Bit of a Close Talker: A Practical Guide to Serverless Cloud Co-Location Attacks

Wei Shao*, Najmeh Nazari*, Behnam Omidi†, Setareh Rafatirad*,
Khaled N. Khasawneh†, Houman Homayoun*, and Chongzhou Fang‡
*University of California, Davis, USA
Email: {wayshao, nnazari, srafatirad, hhomayoun}@ucdavis.edu
†George Mason University, USA
Email: {bomidi,kkhasawn}@gmu.edu
‡Rochester Institute of Technology, USA
Email: cxfeec@rit.edu

***Abstract*—Serverless computing has revolutionized cloud computing by offering users an efficient, cost-effective way to develop and deploy applications without managing infrastructure details. However, serverless cloud users remain vulnerable to various types of attacks, including micro-architectural side-channel attacks. These attacks typically rely on the physical co-location of victim and attacker instances, and attackers need to exploit cloud schedulers to achieve co-location with victims. Therefore, it is crucial to study vulnerabilities in serverless cloud schedulers and assess the security of different serverless scheduling algorithms. This study addresses the gap in understanding and constructing co-location attacks in serverless clouds. We present a comprehensive methodology to uncover exploitable features in serverless scheduling algorithms and to devise strategies for constructing co-location attacks via normal user interfaces. In our experiments, we successfully reveal exploitable vulnerabilities and achieve instance co-location on prevalent open-source infrastructures and Microsoft Azure Functions. We also present a mitigation strategy, the *Double-Dip* scheduler, to defend against co-location attacks in serverless clouds. Our work highlights critical areas for security enhancements in current cloud schedulers, offering insights to fortify serverless computing environments against potential co-location attacks.**

I. Introduction

In recent years, serverless computing, a relatively new form of cloud computing, has become prevalent and attracts a great number of users. It fully offloads t he t ask o f infrastructure management to cloud providers, enabling developers to focus more on the functionality of their applications and to host services more cost-effectively [1], [2]. Today, we can still see an increasing number of commercial services pivoting to this cloud computing model. This computing paradigm has been supported by mainstream cloud providers, e.g., Amazon AWS Lambda [3], Google Cloud Run [4], and Microsoft Azure Functions [5]. Compared to traditional cloud computing (often referred to as "serverful computing"), serverless computing has the following benefits: (1) Flexibility, as the cloud provider-managed light-weight serverless instances can flexibly scale depending on the load of clients' services and cloud providers can achieve higher resource efficiency; (2) Convenience, since client developers now only need to develop functional code and directly submit code or container images to cloud providers, with all resource management details being hidden from them; (3) Customer cost efficiency, as customers are only charged for the actual resources they utilize (often measured by the number of function invocations), unlike in serverful cloud computing, where customers are charged by instance time and a lot of costs are wasted on instance idling.

Resource provisioning, i.e., scheduling, is an important aspect of serverless cloud operation. It impacts the performance of customer applications and can harm the profits of cloud providers if serverless instances are not properly placed and provisioned. Apart from resource scheduling challenges that are similar to serverful clouds, one unique challenge in serverless scheduling is the existence of cold-start [6], i.e., the time-consuming start-up process of serverless instances. Another unique challenge arises from the flexible nature of resource management of serverless systems, and serverless cloud providers need to adaptively perform auto-scaling to accommodate the dynamic service workloads. There are lots of research works dedicated to improving the performance of serverless systems from scheduling policies [7], [8], [9].

Despite being convenient and cost-effective for cloud users, serverless clouds still face security challenges posed by various types of micro-architectural side-channel attacks. For example, remote micro-architectural attacks like Prime+Probe [10] remain feasible in serverless clouds with certain optimizations [11], threatening the security of serverless users. These attacks depend on the co-location of attackers' and victims' serverless instances on the same physical server within the cloud cluster. Although perfect isolation can be obtained by switching to dedicated hosts, this comes at a prohibitive cost: the cheapest dedicated servers from AWS and Azure start at $323 and $444 per month, respectively, which is over 10 times higher than running the same workload on serverless platforms, as discussed in Section IX-B. Therefore, beyond

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241376
www.ndss-symposium.org

understanding the mechanics of the actual attack, it is crucial to study how this critical prerequisite of co-location can be achieved and to analyze potential attacker strategies. This analysis will inspire the design of more secure serverless schedulers. Co-location attack methods have been investigated in traditional clouds [12], [13], [14], [15] and Google Cloud Run [16]. However, there is still a lack of studies on a universal methodology to exploit serverless cloud schedulers.

In this work, we aim to bridge this gap in the literature and provide universal guidance on the construction of co-location attacks targeting different serverless clouds. We first provide a method to reveal exploitable features of serverless cloud scheduling algorithms, such as package locality optimizations [17]. Then, based on the found features, we offer a strategy to construct attacks accordingly. All these steps are done through a normal user interface, i.e., besides deploying serverless applications and invoking corresponding services, attackers do not have other access permissions to the cloud infrastructure. The main design effort is hence dedicated to constructing a set of attacks against serverless services from the user side.

The main contributions of this paper are summarized as follows:

- We propose a method to analyze and reveal serverless scheduler features;
- We develop a strategy for constructing co-location attacks based on the identified scheduler features;
- We conduct evaluations on different types of schedulers, uncovering key insights and offering guidance for improving scheduler design;
- As a case study, we successfully attack Azure Functions and manage to achieve co-location of serverless instances from different accounts.
- Additionally, we propose a scheduling algorithm called *Double-Dip* that aims to provide defense by offering "soft" user isolation.

The remainder of this paper is organized as follows. In Section II, we provide an introduction to background knowledge like serverless scheduling algorithms. We introduce our scheduler fingerprinting strategy in Section III and evaluate the effectiveness of our method in Section IV. Section V proposes the strategies to construct attacks, and the evaluation results are presented in Section VI. The case study on Microsoft Azure Functions is provided in Section VII. Our proposed *Double-Dip* scheduler as a mitigation is discussed and evaluated in Section VIII. Additional discussions are included in Section IX. Finally, a review of the literature and our conclusion are presented in Section X and Section XI, respectively.

II. Background

A. Serverless Schedulers

Serverless cloud offloads all infrastructure-related tasks to cloud providers, making cloud providers fully responsible for managing serverless function instances and properly performing operations like host launching, scaling out, etc. It is hence important to develop scheduling policies that determine the optimal distribution of resources.

Serverless scheduling is different from traditional scheduling in serverful clouds. In serverless clouds, user code is executed in encapsulated environments that pack required dependencies, e.g., containers or vendor-specific instances [18]. Serverless schedulers are responsible for deciding the placements of these instances. Like in traditional cloud scheduling and cluster scheduling, serverless schedulers need to determine how resources are assigned to user instances in the system based on resource availability. Additionally, serverless schedulers need to make decisions on how to dispatch serverless function invocations to instances. A major thread of research is dedicated to optimizing start time and avoiding unnecessary cold-start latency by optimizing scheduling policies [19], [20], [21]. Besides, as serverless cloud providers are also responsible for dynamically scaling function instances [22], auto-scaling and load-balancing are also important aspects of serverless schedulers. There are research works that consider different kinds of locality [23], [17] to optimize for load balancing performance. Later in this paper, we will show that from an attacker's view, these are features that can be exploited to increase the co-location attack success rate.

B. Cloud Co-Location Attacks

To launch malicious attacks like micro-architectural side-channel attacks [24], [25], [26], [27], an important prerequisite that attackers must achieve is the co-location of victim and attack instances. Without obtaining co-location, attackers cannot gain access to shared resources like cache and will be unable to launch subsequent attacks. Cloud co-location attacks [13], [12], [28], [16] aim to exploit scheduler features to manipulate instance placement results in cloud systems and force attacker and victim instances to be placed together on the same physical machine.

In both serverful and serverless clouds, a side-channel attacker needs to go through the following steps to eventually complete an attack, as shown in Fig. 1.

Step 0. Use trial submissions to collect information about the scheduler of the target cloud, and optimize submission strategies accordingly;
Step 1. Invoke attack functions following a predefined policy;
Step 2. Fingerprint co-residing applications and verify if any attacker instance is co-located with a victim;
Step 3. Launch attack and steal secret information from the victim.

Step 3 is the focus of most side-channel attack works, while Steps 0 - 2 are usually neglected by researchers. However, achieving co-location is a non-trivial process, especially as the size of datacenters increases over time. Prior works in recent years [13], [12], [16] indicate that special strategies should be constructed according to scheduler features to tamper with targeted cloud systems.

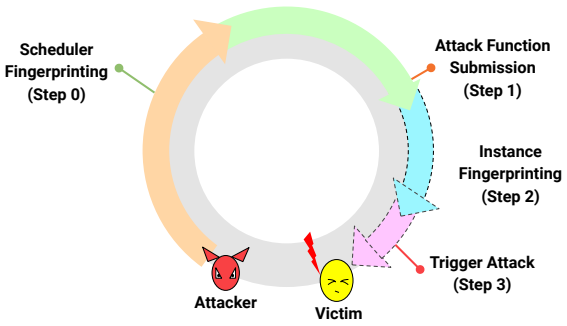


Fig. 1: Diagram of the overall attack flow.

C. Definitions

In the remainder of this paper, user-deployed serverless services are denoted as 'functions', while corresponding cloud instances are referred to as 'function hosts'. The requests to execute a function in the cloud, regardless of trigger types, are called function invocations.

D. Threat Model

In this paper, we only study Step 0 (Section III) - Step 1 (Section V) of the co-location attack. We assume non-privileged attackers, i.e., attackers can only launch attacks by invoking functions and collecting the returned execution results. The goal of attackers is to place **at least** one of their attack function hosts together with a victim's function host on the **same** physical machine to enable subsequent attack efforts. In addition, we assume that the attacker does not have any knowledge about the targeted serverless scheduler. Scheduling strategies need to be reverse-engineered (using our method in Section III), based on which the attacker constructs an attack strategy accordingly to target a specific scheduler. However, they can have certain knowledge about the targeted application, e.g., dependencies, which is realistic in the real world. Examples include Airbnb's disclosure of their usage of open-source projects [29] and Netflix's publication of their internal tool as an open-source project [30].

III. Scheduler Fingerprinting

We first propose a scheduler fingerprinting method to reveal certain important features in the scheduling algorithm designs. This will be the foundation of subsequent attack construction.

A. Algorithm Description

The goal of scheduler fingerprinting is to identify exploitable features of the scheduling algorithm of the target serverless cloud. The obtained information can be utilized to construct attacks targeting the cloud scheduler. By carefully crafting the function invocation sequence, an attacker will be able to obtain information about the scheduling algorithms based on the collected placement sequences.

In this part, we assume a reliable server identification method is provided, i.e., when invoking functions, the attacker will be able to differentiate servers where the function host is placed. This can be based on methods such as time and frequency probing [16] or network-based probing [14]. With this server fingerprinting method, an attacker can continuously 'discover' servers in the cloud system and provide unique identifiers to each discovered physical server. Following the same function invocation sequence, the collected trace of such identifiers will be different for different scheduling algorithms, hence they can be used to identify scheduling policies utilized by the cloud provider.

We denote the aforementioned server identification method as a function $\mathcal{F}$. Every time a serverless function ϕ is executed in the cluster, a hypothetical fingerprint of the corresponding physical server will be reported [denoted as $\mathcal{F}(\phi)$]. We maintain a mapping record ($\mathcal{R}$) between physical server fingerprints and physical server IDs. When a fingerprint that does not map to any server is returned, a new server in the cluster is discovered. A new ID will be assigned to the server, and a new entry in $\mathcal{R}$ will be established to record the mapping between the server fingerprint and the assigned server ID. The trace of server IDs associated with returned server fingerprints after each function invocation is recorded and will later be used for analysis. As an example, consider a function that is invoked 6 times. If the first 3 executions are handled by one server in the cluster, and the last 3 executions are on another server, the resulting sequence will be $(1, 1, 1, 2, 2, 2)$.

Shown in Fig. 2, the submission of function invocation consists of 3 phases (a detailed pseudocode description can be found in Appendix A):

Phase 1. In this step, the attacker constructs a serverless function (ϕ_0) and continuously triggers it for a specific length of time (e.g., 10 minutes). Then, the attacker will stop all activities and wait for a sufficiently long amount of time (e.g., 1 hour) so that all function hosts are cleared/terminated by the cloud system.

Phase 2. In this step, the attacker constructs a second serverless function ϕ_1, which is a copy of the previous serverless function ϕ_0. Then, the attacker will invoke the two functions alternately for a specific length of time (e.g., 10 minutes), and then wait for all function hosts to be terminated.

Phase 3. In this step, the attacker considers possible locality optimization that the cloud provider may use and tests these options. Possible target locality optimization techniques include data locality [23], package locality [17], etc. The attacker will construct a list of serverless functions $\{\phi_0', \phi_0'', ...\phi_0^{(n)}\}$ that is functionally identical with ϕ_0 but vary in locality-related attributes. For example, when targeting package locality, an attacker will construct variants of ϕ_0 that request different packages to be installed.

Analysis. Features of the scheduling algorithm can be revealed by analyzing and comparing different parts of the collected trace. In Phase 1, the attacker invokes the same function with a relatively high frequency for a sufficiently long pre-defined length of time. The placement sequence in the collected trace can reveal the following features (F_i) of the scheduler:

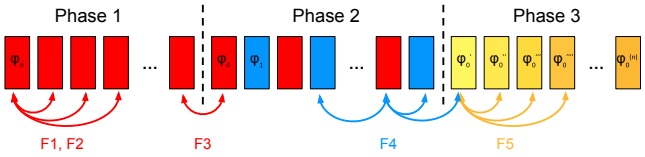


Fig. 2: The scheduler fingerprinting process.

F_1. *Invocation Locality*: If these functions are consistently executed on the same machine over a continuous period, it indicates that the scheduler considers reusing already placed function hosts to avoid cold-start delays.

F_2. *Auto-Scaling*: If F_1 is observed and new servers are discovered continuously, it indicates that there are auto-scaling mechanisms embedded in the scheduling algorithm.

In Phase 2, after all function hosts are terminated, the attacker re-invokes the same function in Phase 1, together with another function with the same functionality but disguised as a different function. This enables the attacker to compare placement results between (1) invocations to the same function that are separated in time, and (2) invocations to different functions. The attacker will be able to answer the following questions:

F_3. *Cold-Start Locations*: By comparing the placement results of the first function in Phase 1 and Phase 2, the attacker will be able to obtain information regarding the scheduler's behavior when rescheduling a function instance that has been terminated before. The attacker will be able to answer the following question: for the same function, is the cold-start location always the same?

F_4. *Account Locality*: By comparing placement results of different functions, the attacker will be able to infer if user locality is considered by the scheduler and answer this question: do the schedulers tend to place function hosts from the same user on the same set of physical servers? This can potentially infect the construction of an attack strategy. This also needs to be cross-checked with placement results from Phase 3.

The goal of Phase 3 is to explore whether there are other locality optimizations that rely on specific configurations of serverless functions (e.g., package required) and are employed by the scheduler. By comparing the placement results of different functions, the attacker will be able to identify:

F_5. *Configuration-Based Locality*: By invoking functions with different configuration parameters (packages, required data, etc.), the attacker will be able to identify certain configuration-based locality optimizations for further exploitation. For example, the scheduler considers package locality and places function hosts with similar package specifications on the same machine.

IV. Fingerprinting Evaluation

A. Simulator Details

We conduct our fingerprinting experiments in a simulator written in C++. The usage of this simulator allows us to launch large-scale Monte Carlo experiments. The simulator consists of ∼2000 Line of Code and simulates the functionality of a serverless system, with a focus on the scheduler. There are 4 different components in the simulator:

- `Node`, which maintains node resource information;
- `Function`, which consists of attribute information of user-submitted functions;
- `Scheduler`, which models scheduling algorithms of different serverless platforms;
- `User`, which models user function invocation behaviors.

We implement these components as plug-and-play modules that interact with each other using uniform API interfaces, as shown in Fig. 3.

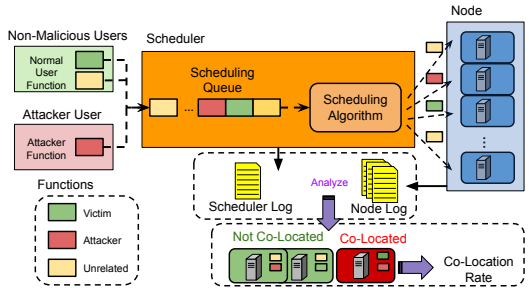


Fig. 3: Architecture of our serverless system simulator.

In our design, the `User` modules are connected to the `Scheduler` module via a scheduling queue. `Users` will push `Function` objects to the scheduling queue, and `Scheduler` will consume the queue in a first-come-first-serve order. The scheduler will dispatch these function invocations to servers (`Node`) in the cluster using predefined scheduling algorithms. `Node` modules are responsible for maintaining server resource information and running function information. When a `Node` receives a `Function`, it will register the `Function` (if it is a cold-start invocation) and update the `Function`'s time-to-live (TTL) information. All `Functions` with TTL$=0$ will be eliminated from the system. `Scheduler` and `Node` modules will both produce log files for analysis.

At the beginning of the simulation, a list of users will be randomly generated, and an attacker will be initiated to target a specific user. Both non-malicious users and attackers will generate `Function` instances when they are initialized. The simulation will go through a pre-defined number of rounds of `Function` submissions by different users, and the log files will be collected for further analysis. We have implemented different scheduling algorithms in different `Scheduler` modules.

B. Selected Scheduling Algorithms

We selected to implement the following schedulers in both the simulator and the Dask-based serverless platform:

1. OpenWhisk [31]: OpenWhisk is an open-source serverless system. In this work, we select to use the `ShardingContainerPoolBalancer` algorithm provided in their repository [32] as our target, as it involves an interesting invocation locality optimization. For each function to be scheduled, the scheduling algorithm selects a list of servers based on the hash value of the function. Every time the function is scheduled, the scheduler will examine the availability of servers on the list in order. This algorithm tries to balance cold-start reduction and load-balancing.
2. Helper, which mimics the critical load-balancing behavior of Google Cloud Run [4] described in [16]. For each function to be scheduled, the scheduler selects a server as the base server first. As the load increases, the scheduler creates function host instances on other servers to perform auto-scaling.
3. Package-Aware Scheduler (PASch) [17], which is a serverless algorithm that utilizes package locality to improve the performance of serverless platforms. The scheduler maintains a consistent hash ring [33] during operation and maps serverless functions to servers based on the largest package that is required to increase the package cache hit rate in the system and at the same time balance the load of each server in the system.
4. Random, which is a scheduler that randomly dispatches functions to servers in the system.

We utilize OpenWhisk [31] as an example of cold-start-optimized schedulers. Also, as it is one of the most popular open-source serverless platforms, our attack results will be representative. We selected Helper [16] since it is reported to be part of the scheduling algorithm of Google Cloud Run, and we also observe similar behaviors in Microsoft Azure (see Section VII). PASch is selected to represent schedulers with F_5 (configuration-based locality) optimizations, and the attack methods on PASch can be easily expanded to target other schedulers with similar optimization features. Finally, Random is selected as the baseline and helps demonstrate if increased randomness can help defend against co-location attacks, as shown by [28] in serverful clouds.

1) Results: We conduct this part of the experiments in simulation, in which we strictly follow the aforementioned submission strategies. The 3 phases of function invocations include:

Phase1. Invoking one function repeatedly;
Phase2. Invoking two identical functions repeatedly;
Phase3. Invoking multiple functions with identical functionality and different package requirements.

Each scheduler processes a total of 8000 invocations per phase. The collected server ID traces are shown in Fig. 4. The discovered features of each scheduler are provided as follows:

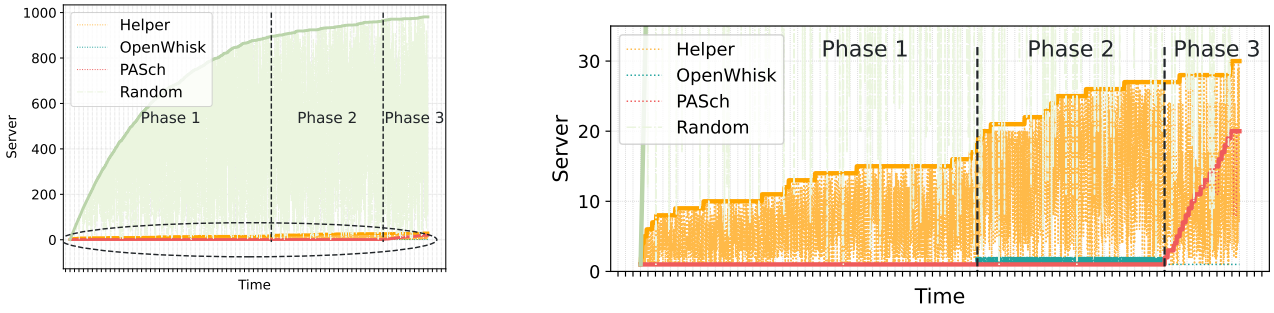
Random Scheduler. It can be seen from Fig. 4 that our function invocation sequence covers more servers than other types of schedulers, with 981 out of 1000 servers being covered in the cluster in our simulation. By more careful examination of the placement data, one can also find that the execution locations of serverless functions present no locality. These observations from the fingerprinting results match the features of the Random scheduler.

Helper Scheduler. By examining the placement trace in Phase 1, we can clearly observe that the scheduler has optimizations to reduce cold-starts, as functions are invoked repeatedly on the same set of servers (F_1). As the number of function invocations increases, the scheduler starts to scale out function hosts by creating more instances to be placed on other servers (F_2). In Phase 2, we can obtain the same observation as we start to involve a second function. By comparing the placement results of the overlapped function in Phase 1 and Phase 2, we find that if all hosts of a function are terminated, when re-launching these function hosts, the target scheduler will still choose the same set of servers, i.e., the Helper scheduler presents cold-start locality (F_3). Also, we notice that the two different functions in Phase 2 are placed in different locations, indicating the Helper function does not confine the host placement to a small set of servers even though they are from the same user (F_4). However, the behaviors of the Helper scheduler in Phase 3 are similar to Phase 1 & 2, indicating that when making scheduling decisions, the scheduler does not consider package locality (F_5).

OpenWhisk Scheduler. In Phase 1, when there is only one function involved, we can see that the function executions consistently stay on one machine, indicating that there is optimization to reduce cold-start (F_1), yet no auto-scaling is involved (F_2). In Phase 2, as we involve a second function, we can see that the number of covered servers increases by 1, indicating that function hosts from the same user are not bonded together (F_4), and the location is the same when a function host experiences termination and cold-start (F_3). In Phase 3, however, we do not notice the scheduling results change, indicating that the OpenWhisk scheduler does not use package information when making scheduling decisions.

PASch Scheduler. Similar to the OpenWhisk scheduler, in Phase 1, traces collected from PASch show instance locality (F_1). However, since the placement consistently stays at the same machine, we can conclude that it does not perform auto-scaling based on the function invocation load (F_2). In Phase 2, when a second function is involved, there is still only one server covered. This could potentially be due to (1) user locality, i.e., the scheduler places all functions from a user to a single machine; or (2) the identical configurations of the two functions result in the same placement decision. By examining Phase 3 of the trace, we can exclude possibility (1) since there are different function hosts in this phase. Besides, by comparing scheduling results in Phase 3, an attacker would be able to notice that the scheduler places function hosts with the same package requirements at the same node, demonstrating the existence of package locality-related policies in the scheduling algorithm.

A summary of the exposed exploitable features is provided in Table I. The above analysis results show that an attacker is able to decipher scheduler features based on the



(a) Full collected placement traces.

(b) Zoom-in view of the circled area in (a).

Fig. 4: Collected traces.

TABLE I: A summary of the reverse-engineered features.

Scheduler	F_1	F_2	F_3	F_4	F_5
Random	×	×	×	×	×
Helper	✓	✓	✓	×	×
OpenWhisk	✓	×	✓	×	×
PASch	✓	×	✓	×	✓(package)

collected traces. The discovered features correctly match our example scheduler implementation, indicating that an attacker can obtain scheduler information simply by observing and comparing different parts of a collected trace. In reality, an attacker can flexibly perform logical reasoning based on the information contained in the collected traces and identify exploitable scheduler features.

V. Recipe for Generating Co-Location Attacks

A. Attack Construction Guide

Based on the revealed scheduler features, an attacker can construct the attack method correspondingly and improve the efficiency of the co-location attack. Features identified by the previous scheduler fingerprinting step center around: (1) Policies that render the placement decisions more deterministic (locality); and (2) Policies that render function hosts to spread in the system (auto-scaling). These scheduling policies are related to the placement of function hosts and can thus be exploited by attackers to improve the attack success rate.

Our proposed attack consists of the following steps:

Step 1. We first examine F_4 (Account Locality) from Section III. Schedulers employing this scheduling policy tend to place function hosts from the same account in a relatively confined set of servers, which limits the number of physical servers a single account can cover. This scheduling feature has been observed in AWS [34] and Google Cloud Run [16]. As a result, an attacker will need to employ multiple accounts to increase the chance of achieving co-location.

Step 2. In this step, we check if F_1 (Invocation Locality) is employed and construct the attack accordingly. Warm-start-related optimizations are usually employed by cloud providers to avoid the costly cold-start process. Consequently, the placement of a single function will be limited to a small set of physical servers. Therefore, if feature F_1 is identified, to improve attack efficiency, an attacker will need to construct multiple attack serverless functions.

Step 3. In this step, we explore how to exploit identified locality optimizations F_5 (Configuration-Based Locality) from Section III. When this scheduling policy is revealed, an attacker can:

1. Focus on one specific choice of the corresponding scheduling parameter, and accurately target the victim's function. This occurs when the choice of scheduling parameter can be known by an attacker. For example, in a package-aware scheduler [17], when a victim hosts a popular serverless service like data analysis, the used packages are usually predictable. This tends to be the most efficient way of launching attacks, as shown by our results in Section VI.
2. Vary configuration parameters to enable the attack functions to spread in the cluster. For example, for package locality-aware schedulers [17], an attacker can construct serverless functions with different package dependencies to fully exploit the feature and force the corresponding function hosts to occupy different physical machines with a higher probability.

Step 4. In Step 4, we observe whether F_2 (Auto-Scaling) is involved. Auto-scaling tends to happen when a serverless function is in high demand. The scheduler will assign more function hosts in the system, causing the function hosts to spread across the cloud system. An attacker can exploit this feature to increase the physical server coverage of a single function by creating repeated invocation bursts, which is similar to the attack method used in [16].

B. Constructed Attacks for Selected Scheduling Algorithms

Based on the discovered scheduler features shown in Table I, we construct the following attack methods to tackle targeted schedulers:

M1. Targeting OpenWhisk, we construct an attack method that spawns multiple function invocations every time an attacker targets a victim. These serverless functions are

of different names, and in reality, these functions can create hash collisions to enable co-location.

M2. Targeting the auto-scaling feature (Helper), in this attack strategy, attackers stress one of their own serverless functions by creating a burst of invocations, aiming to exploit the auto-scaling mechanism to achieve co-location.

M3-1 Targeting PASch, the attacker exploits the locality-based scheduling algorithm by creating functions with the same package dependencies as the victim function, increasing the likelihood of co-location with the victim function hosts.

M3-2 The attacker targets the same feature as M3 but varies the choices to increase coverage.

We categorize M1, M2, and M3-2 as **scatter-based attacks**, since these attacks achieve co-location by scattering function hosts to increase the coverage, while M3-1 only targets the correct physical server. It is worth noting that M3-2 and M1 are different, although both involve creating multiple functions and deploying them. In M3-2, the attacker has identified the exploitable locality feature and can hence use this feature to scatter instances, while in M1, the attacker simply creates multiple copies of a single function, and all parameters remain the same.

VI. Attack Evaluation

A. Serverless System Implementation

Besides conducting experiments in the aforementioned simulator, we also deploy serverless cloud systems with different schedulers to a cluster. Due to resource and cost constraints, we choose to evaluate on a 50-node cluster on CloudLab [35] consisting of 50 `r320` instances. Each node has one 8-core E5-2450 CPU and 16GB of Memory. We use one node as the central scheduler, and the others as workers.

The implementation of the serverless platform follows Abdi *et al.* [23]. We implement the scheduling and function invocation functionalities entirely in the Dask Distributed framework [36], which is a Python-based distributed computing framework. Our implementation involves approximately 500 lines of changes across two files that together comprise around 16,000 lines of Python code. The changes are distributed as follows: modifications to `scheduler.py` for the scheduling algorithm implementation, and updates to `worker.py` to adapt the system for serverless operations. Functions are executed through the Azure Functions Host runtime [37]. Upon cold-start, the server that is scheduled to execute the function will pull the function code from a remote destination and invoke the Azure Functions Host runtime. We use function benchmarks from benchmark sets such as FunctionBench [38]. It is worth noting that this research focuses completely on the scheduler side; hence, we can safely omit environment configuration and execution details during the function execution process. We will use the log files produced by the scheduler node and worker nodes to obtain function placement information and calculate the required metrics. The four selected algorithms are implemented in this system.

B. Metrics

Simulation Experiments. The simulator enables us to flexibly change experiment parameters, e.g., cluster size, and conduct a large number of experiments. We launch 50 short-duration experiments in a 1000-node simulated cluster and use the percentage of experiments where the attacker successfully co-locates with the victim as our metric.

Cloud Experiments. Unlike in the traditional cloud instance placement situation, where instances are relatively fixed [12], [28], the flexible and scalable nature of serverless scheduling complicates the co-location measurements. Simply counting how many of the invocations to victim functions are co-located with attacker functions does not take the constant termination and start of instances into account. For example, an attacker's function invocation causes the corresponding function host to be placed together with a victim, resulting in subsequent invocations to this victim function being risky. The success of these subsequent invocations should be attributed to the specific function invocation that leads to co-location and shouldn't be counted multiple times.

We propose to use the following two metrics:

- Attacker Efficiency (**AE**), i.e., the percentage of attacker function instance placement that results in co-location with victim instances;
- Placement Accuracy (**PA**), i.e., the percentage of victim instances that co-locate with attack instances.

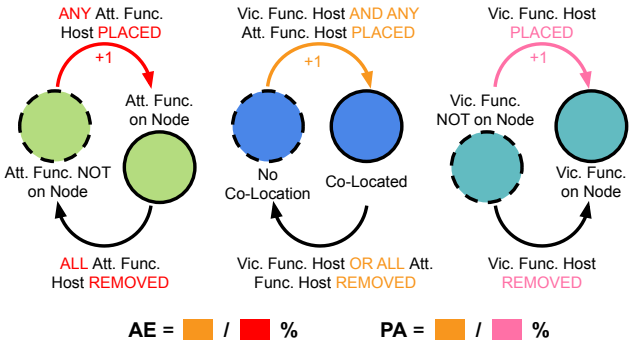


Fig. 5: Diagrams of the AE and PA calculation process. Each sub-diagram is a state machine, and we show the involved states and state transition conditions. To calculate AE and PA, we scan the collected log files and count the number of state transitions marked in red, orange, and pink. These counts represent the number of attack instance placements, the number of times co-location happens, and the number of victim instance placements, respectively.

The calculation process of AE and PA is shown in Fig. 5. By counting state transitions marked as colored edges in Fig. 5, we can obtain the number of function host placement activities that lead to co-location and the total number of attacker and victim placement activities.

C. Results

We perform the evaluation of our attack in both a simulation and a cluster. In the simulation experiments, the metric of focus is the short-term success rate, since it is easier to launch massive experiments in simulation. In each experiment, for a relatively short duration, we specify a victim user, and the attacker launches attack functions. We then collect the generated log files and calculate the percentage of successful experiments. In cluster experiments, we focus on a more realistic attack process with a longer duration and obtain the AE and PA data to measure the attack efficiency.

Simulation. The results we obtained from simulation are shown in Fig. 6.

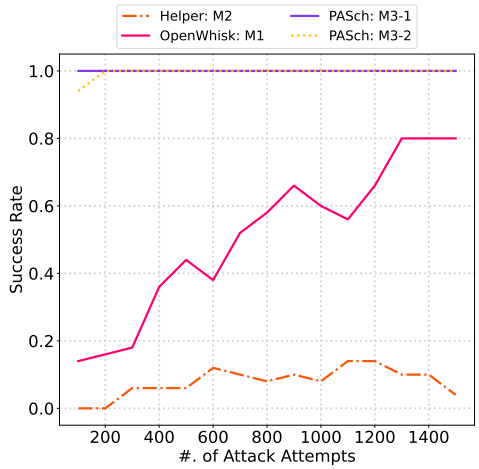


Fig. 6: Short-term attack success rate obtained from the simulator.

From Fig. 6, we can see that in both PASch and OpenWhisk, the attackers are able to achieve relatively high co-location performance as they increase the number of attack functions. Also, it can be seen from the results of M3-2 that even if attackers do not possess full knowledge of the victim package dependencies, they can still achieve co-location by scattering their function instances. Below are some additional findings.

Finding 1: Exploiting F_5 (Configuration-based Locality) scheduling feature is efficient.

By observing the two curves in Fig. 6 regarding PASch (the scheduler that considers package locality), we can see that the attacker is able to reach almost 100% success rate. This indicates that as long as the locality feature of the scheduler is identified by the attacker, the attacker is able to easily achieve a high co-location success rate. If the user-submitted information is known by the attacker, e.g., the user utilizes certain packages, an attacker can precisely target the victim user's function host in the cluster by providing identical package requirements, which is the M3-1 attack strategy we provided in Section V. Otherwise, the attacker can construct multiple functions with various settings to ensure coverage, which is the M3-2 strategy. Either way, the attacker is capable of achieving a high success rate.

Finding 2: Exploiting F_2 (auto-scaling features) is not efficient.

In Fig. 6, as shown by the curve corresponding to Helper Scheduler (which has an auto-scaling feature), increasing the number of function invocations per attack attempt only slightly increases the success rate. This is due to the fact that not all function invocations can lead to the creation of a new function host. A more detailed theoretical analysis can be found in Appendix B.

Cluster Experiments. We also conduct experiments in a 50-node CloudLab [35] cluster. All functions are invoked continuously during the experiments, with function invocation being submitted every two seconds. It takes 3-6 hours to run one experiment. We select to report AE and PA results of these experiments, which reflect the efficiency of the attacker's attempts and the effects of attacks on the victim function hosts. The results on the CloudLab cluster [35] are shown in Fig. 7. We can observe that the PASch scheduler and the Random scheduler have the highest PA values, indicating that most users' placed instances are co-located with a victim instance during its lifetime. By looking at the two curves corresponding to the Helper Scheduler (Helper: M1 vs Helper: M2), we can conclude that compared to using one attack function and repeatedly invoking to trigger the auto-scaling policies of the scheduler, using multiple functions results in higher PA values, i.e., more user function hosts are co-located with attack function hosts.

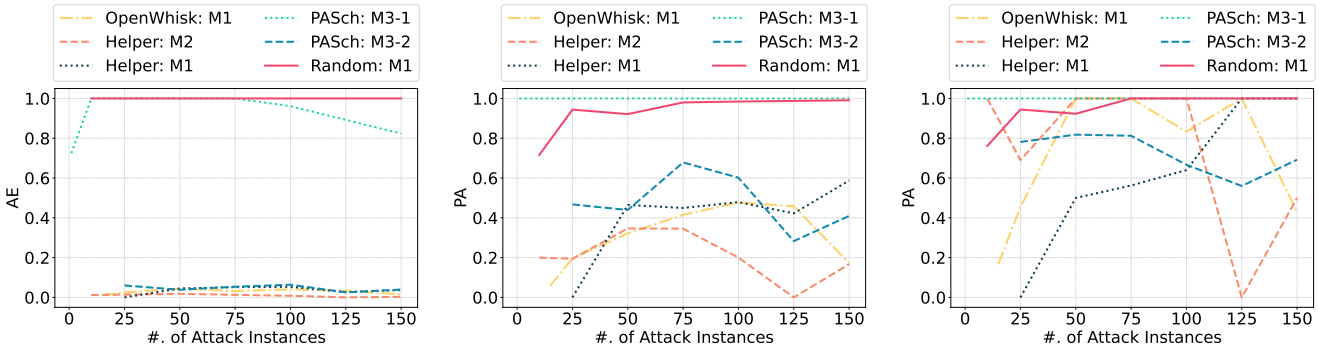
It is worth noting that, as we only specify a single victim and only repeat our experiments a few times, unlike the simulation experiments that are repeated hundreds of times, we can see random fluctuations in the results, which is close to the nature of this task in the real world. However, we are still able to see that with a relatively small number of attack functions, the attacker can achieve co-location with $\sim 50\%$ victim function hosts in most schedulers.

We also observe that:

Finding 3: Attack efficiency of scatter-based attacks (i.e. M1, M2, M3-2) is relatively low.

From Fig. 7 (a), we can see that most of the AE values of scatter-based attacks are below 0.1 (OpenWhisk: M1, Helper: M1, and PASch: M3-2), meaning only a small portion of those placed attack function hosts achieves co-location with victim function hosts. The only exception is the Random scheduler, where the M1 attack strategy achieves an AE value of 1. An explanation of this is provided in Appendix B. Overall, our observations indicate that attacks would be more efficient if locality-optimization features of schedulers could be identified and the attacker could obtain related information about the victim.

The results in both the simulation and a cluster show that our constructed attack can enable attackers to effectively achieve



(a) AE results. (b) PA results. (c) Maximum PA results.

Fig. 7: AE and PA results collected from the cloudlab cluster.

co-location with victim function hosts. We also show that the most important thing an attacker can do is to identify scheduling locality features, as exploiting these features can greatly increase the success rate, as well as improve attack efficiency.

Transferability. We further examine whether the attack methods are transferable. We apply attack methods in Section V to mismatched types of schedulers. The experiments are conducted in the aforementioned simulator. The results are presented in Fig. 8.

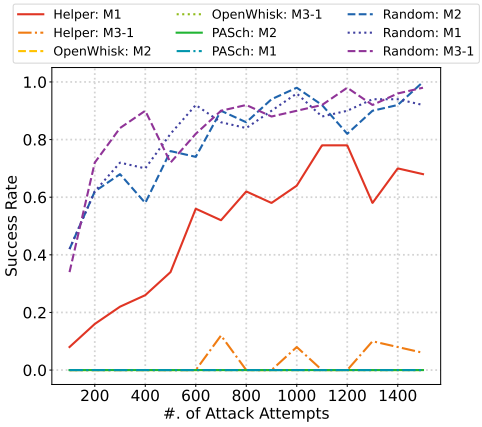


Fig. 8: Attack transferability results.

Fig. 8 shows that 4 out of the 9 curves (OpenWhisk: M2, OpenWhisk: M3-1, PASch: M1, and PASch: M2) consistently remain at 0, indicating that the corresponding attack methods do not apply to targeted schedulers, even if he/she prepares a large number of functions to be deployed. This demonstrates the necessity of attackers to find the most appropriate attack method for a targeted scheduler. We have the following findings:

Finding 4: Full randomness in scheduling is unsafe.

We can see from Fig. 8 that all 3 attack methods (M1, M2, M3-1) on the Random Scheduler achieve similar performance, all better than other attack results obtained from mismatched schedulers and attack methods. This shows that randomness in the scheduling process can actually be used by an attacker to increase attack coverage.

Finding 5: Using multiple functions to target schedulers equipped with F_2 (auto-scaling) feature is effective.

It can be seen from Fig. 8 that when facing schedulers with auto-scaling features (the Helper Scheduler), launching multiple different functions can enable attackers to reach a relatively high co-location success rate. Compared to results obtained from the Helper Scheduler in Fig. 6, we can conclude that exploiting F_2 (auto-scaling) feature with multiple functions is the more ideal solution.

Finding 6: Locality-optimized schedulers are relatively safe when incorrect attack methods are used.

The PASch [17]-related curves in Fig. 8 show that it is hard to achieve co-location if attackers do not exploit the locality features. The success rates of both curves (PASch: M1 and PASch: M2) remain constant at 0. This is due to the fact that this type of scheduler tends to restrict the placement of functions to specific locations, and without targeting the correct scheduling features, the attacker will not be able to identify the victim's location.

Results from our simulation show that generally, without targeting correct scheduling features, an attacker will not be able to achieve co-location even with massive function invocations. The only exception is the scheduler with auto-scaling features, where it is more efficient to deploy multiple functions instead of simply triggering the auto-scaling policy to scale out. The results show that it is therefore important to identify exploitable features of schedulers, i.e., scheduler fingerprinting.

VII. Case Study: Microsoft Azure Functions

As a case study, we also apply our fingerprinting method to Microsoft Azure Functions [5], which is one of the most

prevalent serverless cloud platforms, and report our discovered scheduler features. Then we construct an attack accordingly and perform co-location attack experiments on Azure Functions. We create accounts in Azure Functions and utilize the Consumption Plan [39] hosting model. In the Azure Functions Consumption Plan, users are billed based on cumulative memory resource usage, number of executions, and execution time. Users can only submit code to Azure, and are not allowed to deploy container-based applications in the Consumption Plan. This represents the least expensive and possibly the most common usage pattern of serverless platforms. All experiments are conducted using Consumption Plan [39] in the `westus` region in Microsoft Azure.

A. Scheduler Fingerprinting

In Azure Functions, users can construct applications that contain multiple functions. In Phase 1 and Phase 2, we build function applications that only contain 1 function. In Phase 3, we also include function applications with multiple functions. We utilize the timestamp counter (TSC)-based method provided in [16] as our server fingerprinting method. This method reads the value of TSC registers inside CPUs, parses the base frequency of CPUs, and calculates the boot time as the watermark of a server. We construct an Azure Consumption Plan Function App by directly submitting the code directory, without using containers. We wait for at least 10 minutes after the experiment of each phase to allow function hosts to be terminated. The fingerprinting result sequences are shown in Fig. 9.

We have the following observations.

Observation 1: The scheduling algorithm of Azure Functions considers invocation locality (F_1) and auto-scales if there is a high volume of incoming function invocations (F_2).

By examining the trace from Phase 1, we can see that the execution locations are relatively restricted, and these functions are executed at the same set of locations repeatedly. Also, in the experiment, as we provide a large number of function invocations per second ($\sim$ 100/s), we observe that the cloud scheduler gradually auto-scales function hosts to accommodate the large number of incoming function invocations per second. However, we are only able to increase the number of occupied physical servers to at most 4.

Observation 2: In Azure Functions, function host cold-start locations are different (F_3), and we do not observe account locality (F_4).

Compared to Phase 1, we can see that in Phase 2, the Azure Functions scheduler selects a completely different set of physical servers to execute function hosts, indicating that the function start locations are irrelevant to previous placement results. Besides, in Phase 2, the two constructed functions each cover 4 physical machines, showing that the scheduler of

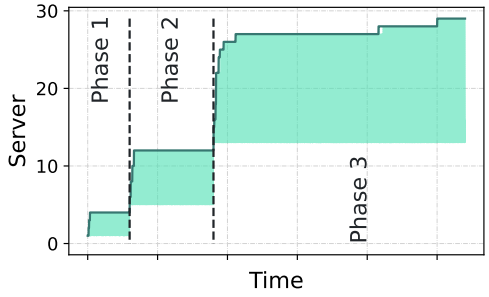


Fig. 9: Fingerprinting traces collected from Azure Functions.

Azure Functions does not have the tendency to place different functions from the same user on the same physical machines.

Observation 3: Azure Functions places function hosts belonging to the same application on the same physical machine (F_5).

In Phase 3, we invoke multiple functions belonging to different Azure Functions applications. Some of these functions belong to the same function application. We observe that Azure Functions tend to place these functions on identical physical machines, even after auto-scaling. Meanwhile, functions belonging to different function applications are still placed on different machines. We were unable to observe other forms of F_5 locality on Azure Functions.

B. Co-Location Attack

Based on the revealed scheduler features, we construct the following attack strategy:

Attack:
- Use one attack account;
- Construct multiple attack functions belonging to different function applications;
- Create bursts of function invocations to trigger auto-scaling policies.

We create two accounts, one as the victim and the other as the attacker. The victim user runs one function in Azure Functions, which is our target. The attacker creates 64 different function applications, each containing one function. Both users utilize the Azure Consumption Plan, and the functions created by both users are the fingerprinting function, so that we can verify co-location. We deploy these functions, invoke them, and analyze the collected traces to determine if co-location is achieved.

We launch the experiment 5 times. We observe that the attacker achieves co-location with at least one of the victim hosts in every experiment. Fig. 10 shows the results of the five co-location attempts. The 64 attack functions manage to cover 230 machines on average, and on average 1.8 victim function hosts are under attack. All the experiments, including fingerprinting and subsequent attack attempts, cost less than 25 USD combined from the attacker's side, indicating that it

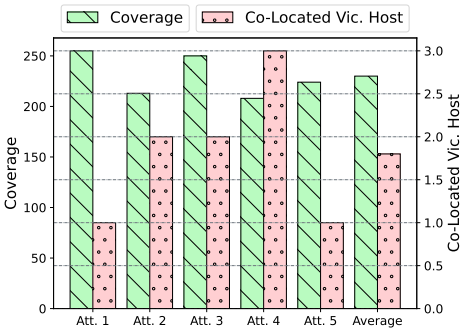


Fig. 10: Co-location attack results on Azure.

is not costly to achieve co-location with another user in the Azure Functions Consumption Plan.

VIII. Mitigation

As demonstrated in our previous attacks, the vulnerability stems from the lack of enforced server isolation between users. Current scheduling algorithms enable attackers to easily reach servers hosting other users' instances, either through instance spreading or precise targeting facilitated by scheduler locality optimizations. By intentionally minimizing overlap between servers occupied by different users in the scheduler's design, the risk of co-location could be significantly reduced.

A. Double-Dip Scheduler

We propose *Double-Dip*, a scheduling algorithm that aims to provide "soft" server isolation for different users. The goal of *Double-Dip* is to minimize the overlap of physical servers between different users. *Double-Dip* works as follows:

1) When a function f_u, belonging to user u, needs to be scheduled, the algorithm first checks if there exists a physical server h such that:
   - u is already active on h, i.e. $u \in C(h)$, where $C(h)$ is the set of users served by h, and
   - h has sufficient resources to accommodate f_u (i.e., $r(f) \leq R(h)$, where $r(f)$ is the resource requirement of f_u, and $R(h)$ is the available resources on h).

   If such a server exists, assign f_u to it. (This is where the name *Double-Dip* comes from.)
2) If no such host exists, the algorithm selects a physical server h^* that:
   - h^* has the least variety of users (i.e., $h^* = \underset{h \in H}{\operatorname{argmin}} |C(h)|$), and
   - h^* has sufficient resources to accommodate f_u ($r(f) \leq R(h^*)$).

The pseudocode of the *Double-Dip* scheduling algorithm is provided as in Fig. 1.

Integrating *Double-Dip* into commercial serverless clouds requires only minimal modifications to existing scheduling pipelines, as it operates entirely at the scheduling layer and leaves execution mechanisms unchanged. Major platforms such as AWS Lambda, Azure Functions, and Google Cloud Run already maintain per-function metadata; *Double-Dip* simply leverages this to keep a lightweight active-user map per host and apply a user-aware rule during placement, and remains fully compatible with modern elastic environments. The scheduling overhead is small, consisting of constant-time user checks and a single pass to identify the least-user-variety host—operations already common in production load balancers. Our cost experiments further show that this logic scales well: even as we increase users and requests by an order of magnitude, warm-start behavior remains close to Helper and above OpenWhisk, with overhead saturating once workers have seen all function types.

Algorithm 1 *Double-Dip* Algorithm.

Require: $nodeList$: List of nodes (hosts), $appToRun$: Application (function) to be scheduled
Ensure: Application is scheduled on a node with minimal co-location rate increase.

for all $node \in nodeList$ **do**
  if checkResource($node, appToRun$) **and** checkActiveApps($node, appToRun$) **then**
    $node$.run($appToRun$) ▷ Run the app on the same host with active apps
    return 0
  end if
end for
▷ No active apps found
while true **do**
  $node \leftarrow$ FindLeastUserVarietyHost($nodeList$) ▷ See Algorithm 3
  if checkResource($node, appToRun$.resource) **then**
    break
  end if
end while
$node$.run($appToRun$) ▷ Run app on the least user variety host
return 0

B. Evaluation

We conducted our experiments in a simulated environment to evaluate the performance of the *Double-Dip* scheduling algorithm in a serverless cloud setting, comparing it with Helper. The simulation setup consisted of 100 nodes, each with the capacity to handle up to 1024 functions. The environment featured one victim function and a variable number of attackers, ranging from 5 to 50. Each attacker was programmed to perform 10 attack attempts during the simulation. To ensure statistical significance and reduce variability, each result was obtained by averaging data from 1000 independent simulation runs.

The results are visualized in Fig. 11 with two lines, one representing our proposed scheduler and the other representing the Helper scheduler. The x-axis denotes the number of attackers, while the y-axis indicates the co-location success rate, where a lower success rate signifies better performance. At 5, 10, 20, 30, and 50 attacker accounts, our proposed scheduler achieved co-location success rates of 0.056, 0.118, 0.203, 0.260, and 0.306, respectively. In contrast, the Helper

scheduler showed success rates of 0.495, 0.781, and 1.000 for attacker counts of 5, 10, and beyond.

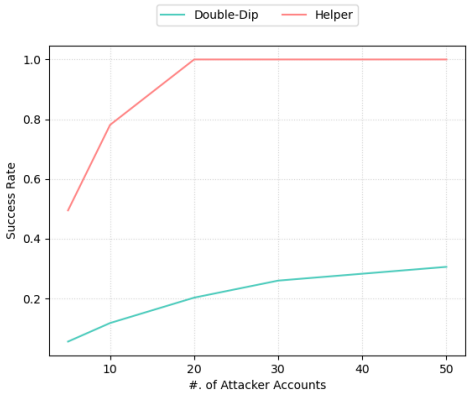


Fig. 11: Co-location attack results on *Double-Dip* and Helper

These results demonstrate that *Double-Dip* lowers the co-location rate an attacker can achieve even with multiple accounts, especially as the number of attackers increases. While the Helper scheduler consistently reaches a co-location success rate of 1.0 (indicating complete co-location success by attackers) for scenarios with 20 or more attacker accounts, the proposed scheduler maintains a much lower success rate, offering better resistance against co-location attacks.

C. Cost Evaluation

To quantify the overhead of *Double-Dip*, we measure the warm-start behavior of each scheduler under varying levels of multi-tenancy. We compare three schedulers: the OpenWhisk scheduler, the Helper scheduler, and our *Double-Dip* scheduler. Similar to attack evaluation, we conduct experiments in a 50-node CloudLab [35] cluster. Following the co-location experiments, we vary the number of users (5, 10, 20, 30, 50) and the number of total requests (500, 1000, 2000, 3000, 5000). All tasks are submitted concurrently, and warm-start ratios are computed from logs by tracking, for each worker, whether it has previously observed the corresponding function type.

Fig. 12 reports the warm-start ratios for all configurations. Across all settings, we observe that Helper consistently has the highest warm-start ratio, followed by *Double-Dip*. This behavior is expected: Helper aggressively reuses recently active nodes, maximizing warm-starts but providing no user-level isolation. *Double-Dip* lies in between, yet it still retains most of the warm-start advantages of Helper while offering substantially stronger isolation guarantees. The lower warm-start ratios in the smallest configuration reflect the system's rapid scale-up, where new workers are activated quickly; this behavior is expected in serverless platforms and further highlights that *Double-Dip* scales smoothly as the cluster expands.

We also observe that as the number of users and the total number of function invocations increase, the differences among the schedulers diminish; there is a 0.39% difference between Helper and *Double-Dip* at the 50 users, 5000 invocations configuration, indicating that *Double-Dip* achieves lower relative overhead under high-intensity cloud-use scenarios.

Overall, *Double-Dip* achieves warm-start performance that is close to Helper, while providing substantially stronger user isolation. These results indicate that *Double-Dip* introduces only a modest performance cost.

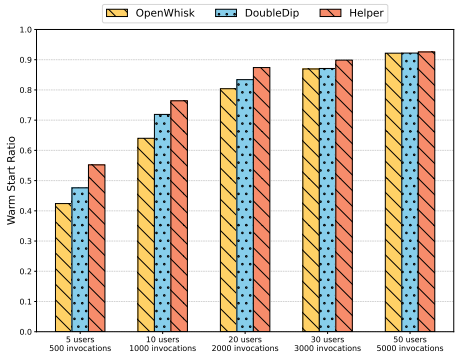


Fig. 12: Warm-start ratios.

IX. Discussion

A. Security Implications in Scheduler Design

Cloud scheduler is an important part of cloud systems. They are crucial to the performance of the whole system, and at the same time, the first barrier to cloud attackers. A good scheduler should optimize performance by fully exploiting function locality features while simultaneously ensuring unpredictability in scheduling results to enhance security.

Our results help provide some insights into the security aspect of serverless scheduler design. On the one hand, we show that pure random schedulers are susceptible to co-location attacks, and attackers are guaranteed to achieve a high success rate as long as they prepare a sufficient number of attack functions to deploy. On the other hand, optimization policies that result in more deterministic instance placements, e.g., package-aware scheduling, can also be exploited by attackers to achieve co-location with victims. These results inspire a hybrid approach, where the scheduler still preserves the original optimization technology, which leads to determinism in instance placement, but at the same time adds randomness to the system to avoid the scheduling results being easily identified. The added randomness should not be excessive as well, otherwise the attacker would still be able to scatter attack function hosts and achieve co-location with victim hosts. It is important for the algorithm designer to strike the balance point and ensure a performant and secure system.

Besides, it is also important to provide user isolation. The co-location vulnerability in this paper largely originates from the fact that different users share the same scheduling pool, i.e., the whole cluster. If different users' scheduling pools are isolated or only partially overlapped, the difficulty of achieving co-location would significantly increase. This should

Vendor	FaaS	Dedicated IaaS (Lower Bound)
AWS [45]	$17.18/mo	$323.28/mo
Azure [46]	$33.3/mo	$444.03/mo

TABLE II: Cost comparison of different service models.

not influence the performance of users' functions, since each function only scales out to a small number of physical servers even under a large burst of invocation requests, as shown by the results from Azure Functions.

B. Other Potential defense

Although the deployment of the *Double-Dip* algorithm proposed in this paper can significantly reduce the co-location threat of serverless clouds, it does not provide a 100% guarantee that the attacker and victim will not be co-located. Given the nature of serverless system design, i.e., it tries to abstract away cloud management details from users and increase server utilization rate from the vendor side, the only solution to guarantee instance isolation is to turn to traditional Infrastructure-as-a-Service (IaaS) clouds and reserve dedicated servers to use. However, this will significantly increase the cost. Below, we outline a ballpark estimation to compare the cost of using dedicated servers to provide an isolation guarantee and *Double-Dip*.

Since the changes we propose in the *Double-Dip* algorithm are oblivious to users, we assume the customers are still charged the same price as before. We calculate using the following settings:

1) 30 Million requests per year (Coca-Cola's vending machine service [40]);
2) Each invocation requires one CPU core and 200ms to handle;
3) Each function host is allocated 2GB of memory and 1GB of storage;

In Table II, we provide a comparison of AWS and Azure. For AWS, we provide results from the US-East-1 region. We use the AWS Lambda Pricing Calculator [41] to estimate the cost of serverless services, and their Elastic Compute Cloud (EC2) dedicated host pricing information [42] to estimate the cost of having a dedicated host. Similarly, for Azure, we use their dedicated host pricing list [43] and their pricing calculator [44] in the West US region.

We estimate the lower bound by using the lowest prices listed in the public pricing pages [42], [43]. For AWS, we use the `a1` instance (0.449 USD/hour $\times$ 720 hour/month = 323.28 USD/month) to compute this lower bound. For Azure, we use the listed monthly cost of the `DCsv2-Type1` VM. We note that other host types can be significantly more expensive.

Using a dedicated host not only incurs substantially higher costs but can also lead to scalability limitations. For example, in AWS, a single dedicated host can accommodate only 16 `a1-medium` instances (based on the capacity information provided in [42]). This level of concurrency is insufficient to handle bursts of requests. In contrast, *Double-Dip*, as a defense integrated within the serverless framework, preserves the platform's ability to scale elastically and can therefore support applications with high and variable demand.

X. Related Works

A. Serverless Cloud Scheduling

Scheduler is considered one of the critical components of a serverless cloud system. In recent years, there have been works that target different features of serverless functions and propose corresponding optimizations to scheduling algorithms. Fuerst *et al.* [7] propose an algorithm named Consistent Hashing with Random Load Updates (CH-RLU), which improves the classic consistent hashing algorithm and takes server loads, function cold-start overheads, etc., into consideration to make trade-offs between server loads and function locality. Kaffes *et al.* [47] offer a taxonomy of serverless scheduling algorithms and summarize a design parameter space of these schedulers. The authors then propose their algorithm, named Hermod, which significantly improves the performance of serverless systems.

Regarding locality optimizations, Aumala *et al.* [17] propose PASch, which also uses consistent hashing and allows serverless systems to utilize the package locality of running functions to accelerate the function start process. Abdi *et al.* [23] propose to use locality hints in schedulers that specify a function's preferences in data locality and optimize the performance of serverless functions. Li *et al.* [8] propose an algorithm to avoid function cold-starts in serverless systems by recycling idle function hosts of other functions and avoiding creating new function hosts.

B. Co-Location Attacks

Achieving co-location is an important and common prerequisite of various types of attacks. Most prior works target serverful clouds. The concept of co-location attacks in the cloud is first introduced and discussed by Ristenpart *et al.* [14]. They launch malicious VMs in AWS EC2 and manage to carry out side-channel attacks. The authors achieve co-location by either issuing multiple attack VMs through brute force or exploiting the scheduling locality of timing. Han *et al.* [48] formalize the co-location problem and theoretically analyze the minimum number of VMs required under various scheduling strategies. Makrani *et al.* [13] propose an attack method that targets machine-learning-based schedulers by disguising attackers' micro-architectural execution traces. Fang *et al.* [12], [28] discuss how to achieve co-location in a heterogeneous cloud and provide a method to quantitatively evaluate the co-location security threat.

The most relevant work is [16]. The authors propose a method to fingerprint physical servers and identify the auto-scaling feature of the Google Cloud Run platform, using which they manage to achieve instance co-location. However, compared to our work, their major focus is on the server fingerprinting step, and they only proved that their method works on one type of cloud scheduler. Our work focuses on the systemization of the co-location attack process, and our contribution is the construction of a universal attack

strategy that can efficiently target different kinds of serverless cloud schedulers. Also, we propose a low-overhead mitigation scheduling algorithm, which is based on "soft" user isolation, to defend against co-location attacks.

XI. Conclusion

In this paper, we focus on serverless cloud schedulers and propose methods to: (1) identify serverless scheduler policies; and (2) exploit revealed algorithm features to achieve serverless instance co-location. This series of techniques forms a practical guide for serverless cloud attackers, enabling them to identify scheduler vulnerabilities and construct targeted attacks. We perform thorough evaluations using simulations, a small-scale cluster, and Microsoft Azure Functions, covering schedulers in open-source platforms, commercial infrastructures, and experimental academic prototypes, proving the existence of co-location vulnerabilities in these serverless schedulers and the effectiveness of our attack. Future work will be dedicated to the design of more secure serverless scheduling systems.

XII. Ethics Considerations

Our research has minimal impact on the public. The experiments conducted on Microsoft Azure involve only functions that read TSC register values, which neither interfere with co-located instances nor cause significant resource utilization. Additionally, the total duration of these experiments is limited to a few hours, ensuring negligible disruption to other users. All other experiments were conducted on our own devices or on servers provided by CloudLab [35], and therefore have no impact on the public.

The study of co-location itself does not directly compromise cloud users' instances. It would require subsequent attacks to cause real damage, which are beyond the scope of this work. We disclosed our findings to Microsoft, although they did not classify the issue as a vulnerability, noting that side-channel defenses are already in place. These defenses, however, are orthogonal to our work, which focuses on the scheduler side.

References

[1] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[2] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: state-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2022.

[3] Amazon, "AWS Lambda," https://aws.amazon.com/lambda/, [Online; accessed Jun. 2024].

[4] Google, "Google Cloud Run," https://cloud.google.com/run, [Online; accessed Jun. 2024].

[5] Microsoft, "Azure Functions," https://azure.microsoft.com/en-us/products/functions, [Online; accessed Jun. 2024].

[6] ——, "Understanding serverless cold start," https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/, 2018, [Online; accessed Jun. 2024].

[7] A. Fuerst and P. Sharma, "Locality-aware load-balancing for serverless clusters," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 227–239.

[8] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, "Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 69–84.

[9] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A deep recurrent-reinforcement learning method for intelligent autoscaling of serverless functions," *IEEE Transactions on Services Computing*, 2024.

[10] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-free high-precision l3 cache attack using intel TSX," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2017, pp. 51–67.

[11] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 582–600.

[12] C. Fang, H. Wang, N. Nazari, B. Omidi, A. Sasan, K. N. Khasawneh, S. Rafatirad, and H. Homayoun, "Repttack: Exploiting cloud schedulers to guide co-location attacks," in *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, 2022.

[13] H. M. Makrani, H. Sayadi, N. Nazari, A. Sasan, K. N. Khasawneh, S. Rafatirad, and H. Homayoun, "Cloak & co-locate: Adversarial railroading of resource sharing-based attacks on the cloud," in *Proceedings of the International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2021, pp. 1–13.

[14] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2009, pp. 199–212.

[15] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2015, pp. 913–928.

[16] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "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 (ASPLOS)*., 2024.

[17] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 282–291.

[18] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[19] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 797–803.

[20] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 1–10.

[21] B. Sethi, S. K. Addya, and S. K. Ghosh, "Lcs: Alleviating total cold start latency in serverless applications with lru warm container approach," in *Proceedings of the 24th International Conference on Distributed Computing and Networking*, 2023, pp. 197–206.

[22] Microsoft, "Azure Functions hosting options," https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale, [Online; accessed Jun. 2024].

[23] M. Abdi, S. Ginzburg, X. C. Lin, J. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 365–380.

[24] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2014, pp. 719–732.

[25] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2016, pp. 279–299.

[26] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploit-

ing speculative execution," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2018, pp. 973–990.

[28] C. Fang, N. Nazari, B. Omidi, H. Wang, A. Puri, M. Arora, S. Rafatirad, H. Homayoun, and K. N. Khasawneh, "Heteroscore: Evaluating and mitigating cloud security threats brought by heterogeneity," in *The Network and Distributed System Security Symposium (NDSS)*, 2023.

[29] Airbnb, "Airbnb infosec tech talk," https://airbnb.tech/events/airbnb-infosec-tech-talk/, [Online; accessed Nov. 2025].

[30] Netflix, "Github repository of bless." https://github.com/Netflix/bless, [Online; accessed Nov. 2025].

[31] OpenWhisk, "OpenWhisk Documentation," https://openwhisk.apache.org/documentation.html, [Online; accessed Jun. 2024].

[32] ——, "OpenWhisk Github Repository," https://github.com/apache/openwhisk, [Online; accessed Jun. 2024].

[33] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 654–663.

[34] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 133–146.

[35] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019, pp. 1–14.

[36] "Dask.distributed," https://distributed.dask.org/en/stable/, Accessed 05/2024.

[37] Microsoft, "Azure functions host," https://github.com/Azure/azure-functions-host, Accessed 05/2024.

[38] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[39] Microsoft, "Azure Functions Consumption plan hosting," https://learn.microsoft.com/en-us/azure/azure-functions/consumption-plan, [Online; accessed Jun. 2024].

[40] T. Rehemäg, "Serverless case study - Coca-Cola," https://dashbird.io/blog/serverless-case-study-coca-cola/, 2020, [Online; accessed Oct. 2025].

[41] AWS, "AWS Lambda pricing calculator," https://calculator.aws/#/createCalculator/Lambda, [Online; accessed Oct. 2025].

[42] ——, "Amazon EC2 dedicated host pricing," https://aws.amazon.com/ec2/dedicated-hosts/pricing/, [Online; accessed Oct. 2025].

[43] Azure, "Azure dedicated host pricing," https://azure.microsoft.com/en-us/pricing/details/virtual-machines/dedicated-host/, [Online; accessed Oct. 2025].

[44] ——, "Pricing calculator — microsoft azure," https://azure.microsoft.com/en-us/pricing/calculator/, [Online; accessed Oct. 2025].

[45] "Amazon EC2," https://aws.amazon.com/pm/ec2/, 2022, [Online; accessed Apr. 2022].

[46] M. Azure, "Microsoft Azure," https://azure.microsoft.com/en-us/, 2022, [Online; accessed Apr. 2022].

[47] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 289–305.

[48] Y. Han, J. Chan, T. Alpcan, and C. Leckie, "Using virtual machine allocation policies to defend against co-resident attacks in cloud computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 95–108, 2015.

APPENDIX A
FINGERPRINTING ALGORITHM

The pseudocode is shown in Algorithm 2.

Algorithm 2 Scheduler Fingerprinting

function UPDATETRACE
  Input: A server fingerprint ζ generated by $\mathcal{F}$
  if $\mathcal{F} \notin \mathcal{R}$ **then** // A new server is discovered
    $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{\zeta \mapsto d\}$
    $d \leftarrow d + 1$
  end if
  $\mathbf{t} \leftarrow (\mathbf{t}, \mathcal{R}(\zeta))$
end function

// Server fingerprint record, $\mathcal{R}(\zeta)$ returns a server ID
$\mathcal{R} \leftarrow \{\}$
// The trace sequence
$\mathbf{t} \leftarrow ()_{0\times 1}$
// Index of the next discovered server
$d \leftarrow 1$

// Phase 1.
Prepare a function ϕ_0
$\Phi \leftarrow \{\phi_0\}$ // Pool of constructed services
for a period of time **do**
  Invoke ϕ_0
  UPDATETRACE($\mathcal{F}(\phi_0)$)
end for
for a period of time **do**
  Wait
end for

// Phase 2.
Prepare ϕ_1: a copy of ϕ_0, and let $\Phi \leftarrow \{\phi_0, \phi_1\}$
for a period of time **do**
  Select $\phi \in \Phi$ and invoke ϕ
  UPDATETRACE($\mathcal{F}(\phi)$)
end for
for a period of time **do**
  Wait
end for

// Phase 3.
Prepare n variations of ϕ_0 under the same name: ϕ_0', ϕ_0'', ...
$\Phi \leftarrow \{\phi_0, \phi_0', \phi_0'', ...\phi_0^{(n)}\}$
for a period of time **do**
  Select $\phi \in \Phi$ and invoke ϕ
  UPDATETRACE($\mathcal{F}(\phi)$)
end for
return $\mathbf{t}$

APPENDIX B
THEORETICAL ANALYSIS OF ATTACK

The co-location attack behaviors can be depicted using classical probability calculations. In this part, we provide theoretical analysis results of co-location attacks on different scheduler models with simplification. These analytical results enable an attacker to evaluate the cost of the attack.

A. Basics

In the rest of this section, we utilize the following notations:

- Victim function: v;
- Attacker functions: $\{a_1, a_2, ...\}$;
- Cluster: $S = \{n_1, n_2, ..., n_N\}$.

We define the following events:

- A_i^α: After invoking the attack function α times, any of the attack function hosts are placed on node n_i;
- V_i^β: After invoking the victim function β times, any of the victim function hosts are placed on node n_i;
- $C_i^{\alpha,\beta}$: Any of the aforementioned $\alpha+\beta$ invocations result in the co-location of victim and attacker function hosts on node n_i.

The indicator variable I is defined as follows:

$$I(X) = \begin{cases} 1, & \text{if event } X \text{ happens} \\ 0, & \text{otherwise.} \end{cases}$$

The placement results of schedulers that consider only resource availability can be approximated as random events in a classical probability model. This can be explained as follows: since cluster information is agnostic to all users, from a user's perspective, the resource availability on physical servers is a random variable following the same distribution. This symmetry indicates that the probability of a user's instance being placed on any given node is equally $1/N$, where N is the size of the cluster.

B. Random Scheduling

We first consider the scenario where every invocation to functions is randomly dispatched to nodes in the serverless cluster. In this scenario, each function invocation is considered independently, and no locality optimization is taken. If the assigned machine does not have a corresponding warm function host running, it results in a cold-start process. This scheduling policy is load-balanced but suffers from low performance due to the lack of locality optimizations.

We first calculate the probability of event A_i^α and V_i^β. $\forall n_i \in S$:

$$\begin{aligned} P(V_i^\beta) &= 1 - P(\text{None of the victim function hosts is on } n_i) \\ &= 1 - \frac{(N-1)^\beta}{N^\beta} \\ &= 1 - \left(1 - \frac{1}{N}\right)^\beta. \end{aligned}$$

Similarly,

$$P(A_i^\alpha) = 1 - \left(1 - \frac{1}{N}\right)^\alpha.$$

As the placements of victim and attacker instances are independent, we have:

$$\begin{aligned} P(C_i^{\alpha,\beta}) &= P(A_i^\alpha)P(V_i^\beta) \\ &= \left[1 - \left(1 - \frac{1}{N}\right)^\alpha\right]\left[1 - \left(1 - \frac{1}{N}\right)^\beta\right]. \end{aligned}$$

The expected number of servers that have co-located victim and attacker function hosts is

$$\begin{aligned} E_C &= E\left[\sum_{i=1}^{N} I\left(C_i^{\alpha,\beta}\right)\right] \\ &= \sum_{i=1}^{N} E\left[I\left(C_i^{\alpha,\beta}\right)\right] \\ &= N\left[1 - \left(1 - \frac{1}{N}\right)^\alpha\right]\left[1 - \left(1 - \frac{1}{N}\right)^\beta\right]. \end{aligned} \tag{1}$$

From Eq. (1) we can see that as the system keeps running,

$$\lim_{\alpha,\beta\to\infty} E_C = N,$$

i.e., co-location tends to happen on almost every physical server if the number of invocations to victim and attacker functions is sufficiently large.

C. Invocation Locality

Now, let's consider the scenario where the scheduler introduces optimization to avoid cold-start and always assigns function executions to existing function hosts if there are any. In this scenario, we can define α as the number of different attack functions and β as the number of launched victim function hosts. In this scenario, we have $\beta \equiv 1$. As there are α different functions, there will be α different function hosts that are placed on physical machines in the cluster.

We can calculate the probability of co-location in the system as follows:

$$\begin{aligned} P_C = P(\text{Co-Location}) &= \frac{\sum_{i=1}^{N} N^\alpha - (N-1)^\alpha}{N^{\alpha+1}} \\ &= 1 - \left(1 - \frac{1}{N}\right)^\alpha. \end{aligned}$$

By substituting $\beta = 1$ in Eq. (1), we have

$$E_C = 1 - \left(1 - \frac{1}{N}\right)^\alpha.$$

As the attacker invests more and increases α,

$$\lim_{\alpha\to\infty} P_C = \lim_{\alpha\to\infty} E_C = 1.$$

D. Auto-Scaling

Now let's consider the scenario where the scheduler additionally applies auto-scaling policies when there is a high demand for a function. The attacker stresses one attack function, enabling the attacker to cover more servers. Let α be the

number of attack function invocations, and every r invocations result in a new function host to be placed. We have

$$P_C = P(\text{Co-Location}) = 1 - \left(1 - \frac{1}{N}\right)^{\frac{\alpha}{r}}.$$

We can see that since it requires r invocations to spread a new function host, stressing the auto-scaling policy is not as efficient as having α different functions. In practice, an attacker should prioritize creating more functions, and stressing the auto-scaling policy can be an auxiliary strategy to increase coverage.

E. Configuration-based Locality Optimizations

Assume that additional locality features (e.g., package locality [17]) are integrated into the scheduler. In this case, if the attacker provides the same specifications as the victim, he/she can increase the probability of having function hosts placed on the machine with a victim host, i.e.,

$$P(A_i^1 | V_i^1) = p > \frac{1}{N}. \tag{2}$$

Assume the attacker prepares α different functions that exploit the locality feature. We have:

$$P_C = 1 - (1-p)^{\alpha} > 1 - \left(1 - \frac{1}{N}\right)^{\alpha}.$$

We can see that the exploitation of these features can increase the chance of co-location.

APPENDIX C
CORRECTNESS OF *Double-Dip* ALGORITHM

A. Assumptions and Notation

- $H = \{h_1, h_2, \ldots, h_m\}$: Set of hosts.
- f_u: A function belonging to user u.
- $C(u)$: set of servers user u occupies
- $R(h)$: Available resources on host h.
- $r(f)$: Resource requirement of function f.
- Target: $min_{i,j}(|C(u_i) \cap C(u_j)|)$

B. Claim: The algorithm minimizes co-location of users and respects resource constraints

Proof: We aim to show that the algorithm minimizes the co-location metric while respecting resource constraints. The proof proceeds by induction on the number of assignments made by the algorithm.

Preliminaries: Let $C(u)$ represent the set of servers user u occupies, and let $C^*(u)$ denote the set of hosts after assigning function f_u to host h. The co-location metric is defined as:

$$\sum_{i,j} |C^*(u_i) \cap C^*(u_j)|$$

and measures the degree of overlap in host assignments for all user functions. The algorithm assigns each function to a host while ensuring that resource constraints are met.

Inductive Step:

1) Case 1: If there exists a host $h \in C(u)$ that has sufficient resources $r(f) \leq R(h)$, the function f_u is assigned to h. In this case, the co-location metric remains unchanged, as the assignment does not introduce any new overlap:

$$\forall i, j,\ |C(u_i) \cap C(u_j)| = |C^*(u_i) \cap C^*(u_j)|.$$

Assigning f_u to any other host $h \notin C(u)$ would unnecessarily increase the co-location metric. Thus, the choice of h here is optimal, and the resource constraint is respected.
2) Case 2: If no such host $h \in C(u)$ exists, the algorithm selects a host h^* that satisfies the resource constraint and minimizes the increase in the co-location metric. Formally, the host is chosen as:

$$h^* = \underset{h \in H}{\operatorname{argmin}} \left(\textstyle\sum_{i,j} |C^*(u_i) \cap C^*(u_j)| - \sum_{i,j} |C(u_i) \cap C(u_j)| \right),$$

subject to:

$$r(f) \leq R(h)$$

Here, we select the host h^* that minimizes the co-location rate increase by calculating the intersection of user co-locations. In the implementation, this is achieved by selecting the host with the least user variety, which corresponds to finding the host that will minimize the increase in the co-location rate. Algorithm 3 is designed to identify such a host by evaluating the user diversity across nodes and selecting the one with the least variety. This approach ensures that the co-location rate increase, as described in the proof, is minimized.

Algorithm 3 findLeastUserVarietyHost Function

Require: $nodeList$: List of nodes
Ensure: A node with the least variety of users

```
minVariety ← ∞
selectedNode ← null
for all node ∈ nodeList do
    userSet ← getUsers(node)
    variety ← size(userSet)  ▷ Calculate the number of unique users
    if variety < minVariety then
        minVariety ← variety  ▷ Update the minimum variety
        selectedNode ← node  ▷ Update the selected node
    end if
end for
return selectedNode
```

Greedy Property: At each step, the algorithm makes a locally optimal choice: either preserving the co-location metric (h) or minimizing its increase (h^*). This greedy approach ensures that the global co-location metric remains as small as possible after each assignment. The inductive hypothesis guarantees that all prior assignments are optimal with respect to the metric and resource constraints, and the current assignment extends this property.

Conclusion: By induction, the algorithm minimizes the co-location metric globally while respecting resource constraints

for every assignment. The proof accounts for all cases and guarantees that no alternative choice at any step would lead to a better solution.

APPENDIX D
VALIDATION OF SERVER FINGERPRINTING METHOD

The TSC-based method was rigorously tested in [16]. However, their tests are conducted on Google Cloud Run [4] and need to go through a relatively complicated process. In this part, we validate that the TSC-based method also works on Microsoft Azure through relatively simple experiments.

Although this behavior is not officially documented, it is reasonable to assume that cloud providers co-locate functions accessing the same data to optimize locality and resource utilization. While co-location is not guaranteed, functions that share data, particularly those belonging to the same serverless application, are likely to have a higher probability of being physically co-located. We have the following assumptions:

Assumption 1: Functions within the same application and access shared data are more likely to be co-located.

Assumption 2: TSC reads can be used as reliable server fingerprints.

If we deploy function pairs $\{\texttt{func1}, \texttt{func2}\}_{(1,2,\ldots)}$ from the same serverless application that access shared data, and function pairs $\{\texttt{func1}', \texttt{func2}'\}_{(1,2,\ldots)}$ from different serverless applications, then only when **BOTH** assumptions hold can we observe statistically significant differences in the TSC readings between functions from the same pair.

In our experiments, we deploy functions with TSC read capabilities. The function pairs are constructed under two conditions (20 pairs each): (1) both functions belong to the same serverless application and access shared data through a common Azure storage account; and (2) the functions belong to different serverless applications and do not access any shared resources.

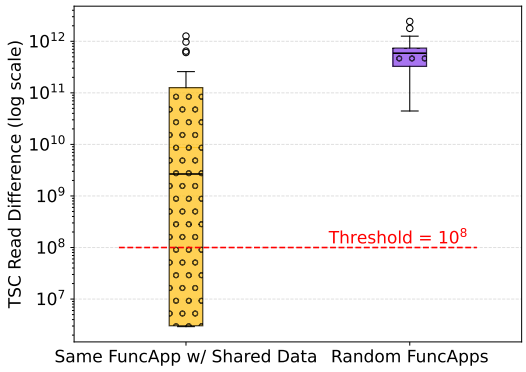


Fig. 13: Comparison of TSC read differences between function pairs.

The results are shown in Fig. 13. As expected, the TSC read differences for condition (1) (*Same FuncApp w/ Shared Data*) are significantly lower than those for condition (2) (*Random FuncApps*); that is, function pairs likely to be co-located indeed exhibit similar TSC readings, whereas random function pairs show much higher TSC read variances.

For accurate co-location measurement, we set the threshold to $t_h = 10^8$ (indicated by the dotted line in Fig. 13). Any two measurements with a difference below this threshold are considered to originate from co-located function hosts.