Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS

Zirui Neil Zhao
University of Illinois Urbana-Champaign, USA
ziruiz6@illinois.edu

Adam Morrison
Tel Aviv University, Israel
mad@cs.tau.ac.il

Christopher W. Fletcher
University of Illinois Urbana-Champaign, USA
cwfletch@illinois.edu

Josep Torrellas
University of Illinois Urbana-Champaign, USA
torrella@illinois.edu

Abstract

Microarchitectural side-channel attacks exploit shared hardware resources, posing significant threats to modern systems. A pivotal step in these attacks is achieving physical host co-location between attacker and victim. This step is especially challenging in public cloud environments due to the widespread adoption of the virtual private cloud (VPC) and the ever-growing size of the data centers. Furthermore, the shift towards *Function-as-a-Service (FaaS)* environments, characterized by dynamic function instance placements and limited control for attackers, compounds this challenge.

In this paper, we present the first comprehensive study on risks of and techniques for co-location attacks in public cloud FaaS environments. We develop two physical host fingerprinting techniques and propose a new, inexpensive methodology for large-scale instance co-location verification. Using these techniques, we analyze how Google Cloud Run places function instances on physical hosts and identify exploitable placement behaviors. Leveraging our findings, we devise an effective strategy for instance launching that achieves 100% probability of co-locating the attacker with at least one victim instance. Moreover, the attacker co-locates with 61%–100% of victim instances in three major Cloud Run data centers.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Security and privacy** → *Side-channel analysis and countermeasures.*

Keywords: Cloud computing, Function-as-a-service (FaaS), Co-location vulnerability, Timestamp counter

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04…$15.00
https://doi.org/10.1145/3617232.3624867

ACM Reference Format:
Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3617232.3624867

1 Introduction

Microarchitectural side-channel attacks [2, 8, 26, 33, 35, 49–51, 64, 65] pose serious threats to modern computer systems, where hardware resources are often shared among mutually distrusting users. In these attacks, an attacker monitors a victim's secret-dependent usage of the shared hardware resources to exfiltrate sensitive victim information, such as cryptographic keys [33, 50, 65]. The first step of these attacks is to ensure that the attacker's processes are *co-located* with the target victim's process on the same physical host [54].

In modern public cloud environments, attaining co-location is challenging for several reasons. First, due to the widespread adoption of the virtual private cloud (VPC) [18], modern cloud infrastructures have become resistant to prior network-based co-location attack techniques [54, 59, 63]. Second, with the rapid expansion of cloud computing and the ever-growing sizes of data centers, the likelihood of attacker-victim co-location has been reduced.

Adding to these challenges, cloud computing is gradually shifting towards the emerging paradigm of *Function-as-a-Service (FaaS)*, exemplified by platforms like AWS Lambda [5], Google Cloud Run [13], and Azure Functions [7]. Co-location attack techniques in these FaaS environments are relatively unexplored and present new challenges for attackers.

In the FaaS paradigm, an application is disaggregated into multiple small standalone components called *functions*. Each function is packed with its dependencies into a lightweight FaaS container [1, 36, 40, 53]. Unlike in conventional virtual machine (VM) environments, where users can specify the placement of VMs in availability zones and choose hosts with certain CPU models, FaaS platforms abstract away these operational details and fully manage the FaaS container placement. Furthermore, due to the dynamic nature of FaaS

environments, container instances are frequently launched and terminated to accommodate the dynamic workload demands. Such characteristics make achieving co-location in FaaS settings particularly difficult.

In this paper, we present the first comprehensive study on risks of and techniques for co-location attacks in modern public FaaS environments. Since public FaaS platforms do not disclose their instance placement policies, reverse engineering these policies is crucial to understand the co-location risk and develop efficient co-location attacks.

To study how container instances are scheduled to physical hosts, we first develop two novel *host fingerprinting* techniques. We show that, despite the use of sandboxing and virtualization technologies [1, 36, 40, 53] in the cloud, the attacker can still learn sensitive host information by directly interacting with the host hardware—specifically, the timestamp counter. Our techniques are applicable to both non-virtualized Linux containers (e.g., Docker [40]) and lightweight VMs (e.g., Firecracker [1]), which are the two mainstream containerization technologies used in FaaS platforms. Armed with host fingerprints, we propose a new method to inexpensively *verify instance co-location* on a large scale. This is essential in vast modern data centers, where the attacker needs to launch numerous instances to achieve co-location.

Using the host fingerprints and our scalable co-location verification methodology, we perform a large-scale study on Google Cloud Run [13] to analyze its instance placement strategy. Our investigation uncovers exploitable instance placement behaviors in Cloud Run. Notably, Cloud Run appears to employ a load balancing mechanism that distributes instances of a function to numerous hosts when the function experiences a high demand within a short time window. We then develop an instance launching strategy that exploits this behavior to deploy attacker instances onto a significant portion of Cloud Run hosts within a data center—drastically increasing co-location efficacy and reducing the financial cost of the attack.

We demonstrate the ability of our attack strategy to achieve 100% probability of co-locating the attacker with at least one victim instance in three major Cloud Run data centers in the US: *us-east1*, *us-central1*, and *us-west1*. Moreover, our strategy effectively co-locates the attacker with 100% of victim instances in *us-west1*, nearly 100% in *us-east1*, and between 61% and 90% in *us-central1*, depending on the victim account. In addition, we observe at least 1702 hosts in the largest data center, and show that our strategy successfully deploys attacker instances that reside on 904 hosts *at once*, with an estimated expense of only 23 USD—showcasing the practicality of co-location attacks in large modern data centers.

This paper makes the following contributions:

- We introduce two effective host fingerprinting techniques as a primitive to study the instance placement policies of modern public FaaS platforms.
- We propose a scalable and inexpensive methodology for instance co-location verification assisted by host fingerprints.
- We systematically study the instance placement policies of Google Cloud Run and identify behaviors exploitable for co-location attacks.
- We devise an efficient attack strategy that achieves high co-location rates with different victim accounts on Google Cloud Run.

Disclosure to Google. We reported our findings to Google in early August 2023. Google identified our findings as an abuse risk and assigned the issue to their Trust & Safety team.

Availability. We open sourced our implementations at https://github.com/zzrcxb/EAAO.

2 Background

2.1 Microarchitectural Side Channels

Microarchitectural side channels exploit shared hardware resources to bypass access control policies and exfiltrate sensitive information. Examples of commonly-exploited shared hardware resources include caches [35, 50, 65], TLBs [33, 57], coherence directories [64], and functional units [2, 8]. These attacks are particularly dangerous in cloud environments, where hardware resources are extensively shared between mutually distrusting parties.

In general, conducting a microarchitectural side-channel attack in the cloud involves two steps [54]. The first step is to *co-locate* the attacker with the victim on the same physical host, as these side channels rely on sharing hardware resources with the victim. The second step is *extraction*, where the attacker monitors the victim's execution through a side channel and exfiltrates sensitive information. While numerous studies explore different side channels for extraction, there has been limited research on achieving the first step in contemporary cloud environments. Our work investigates this first step, co-location.

2.2 Function-as-a-Service

Function-as-a-Service (FaaS) [5, 7, 13] is an emerging cloud computing paradigm that disaggregates a large monolithic application into many small standalone components called *functions*. Each function typically has a single, independent functionality, separate from the others. To perform the task of the original monolithic application, these functions collaborate and communicate with each other over the network. Consequently, functions are usually implemented as web services that can be invoked through various means, such as HTTP requests, WebSockets, or remote procedure calls [16]. In this paper, we use *function* and *service* interchangeably.

Function deployment and management. To simplify function deployment and management, each function is packed with its dependencies into a lightweight, self-contained

container image. The containerization process ensures a consistent execution of the function across various environments.

The FaaS platform orchestrator fully manages function instances at a container-level granularity. When a function is invoked by a user or another function, the orchestrator launches a new container instance of the requested function to process the incoming request. After serving the request, the instance enters an idle state, releasing its CPU, and awaiting further incoming requests. Idle instances are typically charged minimally or not at all. If an instance remains idle for an extended period of time (e.g., 15 minutes), it is terminated and destroyed [15].

Autoscaling. A FaaS platform can dynamically adjust the number of instances of a function based on its demand, a feature known as *autoscaling* [10]. When there is a surge in requests for a function that exceeds its current capacity, the orchestrator *scales out*, deploying additional instances to accommodate the increased demand. Conversely, when the demand for a function declines, the orchestrator *scales in* by terminating excess instances, thus freeing up resources for instances of other functions.

Instance placement. When selecting a host to place a new container instance, a typical FaaS orchestrator first identifies all the hosts that meet specific constraints. Then, among such hosts, it selects the one with the highest score, relying on criteria such as resource utilization and load balancing [20]. The orchestrator can tweak the instance placement algorithm by incorporating additional policies, such as *affinity* and *anti-affinity* rules. Affinity rules aim to place instances from functions that frequently interact with each other on the same host to reduce communication overhead. In contrast, anti-affinity rules attempt to distribute instances of the same function across different hosts for fault tolerance.

2.3 The Cloud Run Platform

In this paper, we primarily focus on the Cloud Run platform [13] from Google Cloud as our target. Cloud Run is a fully-managed serverless computing platform designed for containers, and it powers Google Cloud's FaaS platform. Users of Cloud Run can deploy services using either predefined container templates or custom-built container images. As user containers can run arbitrary programs on Cloud Run, the platform offers two types of sandboxed execution environments to ensure software security.

First generation environment (Gen 1) [11]. Cloud Run uses gVisor [36] to sandbox Linux containers in its first-generation environment *without* host hardware virtualization. Figure 1 shows an overview of gVisor. At a high level, gVisor runs as a userspace kernel that intercepts and emulates normal system calls. This design prevents the untrusted application from directly interacting with the host kernel, reducing the attack surface. Consequently, the user application cannot access sensitive host information. For example, gVisor conceals the host CPU model name and cache sizes by emulating `/proc/cpuinfo`. Additionally, gVisor also virtualizes the host's runtime states, such as its IP address and uptime.

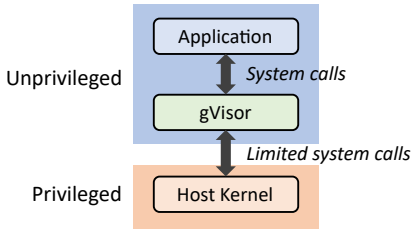


Figure 1. Overview of gVisor container sandbox [36].

Second generation environment (Gen 2) [11]. Cloud Run uses lightweight virtual machines (VMs) to sandbox user programs in its second-generation environment, which was introduced in December 2022 [14]. In Gen 2, the untrusted user program runs inside a guest VM on the virtualized host hardware. Using hardware virtualization, the hypervisor can trap and emulate certain x86 instructions like `cpuid`, thus creating an illusion of the hardware on which the user runs. As a result, the user has no access to sensitive host information.

Comparison between Gen 1 and Gen 2. Both execution environments have their pros and cons, which means that they are complementary rather than substitutional. Since Gen 1 uses Linux containers, it has a small resource footprint and features fast start-up time [11]. This feature is crucial for user-facing web applications that are latency-critical [29, 42, 46], such as web search [46], online collaborative document editing [32], and key-value stores [22, 44, 46]. Increases in latency can negatively impact advertisement revenue [56]. Yet, a limitation of Gen 1 lies in its potential compatibility issues stemming from system call emulation.

Conversely, Gen 2 provides full Linux compatibility, and in a steady state, it performs better than Gen 1. However, its large resource footprint results in longer start-up times [11]. At the time of this writing, Cloud Run uses Gen 1 for services by default [11]. Moreover, Gen 1 is used in other Google Cloud products like Cloud Function [12] (Google Cloud's equivalent of AWS Lambda [5]). Therefore, in this paper, we primarily focus our exploration on Gen 1 and demonstrate the transferability of our results to Gen 2.

2.4 Timekeeping in x86

In recent years, the timestamp counter (TSC) has become a preferable timekeeping option on x86 platforms, as CPU vendors increasingly support *invariant TSC*. An invariant TSC is reset to zero at boot time and increments at a *fixed* rate, irrespective of the CPU's frequency scaling and power state [24].

Compared to other clock sources, TSC offers greater time resolution and lower time retrieval cost. TSC can be conveniently accessed using the unprivileged instructions `rdtsc` and `rdtscp`.

To use the TSC as a clock source in Linux, the kernel needs to determine its frequency. Since the actual TSC frequency usually deviates from the frequency reported by `cpuid` by a constant value (which can be up to a few MHz), the kernel refines the TSC frequency using other hardware clocks in the system and utilizes the refined frequency for more accurate timekeeping [21]. On multi-core systems, Linux also verifies TSC synchronization across cores to prevent time anomalies. Generally, TSC is synchronized among cores across sockets on Intel platforms.

3 Threat Model

Recall that a generalized microarchitectural side-channel attack consists of two steps: co-location and extraction (Section 2.1). In this paper, we consider an attacker aiming to co-locate with instances of a target victim service on a public FaaS platform (step 1). We assume that the victim service processes sensitive information, such as a login service that performs authentication. Since the victim service is usually part of a large web application with public interfaces, we assume that the attacker can either directly or indirectly invoke the victim service through those interfaces. Finally, we assume that, once co-located with the victim, the attacker can detect when the victim program is running and exfiltrate the said sensitive information through techniques discussed in prior work [25, 41, 54, 59, 68].

We assume an unprivileged attacker who is a standard user of a public FaaS platform (e.g., Cloud Run). We also assume that the FaaS platform is trusted and does not collude with the attacker. These two assumptions imply that the attacker can only interact with the platform through standard FaaS interfaces that are available to all platform users, such as deploying custom services and sending requests to services. Using these interfaces, the attacker can execute arbitrary programs on the platform *inside their containers*, and the attacker can launch new container instances through autoscaling (Section 2.2). Additionally, we assume that the attacker has no knowledge of the *exact* host selection policies employed by the platform orchestrator and can only observe their behavior using black-box methods.

4 Host Fingerprinting in the Wild

Public FaaS platforms, such as Cloud Run [13], do not reveal their instance placement policies. In this section, we propose novel, highly accurate physical host fingerprinting techniques suitable for both non-virtualized Linux containers (e.g., the Gen 1 environment) and lightweight VMs (e.g., the Gen 2 environment). Using our fingerprints, attackers can gain insights into the placement policy of the cloud platform, allowing them to develop launching strategies that drastically boost the efficacy of co-location attacks.

As the focus of our paper is on the Gen 1 environment, we organize this section as follows: Section 4.1 provides an overview of host fingerprinting in Gen 1; Section 4.2 discusses two possible implementations to obtain host fingerprints in Gen 1; Section 4.3 proposes a new methodology to verify instance co-location in a scalable manner; Section 4.4 evaluates our fingerprinting for Gen 1 in the wild; and Section 4.5 extends the Gen 1 fingerprinting technique to the Gen 2 environment and evaluates its accuracy.

4.1 Overview

Recall that, in the Gen 1 environment, gVisor sandboxes user programs and hides the host information (Section 2.3), thereby blocking host fingerprinting through IP addresses or statistics in the `/proc` filesystem [54, 59]. However, we find that we can bypass gVisor's software countermeasures to learn sensitive host information by directly interacting with the non-virtualized host hardware.

For example, the attacker can use the unprivileged instruction `cpuid` to extract information like the CPU model and cache hierarchy structure, which are essential for many cache-based side-channel attacks [45, 50, 51, 61][1]. Similarly, the attacker can use the unprivileged instructions `rdtsc` and `rdtscp` to read the host's timestamp counter (TSC). The TSC is reset to 0 on host boot and increments at a fixed rate non-stop (Section 2.4). Therefore, the attacker can use the value of TSC to infer the uptime of the host, which in turn can be used to determine its boot time.

Based on this insight, we propose to use the host's CPU model (*model*) and the host's boot time in real-world time (T_{boot}) to fingerprint a host in the Gen 1 execution environment. The intuition of using T_{boot} is that different hosts very likely have different boot times due to system maintenance, hardware failures, and power management (e.g., powering off the host when the computation demand is low). As a result, T_{boot} can accurately differentiate physical hosts. Since it is trivial to read the CPU model through `cpuid`, we focus our discussion on deriving the host's T_{boot}.

4.2 Deriving the Boot Time from the TSC Value

To derive the host's boot time T_{boot}, the attacker can read the host's TSC value (denoted by *tsc*) through `rdtsc` or `rdtscp`, and simultaneously record the real-world time of this measurement (denoted by T_w) through a system call. Then, the host's boot time is calculated as follows:

$$T_{boot} = T_w - tsc/f, \tag{4.1}$$

[1]Intel introduced the Processor Serial Number (PSN) in the Pentium III processor [23]. The PSN uniquely identifies an individual processor and can be queried through `cpuid`. However, the PSN is discontinued in recent Intel processors due to privacy concerns.

where f is the TSC frequency measured in Hz. Eq. 4.1 assumes that the host CPU supports an invariant TSC (Section 2.4), which holds for all CPU models we observed in Cloud Run. However, even on the *same* host, the derived T_{boot} can exhibit small variations across measurements due to noise. Consequently, we round T_{boot} to a certain precision p_{boot} (e.g., 1 s). With the rounded value, measurements from the same host consistently produce the same fingerprint.

To obtain f, we propose two methods. Neither method relies on any features from gVisor or Cloud Run, making them applicable to other Linux container-based environments.

① **Using the reported TSC frequency.** In this method, the attacker uses the TSC frequency reported by `cpuid`. If `cpuid` does not report the TSC frequency, which is the case on Cloud Run, the attacker can use the labeled base frequency found in the model name. Empirically, this base frequency is equal to the TSC frequency that the clock is supposed to operate on [24]. For example, CPU model "Intel Xeon CPU @ 2.00GHz" has a base frequency and TSC frequency of 2.00 GHz. We refer to a TSC frequency obtained through either way as the *reported TSC frequency*.

Unfortunately, the reported TSC frequency is often slightly inaccurate, deviating from the actual TSC frequency by a constant value [21] (Section 2.4). This inaccuracy can cause the derived T_{boot} to drift over time, causing fingerprinting false negatives. To understand why, let us denote the reported TSC frequency as $f_r = f^* + \epsilon$, where f^* is the actual frequency and ϵ is the constant error. Suppose we collect two fingerprints from the *same* host at two different real-world times T_{w_1} and T_{w_2}, with TSC values tsc_1 and tsc_2, respectively, as illustrated in Figure 2.

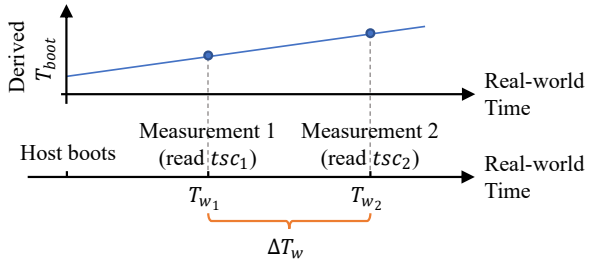


Figure 2. Illustration of drifting in the derived T_{boot} over time.

Using Eq. 4.1, the T_{boot} derived from the two measurements differs by

$$\begin{aligned}\Delta T_{boot} = T_{boot_2} - T_{boot_1} &= (T_{w_2} - tsc_2/f_r) - (T_{w_1} - tsc_1/f_r) \\ &= \Delta T_w - \Delta tsc/f_r \\ &= \Delta T_w - \Delta T_w f^*/f_r \\ &= \Delta T_w \epsilon / f_r. \end{aligned} \tag{4.2}$$

Since both ϵ and f_r are constant, $|\Delta T_{boot}|$ increases linearly as ΔT_w grows, where ΔT_w is the time elapsed between two measurements. If ΔT_w is sufficiently large, $|\Delta T_{boot}|$ will exceed the rounding precision p_{boot}, causing the rounded T_{boot} to differ and leading to a false negative. As a result, we say that the fingerprint exhibits an "*expiration time*" that depends on the frequency error ϵ.

② **Using measured TSC frequency.** An alternative approach is to measure the actual TSC frequency. This approach mitigates the drifting problem. Similar to how Linux refines the TSC frequency at boot time [21], the attacker can read the TSC twice, waiting a real-world time ΔT_w in-between. The TSC frequency can then be calculated as $\Delta tsc/\Delta T_w$. However, unlike the Linux kernel, the attacker cannot access other *hardware* clocks to obtain an accurate ΔT_w in the sandboxed container, as accessing those clocks requires privileged instructions. Consequently, the attacker can only rely on system calls to obtain ΔT_w, which may be subject to noise caused by interrupts and context switches.

We tested this approach on Cloud Run with $\Delta T_w \approx 100$ ms and found that the measured TSC frequency exhibits standard deviations of less than 100 Hz after 10 repetitions on most Cloud Run hosts. However, on a small yet significant portion of the evaluated hosts, we observed large standard deviations ranging from 10 kHz to a few MHz, even after 100 repetitions with an increased ΔT_w. As a result, two co-located instances on such problematic hosts can measure the TSC frequency as two significantly different values—and the T_{boot} derived by the two instances will not match, leading to declaring that the two instances are on two different hosts (i.e., a false negative).

Notably, during the experiment of validating fingerprinting accuracy in Section 4.4.1, we found that 58 out of the 586 evaluated hosts (or about 10%) exhibited such problematic behavior. These affected hosts were largely the same across measurements conducted at different times. Therefore, in the rest of the paper, we obtain f using the first method (i.e., the reported TSC frequency) and, in Section 4.4.2, evaluate the expiration time of fingerprints due to the inaccurate f.

4.3 Verifying Instance Co-location in a Scalable Manner

In Section 4.4.1, we will evaluate the accuracy of our fingerprints by launching multiple instances and measuring whether instances that obtain the same fingerprints are indeed co-located, and vice versa. To do that, in this section, we first develop a new methodology to generate the ground truth of instance co-location in a scalable and inexpensive manner.

To understand our methodology, consider first the conventional approach to test instance co-location using a covert channel. The process involves picking two container instances at a time and instructing both of them to simultaneously put high pressure on a shared resource in the host, such as the random number generator (RNG) [27] or memory bus [62]. If both instances observe resource contention

above a certain threshold, then we conclude that they are co-located. The drawback of this naive pairwise approach is that it has a time complexity of $O(N^2)$, where N is the total number of instances under test.

We propose a new approach to test n instances at once, where $n > 2$. Specifically, consider a covert-channel test primitive for n instances

$$CTest(i_1, i_2, ..., i_n) \rightarrow \{b_1, b_2, ..., b_n\},$$

that takes as input a list of n container instances under test ($\{i_1, i_2, ..., i_n\}$) and instructs all n instances to simultaneously put pressure on the shared resource. This primitive returns a list of boolean values ($\{b_1, b_2, ..., b_n\}$) that indicate whether each corresponding instance observes a contention level above a certain threshold. We assume that it only takes two co-located instances to generate enough contention to go over the contention threshold. In this case, if an instance A does not see contention over the threshold (i.e., it tests negative), then we can conclude that A is not co-located with any of the other $n-1$ instances. Furthermore, if all n instances test negative, then, in a single test, we conclude that no instance is co-located with any other instance.

If an instance A tests positive, we *may* know which instances are co-located with A, depending on the total number of positive instances in the test. To see why, consider as an example a test with four instances $\{A, i_1, i_2, i_3\}$. If three instances, including A, are positive, e.g., $CTest(A, i_1, i_2, i_3) \rightarrow \{T, T, T, F\}$, then we can conclude that $\{A, i_1, i_2\}$ must be co-located, as it takes at least two co-located instances to test positive. However, if all four instances are positive, we *cannot* conclude that they are co-located on the same host; it is possible that these four instances reside on two hosts, e.g., $\{A, i_1\}$ are co-located and $\{i_2, i_3\}$ are co-located. As a result, we can only test $n \leq 3$ instances at once without the confusion of whether they share one or multiple hosts.

We can further improve the test efficiency by (i) either raising the contention level threshold for an instance to test positive, or (ii) reducing the amount of pressure each instance generates. For example, if each instance generates a contention of 1 unit and we set the threshold to m units ($m > 2$), then it takes at least m co-located instances for each one of the m instances to test positive. As a result, if m, $m+1$, ..., or $2m-1$ instances test positive, we verify that these positive instances share the same host in a single test.

Our approach. Based on the above discussion, our approach hierarchically generates the ground truth for fingerprinting validation. The approach is illustrated in Figure 3, where each symbol represents a container instance, and truly co-located instances have the same shape. Assume that we have nine instances and that, using our fingerprinting method, conclude that there are three different fingerprints ($\mathbb{F}_1$, $\mathbb{F}_2$, and $\mathbb{F}_3$), and three instances per fingerprint. To validate our findings, we first group the instances based on their fingerprints (dashed lines in ①). If our fingerprints have a high accuracy, instances in the same group are likely to be indeed co-located, while those in different groups are likely not.

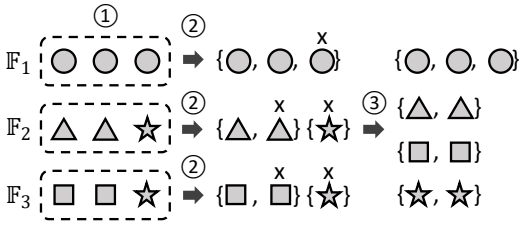


Figure 3. Overview of our fingerprint validation methodology. Each symbol represents a container instance. Instances with the same shape are truly co-located.

Next, we use an appropriate m to test likely co-located instances from each group at once. In the example, we use $m = 2$ (②). After this test, if a group had any false positive (e.g., the group with $\mathbb{F}_2$ and $\mathbb{F}_3$ in Figure 3), it is divided into several clusters, where each cluster includes instances that are verified to be co-located. Otherwise, the entire group remains intact and is considered as a single cluster (e.g., the group with $\mathbb{F}_1$ in Figure 3). Step ② has identified all the false positives. In the best-case scenario where no false positives existed, each fingerprint is verified with one single test. In this case, the total number of tests is the number of fingerprints under validation, which is the number of hosts—except for potential false negatives, as we will consider next.

Note that Step ② tests each fingerprint group in sequence, to avoid interference. We can further reduce the execution time of Step ② by concurrently verifying fingerprints that are *guaranteed* to belong to different hosts, such as those with different CPU models. As will be discussed in Section 4.5, the Gen 2 fingerprint makes crucial use of this optimization.

After this, we want to find the false negatives—i.e., two instances with different fingerprints that are actually co-located. Since instances from the same cluster are verified to be co-located, we pick one instance from each cluster to represent the host they reside on. In Figure 3, we pick the five instances that are decorated with an ×. These selected instances are unlikely co-located. Hence, we set $m = 2$ and test these selected instances all at once (③). Those that test positive are false negatives. In our example, they are the two stars. Then, we further refine our tests on the positive instances to identify co-located instances and merge the clusters they represent. In the example, we end up with four clusters, as the figure shows. In the best-case scenario where no false negatives existed, Step ③ only requires one test.

If an initial group in Step ① is large, we split it into several smaller groups with no more than $2m-1$ instances each, where m is small (in our implementation, we use $m = 2$). Then, we test each group individually. If each small group is verified to be co-located, we pick one instance from each group to hierarchically test their co-location. If some tests

in this process turn out negative, for simplicity, we fall back to pairwise tests within the initial large group.

In summary, our approach's best-case time complexity is $O(M)$, where M is the number of hosts occupied by the instances under validation. This best-case scenario is common if fingerprints are accurate, which we will see is true for our Gen 1 fingerprints in Section 4.4.1.

Comparison with conventional pairwise covert-channel testing [41, 54, 59]. The main goal of our host fingerprinting technique is to improve the attacker's ability to achieve co-location. Remarkably, while conventional pairwise covert-channel testing only confirms co-location at a particular moment, our host fingerprinting allows an attacker to track a host over time. Our technique thus enables the attacker to comprehensively study how container instances are placed onto specific hosts at different times, which can be leveraged to develop efficient instance launching strategies to significantly increase the probability of co-location with a target victim (Section 5). We will see that, if an attacker simply launches instances without insight into the placement policy, the rate of co-location is often zero, whereas our technique attains a co-location rate of 61%–90% in *us-central1* and near 100% in *us-east1* and *us-west1* with minimal cost.

Our co-location verification method is both faster and financially cheaper than conventional pairwise covert-channel testing. In Section 4.4.1, we verify the co-location of 800 container instances. Using pairwise testing, this verification process requires 319,600 pairwise tests. Moreover, these tests are *serialized* to avoid interference. Assuming an optimistic execution time of 100 ms per test, finishing these pairwise tests would take 8.9 hours. In contrast, we find that our approach only takes about 1 to 2 minutes to validate all 800 instances.

To estimate the financial cost of performing these tests on Cloud Run, we use the Cloud Run pricing model [17]. For a standard instance requesting 1 vCPU and 0.5 GB memory, the cost is estimated using the formula $N \times t \times (R_{cpu} + 0.5R_{mem})$. In this equation, N is the number of active instances, t is the active time of these instances in seconds, R_{cpu} is the CPU time cost per vCPU-second in USD, and R_{mem} is the memory cost per GB-second in USD. At the time of this writing, R_{cpu} = ¢0.0024/vCPU-second and R_{mem} = ¢0.00025/GB-second in *us-east1*, *us-central1*, and *us-west1*.

Based on this pricing model and rates, performing the pairwise tests would cost about 645 USD. This cost would be even higher if we used the pairwise test method discussed by Varadarajan et al. [59], which takes several seconds to complete one pairwise test. In contrast, our approach costs about 1 to 3 USD. Importantly, the time and financial cost of pairwise testing grows quadratically with the number of instances being verified for co-location.

Note that prior work [41, 59] speeds up pairwise testing by filtering out instances that do not co-locate with any other instance. İnci et al. [41] call this filtering step Single Instance Elimination (SIE). During SIE, the attacker tests all instances simultaneously and removes instances that test negative. However, SIE is ineffective in a FaaS environment. This is because the FaaS orchestrator tends to place multiple instances onto the same host, as will be discussed in Section 5.2. Consequently, every instance is co-located with some other instances and SIE will fail to remove any instance.

4.4 Evaluating Fingerprinting

4.4.1 Accuracy Results. We evaluate fingerprint accuracy at a large scale on the public Cloud Run platform in three different data centers: *us-east1*, *us-central1*, and *us-west1*. In each data center, we deploy a service and launch 800 concurrently-running container instances. Although a user can launch up to 1000 container instances from the same service on Cloud Run, new instance creation slows down as the instance count approaches 1000, escalating the financial cost. As a result, we launch 800 instances. We accomplish this by configuring each instance to just handle one connection and then establishing 800 WebSocket connections to these instances.

For each instance, we collect its host CPU model name, the TSC value, and the real-world time of the measurement. We also record the true co-locations of the instances (i.e., the ground truth) using the scalable validation methodology discussed in Section 4.3. Our implementation of the methodology utilizes a low-noise covert channel based on contention on the random number generator (RNG) [27]. Because the RNG is rarely used [27], we find that the likelihood of observing RNG contention due to background activities is less than 1%. We require the presence of contention in at least 30 measurements out of 60 to confirm co-location. Hence, the risk of false positives is extremely small. Finally, we evaluate the fingerprint accuracy while varying the rounding precision of T_{boot} (i.e., p_{boot}). We repeat our experiments 5 times at different days and different times of day, totaling 15 measurements across three data centers.

To measure fingerprint accuracy, we examine all unique pairs of instances. For each pair of matching fingerprints, if the instances are indeed co-located, it is a true positive; otherwise, it is a false positive. For each pair of mismatching fingerprints, if the instances are not co-located, it is a true negative; otherwise, it is a false negative. We call the number of true and false positives TP and FP, respectively, and the number of true and false negatives TN and FN, respectively. Then, we compute the Fowlkes-Mallows index (FMI) [31], which is a common metric of clustering performance. FMI is calculated by

$$FMI = \sqrt{precision \cdot recall} = \sqrt{\frac{TP}{TP+FP} \cdot \frac{TP}{TP+FN}}.$$

FMI ranges from 0 to 1, with 1 indicating that the fingerprints are perfect (i.e., there are no false positives or false negatives).

Figure 4 shows the accuracy results averaged across all measurements and the three data centers under evaluation. The top plot of Figure 4 shows the FMI for different values of p_{boot}; the bottom plot shows the recall and the precision for the same values of p_{boot}. In the figure, p_{boot} values are measured in seconds. The error bars represent the standard deviations. From Figure 4, we observe that when p_{boot} is small and, therefore, the rounded T_{boot} has many significant digits (left end of Figure 4), fingerprints suffer from low recall (i.e., many false negatives) and have a low FMI. The reason is that a small p_{boot} cannot overcome the noise in measuring T_{boot}. Hence, the same host gets different fingerprints in different co-located instances. On the other hand, if p_{boot} is very large and, therefore, the rounded T_{boot} has few significant digits (right end of Figure 4), different hosts with similar boot time are rounded to the same value. The result is many false positives, which reduce the precision and the FMI.

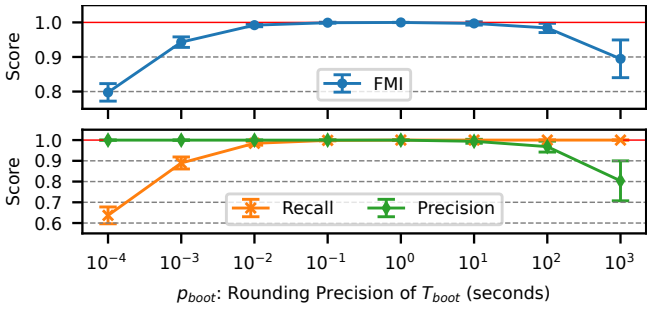


Figure 4. Average fingerprint accuracy with respect to the rounding precision of T_{boot} (i.e., p_{boot}). Error bars show standard deviations. Our Gen 1 fingerprints show near-perfect accuracies with 100 ms $\leq p_{boot} \leq$ 1 s.

The sweet spot for p_{boot} ranges from 100 ms to 1 s, where we reach an average FMI of 0.9999, which is nearly perfect. Since using a large p_{boot} can extend the expiration time of fingerprints, we use p_{boot} = 1 s by default. With this value, our fingerprints are highly accurate: among the 15 experiments conducted across three data centers, we find that 14 generate *perfect* fingerprints, while one generates nearly perfect fingerprints.

4.4.2 Fingerprint Expiration Time Results.

Our fingerprints are subject to drifting because we use the *reported* TSC frequency, which is inaccurate (Section 4.2). However, having long-lived fingerprints is crucial for the attacker to track hosts and understand instance placement behavior across many measurements over the time. Hence, in this subsection, we evaluate the expiration time of Gen 1 fingerprints.

To observe how the T_{boot} of a host drifts over the time, we launch a group of 50 long-running container instances and continuously record their hosts' fingerprints every hour for one week. Note that these instances can still be terminated and restarted on a different host over the course of our measurement. When this happens, we conservatively assume that the restarted instance runs on a different host. Consequently, most hosts have a fingerprint history shorter than a week. We filtered out fingerprint histories that are shorter than 24 hours.

We conducted our experiment in the *us-east1*, *us-central1*, and *us-west1* data centers. After filtering, we obtained 66, 67, and 79 fingerprint histories in each data center, respectively. Since we hypothesize that T_{boot} drifts linearly in Eq. 4.2, we use linear regression to fit T_{boot} as a function of real-world time for each fingerprint history and examine the r-value of the linear regression. An r-value with an absolute value close to 1 corresponds to a strong linear correlation [55]. We found that the *minimum* absolute r-value across all histories is 0.9997, suggesting that T_{boot} indeed drifts linearly.

Based on the T_{boot} of a fingerprint and the slope of its drifting obtained from the linear regression, we can use linear interpolation to accurately estimate the fingerprint expiration time. The expiration time is the amount of time it takes for T_{boot} to drift across a rounding boundary and result in a different rounded value. Figure 5 shows the CDF of the estimated expiration time of fingerprints in each of the three data centers. From the plot, it is clear that most fingerprints can last a few days before they expire. The average estimated time for 10% of the fingerprints to expire is about 2 days.

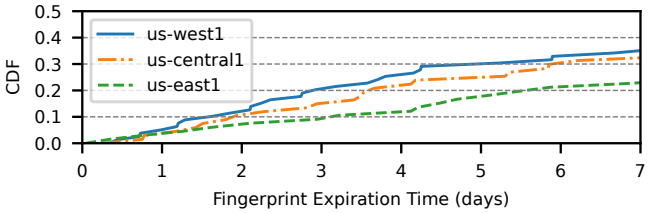


Figure 5. CDF of the estimated fingerprint expiration time. Most Gen 1 fingerprints take many days to expire.

4.5 Host Fingerprinting in the Gen 2 Environment

Our fingerprinting technique for Gen 1 is not directly applicable to the Gen 2 environment due to a hardware virtualization feature known as *TSC offsetting* [24]. With TSC offsetting, the hypervisor can configure the hardware to add an offset to the host TSC when it is read by the guest VM. Typically, when the hypervisor boots a guest VM, it saves the current value of the host TSC (call it tsc_0). Then, when the guest VM asks for a TSC value, the host returns the current TSC value minus tsc_0. This creates the illusion to the guest VM that the TSC was zero when the guest VM booted. Therefore, using Eq. 4.1, one can only derive the boot time of the guest VM instead of the host's.

To circumvent this challenge, we point out that, although the TSC value read by the guest VM has an unknown offset, the guest TSC still increments at the same rate as the host's. As a result, the guest VM can observe the host's actual TSC frequency, which deviates from the reported frequency and

is likely to be unique among hosts. Therefore, we propose to use the actual host TSC frequency to fingerprint a host in the Gen 2 execution environment.

Surprisingly, obtaining the actual host TSC frequency is easier in the VM-based Gen 2 environment than in the Linux container-based Gen 1 environment. This is despite the fact that VMs typically offer good isolation between the guest and host. In the Gen 2 environment, KVM exports the refined host TSC frequency to the guest VM for timekeeping. Since the attacker program has the root privilege within the guest VM, it can simply read the refined frequency from the guest kernel. However, this approach cannot be used to obtain the refined host TSC frequency in the Gen 1 environment and use it as f in Eq. 4.1, as the sandboxed Linux container can only interact with gVisor.

Using the same setup for validating Gen 1 fingerprints, we evaluate the accuracy of Gen 2 fingerprints in *us-east1*, *us-central1*, and *us-west1*. Our evaluation results show that the Gen 2 fingerprint is less accurate than the Gen 1 one, due to its low precision (i.e., its many false positives). Averaged across all measurements in the three data centers, the Gen 2 fingerprint has an FMI of 0.66, and a precision of 0.48. The reason for the low precision is that Linux only refines the TSC frequency to a precision of 1 kHz, causing multiple hosts to share the same refined frequency. In our experiments across three data centers, we find that, on average, 2.0 hosts have the same fingerprint.

Although the Gen 2 fingerprint has relatively low precision, it cannot produce false negatives. This is because the host TSC frequency is refined only once at host boot time, which means that co-located instances must have the same host TSC frequency. Because Gen 2 fingerprints cannot have false negatives, when we use the covert-channel approach of Figure 3 to verify fingerprints, we can perform the tests in Step ② *in parallel* without worrying about interference between tests. Also, we can skip Step ③, which finds false negatives. Consequently, even if the Gen 2 fingerprint is less accurate, we can still efficiently generate the co-location ground truth for numerous container instances.

5 Cloud Run Orchestrator and Co-location

Leveraging our host fingerprinting and co-location verification techniques proposed in Section 4, we can accurately identify physical hosts within a data center and verify instance co-location inexpensively. In Section 5.1, we employ these two techniques to systematically study Cloud Run's instance placement policy, uncovering exploitable behaviors. In Section 5.2, we propose adversarial instance launching strategies that exploit these uncovered behaviors, along with an evaluation of their efficacy and financial cost.

We set up our investigation using three standard Google Cloud accounts: Account 1, Account 2, and Account 3. Account 1 is designated as the attacker account, while Account 2 and Account 3 serve as victim accounts. Following our threat model of Section 3, we assume that once attacker and victim are co-located, the attacker can detect victim program execution and exfiltrate sensitive information using prior techniques [25, 41, 45, 54, 59].

When we rely on fingerprints to identify hosts without verifying them using a covert channel, we refer to these hosts as the *apparent hosts*. As Gen 1 fingerprints are nearly perfect and long-lived (Sections 4.4.1 and 4.4.2), apparent hosts identified by Gen 1 fingerprints should closely match real physical hosts. Lastly, our primary focus is on the Gen 1 environment, unless specified otherwise.

5.1 Understanding the Instance Placement Policy

We perform a set of experiments to study the instance placement policy of Cloud Run. Through these experiments, we seek to answer the following questions about a user launching numerous container instances: (i) How are the instances distributed across hosts and managed by Cloud Run (Experiment 1); (ii) Does the orchestrator exhibit a consistent behavior across launches (Experiment 2); and (iii) What are the major factors that affect the orchestrator's behavior (Experiments 3 and 4). In the remainder of this subsection, we primarily focus on the Gen 1 environment in the *us-east1* data center. We will discuss the different execution environments and data centers at the end of this subsection.

Experiment 1: instance distribution. In this experiment, we launch 800 instances of the same service, and record the set of hosts they are placed onto (i.e., their *host footprint*). We use the covert-channel approach to generate the co-location ground truth. We observe that these 800 instances are placed onto 75 hosts. Moreover, we see that the instance distribution across the hosts used is close to uniform, with the majority of hosts running 10 or 11 instances. This behavior is different from that of a VM environment (e.g., AWS EC2 [4]), where it is observed that instances from the same account do not share a host [54, 63].

Observation 1. Container instances of the same service share hosts, and instance distribution across the hosts used is close to uniform.

Next, we disconnect from these 800 instances, leaving them in an idle state, and observe when they are terminated by the orchestrator (Section 2.2). To record the termination time, we capture the `SIGTERM` signal sent by the orchestrator before it terminates an instance [15]. Upon capturing `SIGTERM`, the container reports the current time to a separate server and terminates. Figure 6 shows the number of idle instances as a function of time since disconnecting. From the plot, we can see that these idle instances are preserved in the first minute. After that, the orchestrator starts to gradually terminate idle instances. After about 12 minutes, almost every instance is terminated. This behavior matches Cloud Run's documentation [15], which states that idle instances are preserved for at most 15 minutes.

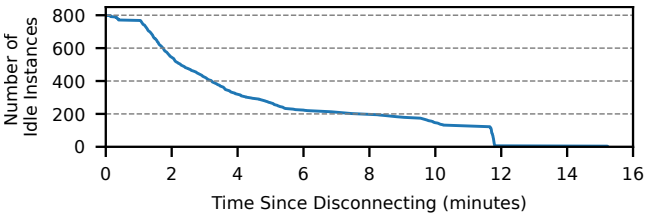


Figure 6. Number of idle instances after disconnecting from 800 instances, as measured in *us-east1*. Practically all instances are terminated in 12 minutes after disconnecting.

Observation 2. Cloud Run gradually terminates idle instances over an approximate period of 12 minutes.

Experiment 2: behavior across launches. We study if this scheduling behavior changes across launches. In this experiment, we repeat six times the launch of 800 instances of the same service, and compare the host footprint of each launch. After each launch, we immediately disconnect from the 800 instances, putting them into an idle state. Then, we wait for 45 minutes before the next launch to make sure that all the old instances are terminated and the service enters a "cold" state. This cold state will be discussed in Experiment 4.

Upon analyzing the experiment results, we observe that the instance distribution remains consistent across launches. We then examine whether instances from different launches share any hosts. Since the old instances from a previous launch are terminated before the next launch, it is impossible to use a covert channel to verify co-location of instances across launches. We rely solely on fingerprints to identify hosts in this experiment, thus reporting the apparent hosts.

Figure 7 shows the number of apparent hosts in each launch (identified by the Launch ID). It also shows the cumulative number of apparent hosts since the first launch. We observe that each launch occupies a similar number of apparent hosts. Moreover, the growth of the cumulative number of apparent hosts is minimal, suggesting that the apparent host footprints are highly overlapped across launches. This behavior can be caused by a data locality optimization, as the orchestrator may prefer hosts that already have the container image from previous launches.

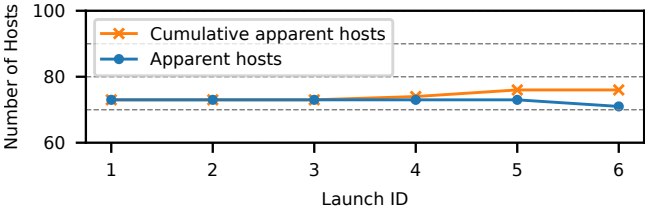


Figure 7. Number of apparent hosts occupied by 800 instances of the same service, as measured in *us-east1*. The footprints of apparent hosts are highly overlapped across launches.

To test the hypothesis of the data locality optimization, we repeat this experiment using a *different* service in each launch. These services are owned by the same account. Before the experiment, we rebuild the container images of the services that will be invoked, to ensure that the images of the services are not cached in any host. Under this configuration, we still observe a pattern that closely resembles Figure 7.

These experiment results suggest that the orchestrator tends to place instances from the *same account* onto a specific set of hosts. This behavior can be explained by affinity scheduling (Section 2.2). Affinity scheduling aims to reduce communication overhead by co-locating instances that frequently interact with each other, which is a likely scenario for services originating from the same account.

Observation 3. Cloud Run exhibits a consistent behavior across launches. It prefers a specific set of hosts for container instances owned by the same account. We refer to these preferred hosts as the ***base hosts***.

Experiment 3: different accounts. In this experiment, we modify Experiment 2 slightly: the services used in launch 1 and 2 are owned by Account 1, the services used in launch 3 and 4 are owned by Account 2, and the services used in launch 5 and 6 are owned by Account 3. Figure 8 illustrates the results in a manner similar to Figure 7. We observe that the cumulative number of apparent hosts establish a step pattern. When a launch uses a service owned by an account different from the accounts in previous launches, we see a large growth in the cumulative number of apparent hosts; otherwise, the growth is minimal. This observation suggests that the orchestrator uses different base hosts for different accounts.

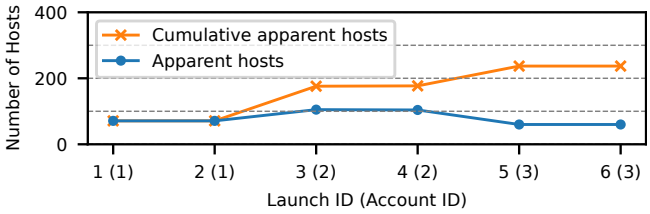


Figure 8. Number of apparent hosts occupied by 800 instances from three different accounts, as measured in *us-east1*. The numbers in parenthesis are the account IDs.

Observation 4. The Cloud Run orchestrator uses different base hosts for different accounts.

Experiment 4: short launch interval. In this experiment, we repeat Experiment 2 with a short time interval between launches of 10 minutes. Under this configuration, we see an interesting orchestrator behavior, as illustrated in Figure 9. Unlike Figure 7 from Experiment 2, we observe that both the number of apparent hosts and the cumulative number of apparent hosts drastically increase after each of the first three launches. Moreover, the difference between the two

curves is small. These results suggest that the orchestrator places instances to both the hosts used in previous launches and the new hosts. As a result, after six launches, we have a host footprint of 264 apparent hosts, which is far higher than the number of base hosts.

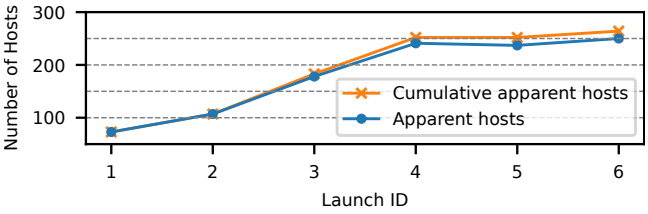


Figure 9. Experiment 2 repeated with a time interval between launches of 10 minutes, as measured in *us-east1*. We observe drastic increases in both the number of apparent hosts and cumulative number of apparent hosts.

To further investigate this behavior, we repeat this experiment with different launch intervals. We observe that this behavior only occurs with an interval smaller than 30 minutes. Furthermore, when the interval is too short, the number of new hosts is small. For example, with the 10-minute interval shown in Figure 9, we observe 177 more apparent hosts after launch 6 than after launch 1. However, with a 2-minute interval, we observe only 12 more apparent hosts. We also repeat the experiment with a different service used in each launch, but we do not observe this behavior.

Based on the aforementioned observations, we hypothesize that this behavior is induced by a load-balancing mechanism of Cloud Run. The mechanism considers the usage of the *same* service within approximately the past 30 minutes. If the service exhibits a high demand (i.e., repeatedly running 800 concurrent instances in our case), then the orchestrator will attempt to place some of the instances of the same service onto hosts that are not base hosts. As a result, it reduces the load on the base hosts. We refer to these extra hosts that are used as *helper hosts*. After a certain number of repeated launches, this behavior saturates.

Observation 5. For a service that has a high demand within less than 30 minutes, Cloud Run appears to use a load balancer that places instances of the service onto hosts that are not base hosts (i.e., ***helper hosts***).

Under this hypothesis, in the first launch, since there is no usage history of the service in the past 30 minutes, instances are placed onto the base hosts. As we wait for some time before the next launch, some of the idle instances are terminated. Therefore, in the next launch, the orchestrator has to create new instances to compensate for the terminated ones. Since the previous launch has primed the service into a high-demand state, the orchestrator starts to place the newly-created instances on the helper hosts to relieve pressure on the base hosts. This cycle repeats for a few iterations. When the wait interval between launches is small, the number of terminated idle instances before the next launch is also small. Consequently, the orchestrator creates fewer new instances, thereby occupying fewer helper hosts.

We consistently observe this behavior when repeating the experiment at different times of the day or using a different service in the experiment. We also observe that different services use different sets of helper hosts; these sets are not mutually exclusive and do overlap. We demonstrate this fact by repeating Experiment 4 for six episodes; in each episode, we use a different service that is launched six times with 800 instances every time. In each episode of Experiment 4, we measure the helper host footprint by computing the difference between the host footprint after the sixth launch and after the first launch.

Figure 10 shows the results of the experiment. It shows the number of apparent helper hosts and the cumulative footprint of apparent helper hosts after each of the episodes. We see that the cumulative footprint of apparent helper hosts expands after each episode. This expansion suggests that each episode uses new helper hosts not seen in previous episodes. The increase in the cumulative footprint of helper hosts after a single episode is less than the number of helper hosts in that episode, indicating overlaps in helper hosts across different services. As will be discussed later, an attacker can exploit this behavior by repeatedly launching instances of multiple services and therefore obtaining residence on a substantial portion of Cloud Run hosts within a data center.

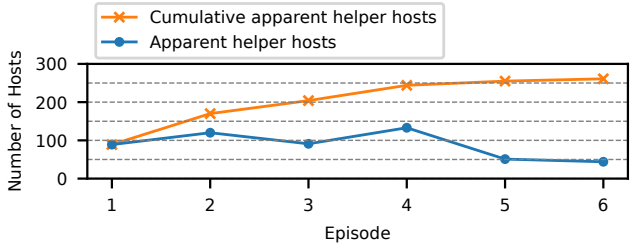


Figure 10. Experiment 4 repeated in six episodes, with a different service used per episode, as measured in *us-east1*. We see growth in the cumulative *helper* host footprint after each episode.

Observation 6. Different Cloud Run services use different but overlapping sets of helper hosts.

Other factors. We investigate other factors that influence the orchestrator's behavior. We report four findings. First, we observe similar placement behavior when launching on different dates and at different times of day. Second, container instances with different resource specifications (such as CPU and memory) share the same base hosts. Third, all nine Cloud Run data centers in the US exhibit similar placement behavior except for *us-central1*, where instance placement is more dynamic. In *us-central1*, many instances are placed onto different hosts across launches, even if we launch from a cold

service in each launch. Fourth, the Gen 2 execution environment shows similar placement behavior, and Gen 2 instances can share hosts with Gen 1 instances.

Implications. The existence of base hosts is a double-edged sword for the attacker. On the one hand, it reduces the uncertainty on where the victim instances are likely to reside, making co-location with the victim easier. On the other hand, base hosts limit the set of hosts where the attacker can reside and, therefore, the set of hosts that the attacker can explore. As a result, naively launching attacker instances has a low chance of co-locating with the victim (Section 5.2), as different accounts often use different base hosts. In practice, the load-balancing behavior of Cloud Run helps the attacker to overcome this challenge. For example, the attacker can prime their services into a high-demand state through repeated launches, which help spread attacker instances onto many helper hosts. This strategy drastically improves the efficacy of co-location attacks (Section 5.2).

5.2 Co-location with Victims

In this subsection, we evaluate two instance launching strategies for co-location with victims. Our primary metric is the *victim instance coverage*, which is the percentage of victim instances that are co-located with the attacker. Then, we report the financial cost of the attack, the estimated size of the Cloud Run data centers, and the transferability of our results to the Gen 2 environment. We conclude with a discussion on potential optimizations that can enhance attack efficacy.

Evaluation setup. For this evaluation, Account 1 is designated as the attacker, while Account 2 and Account 3 serve as the victims. We conduct the evaluation across three data centers: *us-east1*, *us-central1*, and *us-west1*. For each combination of data center and victim account, we repeat the measurement three times on different days and at different times of day. Co-location between attacker and victim instances is verified using the covert-channel method described in Section 4.3.

In each experiment, we vary the number of victim instances. As the default configuration for Cloud Run services allows a maximum of 100 instances, we assess configurations with 20, 50, 100, and 200 victim instances, setting the 100-instance configuration as the default. Prior work [28] suggests that orchestrators might prefer co-locating instances with similar resource specifications (such as CPU and memory) in the same nodes. Accordingly, we vary the size of the victim instances, using the sizes outlined in Table 1. We choose Small as the default victim size since it is the standard configuration for Cloud Run services. We also fix the attacker instance size to Small.

Strategy 1: naive instance launching. Here, the attacker simply launches numerous instances from services in a cold state. This strategy represents a naive attacker who has no insight into the Cloud Run's instance placement behavior. In our experiment, this naive strategy launches 4,800 instances from six services.

Table 1. Various container sizes used in our evaluation. Note that we define these four container sizes for the purpose of this study; a user can use a size different than these four.

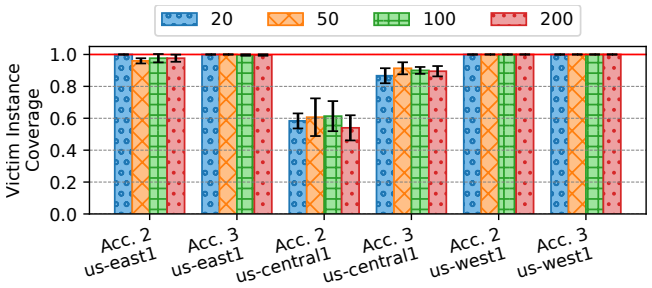
Size	# of CPUs	Memory
Pico	0.25	256 MB
Small (Default)	1	512 MB
Medium	2	1 GB
Large	4	4 GB

Despite the large number of attacker instances, we observe *zero* co-location with Account 2 in *us-east1* and *us-central1*, or with Account 3 in *us-east1* and *us-west1*. We see high average victim instance coverage only with Account 2 in *us-west1* (100.0%) and with Account 3 in *us-central1* (81.0%), as the base hosts of the attacker and victim happen to be highly overlapped in the corresponding data centers. Changing the number of victim instances or their size does not yield significant variations. This is consistent with our observation that services from the same account share the same base hosts, even when they have different resource specifications (Section 5.1). Overall, the data indicates that a naive launching strategy without any insight into Cloud Run's placement behaviors is often ineffective.

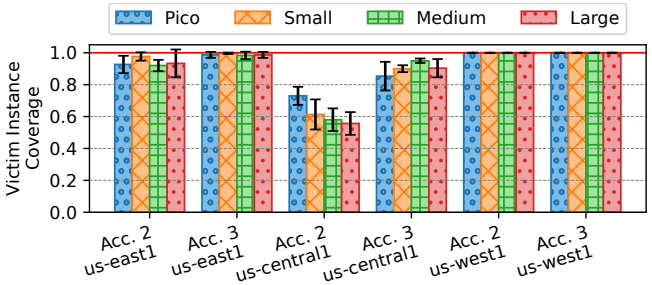
Strategy 2: optimized instance launching. This strategy exploits the load-balancing behavior of Cloud Run. The high-level idea is to prime the attacker service into a high-demand state by repeatedly launching many instances with an appropriate time interval. This action enables the attacker to deploy instances onto numerous helper hosts. Given our observation that different services use different but overlapping sets of helper hosts (Section 5.1), attacker instances can reside on more helper hosts if they utilize multiple services. In our experiment, the attacker deploys six services. Similar to Experiment 4 in Section 5.1, the attacker repeatedly launches 800 instances of each service at a 10-minute interval, killing the instances after each launch except after the last one.

Figure 11 shows the average victim instance coverage using the optimized launching strategy, with the error bars indicating standard deviations. In Figure 11a, we vary the number of victim instances while fixing the victim size to Small. Conversely, in Figure 11b, we vary the victim size while keeping the number of victim instances set to 100.

Figure 11a illustrates that our optimized instance launching strategy is highly effective. With the default configuration of 100 Small victim instances, we observe high victim instance coverage. From left to right, we see, in *us-east1*, victim instance coverages of 97.7% and 99.7% with Account 2 and Account 3, respectively. In *us-central1*, we see lower coverages of 61.3% with Account 2 and 90.0% with Account 3. Finally, in *us-west1*, we see 100.0% coverage with



(a) Varying the number of victim instances (20, 50, 100, and 200). The victim instance size is fixed to Small.



(b) Varying the victim instance size (Pico, Small, Medium, and Large). The number of victim instances is fixed to 100.

Figure 11. Average victim instance coverage across three measurements. Error bars represent standard deviations. "Acc." is an abbreviation for "Account". Our optimized launching strategy can achieve a high victim coverage in different evaluated data centers.

both Account 2 and Account 3. One potential reason for the reduced coverage in the *us-central1* data center is its vast size, as will be shown later in this section. Another factor might be that *us-central1* has a more dynamic instance placement behavior, as discovered in Section 5.1.

Figure 11b shows a similar behavior, as we vary the victim size while keeping the number of victim instances set to 100. Overall, considering the data from both Figure 11a and 11b, we conclude that, in the large majority of cases, the number or the size of victim instances has no significant influence on the average victim instance coverage.

Financial cost of the attack. FaaS platforms such as Cloud Run only charge users for the active time of instances. Since we disconnect from all the instances after each launch, these instances are in the idle state between launches and do not contribute to the cost during this time. The main cost comes from launching instances. On average, to set up a co-location attack with our configuration (six attacker services, six launches per service, and 800 instances per launch), the estimated average costs are 24 USD, 23 USD, and 27 USD in *us-east1*, *us-central1*, and *us-west1*, respectively. These costs are small.

Scale of Cloud Run clusters. To estimate the size of a Cloud Run cluster, we deploy eight services from each of the three accounts (Account 1, Account 2, and Account 3) and use the total 24 services to explore hosts that run Cloud Run services in the data center. We use the optimized strategy to launch instances of these services and record the apparent host footprint of each launch. We launch each service four times. Then, the size of the Cloud Run cluster is estimated by counting the number of unique host fingerprints across *all* launches. The intuition for using services from different accounts instead of more services from the same account is that we can start exploration from different base hosts, and thus discover new hosts more efficiently.

Figure 12 shows the cumulative number of unique apparent hosts as we aggregate apparent hosts across launches. In total, these launches found 474 apparent hosts in *us-east1*, 1702 apparent hosts in *us-central1*, and 199 apparent hosts in *us-west1*. Since the growth of the cumulative number of unique apparent hosts gradually flattens out in all three data centers as we include more launches, it is reasonable to use the total number of unique apparent hosts that we found to estimate the size of the Cloud Run cluster in each data center. Using this estimation, in the co-location experiment that we performed using Strategy 2, the attacker (Account 1) covered 59%, 53%, and 82% of the hosts in *us-east1*, *us-central1*, and *us-west1* on average, respectively.

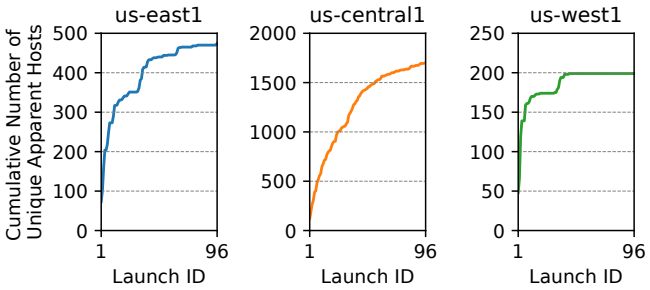


Figure 12. Cumulative number of unique apparent hosts across launches.

Co-location in the Gen 2 environment. We evaluate our optimized launching strategy in the Gen 2 environment, with both attacker and victims launching Gen 2 instances. Averaging three measurements in each data center, we observe victim instance coverage of 87.3% with Account 2 and 88.7% with Account 3 in *us-east1*; 40.7% with Account 2 and 75.3% with Account 3 in *us-central1*; and 96.0% with Account 2 and 97.3% with Account 3 in *us-west1*. No significant coverage differences arise when varying the number of victim instances or size. These results indicate that our launching strategy is highly effective in the Gen 2 environment as well.

Potential attack optimizations. To occupy an even larger fraction of Cloud Run hosts within a data center, the attacker can create more accounts and deploy more services per account. This approach is similar to the experiment where we measured the scale of Cloud Run clusters. However, a

challenge arises as cloud providers often cap new accounts to limited resources—e.g., allowing a maximum of only 10 instances per service. For an attacker to be eligible for higher quotas, the new account needs to sustain a consistent usage *over several months*. This limitation results in additional time and financial costs.

If the attacker intends to repeatedly attack services from the same victim account, an optimization is to record the fingerprints of hosts used by the victim during the first attack. These hosts can be the base hosts preferred by the victim. Therefore, in the subsequent attacks targeting the same victim, the attacker can focus side-channel attack efforts on hosts with fingerprints that match the fingerprints recorded in the first attack.

6 Potential Mitigations

Our two fingerprinting techniques exploit the fact that the timestamp counter (TSC) value or its frequency are shared between the host and untrusted user containers. Therefore, to defend against our techniques, one could mask both the value and frequency of the host's TSC through either TSC emulation or hardware-assisted TSC virtualization.

In the non-virtualized Gen 1 environment, the host can disable `rdtsc` and `rdtscp` in Ring-3 (i.e., userspace) by configuring the `CR4` model specific register [24]. By doing so, the kernel can trap and emulate both instructions. However, this mitigation also forces user applications to switch to kernel space when accessing high-precision timers, incurring high timer access overhead.

The impact of the slower timer accesses depends on the specific application and its use case. Hence, the actual end-to-end execution overhead in an application can only be determined through benchmarking. We identify some applications where this added end-to-end execution overhead is likely to matter: (1) real-time systems that process high-frequency events such as live media or financial data, (2) database systems using fine-grained timestamps for concurrency control, (3) distributed systems employing fine-grained timestamps for synchronization, and (4) applications characterized by intensive logging and journaling. For instance, Cassandra's [30] write latency is reportedly reduced by 43% on Amazon EC2 instances after switching from the `xen` clock source to TSC (the `xen` clock source requires a context switch to kernel space to access) [34].

In the virtualized Gen 2 environment, the hypervisor can also trap and emulate both `rdtsc` and `rdtscp`, which, as in Gen 1, leads to significant timer access overhead. An alternative that does not add overhead is for the host to support hardware-assisted TSC virtualization features, such as TSC offseting and scaling [24, 39]. These features are available on modern Intel and AMD processors and were primarily developed for live VM migration.

Besides emulating or virtualizing the TSC, cloud vendors can also adopt scheduling algorithms that reduce the chance of co-location [6, 37] or mitigate the risk of side-channel attacks after co-location is achieved [58]. Finally, they can detect and stop ongoing side-channel attacks [19, 38, 66, 67].

7 Related Work

Co-location attacks in the public cloud. In 2009, Ristenpart et al. [54] conducted the first study of VM co-location attacks on Amazon's EC2 service using network probing. To assess Amazon's defenses in response to this work, Xu et al. [63] investigated VM co-location attacks on Amazon EC2 service in 2015, by employing network scanning. However, these network-based techniques have become obsolete in the modern cloud environment, due to the widespread adoption of the virtual private cloud (VPC) [18], which logically isolates the networking environments of different accounts. To overcome the challenge posed by VPC, Varadarajan et al. [59] employed a pairwise covert-channel, based on memory bus contention explored by Wu et al. [62], to investigate VM co-location attacks. However, due to the scalability issue of pairwise testing, their approach is unsuitable for the modern FaaS environment, where it is necessary to verify co-location of thousands of instances.

Exploiting cloud orchestrators. Makrani et al. [47] proposed an attack named Cloak & Co-locate, which employs adversarial machine learning to generate fake resource usage traces, fooling machine learning-based resource provisioning systems [3, 60] to co-locate attacker instances with the victim. However, their evaluation is limited to a private mini-cloud running on local clusters. Concurrent to Cloak & Co-locate, Fang et al. [28] proposed Repttack, suggesting that the attacker can launch instances with requirements and preferences that replicate the victim's to increase the likelihood of co-location. However, on Cloud Run, we do not observe any significant increase in co-location rate when the attacker instance has the same configuration as the victim (Section 5.2).

Remote device fingerprinting. Kohno et al. [43] exploited the clock skew in system time that is accumulated over time to fingerprint remote physical devices. They monitor such skew through timestamps included in TCP packets. However, for contemporary systems where clocks are well synchronized to the real-world time through the network time protocol (NTP) [48], such accumulated clock skew is not detectable using coarse-grained TCP timestamps [52], which have the resolution of only one millisecond [9]. Compared to their approach, our fingerprinting method for Gen 1 relies on the host's boot time instead of clock skew. Further, our fingerprinting method for Gen 2 detects clock *frequency* skews, making it effective even if clocks are well synchronized through NTP.

8 Conclusion

In this paper, we presented the first comprehensive study on risks of and techniques for co-location attacks in modern public cloud FaaS environments. We introduced two novel physical host fingerprinting techniques based on the timestamp counter to aid in reverse engineering instance placement policies. Using host fingerprints, we proposed a cost-effective methodology for large-scale instance co-location verification. With these techniques, we conducted an extensive study on Google Cloud Run, discovering an exploitable instance placement behavior. Based on our findings, we devised an efficient instance launching strategy that deploys attacker instances across a large portion of Cloud Run cluster hosts within a data center. Our strategy attains high attack efficacy at minimal financial cost.

Acknowledgments

We thank the anonymous reviewers and the paper's shepherd, Jacob R. Lorch, for their valuable feedback and comments. We particularly appreciate the suggestion of using TSC scaling as a mitigation. This research was funded in part by an Intel Resilient Architectures and Robust Electronics (RARE) gift; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by NSF grants 1942888, 1954521, and 1956007.

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 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 870–887. https://doi.org/10.1109/SP.2019.00066

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. USENIX Association, 469–482.

[4] Amazon AWS. 2023. Secure and resizable cloud compute - Amazon EC2 - Amazon Web Services. https://aws.amazon.com/ec2/.

[5] Amazon AWS. 2023. Serverless Computing - AWS Lambda - Amazon Web Services. https://aws.amazon.com/lambda/.

[6] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and F. Bruce Shepherd. 2014. Co-Location-Resistant Clouds. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*. ACM, 9–20. https://doi.org/10.1145/2664168.2664179

[7] Microsoft Azure. 2023. Azure Functions – Serverless Functions in Computing | Microsoft Azure. https://azure.microsoft.com/en-us/products/functions.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 785–800. https://doi.org/10.1145/3319535.3363194

[9] David A. Borman, Bob Braden, Van Jacobson, and Richard Scheffenegger. 2014. TCP Extensions for High Performance. *RFC* 7323 (2014), 1–49. https://doi.org/10.17487/RFC7323

[10] Google Cloud. 2023. About container instance autoscaling | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/about-instance-autoscaling.

[11] Google Cloud. 2023. About execution environments | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/about-execution-environments.

[12] Google Cloud. 2023. Cloud Functions | Google Cloud. https://cloud.google.com/functions.

[13] Google Cloud. 2023. Cloud Run: Container to production in seconds | Google Cloud. https://cloud.google.com/run/.

[14] Google Cloud. 2023. Cloud Run release notes | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/release-notes.

[15] Google Cloud. 2023. Container runtime contract | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/container-contract.

[16] Google Cloud. 2023. Invoking with an HTTPS Request | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/triggering/https-request.

[17] Google Cloud. 2023. Pricing | Cloud Run | Google Cloud. https://cloud.google.com/run/pricing.

[18] Google Cloud. 2023. Virtual Private Cloud (VPC) | Google Cloud. "https://cloud.google.com/vpc".

[19] Cloudflare. 2023. Security Model - Cloudflare Workers docs. https://developers.cloudflare.com/workers/learning/security-model/.

[20] Kubernetes Contributors. 2023. Kubernetes Scheduler | Kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[21] Linux Contributors. 2023. Linux Source Code. https://github.com/torvalds/linux/blob/e62252bc55b6d4eddc6c2bdbf95a448180d6a08d/arch/x86/kernel/tsc.c.

[22] Memcached Contributors. 2018. memcached - a distributed memory object caching system. https://memcached.org/.

[23] Wikipedia Contributors. 2023. Pentium III - Wikipedia. https://en.wikipedia.org/wiki/Pentium_III.

[24] Intel Corparation. Dec, 2021. Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes.

[25] Christina Delimitrou and Christos Kozyrakis. 2017. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 599–613. https://doi.org/10.1145/3037697.3037703

[26] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 51–67. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[27] Dmitry Evtyushkin and Dmitry V. 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, CCS 2016, Vienna, Austria, October 24-28, 2016*. ACM, 843–857. https://doi.org/10.1145/2976749.2978374

[28] Chongzhou Fang, Han Wang, Najmeh Nazari, Behnam Omidi, Avesta Sasan, Khaled N. Khasawneh, Setareh Rafatirad, and Houman Homayoun. 2022. Repttack: Exploiting Cloud Schedulers to Guide Co-Location Attacks. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-237/

[29] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, Tim Harris and Michael L. Scott (Eds.). ACM, 37–48. https://doi.org/10.1145/2150976.2150982

[30] Apache Software Foundation. 2023. Apache Cassandra | Apache Cassandra Documentation. https://cassandra.apache.org/_/index.html.

[31] Edward B Fowlkes and Colin L Mallows. 1983. A method for comparing two hierarchical clusterings. *J. Amer. Statist. Assoc.* 78, 383 (1983), 553–569.

[32] Google. 2023. Google Docs. https://docs.google.com/.

[33] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[34] Brendan Gregg. 2021. The Speed of Time. https://www.brendangregg.com/blog/2021-09-26/the-speed-of-time.html.

[35] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[36] gVisor Contributors. 2023. The Container Security Platform | gVisor. https://gvisor.dev/.

[37] Yi Han, Tansu Alpcan, Jeffrey Chan, Christopher Leckie, and Benjamin I. P. Rubinstein. 2016. A Game Theoretical Approach to Defend Against Co-Resident Attacks in Cloud Computing: Preventing Co-Residence Using Semi-Supervised Learning. *IEEE Transactions on Information Forensics and Security* 11, 3 (2016), 556–570. https://doi.org/10.1109/TIFS.2015.2505680

[38] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 639–650.

[39] Advanced Micro Devices Inc. June, 2023. AMD64 Architecture Programmer's Manual. Volumes 1-5.

[40] Docker Inc. 2023. Docker: Accelerated, Containerized Application Development. https://www.docker.com/.

[41] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptol. ePrint Arch.* (2015), 898. http://eprint.iacr.org/2015/898

[42] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Workshop/Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.

[43] Tadayoshi Kohno, Andre Broido, and Kimberly C. Claffy. 2005. Remote Physical Device Fingerprinting. In *2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA*. IEEE Computer Society, 211–225. https://doi.org/10.1109/SP.2005.18

[44] Redis Labs. 2022. Redis In-Memory Data Structure. https://redis.io.

[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 605–622. https://doi.org/10.1109/SP.2015.43

[46] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 301–312. https://doi.org/10.1109/ISCA.2014.6853237

[47] Hosein Mohammadi Makrani, Hossein Sayadi, Najmeh Nazari, Khaled N. Khasawneh, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. 2021. Cloak & Co-locate: Adversarial Railroading of Resource Sharing-based Attacks on the Cloud. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021*. IEEE, 1–13. https://doi.org/10.1109/SEED51797.2021.00011

[48] David L. Mills, Jim Martin, Jack L. Burbank, and William T. Kasch. 2010. Network Time Protocol Version 4: Protocol and Algorithms Specification. *RFC* 5905 (2010), 1–110. https://doi.org/10.17487/RFC5905

[49] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. MemJam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* 47, 4 (2019), 538–570.

[50] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' track at the RSA conference*. 1–20.

[51] Colin Percival. 2005. Cache missing for fun and profit. https://www.daemonology.net/papers/cachemissing.pdf.

[52] Libor Polčák, Jakub Jirásek, and Petr Matoušek. 2013. Comment on "remote physical device fingerprinting". *IEEE Transactions on Dependable and Secure Computing* 11, 5 (2013), 494–496.

[53] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *Sixth International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019, Granada, Spain, October 22-25, 2019*. IEEE, 209–214. https://doi.org/10.1109/IOTSMS48152.2019.8939164

[54] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. 199–212. https://doi.org/10.1145/1653662.1653687

[55] Sheldon M Ross. 2017. *Introductory Statistics*. Academic Press.

[56] Eric Schurman and Jake Brutlag. 2009. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*. O'Reilly Media.

[57] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. 2022. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *31st USENIX Security Symposium (USENIX Security 22)*. 989–1007.

[58] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M. Swift. 2014. Scheduler-based Defenses against Cross-VM Side-channels. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 687–702. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan

[59] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. USENIX Association, 913–928. https://www.usenix.org/conference/usenixsecurity15/

technical-sessions/presentation/varadarajan

[60] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. USENIX Association, 363–378.

[61] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and Practice of Finding Eviction Sets. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 39–54. https://doi.org/10.1109/SP.2019.00042

[62] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. USENIX Association, 159–173. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu

[63] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co-residence Threat inside the Cloud. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. USENIX Association, 929–944. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu

[64] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 888–904. https://doi.org/10.1109/SP.2019.00004

[65] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[66] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2016. CloudRadar: A real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*. Springer, 118–140.

[67] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 313–328. https://doi.org/10.1109/SP.2011.31

[68] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 990–1003. https://doi.org/10.1145/2660267.2660356