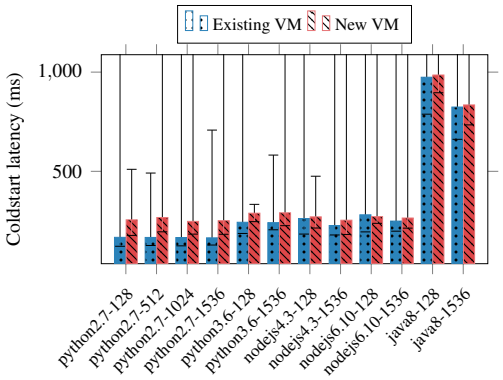


Figure 6: Median coldstart latency with min-max error bars (across 1,000 rounds) under different combinations of function languages and memory sizes in **AWS**. Y-axis is truncated at 1,000 ms.

So, AWS has a pool of ready VMs. The extra delays in case (1) are more likely introduced by scheduling (e.g., selecting a VM) rather than launching a VM.

Our results are consistent with prior observations: function memory and language affect coldstart latency [10], as shown in Figure 6. Python 2.7 achieves the lowest median coldstart latencies (167–171 ms) while Java functions have significantly higher latencies than other languages (824–974 ms). Coldstart latency generally decreases as function memory increases. One possible explanation is that AWS allocates CPU power proportionally to the memory size; with more CPU power, environment set up becomes faster (see §6.1).

A number of function instances may be launched on the same VM concurrently, due to AWS's instance placement strategy. In this case, the coldstart latency increases as more instances are launched simultaneously. For example, launching 20 function instances of a Python 2.7-based function with 128 MB memory on a given VM took 1,321 ms on average, which is about 7 times slower than launching 1 function instance on the same VM (186 ms).

Azure and Google. The median coldstart latency in Google ranged from 110 ms to 493 ms (see Table 7). Google also allocates CPU proportionally to memory, but in Google memory size has greater impact on coldstart latency than in AWS. It took much longer to launch a function instance in Azure, though their instances are always assigned 1.5 GB memory. The median coldstart latency was 3,640 ms in Azure. Anecdotes online [3] suggest that the long latency is caused by design and engineering issues in the platform that Azure is both aware of and working to improve.

Latency variance. We collected the coldstart latencies of 128 MB, Python 2.7 (AWS) or Nodejs 6.* (Google and Azure) based functions every 10 seconds for over

Provider-Memory	Median	Min	Max	STD
AWS-128	265.21	189.87	7048.42	354.43
AWS-1536	250.07	187.97	5368.31	273.63
Google-128	493.04	268.5	2803.8	345.8
Google-2048	110.77	52.66	1407.76	124.3
Azure	3640.02	431.58	45772.06	5110.12

Table 7: Coldstart latencies (in ms) in AWS, Google, and Azure using Nodejs 6.* based functions for comparison.

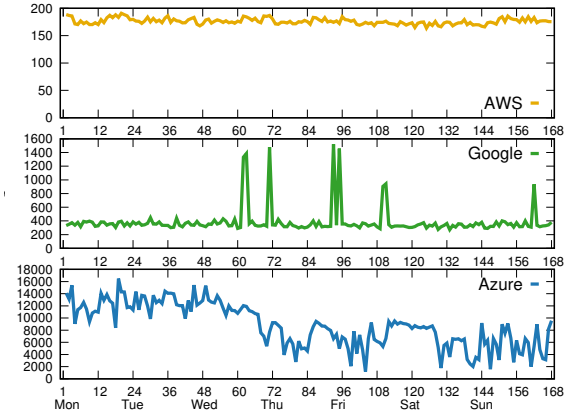


Figure 8: Coldstart latency (in ms) over 168 hours. All the measurements were started at right after midnight on a Sunday. Each data point is the median of all coldstart latencies collected in a given hour. For clarity, the y-axes use different ranges for each service.

168 hours (7 days), and calculated the median of the coldstart latencies collected in a given hour. The changes of coldstart latency are shown in Figure 8. The coldstart latencies in AWS were relatively stable, as were those in Google (except for a few spikes). Azure had the highest network variation over time, ranging from about 1.5 seconds up to 16 seconds.

We repeated our coldstart measurements in May 2018. We did not find significant changes in coldstart latency in AWS. But, the coldstart latencies became 4x slower on average in Google, probably due to its infrastructure update in February 2018 [15], and 15x better in Azure. This result demonstrates the importance of developing a measurement platform for serverless systems (similar to [39] for IaaS) to do continuous measurements for better performance characterization.

5.3 Instance lifetime

A serverless provider may terminate a function instance even if still in active use. We define the longest time a function instance stays active as *instance lifetime*. Tenants prefer long lifetimes because their applications will be able to maintain in-memory state (e.g., database connections) longer and suffer less from coldstarts.

To estimate instance lifetime, we set up functions of different memory sizes and languages, and invoked

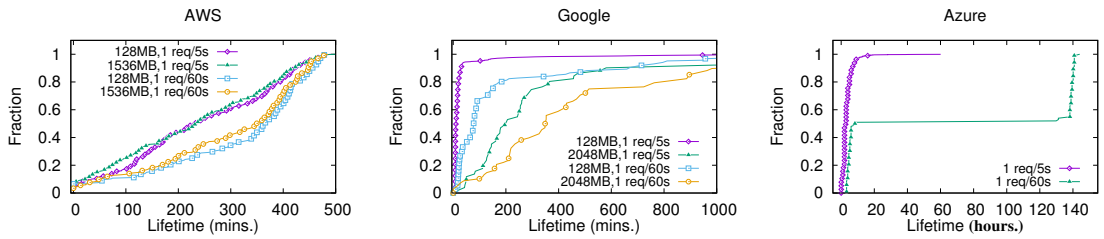


Figure 9: The CDFs of instance lifetime in AWS, Google, and Azure under different memory and request frequency.

them at different frequencies (one request per 5/30/60 seconds). The lifetime of a function instance is the difference between the first time and the last time we saw the instance. We ran the experiment for 7 days (AWS and Google) or longer (Azure) so that we could collect at least 50 lifetimes under a given setting.

In general, Azure function instances have significantly longer lifetimes than AWS and Google as shown in Figure 9. In AWS, the median instance lifetime across all settings was 6.2 hours, with the maximum being 8.3 hours. The host VMs in AWS usually lives longer: the longest observed VM kernel uptime was 9.2 hours. When request frequency increases instance lifetime tends to become shorter. Other factors have little effect on lifetime except in Google, where instances of larger memory tend to have longer lifetimes. For example, when being invoked every five seconds, the lifetimes were 3–31 minutes and 19–580 minutes for 90% of the instances of 128 MB and 2,048 MB memory in Google, respectively. So, for functions with small memory under a heavy workload, Google seems to launch new instances aggressively rather than reusing existing instances. This can increase the performance penalty from coldstarts

5.4 Idle instance recycling

To efficiently use resources, Serverless providers shutdown idle instances to recycle allocated resources (see, e.g., [32]). We define the longest time an instance can stay idle before getting shut down as *instance maximum idle time*. There is a trade-off between long and short idle time, as maintaining more idle instances is a waste of VM memory resources, while fewer ready-to-serve instances cause more coldstarts.

We performed a binary search on the minimum delay t_{idle} between two invocations of the function that resulted in distinct function instances. We created a function, invoked it twice with some delay between 1 and 120 minutes, and determined whether the two requests used the same function instance. We repeated until we identified t_{idle}. We confirmed t_{idle} (to minute granularity) by repeating the measurement 100 times for delays close to t_{idle}.

AWS. An instance could usually stay inactive for at most 27 minutes. In fact, in 80% of the rounds instances were shut down after 26 minutes. When their host VM is "idle", i.e., no active instances on that VM, idle function instances will be recycled the following way: Assuming that the function instances of N functions $f_1,\ldots,f_N$ are coresident on a VM, and k_{f_i} instances are from f_i. For a given function f_i, AWS will shut down $\lfloor k_{f_i}/2 \rfloor$ of the idle instances of f_i every 300 (more or less) seconds until there are two or three instances left, and eventually shut down the remaining instances after 27 minutes (we have tested with $k_{f_i} = 5, 10, 15, 20$). AWS performs these operations to $f_1,\ldots,f_N$ on a given VM independently, and also on individual VMs independently. Function memory or language does not affect maximum idle time.

If there are active instances on the VM, instances can stay inactive for a longer time. We kept one instance active on a given VM by sending a request every 10 seconds and found: (1) AWS still adopted the same strategy to recycle the idle instances of the same function, but (2) somehow idle time was reset for other coresident instances. We observed some idle instances could stay idle in such cases for 1–3 hours.

Azure and Google. In Azure, we could not find a consistent maximum instance idle time. We repeated the experiment several times on different days and found the maximum idle times of 22, 40, and more than 120 minutes. In Google, the idle time of instances could be more than 120 minutes. After 120 minutes, instances remained active in 18% of our experiments.

5.5 Inconsistent function usage

Tenants expect the requests following a function update should be handled by the new function code, especially if the update is security-critical. However, we found in AWS there was a small chance that requests could be handled by an old version of the function. We call such cases *inconsistent function usage*. In the experiment, we sent $k = 1$ or $k = 50$ concurrent requests to a function, and did this again without delay after updating one of the following aspects of the function: IAM role, memory size, environment variables, or function code. For a given setting, we performed these operations for 100 rounds. When $k = 1$, 1%–4% of the tests used an inconsistent function. When there were more associated instances before the update ($k = 50$), 80% of our

rounds had at least one inconsistent function. Looking across all tests from all rounds, we found that 3.8% of instances ran an inconsistent function. Examining the cases, we found two situations: (1) AWS launched new instances of the outdated function (2% of all the cases), and (2) AWS reused existing instances of the outdated function. Inconsistent instances never handle more than one request before terminating (note that max execution time is 300 s in AWS), but still, a considerable faction of requests may fail to get desired results.

As we waited for a longer time after the function update to send requests, we found fewer inconsistent cases, and eventually zero cases with a 6-second waiting time. So, we suspect that the inconsistency issues are caused by race conditions in the instance scheduler. The results suggest coordinating function update among multiple function instances is challenging as the scheduler cannot do an atomic update.

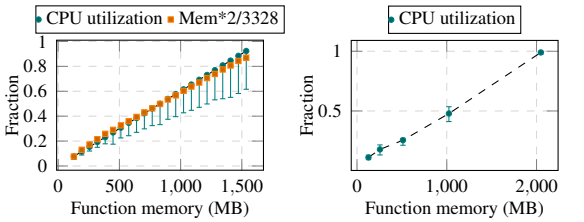
5.6 Discussion

We believe our results motivate further study on designing more efficient instance scheduling algorithms and robust schedulers to further improve VM resource utilization, i.e., to maximize VM memory usage, reduce scheduling latency, and promptly propagate function updates while guaranteeing consistency.

Loading modules or libraries could introduce high latency during coldstart [1, 3]. To reduce coldstart latency, providers might need to adopt more sophisticated library loading mechanisms, for example, using library caching to speed up this process, and resolving the library dependence before deployment and only loading required libraries.

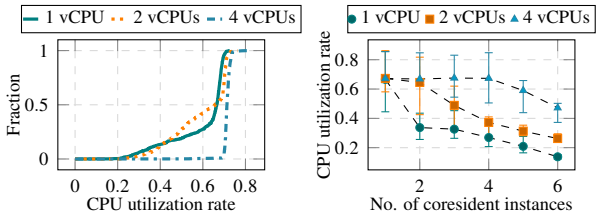
Cross-tenant VM sharing in Azure plus the ability to run arbitrary binaries in the function instance could make applications vulnerable to many kinds of side-channel attacks [16, 17, 20, 45]. We did not examine how well Azure can tackle the potential threats resulting from cross-tenant VM sharing, and leave the actual security vulnerable as an open question.

AWS's bin-packing placement may bring security issues to an application, depending on its design. When a multi-tenant application in Lambda uses IAM roles to isolate its tenants, function instances from different application tenants still share the same VMs. We found two real services that use this pattern: Backand [8] and Zapier [44]. Both allow their tenants to deploy functions in Lambda in some way. We successfully achieved cross-account function coresidency in Backand in just a few tries, while failing in Zapier due to its rate limits and large user base (1 M+). Nevertheless, we could still observe the changes of `procfs` caused by other Zapier tenants' applications, which may admit side-channels [9, 21, 46]. For these multi-tenant applications



(a) AWS (b) Google

Figure 10: The median instance CPU utilization rates with min-max error bars in AWS and Google as function memory increases, averaged across 1,000 instances for a given memory size.



(a) Azure: CPU utilization CDF (b) Azure: CPU vs. coresideny

Figure 11: (a) CDFs of CPU utilization rates of instances (1,000 for each type) and (b) the median CPU utilization rates across a given number of coresident instances (50 rounds) in Azure, with min-max error bars.

to isolate their tenants and achieve better security and privacy, AWS may need to provide a finer-grained VM isolation mechanism, i.e., allocating a set of VMs to each IAM role instead of to each account.

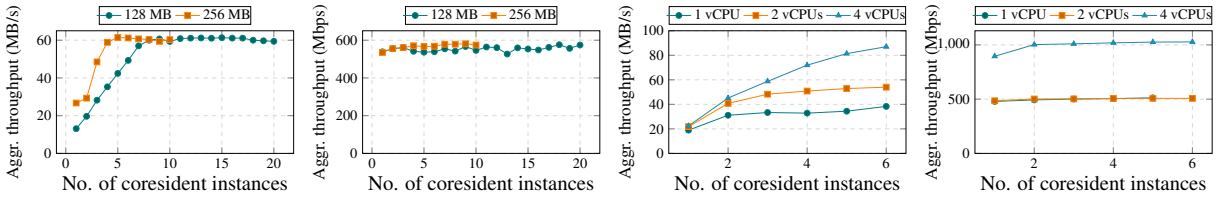
6 Performance Isolation

In this section, we investigate performance isolation. We mainly focus on AWS and Azure, where our ability to achieve coresidency allows more refined measurements. We also present some basic performance statistics for instances in Google that surface seeming contention with other tenants.

6.1 CPU utilization

To measure CPU utilization, our measurement function continuously records timestamps using `time.time()` (Python) or `Date.now()` (Nodejs) for 1,000 ms. The metric *instance CPU utilization rate* is defined as the fraction of the 1,000 ms for which a timestamp was recorded.

AWS. According to AWS, a function instance's CPU power is proportional its pre-configured memory [26]. However, AWS does not give details of how exactly CPU time is allocated to instances. We measured the CPU utilization rates on 1,000 distinct function instances and show the median rate for a given memory size in



(a) AWS: I/O (b) AWS: Network (c) Azure: I/O (d) Azure: Network

Figure 12: Aggregate I/O and network throughput across coresident instances as concurrency level increases. The coresident instances perform the same task simultaneously. The values are the median values across 50 rounds.

Figure 10a. Instances with higher memory get more CPU cycles. The median instance CPU utilization rate increased from 7.7% to 92.3% as memory increased from 128 to 1,536 MB, and the corresponding standard deviations (SD) were 0.7% and 8.7%. When there is no contention from other coresident instances, the CPU utilization rate of an instance can vary significantly, resulting in inconsistent application performance. That said, an upper bound on CPU share is approximated by $2 * m/3328$, where m is the memory size.

We further examine how CPU time is allocated among coresident instances. We let colevel be the number of coresident instances and a colevel of 1 indicates only a single instance on the VM. For memory size m, we selected a colevel in the range 2 to $\lfloor 3328/m \rfloor$. We then measured the CPU utilization rate in each of the coresident instances. Examining the results over 20 rounds of tests, we found that the currently running instances share CPU fairly, since they had nearly the same CPU utilization rate (SD $<$0.5%). With more coresident instances, each instance's CPU share becomes slightly less than, but still close to $2 * m/3328$ (SD $<$2.5% in any setting).

The above results indicate that AWS tries to allocate a fixed amount of CPU cycles to an instance based only on function memory.

Azure and Google. Google adopts the same mechanism as AWS to allocate CPU cycles based on function memory [13]. In Google, the median instance CPU utilization rates ranged from 11.1% to 100% as function memory increased. For a given memory size, the standard deviations of the rates across different instances are very low (Figure 10b), ranging from 0.62% to 2.30%.

Azure has a relatively high variance in the CPU utilization rates (14.1%–90%), while the median was 66.9% and the SD was 16%. This is true even though the instances are allocated the same amount of memory. The breakdown by vCPU number shows that the instances on 4-vCPU VMs tend to gain higher CPU shares, ranging from 47% to 90% (Figure 11a). The distributions of utilization rates of instances on 1-vCPU VMs and 2-vCPU VMs are in fact similar; however, when colevel increased, the CPU utilization of instances on 1-vCPU VMs drops more dramatically, as shown in Figure 11b.

6.2 I/O and network

To measure I/O throughput, our measurement functions in AWS and Google used the `dd` command to write 512 KB of data to the local disk 1,000 times (with `fdatasync` and `dsync` flags to ensure the data is written to disk). In Azure, we performed the same operations using a Python script (which used `os.fsync` to ensure data is written to disk). For network throughput measurement, the function used `iperf 3.13` with default configurations to run the throughput test for 10 seconds with different same-region iperf servers, so that iperf server-side bandwidth was not a bottleneck. The iperf servers used the same types of VMs as the vantage points.

AWS. Figure 12 shows aggregate I/O and network throughput across a given number of coresident instances, averaged across 50 rounds. All the coresident instances performed the same measurement concurrently. Though the aggregate I/O and network throughput remains relatively stable, each instance gets a smaller share of the I/O and network resources as colevel increases. When colevel increased from 1 to 20, the average I/O throughput per 128 MB instance dropped by 4x, from 13.1 Mbps to 2.9 Mbps, and network throughput by 19x, from 538.6 MB/s to 28.7 MB/s.

Coresident instances get less share of the network with more contention. We calculate the Coefficient of Variation (CV), which is defined as SD divided by the mean, for each colevel. A higher CV suggests the performance of instances differ more. For 128 MB instances, the CV of network throughput ranged from 9% to 83% across all colevels, suggesting significant performance variability due to contention with coresident instances. In contrast, the I/O performance was similar between instances (CV of 1% to 6% across all colevels). However, the I/O performance is affected by function memory (CPU) for small memory sizes ($\leq$ 512 MB), and therefore the I/O throughput of an instance could degrade more when competing with instances of higher memory.

Azure. In Azure, the I/O and network throughput of an instance also drops as colevel increases, and fluctuates due to contention from other coresident instances. Even more interestingly, resource allocation is differentiated based on what type of VM a function instance happens to be scheduled on. As shown in Figure 12, the 4-vCPU VMs could get 1.5x higher I/O and 2x higher network throughput than the other types of VMs. The 2-vCPU VMs have higher I/O throughput than 1-vCPU VMs, but similar network throughput.

Google. In Google, both the measured I/O and network throughput increase as function memory increases: the median I/O throughput ranged from 1.3 MB/s to 9.5 MB/s, and the median network throughput ranged from 24.5 Mbps to 172 Mbps. The network throughput measured from different instances with the same memory size can vary substantially. For instance, the network throughput measured in the 2,048 MB function instances fluctuated between 0.2 Mbps and 321.4 Mbps. We found two cases: (1) all instances throughputs' fluctuated during a given period of time, irrespective of memory sizes, or (2) a single instance temporarily suffered from degraded throughput. Case (1) may be due to changes in network conditions, while case (2) leads us to suspect that GCF tenants actually share hosts and suffer from resource contention.

6.3 Discussion

AWS and Azure fail to provide proper performance isolation between coresident instances, and so contention can cause considerable performance degradation. In AWS, the fact that they bin-pack function instances from the same account onto VMs means that scaling up a function places the same function on the same VM, resulting in resource contention and prolonged execution time (not to mention a longer coldstart latency). Azure has similar issues, with the additional issue that contention within VMs arises between accounts. The latter also opens up the possibility for cross-tenant degradation of service attacks.

We leave developing new, efficient isolation mechanisms that take the special characteristics of serverless (e.g., frequent instance creation, short-lived instances, and small memory-footprint functions) as considerations for future work.

7 Resource accounting

In the course of our study, we found several resource accounting issues that can be abused by tenants.

Background processes. We found in Google one could execute an external script in the background that continued to run even *after* the function invocation concluded. The script we ran posted a 10 M file every 10 seconds to a server under our control, and the longest time it stayed alive was 21 hours. We could not find any logs of the network activity performed by the background process and were not charged for its resource consumption.[6][7] In contrast, one could run such background script in Azure but Azure logged all the activity. Our observations suggest that: (1) In Azure and Google the function instance execution context will not be frozen after an invocation, as opposed to AWS; and (2) Google does resource accounting via monitoring the Node.js process rather than the entire function instance.

One can exploit the billing issue in Google to run sophisticated tasks at negligible cost. For a function instance with 2 GB memory and 2.4 GHz CPU, one only needs to pay for a few invocations ($0.0000029/100 ms, with 2 M free calls) to get the same computing resources as using a g1-small instance ($0.0257/hour) on Google Cloud Platform.

CPU accounting. In Google, we found there was an 80% chance that a just-launched function instance (of any memory size other than 2,048 MB) could temporally gain more CPU time than expected. Measuring the CPU utilization rates and the completion times of a CPU-intensive task, we confirmed that the instances that one expects to have 8%–58% of the CPU time (see §6) had near 100% of the CPU time, the same as that given to 2,048 MB instances. The instance can retain the CPU resources until the next invocation. Note that if one wants to conduct performance measurements in Google, this issue could introduce a lot of noise (we appropriately controlled for it in previously reported experiments).

8 Conclusion

In this paper, we provided insights into architectures, resource utilization, and the performance isolation efficiency of three modern serverless computing platforms. We discovered a number of issues, raised from either specific design decisions or engineering, with regard to security, performance, and resource accounting in the platforms. Our results surface opportunities for research on improving resource utilization and isolation in future serverless platform designs.

Acknowledgements

The authors thank engineers from Microsoft, Amazon, and Google for their feedback and helpful discussions. This work was supported in part by NSF grants 1558500, 1330308, and 1718084.

[6] Google has a free tier of service, but even after that is used up the background process consumption went unbilled.

[7] We have reported this issue to Google and Google has been working on fixing it as of May 2018.

References

[1] SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association.

[2] Azure app service, virtual machines, service fabric, and cloud services comparison. https://docs.microsoft.com/en-us/azure/app-service/choose-web-site-cloud-service-vm, 2017.

[3] Cold start taking a long time in consumption mode for C# Azure Function. https://github.com/Azure/azure-functions-host/issues/838, 2017.

[4] Consumption plan scaling issues. https://github.com/Azure/azure-webjobs-sdk-script/issues/1206, 2017.

[5] Create your first function in the Azure portal. https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-azure-function, 2017.

[6] Azure runtime environment. https://github.com/projectkudu/kudu/wiki/Azure-runtime-environment, 2017.

[7] Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale, 2017.

[8] Backand. https://www.backand.com/, 2018.

[9] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *USENIX Security Symposium* (2014), pp. 1037–1052.

[10] How does language, memory and package size affect cold starts of AWS Lambda? https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76, 2017.

[11] Understanding AWS Lambda performance. https://blog.newrelic.com/2017/01/11/aws-lambda-cold-start-optimization/, 2017.

[12] Understanding AWS Lambda coldstarts. https://www.iopipe.com/2016/09/understanding-aws-lambda-coldstarts/, 2016.

[13] Google Cloud Functions quotas. https://cloud.google.com/functions/quotas, 2017.

[14] GLIKSON, A., NASTIC, S., AND DUSTDAR, S. Deviceless edge computing: extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference* (2017), ACM, p. 28.

[15] Google cloud functions release notes. https://cloud.google.com/functions/docs/release-notes, 2018.

[16] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 279–299.

[17] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium* (2015), pp. 897–912.

[18] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (2016), USENIX Association, pp. 33–39.

[19] How long does AWS Lambda keep your idle functions around before a cold start? https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810, 2017.

[20] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: a shared cache attack that works across cores and defies vm sandboxing and its application to AES. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 591–604.

[21] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 143–157.

[22] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 445–451.

[23] KRUG, A. Hacking serverless runtimes profiling Lambda, Azure, and more., 2017.

[24] Lambda CPU relative to which instance type? https://forums.aws.amazon.com/message.jspa?messageID=614558, 2014.

[25] AWS Lambda in production. https://blog.newrelic.com/2017/11/21/aws-lambda-state-of-serverless/, 2017.

[26] Configuring Lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html, 2017.

[27] How does proportional CPU allocation work with AWS Lambda? https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac, 2018.

[28] The occasional chaos of AWS Lambda runtime performance. https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e, 2017.

[29] My accidental 35x speed increase of AWS Lambda functions. https://serverless.zone/my-accidental-3-5x-speed-increase-of-aws-lambda-functions-6d95351197f3, 2017.

[30] Comparing AWS Lambda performance when using Node.js, Java, C# or Python. https://read.acloud.guru/comparing-aws-lambda-performance-when-using-node-js-java-c-or-python-281bef2c740f, 2017.

[31] AWS Lambda performance issues. https://stackoverflow.com/questions/43089879/aws-lambda-performance-issues, 2017.

[32] Understanding container reuse in AWS lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2014.

[33] LLOYD, W., RAMESH, S., CHINTHALAPATI, S., LY, L., AND PALLICKARA, S. Serverless computing: An investigation of factors influencing microservice performance.

[34] MCGRATH, G., AND BRENNER, P. R. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 405–410.

[35] PETERSON, E. Serverless security and things that go bump in the night. https://www.infoq.com/presentations/serverless-security, 2017.

[36] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 199–212.

[37] Security and serverless. https://read.acloud.guru/security-and-serverless-ec52817385c4, 2017.

[38] VARADARAJAN, V., ZHANG, Y., RISTENPART, T., AND SWIFT, M. M. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security Symposium* (2015), pp. 913–928.

[39] WANG, L., NAPPA, A., CABALLERO, J., RISTENPART, T., AND AKELLA, A. Whowas: A platform for measuring web deployments on iaas clouds. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 101–114.

[40] WILLAERT, F. AWS Lambda container lifetime and config refresh. https://www.linkedin.com/pulse/aws-lambda-container-lifetime-config-refresh-frederik-willaert, 2016.

[41] XU, Z., WANG, H., AND WU, Z. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium* (2015), pp. 929–944.

[42] YAN, M., CASTRO, P., CHENG, P., AND ISHAKIAN, V. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs* (2016), ACM, p. 5.

[43] YAROM, Y., AND FALKNER, K. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium* (2014), pp. 719–732.

[44] Backand. https://zapier.com/, 2018.

[45] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 990–1003.

[46] ZHOU, X., DEMETRIOU, S., HE, D., NAVEED, M., PAN, X., WANG, X., GUNTER, C. A., AND NAHRSTEDT, K. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1017–1028.