Gringotts: Fast and Accurate Internal Denial-of-Wallet Detection for Serverless Computing

Junxian Shen
INSC, Tsinghua University
Zhongguancun Laboratory
shenjx22@mails.tsinghua.edu.cn

Han Zhang
INSC, Tsinghua University
Zhongguancun Laboratory
zhhan@tsinghua.edu.cn

Yantao Geng
INSC, Tsinghua University
Zhongguancun Laboratory
gyt22@mails.tsinghua.edu.cn

Jiawei Li
INSC, Tsinghua University
li-jw19@mails.tsinghua.edu.cn

Jilong Wang
INSC, Tsinghua University
BNRist, Tsinghua University
Zhongguancun Laboratory
Peng Cheng Laboratory

Mingwei Xu
INSC, Tsinghua University
BNRist, Tsinghua University
Zhongguancun Laboratory
Peng Cheng Laboratory

ABSTRACT

Serverless computing, or Function-as-a-Service, is gaining continuous popularity due to its pay-as-you-go billing model, flexibility, and low costs. These characteristics, however, bring additional security risks, such as the Denial-of-Wallet (DoW) attack, to serverless tenants. In this paper, we perform a real-world DoW attack on commodity serverless platforms to evaluate its severity. To identify such attacks, we design, implement, and evaluate Gringotts, an accurate, easy-to-use DoW detection system with a negligible performance overhead. Gringotts addresses the information ambiguity inherent in serverless functions by introducing a well-designed performance metrics collection agent. Then, Gringotts uses the Mahalanobis distance to discover anomalies in the distribution of the metrics. We implement Gringotts as a real system and conduct extensive experiments using a testbed to evaluate the performance of Gringotts. Our results indicate that Gringotts has a performance overhead of less than 1.1%, with an average detection delay of 1.86 seconds and an average accuracy of over 95.75%.

CCS CONCEPTS

• **Networks** → *Network experimentation*; • **Security and privacy** → **Denial-of-Service attacks**.

KEYWORDS

Serverless Computing; Denial-of-Wallet Attack; Performance Anomalies Detection

ACM Reference Format:
Junxian Shen, Han Zhang, Yantao Geng, Jiawei Li, Jilong Wang, and Mingwei Xu. 2022. Gringotts: Fast and Accurate Internal Denial-of-Wallet Detection for Serverless Computing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560629

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560629

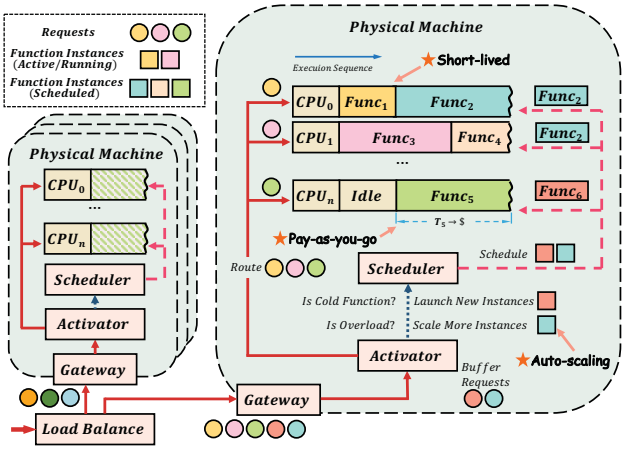


Figure 1: Typical serverless platform architecture. Functions 2, 4, and 5 are functions that are scheduled to execute after functions 1 and 3 yield their allocated CPUs.

1 INTRODUCTION

Serverless computing, or Function-as-a-Service (FaaS), has continued to evolve at a tremendous speed since its inception [9]. With the serverless computing paradigm, tenants entrust the platforms with additional responsibilities such as task deployment, environment configuration, and auto-scaling. More companies, especially startups, are considering moving their businesses to serverless platforms since they offer appealing advantages, e.g., flexibility, low cost, and the pay-as-you-go billing model.

Figure 1 shows a typical serverless platform architecture. Users usually upload their code to the serverless platform, and the platform will generate an execution instance based on the code, where the instance may be a virtual machine (VM), container, or process, depending on the implementation of the platform. When a new request is dispatched to an instance via the load balance, the activator determines whether to cold start, scale, or route the request directly to the associated execution instance. Throughout the procedure, serverless embodies three key *features*: (i) pay-as-you-go, i.e., users only pay for the consumed resources; (ii) auto-scaling, i.e., the platform automatically scales in and out in response to the

Commercial Platforms	Duration Costs	Requests Costs (/1M requests)	Other Costs
AWS Lambda[50]	$\$1.667 * 10^{-5}$/GB-s	\$0.20	Networking + Resource reserved
GCF[51]	$\$2.5 * 10^{-6}$/GB-s + $\$1.0 * 10^{-5}$/GHz-s	\$0.40	Networking + Deployment costs
Azure Functions[52]	$\$1.6 * 10^{-5}$/GB-s	\$0.20	Networking + Storage costs
Alibaba Cloud FC[49]	$\$1.6384 * 10^{-5}$/GB-s	\$0.20	Networking

Table 1: Current billing models of major commercial serverless platforms.

traffic; and (iii) short-lived, i.e., several functions are multiplexed on the same CPU core and recycled when they are idle.

These features distinguish FaaS from Infrastructure-as-a-Service (IaaS), but they also expose serverless tenants to traditional attacks as well as additional security threats. One of these security threats is the Denial-of-Wallet (DoW) attack, which can be seen as a variant of the Denial-of-Service (DoS) attack specifically conducted on serverless platforms. There are two primary DoW attack types, i.e., external DoW attacks and internal DoW attacks. The external DoW attackers will drain money from the victims by repeatedly *invoking the APIs that the victims unwittingly expose*, and the internal DoW attackers will try to cause *resource contention* on the function instances (e.g., containers or VMs) of the victims, slowing down their programs. In this paper, we focus on **internal** DoW attack detection since external DoW attacks can be detected by existing technologies, e.g., ingress filtering [13, 74], traceback [63, 78], and source validation [33, 69].

Similar to a DoS attacker, an internal DoW attacker conducts such attacks to acquire advantages in *malicious industry competition, extortion, or hacktivism.* By delaying the execution of a victim's function, an internal DoW attack can have two direct effects on the victim. First, the internal DoW attack can cause victims to go bankrupt due to platform expenses resulting from extended execution duration and the "pay-as-you-go" billing model. For example, a startup has reported spending $72,000 and nearly declaring bankruptcy due to improperly configured serverless functions [66]. Second, the critical services of the victim will be severely disrupted due to the increase in function duration. Amazon has claimed a 1% loss in sales for every 100*ms* increase in page load time [26].

For internal DoW attack detection, the platforms must monitor and analyze the low-level resource usage of specific bare-metal machines. Fast and precise internal DoW attack detection is a big **challenge**, as it must satisfy two objectives. First, improper sampling of performance data can induce inaccuracies in the detection. Second, serverless functions are transient and highly dynamic, contributing to a noisy environment. There are two kinds of performance analysis methods. The first is based on application-level approaches: metrics tools (e.g., Prometheus [53]), tracing tools (e.g., Jaeger [22]), and logging tools (e.g., Fluentd [14]). These tools collect metrics at or above the operating system level, such as CPU utilization and application logs. They do not, however, collect low-level information such as bus locking and cache misses. The second is based on collecting hardware metrics, such as opcm [46], perf [18], and likwid [31]. These tools determine the physical machine's status based on the number of operations performed by the hardware. But they collect metrics using timers, which leads to data ambiguity. Additionally, because these solutions lack knowledge about application behavior, they are unable to determine if changes in the underlying metrics are the result of load fluctuation or an attack.

Facing these challenges, we make the following **contributions**:

Firstly, we perform a comprehensive analysis of the DoW attack, including its types, root causes, etc. (see § 2). Then we conduct a real-world internal DoW attack on commodity serverless platforms to show DoW jeopardization and understand its root cause. We also generalize the challenges of internal DoW attack detection on serverless platforms (see § 3).

Secondly, we develop a generic system, named Gringotts, for the accurate and timely detection of internal DoW attacks on serverless platforms (see § 4). Gringotts has an easy-to-use and fast **request-oriented sampling** tool that is aware of the function behavior (see § 5). On the one hand, the implementation of Gringotts is loosely coupled with the function and is independent of the metrics selected. On the other hand, its execution is tightly coupled with the function, guaranteeing that metrics collection is unaffected by multiplexing, runtime destruction, or changes in the function's behavior. Additionally, Gringotts discovers the **linear relationship** between execution time and the HPCs. It incorporates a fast and efficient internal DoW detection algorithm based on **HPC-based workload verification** and utilizes the novel concept that the genuine workload remains constant with or without an internal DoW attack. It determines if an internal DoW attack happened based on changes in the workload distribution. We assume that the performance metrics follow a learnable overall trend under normal conditions and represent this distribution using a hyperplane. Then Gringotts describes the noise in the metric collection using a multivariate Gaussian distribution. Finally, by calculating the Mahalanobis distance between the sample and the center of the Gaussian distribution, we are able to accurately assess the occurrence of the internal DoW attack (see § 6).

Thirdly, we implement Gringotts as a real system and conduct extensive experiments using a testbed (see § 7). Our experiments demonstrate that Gringotts can: (1) impose a negligible overhead of less than 1.1%; (2) detect the onset of a DoW attack in as little as 1.86 seconds; and (3) achieve an average accuracy of over 95.75%.

The remainder of this paper is organized as follows: In Section 2, we introduce the background. In Section 3, we define the Denial-of-Wallet attack (Sec 3.1), conduct a real-world DoW attack (Sec 3.2), and discuss the detection challenges (Sec 3.3). We provide an overview of Gringotts' architecture in Section 4, discuss its sampling method in Section 5, and introduce the detection model in Section 6. We conduct thorough experiments on Gringotts in Section 7. Sections 8, 9, and 10 are the discussion, related work, and ethical concerns. Finally, we conclude the paper in Section 11.

2 BACKGROUND

In this section, we first briefly introduce the serverless concept and its billing model (Sec 2.1). Then we discuss the memory bus locking technique (Sec 2.2) and the performance background noise of serverless platforms (Sec 2.3).

2.1 Serverless Computing and Billing Models

As an emerging cloud programming paradigm, serverless computing derives its attractiveness from the pay-as-you-go billing model, the facilitation of deployment, the seemingly infinite scalability, and competitive performance.

The billing model for serverless computing is based on the idea of "pay-as-you-go" and is widely adopted by cloud providers like AWS [28], Google [16], Microsoft [17], and Alibaba [7], as shown in Table 1. The most important component of the billing model is the **duration costs**, which start when a new request is received and end when a response is returned. For instance, a function on AWS Lambda that processes a request in 1 second and has 1024 MB of memory will be charged $\$1.667 * 10^{-5}$ for duration costs.

This duration-based billing model gives FaaS a price advantage over IaaS. However, clients may be overcharged if their functions slow down or for time when they do not run at all. Functions can either actively (e.g., sleep) or passively (e.g., wait for I/O operations) yield their allocated CPUs during the execution. Experiments show that the CPUs can execute other tasks after being yielded, but the victims will still be charged until **their functions return the response** on commodity platforms. Attackers can cause resource contention and, by leveraging this flaw, convert the victim's prolonged execution time into financial exhaustion. We refer to this attack as a "Denial-of-Wallet attack" (Sec 3.1).

2.2 Memory Bus Locking

Modern processors employ a special technique to preserve the atomicity of certain instructions. When an atomic memory operation (e.g., `ATOMIC_FETCH_ADD`) crosses the cache line boundary, the hardware will briefly halt other memory operations in order to comply with the cache coherence protocols [2]. This process, represented in Figure 2, is known as memory bus locking and can be leveraged by attackers to discover function co-location or slow down the execution of the victim.

Memory bus locking is permitted on the majority of the cloud platforms, and it is simple to implement without any additional privileges. Atomic memory operations such as `ATOMIC_FETCH_ADD` are commonly used in multithreaded applications for communication over shared resources, and even the language runtimes (e.g., Python) contain these instructions. Normal users can execute such instructions either by directly coding them in their functions, as in Listing 2, or by implicitly invoking them through implementations of the language runtime. Given the prevalence of these operations, cloud providers cannot defend themselves with simple patches, such as banning all types of atomic operations. Moreover, it is a hardware vulnerability that affects all x86 hardware (Intel [2] and AMD [36]) and the majority of cloud apps (e.g., machine learning applications, HTTP servers, key-value stores, etc.). These instructions cause hardware-level congestion that OS-level tools (e.g., cgroups) are unable to detect or control.

To demonstrate that existing cloud platforms still support the above-mentioned atomic memory operation, we validated AWS [28], Google Cloud Function [16], Microsoft Azure Function [17], and Alibaba Function Compute [7]. We allocate a contiguous memory region and then perform an eight-byte atomic operation with an increasing offset on this region. The results of the experiment

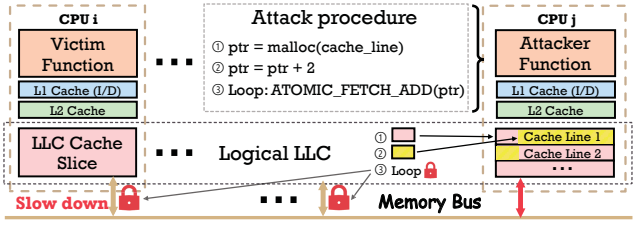


Figure 2: Overview of memory bus locking.

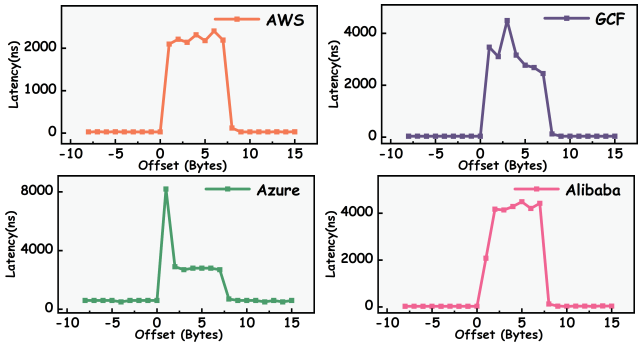


Figure 3: Latency spikes caused by atomic operations.

are shown in Figure 3. When an 8-byte atomic memory operation crosses the cache line boundary (Offset 1-7), the instruction's execution time increases from tens of milliseconds to thousands of milliseconds. This provides convincing evidence that cloud service providers enable memory bus locking.

2.3 Performance Background Noise

Despite the strong isolation provided by VMs and the resource consumption restrictions enforced by Linux cgroups interfaces [34], commodity FaaS platforms display *pervasive performance background noise*, which has a significant impact on the precision of the performance monitoring model. Numerous factors contribute to this noise, including the short-lived nature of the function, the scheduling of the platform, etc.

```
1 def lambda_handler(event, context):
2     return {'statusCode': 200, 'body': b'Hello-World!'}
3
4 server(lambda_handler).run()
```

Listing 1: An example of Python Hello-World function.

We measure the execution time of a simple Python Hello-World function similar to Listing 1. Other impacts, such as gateway congestion, are excluded since they are not counted in the execution time [49–52]. Figure 4 shows a periodic "traffic tide" on AWS Lambda and Alibaba Cloud FC over a four-day period. On the same day, gaps between the maximum and minimum execution duration of the same function were exceeded *sixfold* on AWS Lambda and *17 times* on Alibaba Cloud FC.

3 DENIAL-OF-WALLET ATTACK

In the following section, we first introduce the Denial-of-Wallet attack (Sec 3.1). Then, we present a thorough and comprehensive description of the internal DoW attack by providing a real-world

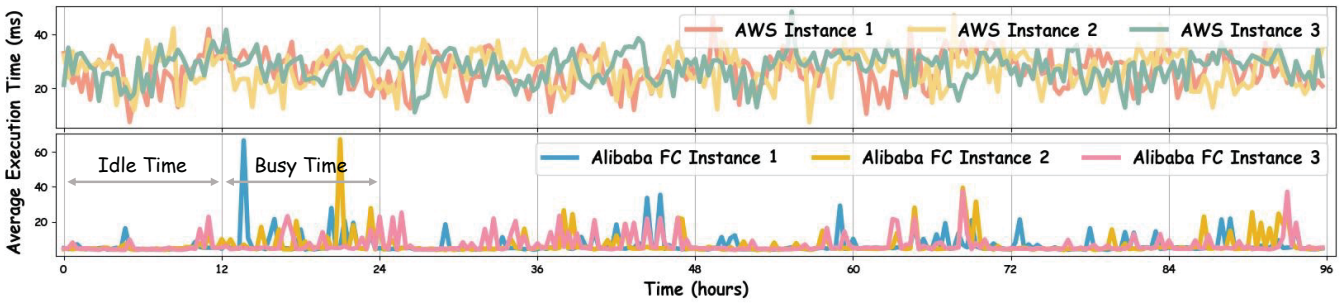


Figure 4: Ubiquitous background noise in AWS Lambda[28] and Alibaba Function Compute[7].

example (Sec 3.2) and discussing the challenges of internal DoW attack detection (Sec 3.3).

3.1 What Is Denial-of-Wallet Attack

We define the Denial-of-Wallet (DoW) attack as *a variant of the Denial-of-Service (DoS) attack specifically conducted on serverless platforms*. Generally, DoW attackers leverage DoS techniques such as generating request flooding or imposing resource contention to stress *the target tenant*. However, there are three key differences between DoW and classic DoS attacks.

First, although both DoS and DoW attacks try to drain the resources of the victim, DoS attackers try to exceed resource limitations, whereas DoW attackers focus on the pay-as-you-go billing model. Second, DoS attacks often result in the complete unavailability of services, while DoW attacks leave services accessible for normal users. Finally, DoS and DoW attacks have distinct financial consequences. DoS attacks will result in business interruption, reputation damage, and customer loss. DoW attacks, on the other hand, turn the stress on the target tenants into direct charges from ISPs because the victims have to pay for the used resources or instances.

DoW attacks can be classified into two types depending on where the malicious resource consumption is launched: *external DoW attacks* and *internal DoW attacks*. External DoW attacks take advantage of the victims' *service or resource APIs* [24, 38] and generate massive invocations of these APIs. External DoW attacks can be defended by existing DoS detection mechanisms such as ingress filtering [13, 74], traceback [63, 78], and source validation [33, 69]. *External DoW attacks are not the focus of this paper.*

3.2 How to Launch an Internal DoW Attack

Internal DoW attacks are carried out by triggering resource contentions on shared hardware. Attackers deploy malicious functions utilizing co-location detection techniques and invoke them to excessively utilize unmonitored yet vital shared resources. The victim will obtain the correct function execution result, but due to the serverless billing model (Sec 2.1) and the longer execution time, the victim will be charged multiple times. In an internal DoW attack, the **target victim's functions** are a set of functions belonging to a specific user. After choosing the function of the intended victim, an attacker can execute a standard internal DoW attack with the following three steps:

Step 1: Malicious function placement. The attacker deploys hundreds of functions as normal users and expects them to reside on the same physical machine as the victim. Using co-location verification methods, the attacker then terminates functions that are not co-located. In this paper, we employ a covert channel similar to [76] for co-location verification; its pseudo-code is included in Section 8.

```
#define LEN  8192

void memory_bus_locking() {
    atomic_llong* ptr=(atomic_llong*) malloc((LEN+1)*8);
    atomic_llong* unalign_ptr=ptr+2;

    // start locking loops
    for (int i=0; i<DEFINED_LOOP_TIMES; i++) {
        for (int j=0; j<LEN; j++) {
            atomic_fetch_add(unalign_ptr+j, 1);
        }
    }
}
```

Listing 2: An example of memory bus locking.

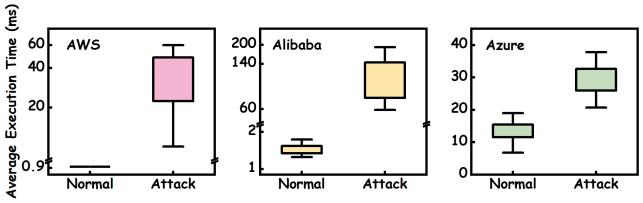
Step 2: Create resource contentions. The attacker then executes the co-located malicious functions deployed in Step 1 to slow down the execution of the target functions. Memory bus locking (Sec 2.2) will be executed by these functions when invoked.

To perform memory bus locking, an attacker can program *atomic memory instructions that cross cache line boundaries* into their functions. For the Intel x86 architecture [2], these operations are language instructions (e.g., `Interlocked.Increment` in C#, `atomic.AddUintptr` in Golang, `ATOMIC_FETCH_ADD` in C++, etc.) that will be translated into machine code that will trigger the automatic locking protocol (e.g., `XCHG`) or have the LOCK# prefix (e.g., `XADD`).

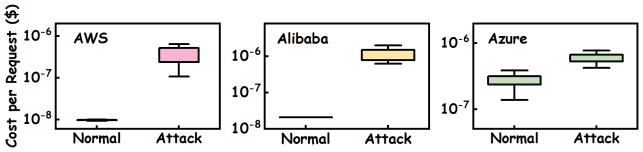
We use memory bus locking (shown in Listing 2) to create shared resource contentions, but this is not fundamental. Contentions on numerous shared resources, such as PCIe I/O switches [65], caches [23, 32, 81], and even power management [6, 20], have been reported to degrade the performance of the entire physical machine. These contentions on shared hardware resources are also easy to conduct and difficult to defend.

Step 3: Direct financial exhaustion. Finally, the pay-as-you-go billing model will turn the performance loss into the victim's financial exhaustion. Given that resource allocation remains as configured and execution time is multiplied, the amount of money spent by the user will increase as a consequence of the attack.

To show the practicality and feasibility of the internal DoW attack, we executed a real-world internal DoW attack against AWS Lambda [28], Microsoft Azure Function [17], and Alibaba Function Compute [7] following the three steps outlined above.



(a) Influence of the DoW attack on function execution time.



(b) Influence of the DoW attack on per-request cost.

Figure 5: An example of real-world Denial-of-Wallet attack.

3.2.1 Attack Settings. For simplicity and generality, we implement a Hello-World function similar to Listing 1 as the victim function. To detect co-residency, we implement a covert channel similar to [76]. The functions of the victim and the attacker are deployed from two different accounts, both controlled by us. We use code similar to Listing 2 to implement memory bus locking in the malicious functions. To ensure that the results of the victim function are not affected by background noise, we alternate several times between normal execution and attack procedure. Figure 4 depicts hourly fluctuations in performance, whereas our experiment is conducted on a minute scale. Before and after the attack, we observed a significant and stable shift in the duration of the victim function. We retrieve the experiment results (resource usage or per-request billing information) using APIs offered by cloud platforms. For platforms that do not give direct pricing information, we manually calculate the amount using the corresponding billing model [49, 50, 52].

3.2.2 Attack Results. Figure 5a depicts the execution time of the victim function before and after the attack. The attack drastically reduces the victim's performance and raises the charge. On AWS Lambda, the victim's original average function execution time was 0.93ms, whereas this number increased to 34.44ms after being attacked, increasing by 37.03 times. On Alibaba FC, the victim's average duration has increased by 78.88 times, from 1.45ms to 114.37ms.

The average duration increase of Azure (from 13.71ms to 35ms) was not as great as that of AWS Lambda and Alibaba FC. This is likely due to the fact that the Azure platform does not permit users to adjust the number of CPUs utilized by a function [17]. It is essential to mention that all platforms execute the identical functions. The average execution time of Azure functions during normal execution (13.71ms) is much longer than that of AWS (0.9ms), indicating that Azure scheduled the victim functions. During an attack, Azure and AWS both had to allocate an entire CPU to the victim function, so their performance became nearly identical (~ 35ms).

Figure 5b shows the rise in the per-request cost. Since the memory usage of the victim function does not change before and after the attack, the increased cost of the function is proportional to its execution duration. Notably, in Alibaba FC, the execution time and resource consumption do not rise by the same factor. This, we believe, is because the platform rounds up execution time in request units when invoicing for resources.

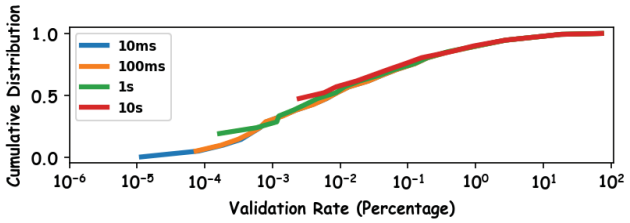


Figure 6: Sample validation rate on Azure traces [80].

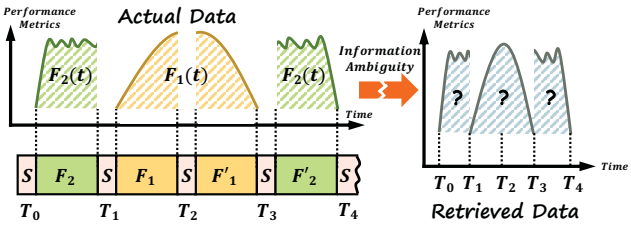


Figure 7: Sampling with a fixed-length window.

A hypothetical victim scenario. Consider the following scenario described in the AWS Lambda pricing example [50]: A victim deploys a mobile backend application. The program is configured with 1536 MB of memory and handles three million requests per month with a 120ms response time for each request. The application is expected to use $5.4 * 10^5 GB * s$ of resources, for which the victim should be charged $2.73 after subtracting the $4 * 10^5 GB * s$ free tier amount. However, if a malicious tenant conducts an internal DoW attack, the victim may consume $1.998 * 10^7 GB * s$ of resources and will be charged $326.33 for the single application.

3.3 Accurate Detection Is Challenging

DoW attacks are carried out by maliciously utilizing shared hardware resources amongst tenants on the same bare-metal machine. To detect such misbehavior, platforms must monitor and analyze the resource usage characteristics of specific bare-metal machines. However, because serverless workloads are transient and extremely dynamic, accurately detecting DoW attacks presents two significant challenges.

Errors introduced by improper sampling. Previous studies such as memory DoS detections [30, 79], performance and resource interference predictions [4, 15, 29, 35, 41, 54, 62, 70, 75] have all used a fixed-length window sampling approach to record performance metrics and resource consumption attributes. Every few milliseconds, the sample agent obtains resource-related statistics via hardware Processor Counter Monitor (PCM) tools. This method of sampling with a fixed-length window overlooks the regularly changing function contexts in serverless systems, resulting in data invalidation and corruption.

Data invalidation is caused by the well-known short-lived nature of serverless functions, as functions may not be executed when sampled. To demonstrate, we conduct a simulation using the traces in [80], as shown in Figure 6. We calculate the cumulative fraction of invalid data collected for each function using a fixed sampling window size of 10ms to 10s. More than 80% of the functions have a valid rate of less than 20%.

Data corruption is caused by the differences in performance metrics between functions and between requests for the same function,

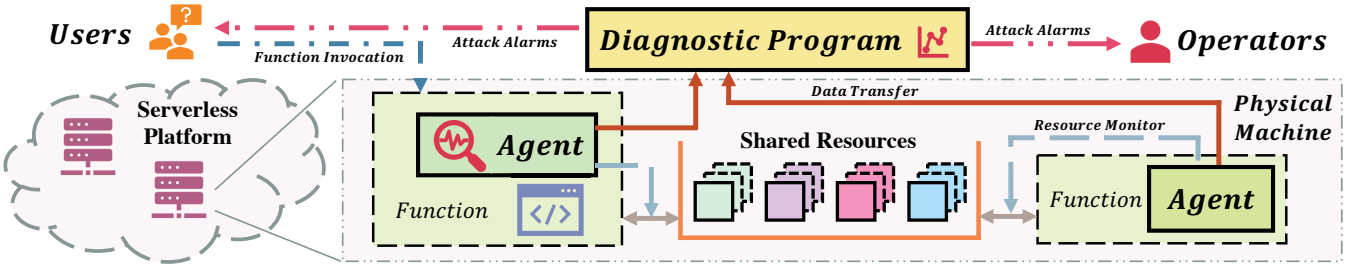


Figure 8: The architecture of Gringotts. Users and operators receive attack alarms and take additional actions.

as shown in Figure 7. On the one hand, serverless solutions enable high resource utilization by multiplexing different functions on the same CPU core. Due to the fact that different apps present significantly different metrics, this multiplexing will combine them. On the other hand, performance metrics may vary significantly depending on the nature of the request. [40] demonstrates how an image blurring function's memory consumption changes significantly depending on the size of the input image. Additionally, metrics widely used in [30], such as Last Level Cache (LLC) hits/misses, also change significantly when various requests are processed.

Errors introduced by the noisy environment. Serverless systems are more congested than IaaS, which co-locates VMs less or equal to the core numbers. The number of serverless instances on a bare-metal machine might easily exceed 100 [64], posing a bigger denoising challenge. Furthermore, bursty workloads of serverless functions [61] complicate the detection model. Platforms must determine whether the changes in performance metrics are caused by traffic surges or a DoW attack.

4 SYSTEM OVERVIEW

Architecture. Gringotts consists of two high-level components: (i) an **Agent** and (ii) a **Diagnostic Program**.

Figure 8 shows a simplified system architecture of Gringotts. To utilize the serverless service provided by the platform, users must integrate their code into the runtime template provided by the cloud providers [25, 44, 45, 57–59]. Upon receiving requests from tenants, the cloud provider will select a server from the cluster and deploy the uploaded function on that physical machine. The platform will allocate a specified amount of resources based on the subscription of the user, and the function runtime will maintain a processing loop to handle received requests.

The Agent of Gringotts is responsible for monitoring the resource utilization of the target functions. It is incorporated by the platform into the runtime of every monitored function and deployed along with the instance just like any other runtime component (e.g., HTTP request handling, function logging). The Agent will be automatically executed to gather hardware performance metrics each time the users invoke the functions. Function metrics are collected, aggregated into a request-based unit, and then transmitted to the Diagnostic Program. The Agent accepts coordination from the platform and maintains the same lifecycle as the user function.

The Diagnostic Program of Gringotts performs the actual detection of the internal DoW attacks atop physical machines. The aggregated performance metrics received from the Agent can be transmitted through protocols such as HTTP, gRPC [19], or even piggybacking. It does per-request detection in real-time for target functions without requiring local execution on the same machine, and cloud providers might deploy a Diagnostic Program for every few machines based on the workload. Attack alarms are provided to both users and platform operators when a DoW attack is suspected. They may also take additional actions, such as verifying the DoW attack and halting losses.

Metric Name	Metric Descriptions
LOAD_L3_MISS	Retired load uops missed L3.
LOAD_L3_HIT	Retired load uops with L3 cache hits as data sources.
SPLIT_LOADS	Retired load uops that split across a cache line boundary.
SPLIT_STORES	Retired store uops that split across a cache line boundary.
LOCK_DUR	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.
LLC_MISS	Each cache miss condition for references to the LLC.
LLC_REF	Requests originating from the core that reference a cache line in the LLC.
ICACHE_MISS	Instruction Cache (ICACHE) misses.

Table 2: Selected PCM events and correlated descriptions [47].

Request-oriented sampling. As shown in Figure 7, the traditional method of sampling with a fixed-length window will result in information ambiguity in serverless computing. Gringotts adopts a request-oriented sampling method to overcome these drawbacks for the following three reasons: First, by monitoring the workload in a per-request context, metrics are automatically gathered and aggregated in a lightweight manner. The fluctuation of metrics brought on by function multiplexing is prevented. Second, by keeping track of the request context, Gringotts is aware of the lifecycle of the application logic. This understanding of application logic eliminates a substantial amount of invalid and redundant data, which is difficult to accomplish on IaaS or with fixed-length sampling. Third, requested-oriented sampling can capture the execution pattern of functions, using the request as the pattern. Doing so is highly intuitive, as similar inputs are likely to result in comparable execution.

Metric selection. Table 2 shows the exact metrics chosen from Intel's PCM vector [47] to characterize the resource utilization. Gringotts selects performance metrics according to two primary criteria. First, they are relevant to DoW attacks, as internal DoW

attacks utilize memory bus locking to create resource contentions. Second, they reflect the workloads of applications. These metrics capture the time-consuming instructions of the monitored functions, including cache misses and memory accesses. When memory regions across cache line boundaries are accessed, metrics such as `SPLIT_LOADS` and `SPLIT_STORES` will collect information about underlying memory bus locks. Other metrics, such as `LOAD_L3_MISS` or `LOAD_L3_HIT`, are related to retired cache or memory operations.

5 PERFORMANCE METRICS COLLECTION

In this section, we first briefly introduce the Agent (Sec 5.1). Then we discuss how Gringotts solves the data ambiguity challenge (Sec 5.2) and achieves low overhead (Sec 5.3). Finally, we highlight some additional advantages in the design of the Agent (Sec 5.4).

5.1 Workflow

The key element of Gringotts' design is **request-oriented sampling**. Traditional monitoring systems often adopt the fixed-length sampling method and are consequently incapable of resolving the issue of information ambiguity. In contrast, request-oriented sampling accurately captures the runtime characteristics by integrating the application-level features of request processing with the hardware-level metrics of performance counters.

Figure 9 shows a simplified workflow of Gringotts, annotated with ① - ⑥. When a function is deployed as a result of runtime cold-starts, the Agent instantly launches an initialization to construct the associated data structures (Step ①). After that, the Agent performs PCM event registrations (Step ②). A fresh round of metric gathering is triggered by the arrival of a request (Step ③). The Metric Collector utilizes low-level interfaces to gather performance measurements from the hardware performance counters (HPC) (Step ④). Step ⑤ enqueues the sampled data into a ring buffer. All of the buffered data will be delivered to the diagnostic program in a single, straightforward manner once the request has completed processing (Step ⑥). The aggregated performance metrics are stored in a JSON-style string. For instance, the performance metrics for $Request_i$ of $Function_j$ might be formatted as:

$$\{Req\_id : r_i, \quad Func\_id : f_j, \quad Process\_t : T' - T,$$
$$Metric_1 : v_1' - v_1; \quad Metric_2 : v_2' - v_2; \quad \ldots \quad \}$$

5.2 Function-Metric Coordination

Information ambiguity is produced as a result of multiplexing, runtime destruction, and changes in the behavior of functions in which one kind of metric is mixed with invalid samples or other kinds of metrics. The key idea for Gringotts to effectively address the issue of information ambiguity is to *coordinate the high-level function behaviors while gathering low-level performance metrics.*

To address the multiplexing-related issue, we recommend that metric collection behaviors be bound to the runtime. More precisely, when the context changes, we save or restore the metrics. We enable this by leveraging the `perf_event_open()` syscall of Linux [37], and it is chosen since this implementation provides the best performance. It should be noted, however, that our solution is only interface-compatible with the `perf_events` tool [18]. The three profiling modes of `perf_events` are period/frequency-based

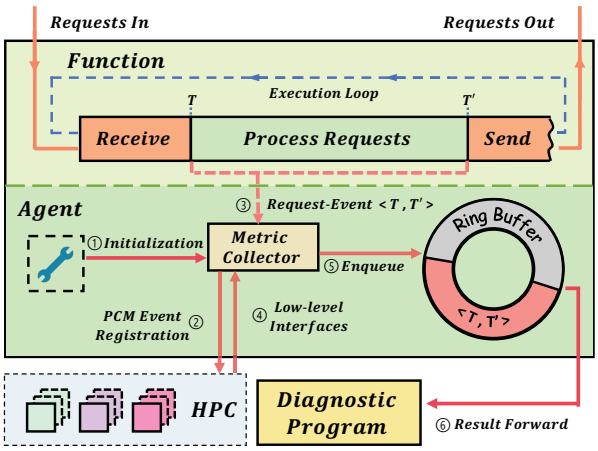


Figure 9: Gringotts' performance metrics collection.

counting, profiling, and tracing. Its target and sampling method are significantly different from those of Gringotts.

To address the issue of runtime destruction, we include an Agent in each runtime. Specifically, we implement this using dynamic library loading, which minimizes the need for program modifications. This ensures that metrics are only collected during the execution of the serverless functions.

To enable proper detection of changes in function behaviors, we creatively leverage the request-oriented nature of serverless architecture. Whenever the runtime receives or sends a request, the Metric Collector module is invoked proactively rather than reactively. In contrast to VMs, which have confusing mapping rules between input, output, and function behaviors, serverless functions receive and send requests in a manner that precisely represents the bounds of function activities. While different requests might cause the function to behave differently, the processing logic is basically comparable when the same request is handled.

5.3 Negligible Performance Overhead

It is not straightforward to perform low-overhead, fine-grained measurements. To accomplish this, we must minimize any unnecessary performance overhead, which influences how the Agent is positioned by Gringotts.

We cannot run the Agent in a process separate from the function being measured, as this would subject the sampling process to the scheduling of the operating system, cross-core HPC accesses, context switching overhead, etc. As a result, Gringotts implements the Agent as a user-level dynamic library that is actively invoked by the function in order to minimize the sampling process's performance overhead. To further optimize sampling efficiency, we read and write performance monitoring counters using user-mode instructions such as `rdpmc`, avoiding system calls that may cause performance issues. Apart from the overhead associated with invoking dynamic libraries within the language (e.g., Python adds an extra $24us$ of overhead), the sampling itself can be performed in less than $1us$.

We construct Gringotts for collecting fine-grained metrics similar to SHIM[75] and [37], as shown in Algorithm 1. At the start of each sampling cycle, Gringotts uses the `RDTSCP` instruction to obtain the

Algorithm 1 Fine-grained metrics collection algorithm

Input: T - Trigger event, M - Metadata, R - Predefined registers
Output: X - Sampled performance metrics

```
1:  if T then
2:      cyc ← RDTSCP()      // Record the timestamp
3:      (Memory Barrier)    // Ordering constraint before reading
4:      for r ← R do
5:          Assert not M[r].multiplexed
6:          x ← rdpmc(r)    // Read the register in user mode
7:          X.push_back(x)  // Store in buffer
8:      end for
9:      (Memory Barrier)    // Ordering constraint after reading
10: end if
11: Return X
```

current timestamp (Line 2). Then, memory barriers [37] are used by Gringotts to impose an ordering constraint on memory operations (Line 3). By reading the metadata, the algorithm guarantees that performance monitoring counters are not multiplexed (Line 5). Next, it accesses the registers predefined by `perf_event_open()` in user mode (Line 6). Finally, the sampled results are instantly stored in a buffer (Line 7), followed by another memory barrier (Line 9).

5.4 Additional Advantages

5.4.1 Event-skid. The difference between the time when the sampling command is issued and the actual sampling action is referred to as the *event-skid*. It can be caused by the scheduling of the operating system, execution delays, or out-of-order execution. Event skid is a typical performance analysis problem, and we demonstrate that it has no effect on Gringotts. To begin with, the collection of metrics occurs in the same user-mode environment as the runtime. Even if the OS schedules a function while it is executing, its metrics are retained without modification during process switching. Second, we utilize memory barriers to inhibit out-of-order execution before and after sampling. Finally, the event skid caused by the execution delay is confined to approximately 10 instructions and has no impact on the results of serverless requests.

5.4.2 Easy-to-use. Gringotts exposes four APIs that allow developers to customize the sampling behavior of the Agent: `init()`, `add()`, `sample()`, and `extract()`. When a process or thread starts, it automatically invokes `init()` to construct the data structure. Then it registers PMC events using the `add()` command. `sample()` is used to perform proactive sampling, while `extract()` is used to extract metrics. This is transparent to the tenants because it is incorporated into the function runtime. Additionally, platform developers can easily override these APIs for certain circumstances.

6 DOW DETECTION MODEL

In this section, we present our major insights of the Diagnostic Program (Sec 6.1). Then we demonstrate the detection model of Gringotts (Sec 6.2).

6.1 HPC-based Workload Verification

Instead of directly detecting whether congestion is occurring on shared resources such as memory buses, the Diagnostic Program of Gringotts is designed to use HPCs to **verify the workload** of the victim functions. Workload verification tackles the problem of attribution difficulty indicated in [35] by adopting the **linear relationship** between execution time and HPCs. The "slowing down" phenomenon caused by DoW can be easily identified if the correlation between execution time and workload has changed.

Gringotts collects the required data from the Agent. It then calculates the incremental data for the metrics between request receiving and response sending. Specifically, $[m_i^1, m_i^2, \cdots, m_i^c]$ represents the changes in the metrics of the function when processing request i, and t_i represents the execution time. We define matrix M as the aggregated metric input, matrix T as the time input, c as the number of selected PCM events, and α as the number of requests.

$$M = \begin{bmatrix} m_1^1 & m_1^2 & \cdots & m_1^c \\ m_2^1 & m_2^2 & \cdots & m_2^c \\ \vdots & & \ddots & \vdots \\ m_\alpha^1 & m_\alpha^2 & \cdots & m_\alpha^c \end{bmatrix}, \qquad T = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_\alpha \end{bmatrix}$$

We do not detect outliers. In the training stage, we presume that all the data is normal. While in the testing stage, outlier data may be the result of an attack. We also do not apply sliding averages to the data. Each piece of data shows the workload associated with processing a request. Sliding averages are in violation of our observation when the inputs are different.

6.2 Detection Model

6.2.1 Assumptions on Application and User Behaviors. For serverless functions and users, we assume the following behaviors:

- The function is activated by user-defined external input, which is consistent with the serverless concept. Even though a function has several triggers, it is not self-executing.
- User inputs to each function conform to the independent identical distribution (i.i.d.). That is, the distribution of inputs remains constant throughout time and is unrelated to one another between invocations.
- When the function is newly deployed, the attack cannot occur instantly. This means that the first α data in each function is normal data that is not under attack.

6.2.2 Detection Model. We highlight DoW detection as a problem that time series exception detection cannot resolve. The underlying rationale is that workloads necessary to finish a request are essentially independent of the prior requests completed. Due to this distinguishing feature of FaaS over IaaS, we are unable to use the previous detection approach [30, 79].

We describe the DoW detection problem as a *multivariate distribution outlier detection* of a function's workload. The workload of a function can be expressed in two ways: its processing time and its content (e.g., the number of instructions completed, the number of LLC misses, etc.). The former can be directly represented by the execution time of the current request, whereas the latter indicates the tasks performed by the function to process the request. Their correspondences are different when the function is in a different state (normal or being attacked). This is intuitive, as when under attack, the function will continue to perform the same task, but its execution time will increase. Thus, the core concept of DoW

detection is to determine if the present workload conforms to the normal distribution at a per-request level.

The overall distribution of the function's workload is learnable, but it cannot be simply expressed as a linear function of LLC misses or references. Rather than that, Gringotts decomposes the workload into a linear combination of the performance metrics that have been chosen. We use $W_i(m_i^j)$ for the denotation of a performance metric's contribution to the function's workload while processing request i.

$$t_i \propto \sum_{j=1}^{c} W_i(m_i^j), \quad W_i(m_i^j) \propto m_i^j$$

At the training stage, the training set is processed with z-score normalization, that is, $m' = (m - \mu_m)/\sigma_m, t' = (t - \mu_t)/\sigma_t$. The transformed data conforms to the Gaussian distribution ($m' \sim \mathcal{N}(0,1), t' \sim \mathcal{N}(0,1)$). Then, we use multivariate linear regression (MLR) to establish the correlation between workloads. Therefore,

$$\hat{T}(\beta, m') = \beta_0 + \beta_1 * m'^1 + \cdots + \beta_c * m'^c, \ \min_{\beta} \left\| M'\beta - T' \right\|_2^2 \quad (1)$$

We use ϵ to denote the MLR prediction error ($\mathcal{E} = \left\{\epsilon | \epsilon = t - \hat{t}\right\}$). Then, putting it all together, we obtain a new multivariate Gaussian distribution:

$$x = (t', m'^1, \cdots, m'^c, \epsilon)^T \sim \mathcal{N}(\mu_x, \sum) \quad (2)$$

μ_x is the N-dimensional mean vector of x and $\sum$ is the $N \times N$ covariance matrix, where N equals $(c+2)$.

At the testing stage, the testing set is also processed with z-score normalization, using μ_m, σ_m, μ_t and σ_t derived at the training stage. Each line in the testing data set corresponds to a point in the high-dimensional space. For each test sample $[\bar{m}, \bar{t}]$, we use the same procedure to construct $m'' = (\bar{m} - \mu_m)/\sigma_m$ and $t'' = (\bar{t} - \mu_t)/\sigma_t$. Then we calculate the prediction error $\bar{\epsilon}$ using the well-trained MLR in (1). Once again, we construct the prediction vector $\bar{x}$ as:

$$\bar{x} = (t'', m''^1, \cdots, m''^c, \epsilon)^T \quad (3)$$

We use the derived $\bar{X}$ to get the distance D between *the error prediction vector of the testing set samples* and *the error prediction center of the training set samples*. We fit the error with a multivariate Gaussian distribution and compute the Mahalanobis distance:

$$D(\bar{X}, \mu_x) = \sqrt{(\bar{X} - \mu_x)^T \sum^{-1} (\bar{X} - \mu_x)} \quad (4)$$

The Mahalanobis distance was chosen over the Euclidean distance since the latter does not account for the connection between strongly correlated variables. In performance measurements, these variables are fairly prevalent. For example, there might be an implicit relationship between `LOAD_L3_MISS` and `LLC_MISS`. Because these variables measure roughly the same feature, the Euclidean distance allocates equal weight to them. In effect, the Euclidean distance gives correlated variables improper weight.

Finally, for a pre-defined threshold $Thres$, a sample of a request is considered normal if and only if its $\bar{x}$ satisfies $D(\bar{x}, \mu_x) \leq Thres$.

7 EVALUATION

In this section, we thoroughly evaluate the performance of Gringotts. We address the following questions through the experiments.

- Is the overhead of Gringotts negligible? (Sec 7.2)
- What advantage does Gringotts have in terms of detection delay over time-series anomaly detection methods? (Sec 7.3)
- How do the parameters of the diagnostic program affect the precision of Gringotts? (Sec 7.4)
- What is the overall prediction accuracy of Gringotts' detection model? (Sec 7.5)

7.1 Experiment Setup

Testbed. We conduct the experiments using a two-node Kubernetes [27] cluster-based testbed (version 1.24.2) with typical configurations [48]. The Kubernetes cluster consists of two identical servers with Intel Xeon E5-2620 v3 CPUs (2.40GHz, 12 physical cores) and 128GB total RAM. The CPU architecture is Haswell-EP/EN/EX, with 8 core PMU generic (programmable) counters and 3 core PMU fixed counters. Ubuntu 20.0.4 is installed on the server, along with kernel version 5.4.0-107-generic.

Baselines. We construct three baselines in order to evaluate the improvements of Gringotts:

- *SDS/B.* SDS/B is a statistical memory DoS detection approach proposed by [30]. It calculates the normal range of the monitored applications based on sliding windows and an exponential weighted moving average (EWMA). If the EWMA values deviate from the typical range for H_c times, SDS/B will trigger an attack alarm. We followed the same parameters as in [30] (i.e., $T_{PCM} = 0.01s$, $W = 200$, $\delta W = 50$, $\alpha = 0.05$, $k = 1.125$, $H_c = 30$).
- *SDS/B-0.* To eliminate the influence of H_c on the final results, we set $H_c = 0$ to measure the detection accuracy of SDS/B itself. Other parameters remain unchanged.
- *Gringotts.* We implement Gringotts using Python (version 3.8.10). The Agent is a Python extension written in C++ and incorporated into a Flask-based example from Knative [25]. The Diagnostic Program is written in Python and utilizes the scikit-learn package [60] (version 0.23.2). Compared with Gringotts, Gringotts-3 will not activate the attack alarm unless it detects three consecutive anomalous values.

Metrics. For detection accuracy, we use recall, specificity, and accuracy as the indicators of our experiments, where T stands for $True$, F for $False$, N for $Negative$, and P for $Positive$.

$$Recall = \frac{TP}{TP + FN}, \qquad Specificity = \frac{TN}{FP + TN}$$

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Function	Description
Blurr	Image blurring function with normalized box filter
LR	The regularized logistic regression function
PCA	The Principal component analysis function
Bayes	The Gaussian Naive Bayes (GNB) function

Table 3: Workloads chosen for serverless functions.

Workload. As listed in Table 3, we develop the victim functions included in our experiments using NumPy [42] (version 1.22.3), scikit-learn [60] (version 0.23.2) and opencv-python [43] (version 4.5.5). We adjust the input parameters in requests uniformly so that the function's execution duration is roughly between $1ms$ and $1s$.

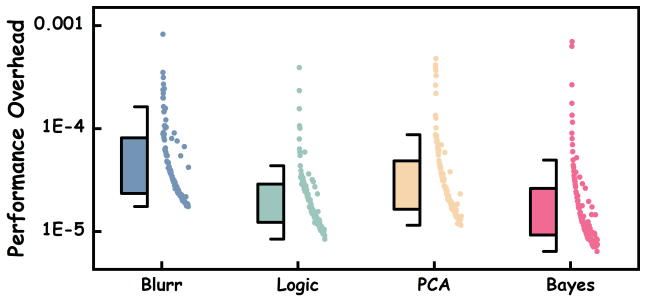


Figure 10: Gringotts' sampling overhead.

The collection of metrics in [30] is accomplished via opcm [46], which is in direct conflict with our sampling approach. So we carried out the experiments using the following method: We compose the sampling results by randomly selecting real-world attacked and unattacked function data and averaging the sampling results. This has no effect on the final results because, in [30], a sliding window of 2*s* and EWMA are used, while the maximum time of function execution is set at around 1*s*.

We consider three attack scenarios. The first two attack scenarios are similar to [30]. In the third attack scenario, we foresee a more sophisticated attack setting under the serverless situation.

Scenario 1. The victim's function ran for 60 minutes. For the first 30 minutes, the victim remains unaffected. We launch the DoW attack between 30 minutes and 60 minutes.

Scenario 2. The victim's function ran for 60 minutes. We launch the DoW attack in a round-robin manner, the duration of which conforms to a uniform distribution between $[10, 50]$ seconds.

Scenario 3. The victim's function ran for 60 minutes. We launch DoW attacks at *any moment* for *any period of time* between requests.

7.2 Performance Overhead

To reduce user-kernel mode switching, scheduling delays, and context switching costs, Gringotts' sampling technique employs a lightweight user-level API, a pro-active sampling trigger, and an embedded sampler. In this experiment, we measured the sampling performance overhead of Gringotts. We did not conduct an attack during the experiment. We thoroughly investigated the sample overhead by changing the inputs, repeating each experiment multiple times, and taking the average to avoid errors.

The performance overhead for Gringotts is depicted in Figure 10 on a per-request basis. The trend in sampled overhead varies nearly similarly for all functions, and the effect of Gringotts on the function's performance being sampled is negligible. The highest performance overhead is below 1.1%, while over 70% of the requests had an overhead of less than 0.002%.

It should be noted, however, that `perf_event_open()` must be executed at the time of each function's setup. If a request is dispatched to a machine that isn't running that function (neither sleeping nor active), the startup of Gringotts will add an additional $\sim 23ms$ to the cold start time on average. Because the loading of the NumPy library alone takes on the order of 100ms at a cold start, we feel this to be a reasonable expense. We believe that, without changing the OS kernel, this is the simplest way to access PMC. That overhead will also be present in other PMC-based implementations.

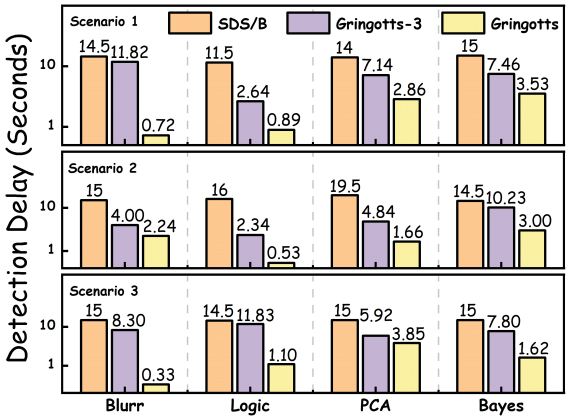


Figure 11: Detection delay comparison.

7.3 Detection Delay

In this experiment, we measured the detection delays of different detection technologies in each of the three scenarios. The findings are shown in Figure 11. The variation in detection delays varies slightly under different settings. Due to the sliding window and Hc, the detection delay of SDS/B will ultimately increase to roughly 15 seconds. This is in line with the results of [30]. The detection delay of Gringotts falls under 3.85 seconds, which is due to the fact that Gringotts detects DoW attacks based on a single request. The detection delay is mostly determined by the execution time of the monitored functions, as the running time of the detection model is insignificant in both of them. Because of the continuous judgment condition, the detection delay of Gringotts-3 increases approximately fourfold compared to the original Gringotts scheme (since it must detect four consecutive positives before assuming the occurrence of a DoW attack), ranging from approximately $3 \sim 12$ seconds.

7.4 Influences of the Factors

We examine the effects of factors on the detection model of Gringotts. In this experiment, we consider the threshold factor and the H_c factor. We demonstrate whether and how these factors will affect the overall detection results.

We quantify the specificity, recall, and accuracy of the four chosen functions. In the recall and accuracy experiments, we set the threshold from 10 to 5000. The threshold is modified between 10 and 110 in the specificity tests because the specificity hits 100% when the threshold is near 100. All of the tests in this experiment are conducted under scenario 3, which is the most severe scenario.

Figure 12 illustrates the constraint that the influence threshold imposes on Gringotts. The specificity of the model is increasing, and it soon reaches 100% for all four functions. This is because, as the threshold is raised, normal samples are no longer estimated as attacked samples. The recall rate peaks when the threshold is set to 10, indicating that all attacked samples are identified. As the threshold climbs progressively, the recall lowers gradually as well. It should be noted that when the threshold is increased to approximately 5000, the recall of function LR drops to 2%. This approximates the distribution of the LR function when it is under a DoW attack. We are not surprised to discover that accuracy

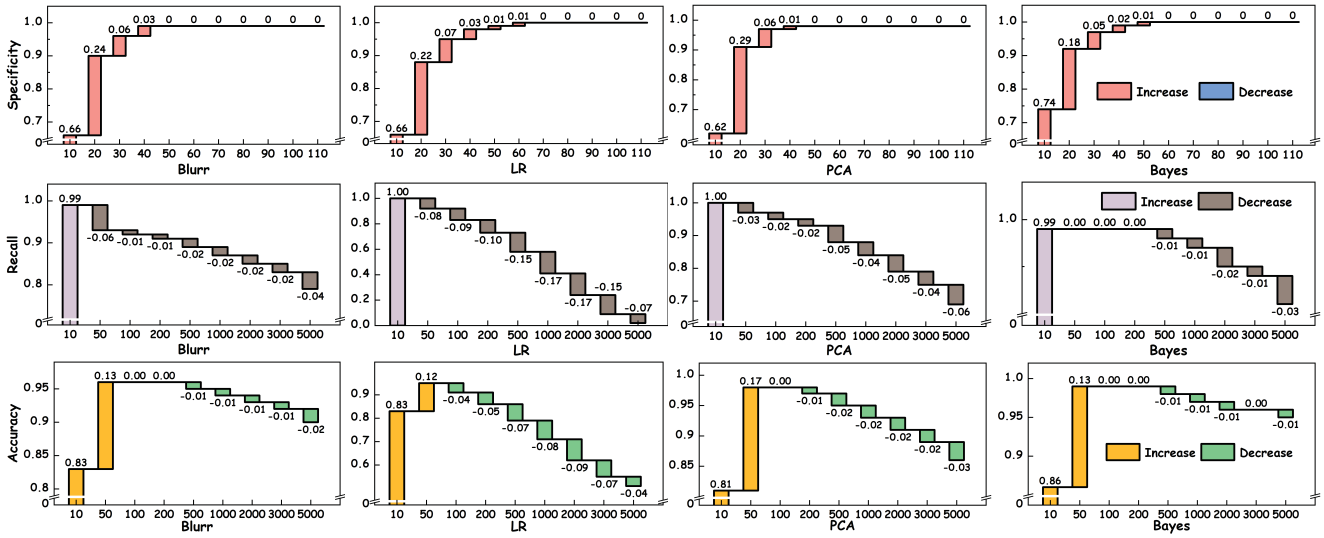


Figure 12: The influence of threshold on Gringotts' detection model. The abscissa is the value of Threshold.

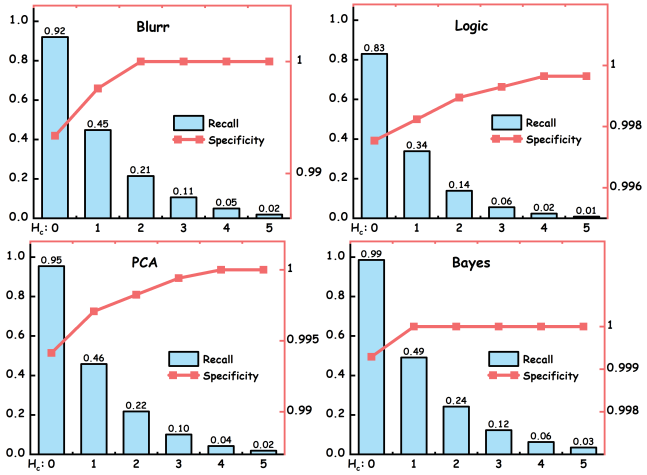


Figure 13: The influence of H_c on Gringotts' detection model.

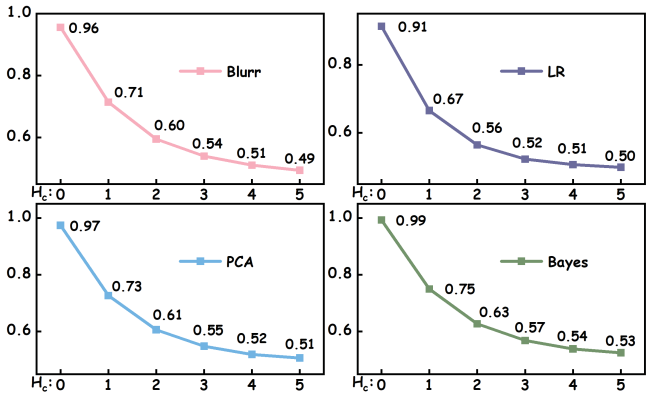


Figure 14: The influence of H_c on Gringotts' detection model.

increases first and later drops. Between the thresholds of 50 and 100, accuracy reaches its peak, ranging from 91.3% to 99.3%. This demonstrates the remarkable accuracy of the detection model in Gringotts.

The detection accuracy of Gringotts is also influenced by factor H_c. Functions are considered to be under a DoW attack only if we have $(H_c + 1)$ successive positive values. Figure 13 illustrates the effect of changing H_c on specificity, recall, and accuracy. We can conclude that when H_c increases, Gringotts does not become more precise. In contrast to time-series anomaly detection, H_c does not improve the final result for request-based detection. As a result, in the end-to-end experiments, we set $H_c = 0$.

7.5 End-to-End Result

Our end-to-end experiments are designed to detect DoW attacks on four selected functions in three scenarios. In this case, we employ SDS/B with a H_c of 0. This is used to determine the extent to which the H_c improves the SDS/B method. We choose accuracy as the metric since it accounts for both false negatives and false positives. The final result is shown in Figure 15.

The first 300 seconds of scenario 1 are comprised of normal data, whereas the attack occurs between 300 and 600 seconds. In scenario 1, all approaches obtain a high level of accuracy. When H_c is set to 30, SDS/B is capable of denoising the sampled data, thus having a higher accuracy than SDS/B-0. After removing the H_c parameter (i.e., SDS/B-0), SDS/B is insufficiently accurate. However, when scenario 2 is considered, the accuracy of SDS/B-0 exceeds that of SDS/B. This is because, unlike with a virtual machine, our workload is selected randomly from a uniform distribution. Even when there is no attack against the request, its performance metrics change on a per-request basis. Not to mention the background noise that makes detection more challenging. The fluctuation becomes more extreme in scenario 3. However, in some cases, this actually compensates for past erroneous estimations caused by request changes and background noise. Thus, the accuracy of the SDS increases in the PCA and Bayes functions. However, regardless of how the scenario changes, Gringotts retains a high level of accuracy, from 91.3% to 99.3%.

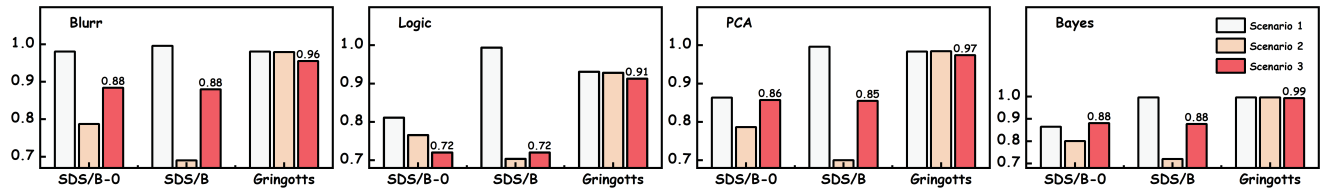


Figure 15: End-to-end evaluation of different models.

Algorithm 2 Covert Channel Implementation

Input: *B* - Number of bits, *is_Sender* - If this instance is a sender

```
WAIT_FOR_TIME_SYNC
current_bit ← 0
id_read ← 0
while current_bit < B do
    if is_Sender then
        WRITE_BIT()
        bit_read ← 1
    else
        bit_read ← READ_BIT()
    end if
    id_read ← 2 * id_read + bit_read
    current_bit ← current_bit + 1
end while
Return is_received_B_bits
```

8 DISCUSSION

Co-residency detection and random DoW attacks. In this paper, we implement the co-residency detection approach similar to the method proposed by [76], which utilizes the memory bus as a covert channel to transmit function information. The sender side of the covert channel will intentionally induce memory bus contentions and transmit B bits to N receivers. When the contention is induced, the latency for non-cached memory accesses will increase and a single bit will be transmitted. If one receiver has received B consecutive bits, it is considered to be co-located with the sender. The covert channel receiver will sequentially operate a continuous memory region and identify latency anomalies. Since both the sender and the receiver reside in the same data center, an attacker can roughly synchronize the behavior of sending and receiving by invoking both at the same time.

For the following three reasons, the number of serverless functions that are concurrently running on the same machine **will not affect the detection accuracy**: First, unlike cache-based techniques, memory-bus-based co-residency detection only requires regular (non-cached) memory accesses and is therefore *unaffected by cache misses* caused by the increasing number of functions [30, 67, 76, 79]. Second, the sender and receiver in the covert channel only occupy a tiny portion of the memory bandwidth, and the present occupancy of the memory bandwidth has no effect on the latency of memory probing. Even without a high memory bandwidth, memory probing can be performed on crowded physical machines. Third, [76] uses time-based synchronous sampling and a variant of the two-sample Kolomogorov-Smirnov (KS) test to further eliminate detection noise.

To enable the co-location of the malicious function and the victim function, attackers can simultaneously start hundreds or thousands of malicious functions, expecting them to co-reside with the victim function. Such methodologies are commonly utilized in investigations of placement vulnerability [55, 67, 73, 76]. Additionally, the short-lived nature of serverless functions enhances the co-location potential of this placement strategy.

Co-residency detection and co-location placement are not the topics of this paper, and both of the co-location placement strategies and co-residency detection [76] are replaceable by any other co-residency detection methods.

In the earlier description (Sec 3.2), we mention that the target functions are a set of functions belonging to a certain user. However, internal DoW attacks can be performed in a blind way without the necessity for co-location detection methods to identify victim functions. As long as some benign functions are running on the same machine as the malicious functions, they are vulnerable to the internal DoW attack. Such an attack may result in platform-level performance degradation and QoS violations, as well as serious collateral damage to affected tenants.

Practicality and feasibility of the internal DoW attack. The internal DoW attacks are practical and feasible for the following four reasons: First, because of the continual launch of new functions and the immediate termination of non-co-located functions, the financial cost to the attacker is negligible, and traditional defenses (e.g., function random placement) are ineffective against this attack. Second, memory bus locking requires no additional privileges, and the hardware-level contention cannot be monitored or controlled by current OS-level tools (such as cgroups). Third, there is no requirement for timing information, and the attack can begin immediately after co-location placement. Finally, attackers can easily leverage the free plans provided by most platforms. A few accounts are sufficient to exhaust the available credit of the victim and throw them into debt.

DoW Mitigation. With the serverless billing model gaining more attention, services such as billing alerts and resource restrictions are currently available on the majority of the serverless platforms [10]. These services, however, are not capable of defending against Denial-of-Wallet attacks. First, when an account has accumulated a huge quantity of charges in a short period of time owing to a DoW attack, the platform will freeze the account or notify the victim. Then the DoW attack is transformed into a primitive DoS attack due to the momentary unavailability of the service before the user may respond. Second, despite the billing alert, customers are still overcharged for a short period of time as a result of their misconfiguration [66]. Third, if the attacker employs a smart approach, the attack should be conducted slowly since, unlike classic DoS attacks, a DoW attack does not necessitate the victim's services being entirely denied. The user's bill will rise slowly over time, while the quality of service will decline steadily.

Metric selection. Based on our experience and knowledge of the internal Denial-of-Wallet attacks, we choose the appropriate hardware performance metrics for Gringotts. In practice, however, selecting indicators is a challenging task. We must guarantee that the metrics accurately reflect the workload of the function while preventing them from being impacted by anything other than the attack. For example, applications operating on other cores that share the same LLC can easily influence the miss and hit rates of the monitored LLC. Although SLOMO [35] provides recommendations for metric selection, it is built on virtualized network functions. In the future, we plan to automatically profile each function separately in order to acquire customized metric sets.

9 RELATED WORK

Serverless security. Serverless security has become a rising problem as a result of the continual growth of serverless applications. From the attackers' perspective, serverless is vulnerable to both of the traditional attacks [1, 5, 21, 56, 77] and the serverless-specific vulnerabilities. Anil Yelam et al. [76] took advantage of the potential co-residency of lambda functions to establish a covert channel between the attacker and the victim. Warmonger [72] claimed that an attacker could taint the shared egress IP pool of a serverless platform in order to launch a platform-wide Denial-of-Service attack against users. From the standpoint of the defenders, VALVE [8] and Trapeze [3] promised to protect against data exfiltration by regulating the information flows of serverless applications. Liang Wang et al. [68] summarized the serverless platform's isolation problem, runtime characteristics, and infrastructure knowledge. The Denial-of-Wallet attacks have been proposed by [24, 38], and they effectively exploit the serverless pay-as-you-go pricing paradigm to convert the Denial-of-Service attack into an economic attack on the victim. However, their definition of "Denial-of-Wallet" only comprises the external DoW attack.

Covert channels and memory DoS attacks. Covert channels are a longstanding problem in the security community [6, 11, 12, 20, 23, 32, 55, 65, 71, 76, 81]. The covert channels make use of shared resources on multi-tenant platforms, which are incapable of being completely isolated by VM/container virtualization. Information is transmitted between tenants in a subtle manner via performance degradation. Memory DoS exploits this congestion even further. Tianwei Zhang et al. [79] performed memory DoS attacks on VMs and used the two-sample Kolmogorov-Smirnov (KS) test [39] for detection. Zhuozhao Li et al. [30] proposed three memory DoS detections with higher specificity, shorter detection delays, and little performance overhead: SDS/B, SDS/P, and DNN. In comparison to previous research, our work is primarily concerned with Denial-of-Wallet attacks in serverless computing. It is more destructive and poses greater detection difficulty on serverless infrastructure.

Performance predictions and resource interference. Numerous studies have been conducted on system performance prediction. SHIM [75] developed a continuous profiler capable of sampling at a resolution of 15 cycles. USAD [4] inferred system abnormal behavior by combining generative adversarial networks (GANs) and an auto-encoder. Silvery Fu et al. [15] examined and assessed several common machine learning techniques for performance prediction. Hansheng Ren et al. [54] suggested utilizing spectral residuals (SR) and convolutional neural networks (CNN) to monitor the time series. [41] assessed the accuracy of different event-based sampling strategies and estimated the effect of some recent advances. SAM [62] used aggregated coherence and bandwidth event counts to distinguish traffic. SLOMO [35] captured the NF pressures and the susceptibility of each NF to performance deterioration. By utilizing holistic resource affinity parameters, RAIE [29] presented a performance prediction model for the non-uniform memory access (NUMA) architecture. Yang Wu et al. presented a temporal provenance generation technique with an experimental debugger named Zeno [70]. However, all of these researches focused on traditional cloud platforms rather than serverless solutions.

10 ETHICAL CONSIDERATIONS

In this section, we discuss ethical considerations in detail. First, with the exception of Section 3, we conduct all of our attack experiments on local servers. Second, we have received permission from the relevant cloud providers (i.e., Amazon, Google, Microsoft, and Alibaba) to conduct the experiments. Under their guidance, we conduct the attacks at the lightest traffic periods. The victim and the attacker are two separate accounts that we control, and we choose the simplest Hello-World application to shorten the duration of the experiment and minimize the possibility of external impact. Finally, compared to earlier studies [76] that used the same technique, our experiment has a smaller impact. We launch fewer functions than [76] and implement our code to limit each memory bus lockdown duration to a few seconds.

11 CONCLUSION

In this paper, we thoroughly analyze the Denial-of-Wallet attack, which is a variant of the Denial-of-Service attack specifically conducted on serverless platforms. We conduct a real-world DoW attack on commercial serverless platforms, and the results indicate that launching a DoW attack on victim functions will lead to a payment increase of more than 37-fold. We implement Gringotts as a real system for potential DoW detection on the serverless platform. Gringotts builds a fast and efficient performance metrics collector to address the data ambiguity of serverless platforms. To eliminate the interference caused by the background noise, Gringotts models the collected performance metrics using a Gaussian distribution. The distance between the data points and the real distribution is then measured using the Mahalanobis distance by Gringotts. Our experiments reveal that Gringotts has a performance overhead of less than 1.1%, an average detection delay of 1.86 seconds, and an average detection accuracy of 95.75%.

ACKNOWLEDGMENTS

We thank our shepherd and anonymous CCS reviewers for their valuable comments. We thank engineers from Amazon, Google, Microsoft, Alibaba, and Shicheng Wang for their genuine help. The research is supported by Joint Research on IPv6 Network Governance: Research, Development and Demonstration (2020YFE0200500) and the National Natural Science Foundation of China under Grant No. 62002009. Han Zhang is the corresponding author.

REFERENCES

[1] Blackhat USA 2017. 2022. Hacking serverless runtimes. Retrieved April, 2022 from https://www.blackhat.com/us-17/briefings/schedule/#hacking-serverless-runtimes-profiling-aws-lambda-azure-functions-and-more-6434

[2] Intel® 64 and IA-32 Architectures Software Developer Manuals. 2022. Retrieved July, 2022 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 118 (oct 2018), 26 pages. https://doi.org/10.1145/3276488

[4] Julien Audibert, Pietro Michiardi, Frédéric Guyard, Sébastien Marti, and Maria A. Zuluaga. 2020. USAD: UnSupervised Anomaly Detection on Multivariate Time Series. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3395–3404. https://doi.org/10.1145/3394486.3403392

[5] Securing Serverless – by Breaking in. 2022. Retrieved April, 2022 from https://www.infoq.com/presentations/serverless-security-2018/

[6] Paizhuo Chen, Lei Li, and Zhice Yang. 2021. Cross-VM and Cross-Processor Covert Channels Exploiting Processor Idle Power Management. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Washington, D.C., 733–750. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-paizhuo

[7] Alibaba Cloud Function Compute. 2022. Retrieved March, 2022 from https://www.alibabacloud.com/product/function-compute

[8] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 939–950. https://doi.org/10.1145/3366423.3380173

[9] Introducing Simplified Serverless Application Deployment and Management. 2016. Retrieved April, 2016 from https://aws.amazon.com/blogs/compute/introducing-simplified-serverless-application-deplyoment-and-management/

[10] AWS Cost Anomaly Detection. 2022. Retrieved April, 2022 from https://aws.amazon.com/aws-cost-management/aws-cost-anomaly-detection/

[11] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, USA, 972–984. https://doi.org/10.1109/ISCA52012.2021.00080

[12] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 843–857. https://doi.org/10.1145/2976749.2978374

[13] Paul Ferguson and Daniel Senie. 2000. rfc2827: network ingress filtering: defeating denial of service attacks which employ ip source address spoofing.

[14] fluentd. 2022. Retrieved April, 2022 from https://www.fluentd.org/

[15] Silvery Fu, Saurabh Gupta, Radhika Mittal, and Sylvia Ratnasamy. 2021. On the Use of ML for Blackbox System Performance Prediction. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, USA, 763–784. https://www.usenix.org/conference/nsdi21/presentation/fu

[16] Google Cloud Functions. 2022. Retrieved March, 2022 from https://cloud.google.com/functions/

[17] Microsoft Azure Functions. 2022. Retrieved March, 2022 from https://azure.microsoft.com/en-us/services/functions/

[18] Brendan Gregg. 2022. Linux perf Examples. Retrieved April, 2022 from https://www.brendangregg.com/perf.html

[19] open source universal RPC framework gRPC. A high performance. 2022. Retrieved July, 2022 from https://grpc.io/

[20] Jawad Haj-Yahya, Jeremie S. Kim, A. Giray Yağlikçi, Ivan Puddu, Lois Orosa, Juan Gómez Luna, Mohammed Alser, and Onur Mutlu. 2021. IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, USA, 985–998. https://doi.org/10.1109/ISCA52012.2021.00081

[21] Gone in 60 Milliseconds: Intrusion and Exfiltration in Serverless Architectures. 2022. Retrieved April, 2022 from https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds#t=253

[22] Jaeger. 2016. Retrieved April, 2016 from https://www.jaegertracing.io/

[23] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE Press, USA, 1–6. https://doi.org/10.1145/2897937.2897962

[24] Daniel Kelly, Frank G. Glavin, and Enda Barrett. 2021. Denial of Wallet - Defining a Looming Threat to Serverless Computing. *CoRR* abs/2104.08031 (2021). arXiv:2104.08031 https://arxiv.org/abs/2104.08031

[25] Knative. 2022. Retrieved March, 2022 from https://knative.dev/

[26] Ron Kohavi and Roger Longbotham. 2007. Online Experiments: Lessons Learned. *Computer* 40, 9 (2007), 103–105. https://doi.org/10.1109/MC.2007.328

[27] kubernetes. 2022. Retrieved July, 2022 from https://kubernetes.io/

[28] Amazon Web Services Lambda. 2022. Retrieved March, 2022 from https://aws.amazon.com/lambda/

[29] Jian Li, Jianmin Qian, and Haibing Guan. 2019. A Holistic Model for Performance Prediction and Optimization on NUMA-based Virtualized Systems. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. IEEE Press, USA, 352–360. https://doi.org/10.1109/INFOCOM.2019.8737447

[30] Zhuozhao Li, Tanmoy Sen, Haiying Shen, and Mooi Choo Chuah. 2022. A Study on the Impact of Memory DoS Attacks on Cloud Applications and Exploring Real-Time Detection Schemes. *IEEE/ACM Transactions on Networking* 30, 4 (2022), 1644–1658. https://doi.org/10.1109/TNET.2022.3144895

[31] likwid. 2022. Retrieved April, 2022 from https://github.com/RRZE-HPC/likwid/

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

[33] Xin Liu, Ang Li, Xiaowei Yang, and David Wetherall. 2008. Passport: Secure and Adoptable Source Authentication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/nsdi-08/passport-secure-and-adoptable-source-authentication

[34] Linux man pages. 2022. cgroups interface. Retrieved March, 2022 from https://man7.org/linux/man-pages/man7/cgroups.7.html

[35] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. 2020. Contention-Aware Performance Prediction For Virtualized Network Functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 270–282. https://doi.org/10.1145/3387514.3405868

[36] AMD64 Architecture Programmer's Manual. 2022. Retrieved July, 2022 from https://www.amd.com/system/files/TechDocs/24592.pdf

[37] Linux manual pages perf_event_open(2). 2022. Retrieved April, 2022 from https://man7.org/linux/man-pages/man2/perf_event_open.2.html

[38] Eduard Marin, Diego Perino, and Roberto Di Pietro. 2021. Serverless Computing: A Security Perspective. *CoRR* abs/2107.03832 (2021). arXiv:2107.03832 https://arxiv.org/abs/2107.03832

[39] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[40] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239

[41] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a Base of Trust with Performance Counters for Enterprise Workloads. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 541–548.

[42] NumPy. 2022. Retrieved April, 2022 from https://numpy.org/

[43] Python OpenCV. 2022. Retrieved April, 2022 from https://pypi.org/project/opencv-python/

[44] OpenFaaS. 2022. Retrieved March, 2022 from https://www.openfaas.com/

[45] Apache OpenWhisk. 2022. Retrieved March, 2022 from https://openwhisk.apache.org/

[46] Processor Counter Monitor (PCM). 2022. Retrieved April, 2022 from https://github.com/opcm/pcm.git

[47] Intel Perfon-Events. 2022. Retrieved April, 2022 from https://perfmon-events.intel.com/index.html?pltfrm=haswell_server.html

[48] Kubernetes Documentation-Configuration Best Practices. 2022. Retrieved July, 2022 from https://kubernetes.io/docs/concepts/configuration/overview/

[49] Alibaba Cloud Function Compute Pricing. 2022. Retrieved March, 2022 from https://www.alibabacloud.com/help/en/doc-detail/54301.htm

[50] AWS Lambda Pricing. 2022. Retrieved March, 2022 from https://aws.amazon.com/lambda/pricing/

[51] Google Cloud Function Pricing. 2022. Retrieved March, 2022 from https://cloud.google.com/functions/pricing

[52] Microsoft Azure Functions Pricing. 2022. Retrieved March, 2022 from https://azure.microsoft.com/en-us/pricing/details/functions/

[53] prometheus. 2016. Retrieved April, 2016 from https://prometheus.io/

[54] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. 2019. Time-Series Anomaly Detection Service at Microsoft. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD*

'19). Association for Computing Machinery, New York, NY, USA, 3009–3017. https://doi.org/10.1145/3292500.3330680

[55] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 199–212. https://doi.org/10.1145/1653662.1653687

[56] CVE-2019-5736: runc container breakout. 2022. Retrieved April, 2022 from https://www.openwall.com/lists/oss-security/2019/02/11/2

[57] AWS Lambda Nodejs Runtime. 2022. Retrieved April, 2022 from https://github.com/aws/aws-lambda-nodejs-runtime-interface-client

[58] AWS Lambda Python Runtime. 2022. Retrieved April, 2022 from https://github.com/aws/aws-lambda-python-runtime-interface-client

[59] AWS Lambda Ruby Runtime. 2022. Retrieved April, 2022 from https://github.com/aws/aws-lambda-ruby-runtime-interface-client

[60] scikit-learn Machine Learning in Python. 2022. Retrieved April, 2022 from https://scikit-learn.org/stable/

[61] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, USA, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[62] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2015. Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 529–540.

[63] Minho Sung, Jun Xu, Jun Li, and Li Li. 2008. Large-Scale IP Traceback in High-Speed Internet: Practical Techniques and Information-Theoretic Foundation. *IEEE/ACM Trans. Netw.* 16, 6 (dec 2008), 1253–1266. https://doi.org/10.1109/TNET.2007.911427

[64] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. An Analysis and Empirical Study of Container Networks. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. IEEE Press, USA, 189–197. https://doi.org/10.1109/INFOCOM.2018.8485865

[65] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. 2021. Invisible Probe: Timing Attacks with PCIe Congestion Side-channel. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Press, USA, 322–338. https://doi.org/10.1109/SP40001.2021.00059

[66] We Burnt $72K testing Firebase + Cloud Run and almost went Bankrupt. 2022. Retrieved April, 2022 from https://blog.tomilkieway.com/72k-1/

[67] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 913–928. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan

[68] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 133–145.

[69] Jianping Wu, Jun Bi, Marcelo Bagnulo, Fred Baker, and Christian Vogt. 2013. Source address validation improvement (SAVI) framework. *RFC7039* 0 (2013).

[70] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

[71] Zhenyu Wu, Zhang Xu, and Haining Wang. 2015. Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks inside the Cloud. *IEEE/ACM Trans. Netw.* 23, 2 (apr 2015), 603–614. https://doi.org/10.1109/TNET.2014.2304439

[72] Junjie Xiong, Mingkui Wei, Zhuo Lu, and Yao Liu. 2021. Warmonger: Inflicting Denial-of-Service via Serverless Functions in the Cloud. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 955–969. https://doi.org/10.1145/3460120.3485372

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

[74] A. Yaar, A. Perrig, and D. Song. 2004. SIFF: a stateless Internet flow filter to mitigate DDoS flooding attacks. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004.* IEEE Press, USA, 130–143. https://doi.org/10.1109/SECPRI.2004.1301320

[75] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with Shim. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 170–184. https://doi.org/10.1145/2872887.2750401

[76] Anil Yelam, Shibani Subbareddy, Keerthana Ganesan, Stefan Savage, and Ariana Mirian. 2021. CoResident Evil: Covert Communication In The Cloud With Lambdas. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 1005–1016. https://doi.org/10.1145/3442381.3450100

[77] Event Injection: Protecting your Serverless Applications. 2022. Retrieved April, 2022 from https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/

[78] Shui Yu, Wanlei Zhou, Robin Doss, and Weijia Jia. 2011. Traceback of DDoS Attacks Using Entropy Variations. *IEEE Trans. Parallel Distrib. Syst.* 22, 3 (mar 2011), 412–425. https://doi.org/10.1109/TPDS.2010.97

[79] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2017. DoS Attacks on Your Memory in Cloud. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 253–265. https://doi.org/10.1145/3052973.3052978

[80] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. https://doi.org/10.1145/3477132.3483580

[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 990–1003. https://doi.org/10.1145/2660267.2660356