Clover: Workload Verification for Real-Time Detection of Contention-Induced Slowdowns in Serverless Platforms

Junxian Shen, Han Zhang, *Member, IEEE*, Weiwei Lin, *Senior Member, IEEE*, Yantao Geng, Jiawei Li, Jilong Wang, *Member, IEEE*, and Mingwei Xu, *Senior Member, IEEE*

***Abstract*—Serverless computing, or Function-as-a-Service, continues to gain popularity due to its pay-as-you-go billing model, flexibility, and cost efficiency. However, these same features introduce significant security risks, such as the Denial-of-Wallet (DoW) attack. In this paper, we conduct real-world DoW attacks on commercial serverless platforms to evaluate their severity. To detect such attacks, we design, implement, and evaluate Clover, an accurate and user-friendly DoW detection system with negligible performance overhead. Clover addresses information ambiguity in serverless environments by deploying a request-oriented metric collection agent. At its core, Clover proposes a workload verification approach to bridge performance metrics and execution duration. Specifically, Clover uses a multivariate linear model to learn the benign relationship between metrics and execution duration, effectively characterizing normal workload behavior. It then continuously monitors runtime workloads by calculating their Mahalanobis distance from this learned benign model. Deviations identified through this distance indicate potential DoW attacks. Implemented as a practical system, Clover introduces performance overhead of less than 3.2%, maintains an average model execution time of only 0.84 microseconds, and achieves an accuracy of 92.7% under the most challenging scenario.**

***Index Terms*—Serverless computing, cloud computing security, denial-of-wallet attack.**

Received 21 September 2025; revised 26 December 2025 and 25 February 2026; accepted 27 March 2026; approved by IEEE TRANSACTIONS ON NETWORKING Editor X. Luo. Date of publication 3 April 2026; date of current version 21 April 2026. This work was supported in part by Guangxi Key Research and Development Project under Grant 2024AB02018, in part by Guangdong Provincial Natural Science Foundation Project under Grant 2025A1515010113, in part by Shandong Provincial Natural Science Foundation Project under Grant ZR2024LZH012, and in part by the National Natural Science Foundation of China under Grant 62572269 and Grant 62394322. An earlier version of this paper was presented at the conference of CCS 2022 [DOI: 10.1145/3548606.3560629]. *(Corresponding author: Han Zhang.)*

Junxian Shen is with the School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China (e-mail: shenjx@scut.edu.cn).

Han Zhang, Yantao Geng, and Jiawei Li are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China (e-mail: zhhan@tsinghua.edu.cn; gyt22@mails.tsinghua.edu.cn; li-jw19@mails.tsinghua.edu.cn).

Weiwei Lin is with the School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China, and also with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: linww@scut.edu.cn).

Jilong Wang and Mingwei Xu are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China (e-mail: wjl@cernet.edu.cn; xumw@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TON.2026.3680829

I. INTRODUCTION

SERVERLESS computing, also known as Function-as-a-Service (FaaS), continues to evolve rapidly since its inception. Under this paradigm, tenants delegate responsibilities such as task deployment, environment configuration, and auto-scaling to the platform. Increasing numbers of companies, especially startups, are migrating their services to serverless platforms due to their flexibility and pay-as-you-go billing model.

Although these advantages differentiate FaaS from Infrastructure-as-a-Service (IaaS), they also expose tenants to traditional as well as novel security threats. One such threat is the Denial-of-Wallet (DoW) attack, a specialized variant of the Denial-of-Service (DoS) attack targeting serverless platforms. DoW attacks fall into two categories: external and internal DoW attacks. External attackers drain financial resources by repeatedly invoking publicly exposed victim APIs, while internal attackers create resource contention within victim function instances (e.g., containers or VMs), causing slowdowns. In this paper, we focus on the detection of internal DoW attacks, as existing technologies such as ingress filtering [1], traceback [2], and source validation [3] can adequately detect external DoW attacks.

Similar to DoS attackers, internal DoW attackers aim to gain benefits through malicious industrial competition, extortion, or hacktivism. By deliberately delaying the execution of victim functions, internal DoW attacks can produce two direct consequences. First, victims will face significant financial losses due to the increased platform costs arising from prolonged function executions under the pay-as-you-go billing model. For instance, one startup reported incurring 72,000 in unexpected charges and nearly declared bankruptcy because of misconfigured serverless functions [4]. Second, the victim's critical services are severely disrupted by the extended execution duration. Amazon, for example, reports a 1% loss in sales for every 100ms increase in page load time [5].

To detect internal DoW attacks, platforms must carefully monitor and precisely analyze hardware resource utilization on specific bare-metal machines. However, fast and accurate internal DoW detection remains challenging due to two critical factors. First, improper sampling of performance metrics may introduce detection inaccuracies. Second, there is a considerable semantic gap between low-level performance metrics and high-level function execution status.

2998-4157 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

TABLE I
CURRENT BILLING MODELS OF MAJOR COMMERCIAL SERVERLESS PLATFORMS

Commercial Serverless Platforms	The Pricing Unit for Duration Costs and Resources Reserved (κ_1)	The Fixed Pricing Unit per Million Requests (C_1)	Other Costs
AWS Lambda [6]	$\$1.667 * 10^{-5}$/GB-s	\$0.20	Network + Resource costs
Google Cloud Function [7]	$\$2.5 * 10^{-6}$/GB-s + $\$1.0 * 10^{-5}$/GHz-s	\$0.40	Network + Deployment costs
Azure Functions [8]	$\$1.6 * 10^{-5}$/GB-s	\$0.20	Network + Storage costs
Alibaba Function Compute [9]	$\$1.6384 * 10^{-5}$/GB-s	\$0.20	Network costs

Unfortunately, existing methods fail to address these challenges effectively. We classify current performance analysis tools into two categories based on their data collection methods. The first category collects application-level metrics, represented by tracing tools such as Jaeger [10] and logging tools like Fluentd [11]. However, these tools cannot capture essential low-level information like bus-locking events or cache misses, making them ineffective for DoW attack detection. The second category targets hardware-level metrics, exemplified by tools like opcm [12] and perf [13]. Although these tools can obtain detailed hardware information, they typically sample metrics at fixed-length intervals. Moreover, lacking insights into application behaviors, these methods struggle to differentiate between normal workload variations and malicious attacks.

Clover. Facing these challenges, we design, implement, and evaluate a fast and accurate internal DoW detection system. It collects performance metrics in a request-oriented manner and detects DoW attacks in real time through workload verification. In summary, our **contributions** include:

First, we comprehensively analyze the DoW attack, including its definition, underlying mechanism, attack procedure, practicality, and feasibility (Sections II and III). We then conduct a real-world internal DoW attack on commodity serverless platforms to illustrate its significant harm and analyze its characteristics. We also discuss the detection challenges posed by internal DoW attacks on serverless platforms.

Second, we introduce Clover (Contention-induced slowdown verifier), which includes a data collection agent (Section V) and a detection model based on workload verification (Section VI). The agent employs a metric collector whose lifecycle closely matches the monitored function, enabling Clover to precisely align hardware performance metrics with function behaviors. The detection model introduces the concept of "workload," which is the linear correlation between performance metrics and request execution time. Clover learns the normal workload distribution during a brief, attack-free period immediately after function deployment. By comparing real-time collected metrics with this learned distribution, Clover accurately detects internal DoW attacks by identifying abnormal slowdowns under similar workloads.

Third, we implement Clover as a real system and evaluate it extensively using a testbed (Section VII). Our experiments demonstrate that Clover imposes negligible metric collection overhead of less than 3.2%, maintains an average model execution time of only 0.84 microseconds, and achieves an average detection accuracy exceeding 92.7%, even in the most challenging scenario.

II. BACKGROUND

A. Serverless Computing and Billing Models

As an emerging cloud programming paradigm, serverless computing derives its attractiveness from its billing model, the facilitation of deployment, the seemingly infinite scalability, and competitive performance. Specifically, its billing model is based on the idea of "pay-as-you-go" and is widely adopted by cloud providers like AWS [14], Google [15], Microsoft [16], and Alibaba [17]. As shown in Table I, the total cost B_f of a function f over a specified period is given by

$$B_f = R_f \kappa_1 \sum_{i=1}^{N} T_i + \kappa_2 C_0 + N C_1 + C_2 \tag{1}$$

$$= R_f \kappa_1 \sum_{i=1}^{N} \sum_{j=1}^{M} \frac{I_j}{\sigma_j} + \kappa_2 C_0 + N C_1 + C_2 \tag{2}$$

Here, R_f indicates the resources allocated by the tenant, κ_1 is a pricing unit based on the execution duration and allocated resources, and κ_2 is a pricing unit for bandwidth usage. C_0 denotes the bandwidth consumed, N is the number of requests processed, C_1 represents a fixed pricing unit per request, and C_2 is a fixed charge covering deployment costs. The most important component of this model is the **duration costs** (i.e., $R_f \kappa_1 \sum_{i=1}^{N} T_i$), which start to accumulate 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 $1.667 * 10^{-5}$ dollars for duration costs.

This model gives FaaS platforms a pricing advantage over IaaS. For tenants with highly variable workloads, choosing FaaS platforms is more economical than leasing VMs based on peak demands. However, tenants risk being overcharged if their functions slow down or remain idle during execution. Functions may yield their allocated CPUs either actively (e.g., by sleeping) or passively (e.g., waiting for I/O operations). Experiments show that once CPUs are yielded on commodity platforms, they can execute other tasks, yet tenants continue being charged until their functions return responses. In Equation 2, we decompose the execution duration, T_i, of a single request into the cumulative time required to complete individual tasks. The execution time for each task is calculated by dividing its workload, I_j, by its execution speed, σ_j, where j ranges from 1 to M, and M denotes the total number of tasks. Attackers can intentionally trigger resource contention to reduce σ_j, thereby extending execution durations and causing financial losses to victims. We term this type of attack an internal Denial-of-Wallet attack (Section III-A).

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

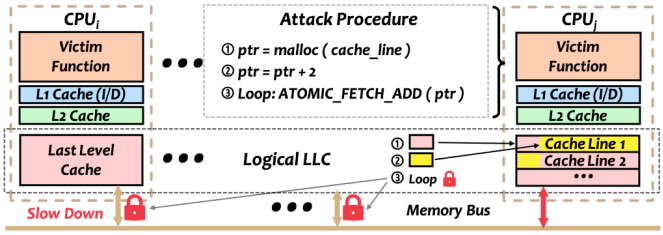

Fig. 1. Overview of memory bus locking.

B. Memory Bus Locking

There are many methods to induce contention on FaaS platforms, and memory bus locking is one of them. As illustrated in Figure 1, it is a specific mechanism introduced by modern processors to ensure the atomicity of certain instructions. When an atomic memory operation crosses a cache line boundary, the hardware temporarily suspends other memory operations to adhere to cache coherence protocols [18].

Memory bus locking is permitted on most cloud platforms and can be easily implemented without additional privileges. Atomic memory operations, such as ATOMIC_FETCH_ADD, are frequently employed in multi-threaded applications for communication over shared resources. Even language runtimes, such as Python, implicitly utilize these instructions. Ordinary users may perform these operations either explicitly in their functions or implicitly through runtime implementations. Given the widespread use of these operations, cloud providers cannot easily mitigate such vulnerabilities through simple patches, such as prohibiting all atomic operations. Furthermore, this issue affects all x86 processors, including those from Intel [18] and AMD [19]. OS-level tools (e.g., cgroups) are currently unable to detect or manage the hardware-level congestion caused by these instructions.

To demonstrate that major cloud platforms remain vulnerable to memory bus locking, we conducted a simple test on the serverless platforms of AWS, Google, Azure, and Alibaba. Because memory bus locking is ISA-level and independent of provisioned CPU share or memory, we used the same per-instance configuration across platforms: 1 vCPU and 2 GB of memory (except on AWS, where 1 vCPU corresponds to 1,769 MB of memory). All of our tests ran on x86-64 and covered mainstream server-grade x86 processors, spanning Intel Xeon CPUs from the Haswell, Skylake, and Ice Lake generations (Family 6 Models 63, 85, and 106) and AMD EPYC Zen-class CPUs (e.g., Family 25 Model 17). During the test, we repeatedly executed an 8-byte atomic operation at increasing offsets within a cache line. When the operand straddled a cache-line boundary (offsets 1–7), the CPU had to acquire a bus lock to preserve atomicity. We launched 20 instances on each platform and report the resulting latency inflation for one arbitrarily selected instance per platform in Figure 3.

Memory bus locking via split-locked atomics is specific to x86 architecture. Nevertheless, **the vulnerability remains widespread in practice** for two reasons. First, x86 remains the dominant execution substrate on mainstream serverless platforms. For example, a recent study [20] attributes all 2,020 sampled invocations on AWS Lambda to x86 CPU models (Intel Haswell-class Xeons and AMD EPYC). In addition, platforms such as Google, Azure, and Alibaba *only* provide x86 serverless environments, thereby effectively restricting the architectural scale. Second, ARM-based architectures do not eliminate contention-driven internal DoW attacks. Analogous mechanisms, such as cache-bank contention, have been demonstrated on multi-core ARM processors [21].

C. Performance Background Noise

VMs and OS-enforced resource limits provide strong isolation, but commodity FaaS platforms still exhibit substantial background performance noise. This noise can reduce the accuracy of performance monitoring models. To illustrate this phenomenon, we measured the execution duration of a simple Python "Hello World" function on AWS Lambda and Alibaba Function Compute. We tested three representative resource tiers per platform (*small*, *medium*, and *large*) for six deployments in total. On AWS Lambda, CPU entitlement scales with the configured memory size. Since 1,769 MB of memory corresponds to approximately one vCPU [14], we used 128 MB (small), 1,769 MB (medium), and 10,240 MB (large). Alibaba enforces similar CPU–memory constraints [17]. Accordingly, we configured 0.05 vCPU with 128 MB (small), 1 vCPU with 1 GB (medium), and 16 vCPU with 32,768 MB (large). We invoked each deployment periodically over four consecutive days. We reported duration metrics that reflected in-platform execution time and were therefore insensitive to client-side networking effects and gateway congestion.

Overall, we launched five independent instances per tier on each platform, for 30 instances in total. For clarity and readability, we plotted one arbitrarily chosen instance per tier on each platform. As shown in Figure 2, both platforms exhibited pronounced periodic "tides" in execution duration. Within a single day, the max-to-min ratio exceeded 6× on AWS and reached 17× on Alibaba. We acknowledge that this experiment covered only a small fraction of each platform's vast configuration space. For example, Lambda offers memory configurations from 128 MB to 10,240 MB in 1-MB increments, and Alibaba supports thousands of valid vCPU–memory combinations. Nevertheless, recent studies [22], [23], [24] on production serverless platforms have reported similar patterns and substantial performance variance, which further strengthen the generality of our observation.

III. Denial-of-Wallet Attack

A. What Is a Denial-of-Wallet Attack

We define the Denial-of-Wallet (DoW) attack as *a variant of the Denial-of-Service attack specifically targeting serverless platforms*. Generally, DoW attackers exploit traditional DoS techniques, such as request flooding or resource contention, to pressure target tenants. Nevertheless, there are three fundamental differences between DoW and DoS attacks. First, although both types of attack drain victims' resources, DoS attacks aim to exhaust resource limits, whereas DoW attacks specifically exploit the pay-as-you-go billing model. Second, DoS attacks often cause complete service unavailability, while DoW attacks typically leave services accessible to legitimate users. Finally, the two attacks have distinct financial implications: DoS attacks typically result in business interruptions,

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

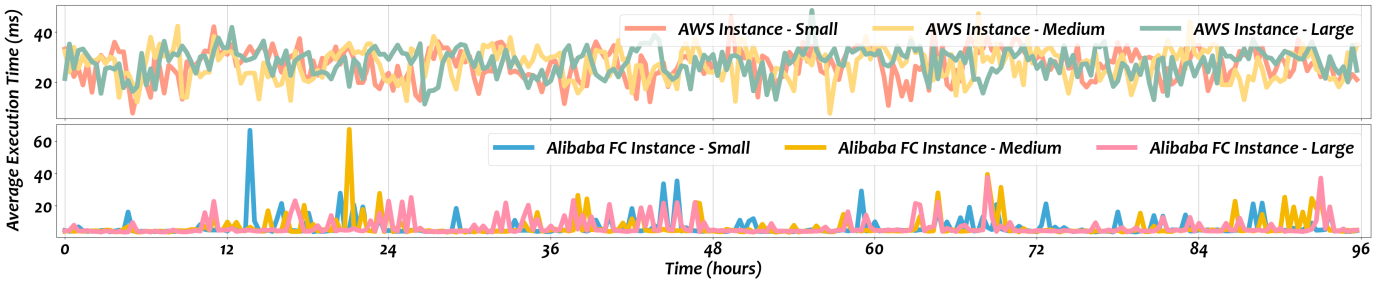

Fig. 2. Ubiquitous background noise in AWS Lambda [14] and Alibaba function compute [17].

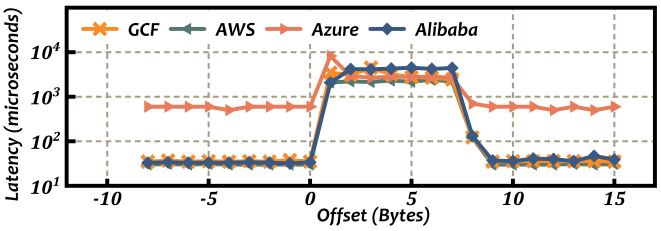

Fig. 3. Latency spikes caused by atomic operations.

reputation damage, and customer loss, whereas DoW attacks directly impose financial charges on tenants, as victims must pay cloud providers for consumed resources.

DoW attacks can be categorized into two types based on the origin of malicious resource consumption: **external DoW attacks** and **internal DoW attacks**. External DoW attacks exploit victims' publicly exposed service or resource APIs [25], [26], generating large volumes of invocations. Referring back to Equation 2, we observe that these attacks specifically aim to increase either the number of requests (N) or the workload per task (I_j). Currently, such attacks can be effectively mitigated by existing DoS detection methods, including ingress filtering [1], traceback [2], and source validation [3]. External DoW attacks are outside the scope of this paper.

B. Threat Model

In this work, we consider a typical commercial attack scenario with three entities: (i) **A multi-tenant FaaS platform**, where functions from different tenants may be co-located on the same physical machine. We assume the platform is trusted and does not collude with the attacker. (ii) **Victim tenants** that run production workloads under a pay-as-you-go billing model. Typical possible victims include enterprises that can deploy up to tens of thousands of function instances [27], [28], as well as budget-constrained startups and individual developers that adopt serverless to reduce financial burden [4]. Also, we define **the victim functions** as the set of functions belonging to a specific victim tenant. (iii) **An unprivileged attacker** who acts as a regular user of the platform and is motivated by malicious industrial competition, extortion, or hacktivism. We assume the attacker has already chosen the victim functions and aims to conduct an internal DoW attack. Before launching the attack, the attacker must place malicious functions on the same physical machines as the victim functions (i.e., function co-location). Prior recent studies [29], [30], [31] show that co-location can be achieved quickly (i.e., in a few seconds), accurately (i.e., with over 90% coverage), and at low cost (i.e., a few to tens of dollars) using only unprivileged mechanisms. Note that serverless function co-location is beyond the scope of this paper.

C. How to Launch an Internal DoW Attack

Internal DoW attacks are carried out by inducing resource contention on shared hardware. After selecting the victim functions, an attacker can launch such an attack in three steps:

Step 1: Malicious function co-location. To induce effective hardware resource contention, the attacker must first achieve function co-location. In practice, the attacker may deploy hundreds of functions as a regular tenant, aiming that some will be scheduled onto the same physical machine as the victim. The attacker then applies existing co-location detection methods to identify co-resident instances. In this paper, we adopt a co-location detection technique similar to that in [31].

Step 2: Create resource contention. Attackers then invoke the co-located malicious functions to create resource contention via memory bus locking (Section II-B). On Intel x86 architectures [18], memory bus locking can be easily triggered using language-level atomic operations such as Interlocked.Increment in C#, atomic.AddUintptr in Golang, and ATOMIC_FETCH_ADD in C++. These operations translate into machine instructions that either activate locking protocols (e.g., `XCHG`) or include the LOCK# prefix (e.g., `XADD`). While we specifically employ memory bus locking in this paper, similar hardware-level resource contentions involving PCIe I/O switches [32], caches [33], [34], [35], and power management units [36], [37] have been demonstrated as equally effective, simple to execute, and challenging to mitigate.

Step 3: Direct financial losses. Once the resource contention begins, the pay-as-you-go billing model converts the prolonged execution time of functions directly into financial losses for the victim. Since resource allocations remain unchanged while execution durations increase significantly, the victim's incurred costs rise correspondingly due to the attack. However, during this period, users still retain access to the affected functions.

D. Internal DoW Attack Example

To demonstrate the practicality of internal DoW attacks, we conducted real-world experiments on AWS [14], Microsoft [16], and Alibaba [17], following the three-step procedure outlined above. For simplicity, we implemented a Hello-World function as the victim, deploying attacker and victim functions

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

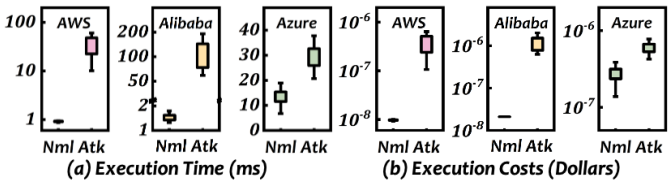

Fig. 4. Impact of the internal DoW attack. *Nml* denotes normal execution, whereas *Atk* denotes execution under internal DoW attack.

from two separate accounts under our control. Note that our experimental results were unaffected by the performance fluctuations shown in Figure 2, as those variations occur on a scale of minutes or hours, while our experiments concluded within a few minutes. We also noted that commodity FaaS platforms schedule a function when its instance reserves fewer than 1 vCPU, which prevents the attack from running continuously. Therefore, the instance hosting a malicious function had to reserve at least 1 vCPU. In our experiments, we provisioned the malicious functions to meet this 1-vCPU requirement and each platform's CPU–memory constraint. On each of the three platforms, we launched 200 malicious functions, and each malicious function ran once (20 seconds) to detect co-location. On AWS, we configured each malicious function with 1 vCPU and 1769 MB of memory, costing approximately $0.12 during co-location. On Alibaba, we configured each function with 1 vCPU and 1 GB of memory, costing approximately $0.06 during co-location. Azure provides a unique consumption plan; under this setting, each function is limited to 1.5 GB of memory and 1 vCPU, and the co-location on Azure costs approximately $0.1. After the attack began, we retained *only one* of the malicious functions co-located with the victim function. Each attack lasted about five minutes, and the maximum cost was $0.009 (on AWS).

Figure 4 illustrates the significant impact of internal DoW attacks on execution duration and costs across three platforms. On AWS, the victim function's execution duration increases dramatically from 0.93ms to 34.44ms (37.03×), directly elevating execution costs. Functions on Alibaba experience an even greater increase, with durations rising sharply from 1.45ms to 114.37ms (78.88×). In contrast, Azure Functions shows a more modest rise, from 13.71ms to about 35ms, primarily due to Azure's CPU allocation policy [16].

Attack motivation and practicality. An internal DoW attack is inexpensive. First, function co-location is inexpensive. Each co-location trial requires only one function deployment and one invocation. The invocation is used to collect physical-machine-level signals (e.g., memory/cache/clock fingerprints) to test whether co-location is achieved. The invocation is short and typically lasts only a few to tens of seconds. If a trial fails, that physical machine does not need to be tested again. Furthermore, deployment-diversification strategies (e.g., launching many attack functions at the same time) can help spread functions across physical machines. In practice, co-location can usually be achieved within a few thousand attempts. In the above example, we achieve co-location at a cost of at most $0.12, while recent work [29] reports covering a much larger fleet (230 instances on average) with a maximum cost of 25. With deployment-diversification strategies, attackers can co-locate with victim functions on more than 900 physical machines for 23 [30]. Second, sustaining internal DoW is also inexpensive. After co-location succeeds, the attacker only needs to keep **one** successfully co-located malicious function running. This malicious function itself requires only 1 vCPU and the minimum memory allowed by the platform. Even if that function runs continuously for one month, the cost is only about 76. Moreover, attackers can further abuse free-tier quotas across zombie accounts. Major serverless platforms offer a monthly free quota of 400,000 GB-s per account, and such abuse has been observed in practice [38].

DoW's inexpensive resource manipulation can severely disrupt regular services. Unlike DoS, which typically requires sustained, high-volume, and complex traffic (often at Gbps or even Tbps), DoW can be sustained by invoking only one function at tens of requests per second. Correspondingly, DoS costs are driven by traffic generation and delivery, whereas DoW costs are mainly driven by function execution. More importantly, malicious functions can run with minimal resources, whereas victim functions often reserve substantially more. For example, Huawei reports that about 50% of their functions reserve at least 2 GB of memory [39]. Similarly, recent IBM data shows that 42% of functions are configured with 4 GB of memory, and some reach 8 vCPUs and 48 GB of memory [40]. If an attacker uses a 1 vCPU/1 GB malicious function against a victim function configured with 8 vCPUs and 48 GB of memory, the direct amplification factor is 384×. Currently, there is no reliable method for accurately obtaining another tenant's CPU and memory configuration. Therefore, attackers might identify critical functions in two ways: by obtaining configuration information through social engineering or sandbox escape, or by probing multiple victim functions and selecting the one that causes the largest increase in end-to-end latency. This latency inflation can propagate and amplify along the function call chain, leading to additional economic loss. For example, Google reported that a 500 ms increase in page-load latency reduced website traffic by 20% [41].

E. Challenges in Accurately Detecting Internal DoW Attacks

DoW attacks maliciously exploit shared hardware resources among co-located tenants. Therefore, detecting such attacks requires platforms to closely monitor resource usage patterns on bare-metal machines. However, the transient and highly dynamic nature of serverless workloads presents two significant challenges for accurately detecting internal DoW attacks.

Limitations of fixed-interval sampling. Previous studies on memory DoS detection and resource interference prediction typically collect performance metrics at fixed intervals [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52]. While straightforward and effective on IaaS platforms, this approach causes data invalidation issues on FaaS platforms for three key reasons. First, functions are short-lived, as they are frequently created, executed, and destroyed within a brief period. Second, FaaS platforms often multiplex multiple functions from one or more tenants onto a single CPU core, making it challenging for fixed-length sampling to identify which metrics belong to which function. Third, even if a function's process remains unchanged, its execution pattern may still vary significantly when handling different requests [53]. To demonstrate these issues, we conduct simulations using a trace dataset collected

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

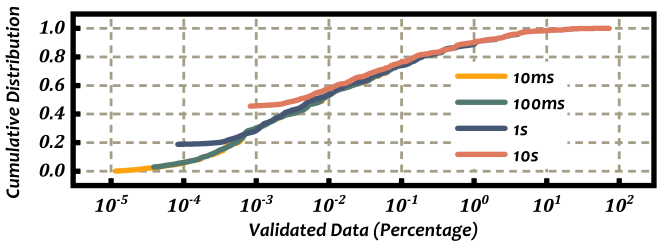

Fig. 5. The proportion of valid samples collected from the Azure dataset [54].

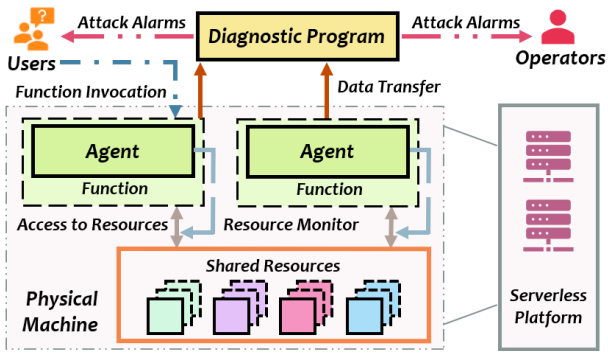

Fig. 6. The architecture of Clover.

from Microsoft Azure Functions [54], as shown in Figure 5. We calculate the cumulative fraction of invalid data for each function under various fixed sampling windows (from 10ms to 10s). The results indicate that over 80% of functions have a valid data rate below 20%.

Challenges in mapping hardware metrics to function execution status. Even with accurate, function-specific hardware metrics, precisely detecting internal DoW attacks remains challenging. First, there is a considerable semantic gap between low-level hardware metrics and high-level function execution status, especially in serverless environments. Unlike prior IaaS-focused studies, we cannot simply perform statistical tests on a single metric (e.g., LLC misses) to identify internal DoW attacks. Instead, we need a detection model that can clearly understand how different hardware metrics relate to real function execution duration, ensuring greater interpretability and precision. Second, the serverless environment is inherently noisy, and function workloads are frequently changing. Therefore, our detection model must effectively adapt to such highly dynamic and rapidly fluctuating scenarios.

IV. System Overview

Architecture. Clover comprises two main components: (i) an **agent** and (ii) a **diagnostic program**. Figure 6 presents a simplified overall architecture of the Clover system.

The Clover agent monitors hardware resource usage of target functions. The platform injects it into each function's runtime and deploys it alongside other components (e.g., HTTP request handling, logging). Upon every invocation, the agent automatically collects hardware performance metrics. It records these metrics per request and forwards them to the diagnostic program. The agent is orchestrated by the platform and shares the lifecycle with the user functions.

TABLE II
Selected PCM Events and Correlated Descriptions [55]

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.

The diagnostic program is a real-time and cluster-level internal DoW attack detection service. It comprises a simple detection model (Clover-S) based on a single-distribution assumption and a more sophisticated detection model (Clover-M) closely tailored to real-world serverless operating environments. In practice, we adopt the Clover-M model by default for real-time per-request detection. When a potential DoW attack is identified, both users and platform operators immediately receive clear attack alerts. They may also perform additional actions, such as verifying the suspected DoW attack, isolating suspicious functions promptly, and effectively mitigating potential financial losses and related risks.

Hardware performance metric selection. Table II summarizes the metrics chosen from Intel's architecture [55] to characterize resource utilization. Clover selects these metrics based on two main considerations. First, metrics must be closely associated with internal DoW attacks, as such attacks leverage memory bus locking to trigger resource contention. For example, when accesses cross cache line boundaries, metrics like SPLIT_LOADS and SPLIT_STORES effectively capture underlying memory bus locking events. Second, selected metrics should accurately reflect the workload characteristics of functions by capturing performance-critical operations, such as cache misses and memory accesses. Metrics such as LOAD_L3_MISS and LOAD_L3_HIT, for instance, clearly indicate retired cache and memory instructions, providing valuable insights into workload behavior.

V. Performance Metric Collection

A. Workflow

The core innovation of the Clover agent is **request-oriented sampling**. To clearly illustrate this feature, we show the workflow of performance metric collection in Figure 7. When a function instance is deployed onto a new physical machine, the serverless platform first retrieves the corresponding function runtime from its repository. Together with this runtime, our embedded agent is also downloaded onto the machine. Once the function runtime is successfully created, the agent immediately initializes itself by constructing the necessary data structures (Step ①). Next, the agent registers hardware counters associated with the selected performance metrics (Step ②). Whenever the function runtime *receives or sends*

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

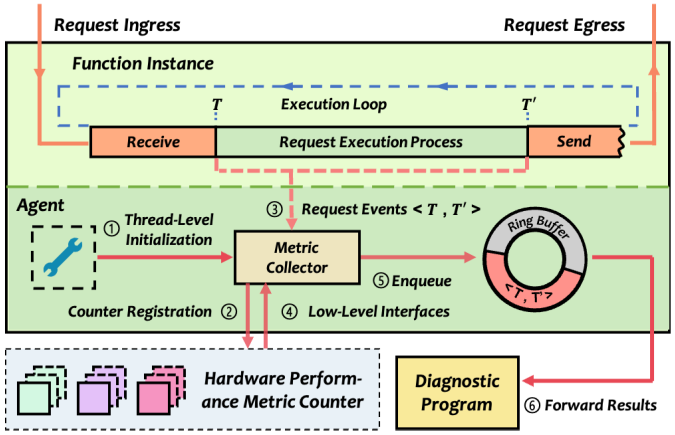

Fig. 7. The metric collection procedure of Clover agent.

a user request, the agent is activated, triggering the metric collector (Step ③). The metric collector then accesses low-level interfaces to gather performance metrics directly from the registered hardware counters (Step ④). The collected data is promptly stored in a ring buffer (Step ⑤). Finally, after the request processing completes, all buffered data is delivered to the diagnostic program for further analysis (Step ⑥).

B. Function-Metric Coordination

Data invalidation mainly results from function multiplexing, runtime destruction, and workload fluctuations. In these scenarios, hardware metrics corresponding to a particular request's execution of a specific function may become mixed with invalid data or metrics from other requests or functions. By adopting request-oriented sampling, Clover effectively addresses this issue by coordinating high-level function behaviors with low-level performance metric collection.

To address the multiplexing-related issue, we bind metric collection behaviors directly to the runtime. Specifically, we save or restore metrics whenever the execution context changes. We achieve this by leveraging the Linux perf_event_open() interface [56], selected for its optimal performance. However, it is important to note that our approach is only interface-compatible with the perf_events tool [13]. The three profiling modes offered by perf_events - period/frequency-based counting, profiling, and tracing - differ substantially from the sampling method of Clover.

To handle the issue of runtime destruction, we embed an agent directly into each runtime using dynamic library loading. This implementation minimizes modifications to the original user program and ensures that metrics are collected exclusively during the execution of the serverless functions.

To accurately detect changes in function behaviors, we take advantage of the request-oriented nature inherent in serverless architectures. Specifically, we proactively invoke the metric collector whenever the runtime receives or sends a request.

C. Negligible Metric Collection Overhead

It is challenging to achieve low-overhead, fine-grained metric collection, as doing so requires eliminating unnecessary overhead wherever possible. This consideration directly informs the positioning of the agent within Clover. In other words, we cannot place the agent in a separate process from the monitored function, since that would introduce overhead from OS scheduling, cross-core hardware counter access, and context switching. To avoid these issues, Clover implements the agent as a user-level dynamic library directly invoked by the function, significantly reducing performance overhead during sampling. Moreover, Clover reads and writes performance counters through user-mode instructions (e.g., rdpmc), thus avoiding the usage of costly system calls.

Algorithm 1 Fine-Grained, Light-Weight Metric Collection

Require: T: Trigger event, M: Metadata, C: Metric counters
Ensure: X - Collected performance metrics
1: **if** T **then**
2: $cyc \leftarrow RDTSCP()$ // Record the timestamp
3: $(Mem\ Barrier)$ // Ordering constraint before reading
4: **for** $c \leftarrow C$ **do**
5: Assert not $M[c].multiplexed$
6: $x \leftarrow rdpmc(c)$ // Read the register in user mode
7: $X.push_back(x)$ // Store in buffer
8: **end for**
9: $(Mem\ Barrier)$ // Ordering constraint after reading
10: **end if**
11: Return X

We implement Clover to collect hardware metrics in a manner similar to SHIM [44], as outlined in Algorithm 1. At the beginning of each sampling cycle, Clover first obtains the current timestamp using the RDTSCP instruction (Line 2). Next, it enforces an ordering constraint on memory operations through memory barriers [56] (Line 3). The algorithm then verifies that hardware counters are not multiplexed by inspecting their metadata (Line 5). Afterward, it accesses the registers predefined by perf_event_open() directly in user mode (Line 6). Finally, the sampled results are immediately stored in a ring buffer (Line 7), followed by another memory barrier to maintain correct operation ordering (Line 9).

D. Additional Advantages of the Clover Agent

1) Event-Skid: The difference between the time when the sampling command is issued and when the actual sampling occurs is known as *event-skid*. It typically arises from OS scheduling, execution delays, or out-of-order instruction execution. Although event-skid is a common challenge in performance analysis, we demonstrate that it has no adverse effect on Clover. First, Clover collects metrics within the same user-mode context as the runtime, ensuring the metrics remain unaffected even if the OS schedules another function during execution. Second, we employ memory barriers before and after sampling to prevent out-of-order execution. Finally, the delay-induced event-skid typically spans only about ten instructions, which is negligible and does not influence the accuracy of request-level metric collection.

2) Easy-to-Use: We provide four APIs to customize the data collection behaviors: init(), add(), sample(), and extract(). When a function starts, it should call init() to construct the necessary data structures. Subsequently, it registers the metric counters using the add() command. The sample()

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

function proactively triggers the data collection, while extract() retrieves collected hardware performance metrics.

These APIs are intended for platform engineers rather than function developers. In the default scenario, developers are required to build functions on a platform-provided runtime [57], [58], [59]. In this case, the platform integrates our agent once into the runtime. All functions developed based on that runtime are then monitored transparently, without any per-function code changes. Even when functions are developed by third parties, our agents can still be integrated in the same way, which avoids imposing an additional burden on the function developers. Note that the platform-side effort also remains modest. A provider typically supports only a limited number of runtimes (about twenty), so the agent is integrated once per runtime rather than per function.

VI. DoW Detection Model

A. Assumptions About Serverless Functions

Before presenting our detection model, we define the assumed behavior of the monitored serverless functions:

- Each function is triggered by external user requests and does not initiate execution independently.
- The execution of each function is assumed to follow an independent and identically distributed (i.i.d.) pattern, with a stable distribution over time and no correlation between invocations under normal conditions.
- When a function is first deployed, it is presumed to operate in a benign environment, so its initial data is considered normal and attack-free.

These assumptions align with common serverless practices [14], [15], [16], [17] and thus do not undermine the practicality or feasibility of Clover. The first assumption allows us to record performance metrics on a per-request basis; the second crystallizes our understanding of the key differences between FaaS and IaaS platforms; and the third allows metrics collected shortly after deployment to serve as training input.

B. Design Choices

Our design choices are summarized in three aspects.

First, unlike traditional time-series approaches [45], [47], Clover emphasizes that DoW detection is essentially **a distribution-outlier problem**. This decision follows the assumed i.i.d. execution pattern in FaaS platforms and therefore targets the serverless environment. Traditional detection approaches focus on IaaS platforms, where an attack persistently affects a tenant's VMs [43]. These methods usually also assume that once an attack starts, it does not stop voluntarily. By contrast, Clover assumes that attackers may launch more complex and random attacks (see Section VII).

Second, Clover employs **workload verification** to detect DoW attacks against general functions. Rather than probing congestion on shared resources such as memory buses, we model the workload produced by each request through performance metrics and compare it with the corresponding execution time to observe changes in their correlation. When a DoW attack occurs, the function still performs the same task, yet its execution time increases; the altered correlation thus readily exposes the slowing-down phenomenon. By integrating multiple metrics, our method also avoids the accuracy loss that arises from relying on a single indicator (Section VII).

Finally, we posit a **learnable high-dimensional linear relationship** among performance metrics, workload, and execution time. This design choice is informed by the billing model (Equation 1 in Section II-A) and by the nature of the selected metrics, which record counts or durations of key events (e.g., cache misses or lock duration). Let M denote the number of metrics (eight in this study; see Table II). Define $\mathbf{x}_i = [x_i^1, x_i^2, \ldots, x_i^M] \in \mathbb{R}^M$ as the metrics that Clover collects when function k processes request i, and let t_i be the corresponding execution time. Then, we have

$$t_i \propto \sum_{j=1}^{M} W_i\left(x_i^j\right), \quad \text{where } W_i\left(x_i^j\right) \propto x_i^j \tag{3}$$

Here, $W_i(x_i^j)$ represents the contribution of metric x_i^j to the workload of request i. Hence, DoW detection aims to decide whether each function-specific correlation pattern conforms to one or more normal distributions.

C. Single-Distribution Detection Model

We first present a basic detection model for functions with simple logic, whose workloads typically follow a unimodal distribution [2], [60]. We describe this model explicitly for two reasons. First, it is the minimal complete definition of our verifier, and it provides a clean baseline that makes the extended design easier to follow. Second, it is also practically useful on its own. Under most single-distribution workloads, it remains stable and achieves higher accuracy.

Within this basic model, DoW detection consists of two phases: training and testing. **During the training phase**, we employ the metrics collected in the initial interval following the deployment of each function, in line with the third assumption in Section VI-A. The agent presented in Section V gathers these metrics, which can be expressed as:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^M \\ x_2^1 & x_2^2 & \cdots & x_2^M \\ \vdots & & \ddots & \vdots \\ x_n^1 & x_n^2 & \cdots & x_n^M \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} \tag{4}$$

Consistent with the above notation, $\mathbf{x}_i \in \mathbb{R}^M$ denotes the metrics Clover collects when function k processes request i, M is the number of metrics, and t_i is the execution time. The training set contains performance metrics from n requests. In practice, n corresponds to the requests processed within five minutes after deployment or to the first 1,000 requests. Unlike prior work [42], [43], we do not apply moving averages to the training data. Instead, we first compute the means ($\boldsymbol{\mu_x} = [\mu_x^1, \cdots, \mu_x^M]$ and μ_t) and the standard deviations ($\boldsymbol{\sigma_x} = [\sigma_x^1, \cdots, \sigma_x^M]$ and σ_t) of $\mathbf{X}$ and $\mathbf{T}$, respectively, and then perform z-score normalization, i.e., $x'^j = (x^j - \mu_x^j)/\sigma_x^j$ and $t' = (t - \mu_t)/\sigma_t$. After this preprocessing, each per-request performance-metric vector $\mathbf{x}'$ and execution time t' follow a standard Gaussian distribution ($\mathbf{x}' \sim \mathcal{N}_M(0,1)$, $t' \sim \mathcal{N}(0,1)$). Next, we employ multivariate linear regression (MLR) to

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

model the relationship among performance metrics, workload, and execution time:

$$\hat{t} = \beta_0 + \sum_{j=1}^{M} \beta_j x'^j, \ \boldsymbol{\beta} = \arg\min_{\boldsymbol{\beta}} \left\| \mathbf{Z}\boldsymbol{\beta} - \mathbf{T}' \right\|_2^2 \tag{5}$$

Here, $\mathbf{Z} = [\mathbf{1}, \mathbf{X}'] \in \mathbb{R}^{n\times(M+1)}$ is the design matrix obtained by augmenting $\mathbf{X}'$ with a leading column of ones; $\boldsymbol{\beta} = [\beta_0, \beta_1, \ldots, \beta_M]^T$ denotes the regression coefficients learned by MLR; $\mathbf{T}' \in \mathbb{R}^n$ is the normalized execution-time vector; $\hat{t}$ predicts the execution time of a single request. Let ε denote the MLR prediction error ($\varepsilon = t - \hat{t}$). Then, combining these elements yields a multivariate Gaussian distribution, where $\boldsymbol{\mu}_y$ and $\boldsymbol{\Sigma}_y$ are the mean vector and covariance matrix:

$$\mathbf{y} = [t', x'^1, \ldots, x'^M, \varepsilon]^T \in \mathbb{R}^{M+2}, \mathbf{y} \sim \mathcal{N}_{M+2}(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y) \tag{6}$$

At the testing stage, we also apply z-score normalization to the testing data with the statistics $\boldsymbol{\mu}_{\boldsymbol{x}}$, $\boldsymbol{\sigma}_{\boldsymbol{x}}$, μ_t, and σ_t obtained during training. For each newly captured performance-metric vector $\mathbf{x}_{\text{test}} = [x^1_{\text{test}}, \cdots, x^M_{\text{test}}]$ and execution time t_{test}, Clover normalizes them as $x'^j_{\text{test}} = (x^j_{\text{test}} - \mu^j_x)/\sigma^j_x$ and $t'_{\text{test}} = (t_{\text{test}} - \mu_t)/\sigma_t$. Using the trained MLR in Equation 5, we compute the prediction error $\varepsilon_{\text{test}}$ and form

$$\mathbf{y}_{\text{test}} = [t'_{\text{test}}, x'^1_{\text{test}}, \ldots, x'^M_{\text{test}}, \varepsilon_{\text{test}}]^T \in \mathbb{R}^{M+2} \tag{7}$$

We measure the deviation of $\mathbf{y}_{\text{test}}$ from the training distribution center via the Mahalanobis distance,

$$D(\mathbf{y}_{\text{test}}, \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y) = \sqrt{(\mathbf{y}_{\text{test}} - \boldsymbol{\mu}_y)^T \boldsymbol{\Sigma}_y^{-1} (\mathbf{y}_{\text{test}} - \boldsymbol{\mu}_y)} \tag{8}$$

We avoid Euclidean or Manhattan distances because the performance metrics, execution time, and MLR residual share substantial correlations. For instance, LOAD_L3_MISS often co-varies with LLC_MISS. Distances that weight all dimensions equally are thus prone to spurious sensitivity, whereas the Mahalanobis distance *explicitly accounts for* these correlations through the covariance matrix and yields a more reliable anomaly score. Finally, with a predefined threshold θ, we classify a request as normal when

$$D(\mathbf{y}_{\text{test}}, \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y) \le \theta \tag{9}$$

D. Multiple-Distribution Detection Model

In Section VI-C, we assume that the workload pattern of each function follows a unimodal normal distribution. In practice, however, function workloads often exhibit mixture behavior [54], [61], [62], [63]. Such multimodality can stem from distinct execution phases (cold start, warm start, and steady state), program-level behaviors (e.g., garbage collection and stochastic effects), and system-level conditions (e.g., cache hits and misses). Under this scenario, Clover-S faces an inherent tradeoff. A strict global threshold raises false positives for samples from other legitimate modes, whereas a relaxed threshold suppresses alarms and increases false negatives (low recall) when attacks occur within a non-dominant mode. To tackle this challenge, we extend the original model and propose an innovative multi-distribution detection model.

During training, the model also ingests the performance metrics and execution time gathered by the agents, identical in format to the single-distribution model. Next, we calculate $\boldsymbol{\mu}_{\boldsymbol{x}}$, $\boldsymbol{\sigma}_{\boldsymbol{x}}$, μ_t, and σ_t only to place every dimension on a common scale and alleviate unit-driven imbalance; these statistics do not participate in workload distribution modeling. After applying z-score normalization and MLR, we obtain $\mathbf{x}'$, t', and ε as in Section VI-C. Then, we concatenate them to construct

$$\mathbf{y} = [t', x'^1, \ldots, x'^M, \varepsilon]^T \in \mathbb{R}^{M+2} \tag{10}$$

Let R denote the total number of workload patterns of a function, i.e., the number of potential Gaussian components. For each $\mathbf{y} \in \mathbb{R}^{M+2}$, we introduce a discrete latent variable:

$$C \in \{1, \ldots, R\}, \ \Pr(C = r) = \pi_r > 0, \ \sum_{r=1}^{R} \pi_r = 1 \tag{11}$$

Conditioned on $C = r$, our multiple-distribution model indicates that $\mathbf{y}$ follows a certain Gaussian distribution:

$$\mathbf{y} \,\big|\, C = r \ \sim \ \mathcal{N}_{M+2}(\boldsymbol{\mu}_r, \boldsymbol{\Sigma}_r) \tag{12}$$

Because each $\mathbf{y}$ should belong to exactly one of the R Gaussian components, the model only needs to learn R together with $\boldsymbol{\mu}_r$ and $\boldsymbol{\Sigma}_r$ to fully characterize the workload distribution. Although R is unknown beforehand, it is fixed for any given function. We therefore employ a Dirichlet–Process Gaussian Mixture Model (DP-GMM) [64], which infers R, $\boldsymbol{\mu}_r$, and $\boldsymbol{\Sigma}_r$ automatically. This method avoids the repeated training and argument selection required by conventional GMMs [61].

At the end of training, we further employ the minimum covariance determinant (MCD) [65] to obtain robust estimates of $\boldsymbol{\mu}_r$ and $\boldsymbol{\Sigma}_r$. This step mitigates the influence of normal performance fluctuations on the sample mean and covariance. The resulting estimates are denoted by $\hat{\boldsymbol{\mu}}_r$ and $\hat{\boldsymbol{\Sigma}}_r$. Consequently, the final multi-distribution model is expressed as:

$$p(\mathbf{y}) = \sum_{r=1}^{R} \pi_r \mathcal{N}_{M+2}(\mathbf{y}; \hat{\boldsymbol{\mu}}_r, \hat{\boldsymbol{\Sigma}}_r) \tag{13}$$

At the testing stage, we apply z-score normalization, MLR prediction, and vector concatenation to obtain $\mathbf{y}_{\text{test}} \in \mathbb{R}^{M+2}$. This pre-processing procedure is identical to that used in the single-distribution model (Equation 7). Then, we traverse the R Gaussian components obtained during training and compute the robust Mahalanobis distance for each component:

$$D(\mathbf{y}_{\text{test}}, \hat{\boldsymbol{\mu}}_r, \hat{\boldsymbol{\Sigma}}_r) = \sqrt{(\mathbf{y}_{\text{test}} - \hat{\boldsymbol{\mu}}_r)^T \hat{\boldsymbol{\Sigma}}_r^{-1} (\mathbf{y}_{\text{test}} - \hat{\boldsymbol{\mu}}_r)} \tag{14}$$

For the r-th Gaussian component, we define the DoW threshold θ_r as the largest Mahalanobis distance between any training sample assigned to this component and its center:

$$\theta_r = \max_{i:\, \hat{C}_i = r} D(\mathbf{y}_i, \hat{\boldsymbol{\mu}}_r, \hat{\boldsymbol{\Sigma}}_r) \tag{15}$$

Let $r^\star$ denote the Gaussian component closest to $\mathbf{y}_{\text{test}}$ (i.e., $r^\star = \arg\min_r D(\mathbf{y}_{\text{test}}, \hat{\boldsymbol{\mu}}_r, \hat{\boldsymbol{\Sigma}}_r)$). Clover determines whether the function is under a DoW attack as follows:

$$\mathbf{y}_{\text{test}} = \begin{cases} \text{normal}, & D(\mathbf{y}_{\text{test}}, \hat{\boldsymbol{\mu}}_{r^\star}, \hat{\boldsymbol{\Sigma}}_{r^\star}) \le \theta_{r^\star}, \\ \text{under attack}, & \text{otherwise}. \end{cases} \tag{16}$$

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

VII. Evaluation

In this section, we comprehensively evaluate Clover's performance and address four key questions:

- Does agent-side metric collection in Clover impose only negligible runtime overhead? (Section VII-B)
- Does Clover's workload-verification model enable real-time detection of DoW attacks? (Section VII-C)
- How do the parameters of the single- and multiple-distribution workload-verification models affect the detection precision of DoW attacks? (Section VII-D)
- What end-to-end prediction accuracy does Clover's detection model deliver? (Section VII-E)

A. Experiment Setup

Testbed. We perform our experiments on a two-node Kubernetes cluster (version 1.24.2) configured according to standard practices [66]. Each node is an identical server equipped with an Intel Xeon E5-2620 v3 CPU (2.40 GHz, 12 physical cores) and 128 GB RAM. The underlying microarchitecture is Haswell-EP/EN/EX, exposing eight programmable and three fixed per-core PMU counters. All nodes run Ubuntu 20.04 with Linux kernel 5.4.0-107-generic.

Evaluation indicators. To evaluate the detection performance, we adopt recall, specificity, precision, and accuracy as evaluation indicators in our experiments, where T denotes *True*, F denotes *False*, N denotes *Negative*, and P denotes *Positive*.

$$\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}, \quad \text{Specificity} = \frac{\text{TN}}{\text{TN}+\text{FP}},$$
$$\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}, \quad \text{Accuracy} = \frac{\text{TP}+\text{TN}}{\text{TP}+\text{FP}+\text{TN}+\text{FN}}.$$

Baselines. We implement several baseline anomaly-detection algorithms to evaluate the advantages of Clover.

- *SDS/B*. Proposed by Li et al. [42], SDS/B is a statistical memory-DoS detector that estimates the normal operating range of a metric using sliding windows and an exponentially weighted moving average (EWMA). When the EWMA departs from this range for H_c consecutive updates, the algorithm triggers an alarm. We adopt the parameters in the original paper: $T_{\text{PCM}} = 0.01$ s, $W = 200$, $\delta W = 50$, $\alpha = 0.05$, $k = 1.125$, and $H_c = 30$.
- *K-Means*. K-Means is a representative distribution-based method. It is adopted in prior SOTA systems such as FineLame [67]. During training, we tune the cluster count K by sweeping candidate values. We then set the detection threshold to the maximum Euclidean distance from any training sample to its nearest centroid.
- *Random forest (RF)*. RF is a supervised classifier that has been used in recent metrics-based anomaly detection systems (e.g., MicroHECL [68]). It requires anomalous data during training; accordingly, we generate pseudo-anomalies by sampling each metric at $\mu \pm 3\sigma$.
- *ModernTCN*. ModernTCN is a representative sequence-based SOTA detector that uses a modernized TCN (i.e., pure temporal convolutions) to learn temporal patterns [69]. We use the settings from the original paper.
- *SensitiveHUE*. SensitiveHUE is another SOTA sequence-based detector. It adopts a transformer architecture to improve anomaly sensitivity via statistical feature removal and heteroscedastic uncertainty estimation [70]. We use the hyperparameter settings from the original paper.
- *Clover*. We denote the implementation based on the single-distribution model as *Clover-S*, and that based on the multiple-distribution model as *Clover-M*.

TABLE III
Victim Functions (Exposed as HTTP Endpoints Using Flask)

Function	Description	Libraries
Blur	Image blurring (norm box filter)	opencv-python
LR	Regularized logistic regression	numpy
PCA	Principal component analysis	scikit-learn
Bayes	Gaussian Naive Bayes	scikit-learn
WebApps	Dynamic HTML generation	jinja2
Thumb	Image thumbnail generation	Pillow
Data-Vis	DNA sequence visualization backend	squiggle
Inference	Image recognition	PyTorch
CfgCheck	YAML parsing and validation	PyYAML
TokenSign	Token signing and verification	cryptography

Function setup. As shown in Table III, we implement victim functions using a diverse set of foundational libraries. These libraries cover many of the most-downloaded PyPI packages, and some are adapted from the SEBS serverless benchmark suite [71]. We adjust each request to ensure its execution time remains between 1ms and 1s. We conduct internal DoW attacks by explicitly scheduling an attacker function with the victim on the same physical node. We deploy all functions as containers managed by Kubernetes, and we constrain the attacker function to 1 vCPU and 2 GB of memory. The attacker function is written in C. It allocates a contiguous memory region and repeatedly executes lock-prefixed atomic updates on cache-line-crossing addresses, which triggers memory bus locking on x86 processors. We also toggle the attack on and off by sending control requests to the attacker function.

We implement the agent of Clover as a C++ extension to the Python runtime and embed it in a Flask-based Knative example [72]. However, time-series models such as SDS/B require inputs collected at fixed intervals, which conflicts with the per-request metrics recorded by Clover. To resolve the mismatch, we interpolate the per-request metrics by timestamp and resample them into a uniformly spaced time series. This preprocessing does not affect detection results because SDS/B uses a 2s sliding window with EWMA, while each function completes in about 1s. Although the interpolated points represent estimates rather than exact values, the sliding window and EWMA will smooth out any resulting errors.

Attack scenarios and workloads. We examine three DoW attack scenarios. The first two replicate the settings in [42], whereas the third introduces a more sophisticated attack configuration tailored to serverless environments.

- *Scenario 1*. The victim function runs for 60min. It stays benign for the first 30min, after which a DoW attack is launched during the remaining 30min.
- *Scenario 2*. The victim function runs for 60min. DoW attacks are launched in a round-robin fashion, with each attack interval uniformly distributed over [10,50]s.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

Fig. 8. The monitoring overhead of Clover's agent.

- *Scenario 3*. The victim function again runs for 60min. DoW attacks may start at *arbitrary moments* and last for *arbitrary durations* between consecutive requests.

We evaluate two types of workloads: *single-distribution workloads* and *multiple-distribution workloads*. In the single-distribution workload, we execute each function in isolation and vary request parameters to maximize intra-function diversity. Because each function is simple, its performance metrics roughly follow a unimodal Gaussian distribution. In the multiple-distribution workload, we merge all ten functions into one large function and intentionally interleave requests among them. Their execution times stay comparable, yet their input patterns become highly heterogeneous. This design yields a multi-modal workload that better approximates real-world conditions [62], [63] and represents the core challenge addressed in Section VI-D.

B. The Monitoring Overhead of Agent

As described in Section V-C, we implement the agent as a user-level dynamically loaded library, which avoids additional process switches during metric collection. To quantify this benefit, we compare three designs: (i) *Baseline*, which does not collect metrics; (ii) *External Collector*, which collects metrics in a separate process; and (iii) *Clover*. We vary request payloads to cover a wide execution-time range, from 1ms to 1s. For each payload a, we take the minimum latency in *Baseline* (i.e., $T^a_{\min}$) and normalize all results under a as $x' = (x - T^a_{\min})/T^a_{\min}$. This removes absolute latency differences and yields relative overhead across parameter settings. We invoke each function 100 times under each setting.

Figure 8 quantifies the end-to-end overhead of our agent, including metric collection, metric extraction, and auxiliary operations in the monitoring pipeline. Across all inputs, Clover increases latency by only 0.28–3.19% over the baseline, and this bound still holds in the most challenging scenario, where invocations are extremely short (around 1ms). In contrast, the external collector incurs about 10% overhead. These results validate our claim that implementing the agent as an in-process dynamic library keeps monitoring lightweight. We also measure the cold-start overhead of the agent, which is about 23ms. Given that loading NumPy alone takes around 100ms during a cold start, this additional delay is acceptable.

C. Model Execution Time

Next, we assess whether Clover can support real-time DoW detection. We run each model on multiple victim functions under a single-distribution workload across three attack scenarios, and we report results over repeated trials for each setting. In deployment, detection delay is the time from attack onset to the first alarm, and it depends on data-collection granularity, model accuracy, function execution duration, and model execution time. To isolate model-side cost, this experiment reports only the execution time of each model.

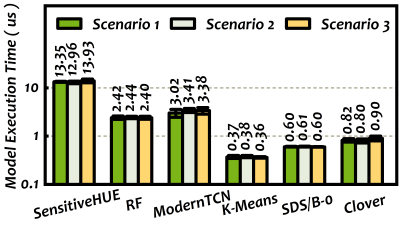

Fig. 9. The per-request execution time for each model. Here we test the Clover-M model.

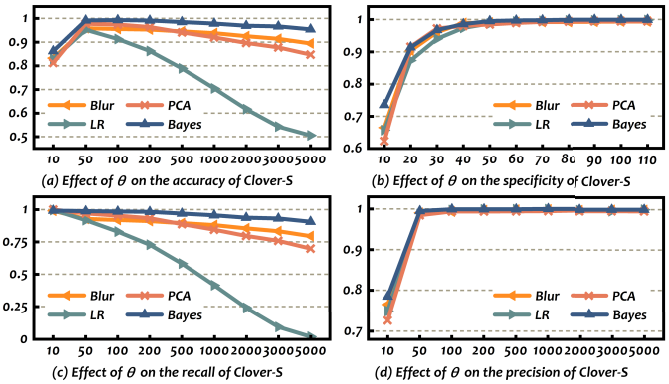

Fig. 10. The effect of threshold θ on the Clover-S model.

As shown in Figure 9, all models can detect DoW attacks in real time. K-Means is the fastest, taking about 0.37μs across all settings, because it requires only a few simple operations. RF and SDS/B run more slowly, at 2.42μs and 0.6μs, respectively. ModernTCN and SensitiveHUE are the slowest, completing per-request detection in about 3.3μs and 13.4μs. Clover completes detection in about 0.84μs.

D. Influences of the Factors

Next, we conduct a sensitivity analysis of key hyperparameters. For Clover-S, we evaluate four victim functions while varying the detection threshold θ and the continuous-judgment threshold H_c. For Clover-M, we run experiments under the multiple-distribution workload and vary the DP-GMM concentration parameter α and the outlier proportion assumed by MCD F. Unless otherwise specified, all tests in this experiment are conducted under attack scenario 3.

To study the effect of the detection threshold, we sweep θ over a wide range and report the results in Figure 10. As θ increases, specificity rises quickly and reaches 100%. Once $\theta > 100$, no normal samples are misclassified as attacks. This gain comes at the expense of recall. Recall is highest at $\theta = 10$ and decreases monotonically as θ grows. Accuracy first improves and then declines, peaking at $\theta \in [50, 100]$. Precision typically follows the same qualitative trend as specificity.

We then vary the DP-GMM concentration parameter α and the outlier proportion F assumed by MCD under the multi-distribution workload. Figure 12 shows that α has a pronounced, non-monotonic effect. Very small α (0.01-0.05) yields consistently lower accuracy (82.3%-87.9%, depending on F). Moderate values (0.5-1.0) provide the best overall performance, reaching 92.7% accuracy at $\alpha = 1.0$ and $F = 0.1$.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

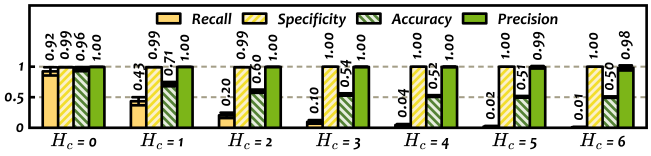

Fig. 11. The effect of H_c on the Clover-S Model.

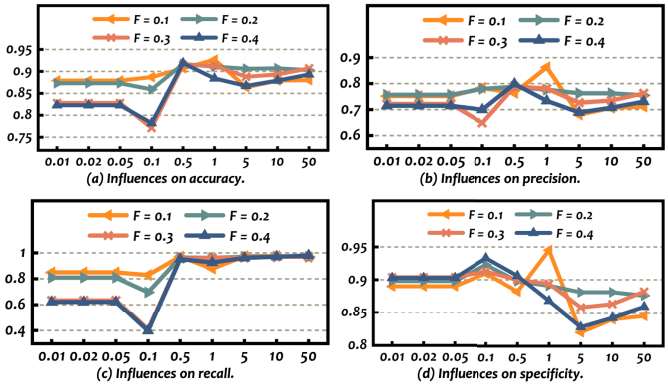

Fig. 12. Hyperparameter effects on Clover-M. The x-axis represents α.

TABLE IV
SINGLE-DISTRIBUTION WORKLOADS - SCENARIO 1

	WebApps	Blur	Inference	PCA	CfgCheck	LR	Data-Vis	Bayes	TokenSign	Thumb
SDS/B	0.7860	0.5062	0.8285	0.5061	0.6531	0.5053	0.6589	0.5068	0.6012	0.5641
SDS/B-0	0.7794	0.5020	0.8197	0.5019	0.6476	0.5011	0.6534	0.5027	0.5961	0.5593
K-Means	0.9598	0.8905	0.3848	0.8969	0.9297	0.8704	0.9245	0.8834	0.7752	0.4153
RF	0.9619	0.8850	0.9399	0.9051	0.8675	0.8588	0.8999	0.8860	0.7711	0.6066
ModernTCN	0.9283	0.8947	0.9303	0.8534	0.9211	0.8923	0.9199	0.8587	0.8377	0.8549
SensitiveHUE	0.9251	0.8921	0.9293	0.8688	0.9162	0.9023	0.9060	0.8881	0.8235	0.9097
Clover-S	0.9284	0.8624	0.9078	0.8566	0.9194	0.9624	0.8967	0.9146	0.8326	0.7623
Clover-M	0.9224	0.9457	0.9648	0.9552	0.8932	0.9354	0.9218	0.9587	0.9230	0.9276

In contrast, $\alpha = 0.1$ shows a clear degradation at larger F (e.g., recall drops to 40.8% at $F = 0.3$). Larger α (5.0-50.0) tends to maintain very high recall (96.2%-98.0%), but precision and specificity decrease, which slightly reduces accuracy. Overall, F mainly shifts the operating point and interacts with α. Moderate values (F = 0.1-0.2) are the most stable, and we use $\alpha = 1.0$ and $F = 0.1$ as the default hyperparameters.

E. End-to-End Results

In the final experiment, we evaluate all models across attack scenarios, victim functions, and workloads. Under the single-distribution workload, we consider three attack scenarios and report accuracy. Under the multi-distribution workload, we focus on attack scenario 3, which best captures key serverless characteristics, and report precision, recall, specificity, and accuracy. Tables IV-VII summarize the results.

Under the single-distribution workload, most detection models achieve high accuracy in scenario 1-2. In scenario 3, fluctuations become more pronounced and may offset modeling errors, which improves several models on certain functions. K-Means, RF, and the sequence-based SOTA methods (SensitiveHUE and ModernTCN) are more sensitive to the workload and scenario, whereas Clover-S remains stable and achieves 83%-97% accuracy. Notably, the Thumb function exhibits *multiple modes* during its normal execution, so Clover-S is less effective and achieves about 75% accuracy.

TABLE V
SINGLE-DISTRIBUTION WORKLOADS - SCENARIO 2

	WebApps	Blur	Inference	PCA	CfgCheck	LR	Data-Vis	Bayes	TokenSign	Thumb
SDS/B	0.7527	0.4814	0.8291	0.5437	0.6409	0.5414	0.6589	0.5799	0.5412	0.5836
SDS/B-0	0.7589	0.4814	0.8202	0.5343	0.6398	0.5373	0.6384	0.5840	0.5353	0.5881
K-Means	0.9282	0.8849	0.4882	0.9024	0.8260	0.8757	0.8740	0.9008	0.8199	0.6183
RF	0.9401	0.7846	0.8855	0.9007	0.8658	0.8684	0.8678	0.9007	0.8723	0.9064
ModernTCN	0.9207	0.8957	0.9331	0.8899	0.9177	0.8900	0.9219	0.8771	0.8433	0.9148
SensitiveHUE	0.9217	0.8902	0.9272	0.8801	0.9186	0.8933	0.9240	0.8778	0.7746	0.9254
Clover-S	0.9291	0.8585	0.9078	0.8589	0.9195	0.9638	0.8945	0.9175	0.8470	0.8494
Clover-M	0.9200	0.9440	0.9648	0.9550	0.8919	0.9390	0.9218	0.9614	0.9219	0.9319

TABLE VI
SINGLE-DISTRIBUTION WORKLOADS - SCENARIO 3

	WebApps	Blur	Inference	PCA	CfgCheck	LR	Data-Vis	Bayes	TokenSign	Thumb
SDS/B	0.9858	0.7583	0.9900	0.8459	0.8917	0.7042	0.9205	0.8959	0.7456	0.7319
SDS/B-0	0.9914	0.7575	0.9988	0.8487	0.8958	0.7051	0.9252	0.8987	0.7493	0.7349
K-Means	0.9661	0.9284	0.3848	0.9570	0.9395	0.9023	0.9362	0.9637	0.7740	0.3959
RF	0.8175	0.8639	0.8192	0.8983	0.7042	0.8126	0.7310	0.9209	0.6268	0.5019
ModernTCN	0.9243	0.9175	0.9319	0.9330	0.9267	0.9125	0.9291	0.9253	0.8832	0.8176
SensitiveHUE	0.9331	0.9193	0.9202	0.9302	0.9257	0.9117	0.9326	0.9214	0.7695	0.9326
Clover-S	0.9241	0.8723	0.9078	0.9679	0.9155	0.9698	0.8861	0.9512	0.8303	0.7494
Clover-M	0.9272	0.9434	0.9648	0.9047	0.9011	0.9463	0.9232	0.9729	0.9233	0.9319

TABLE VII
MULTIPLE-DISTRIBUTION WORKLOADS - SCENARIO 3

	SDS/B	SDS/B-0	K-Means	RF	Modern TCN	Sensitive HUE	Clover-S	Clover-M
Precision	0.4581	0.4916	0.7753	0.9506	0.7018	0.6973	0.6958	0.8631
Recall	0.5014	0.6353	0.6187	0.7009	0.9764	0.9205	0.8570	0.8794
Specificity	0.3586	0.2895	0.8061	0.9561	0.8370	0.8417	0.8528	0.9452
Accuracy	0.4328	0.4691	0.7087	0.8651	0.8763	0.8662	0.8540	0.9267

Under the multi-distribution workload, most methods degrade because they fail to model heterogeneous benign modes. Some become overly conservative and show high specificity but low recall. Others sacrifice precision to improve recall. SensitiveHUE and ModernTCN reach only 86.6% and 87.6% accuracy, respectively. This is partly due to their sequence-based design. They implicitly assume anomalies persist across multiple consecutive time steps and can be separated from short-lived fluctuations. Clover-S drops to 85.4% because its single-distribution assumption breaks under multi-modal workloads. In contrast, Clover-M maintains 92.7% accuracy. This does not imply that Clover-M is always better, since Clover-S can be more accurate when the workload is unimodal, for example, for LR and PCA.

Finally, we conduct an ablation study of Clover-M under scenario 3. We compare five variants: a simple linear model (MLR), Clover-S, a GMM model scored by negative

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

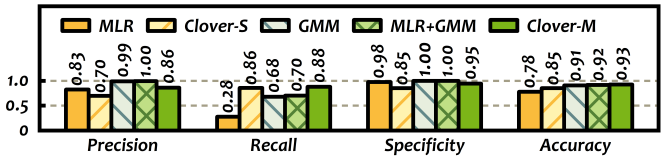

Fig. 13. Ablation study of the Clover-M model.

log-likelihood, MLR+GMM (which adds residuals to encode the workload relationship), and the full Clover-M (which further incorporates MCD denoising and Mahalanobis-distance detection). As shown in Figure 13, the linear baseline is overly conservative (precision 82.6%, specificity 97.7%) and misses most attacks (recall 28.0%, accuracy 78.0%). Clover-S increases recall to 85.7% while achieving high precision and specificity. Modeling heterogeneous benign behaviors further raises the precision of our model to 86.3% with specificity around 95%. As a result, the Clover-M model provides the best balance and reaches 92.7% accuracy.

VIII. Discussion and Limitations

Function placement and co-location detection. To enable the co-location of malicious and victim functions, attackers can initiate hundreds or thousands of malicious functions simultaneously, with the goal of co-residing with the victim function. Such methods are widely utilized in placement vulnerability research [31], [73], [74], [75]. Moreover, the short-lived nature of serverless functions increases the likelihood of co-location. In this paper, we implement a co-location detection method similar to that proposed by Anil Yelam et al. [31], which utilizes the memory bus as a covert channel to transmit information. Note that co-location detection and function placement are outside the scope of this paper, and the strategies used can be replaced with alternative methods.

Practicality and feasibility of internal DoW attacks. Internal DoW attacks are practical and feasible for the following four reasons: First, because of the ongoing launch of new functions and the immediate termination of non-co-located functions, the attack's financial cost to the attacker is negligible. Traditional defenses, such as random placement of functions, do not effectively counter this attack. Second, memory bus locking requires no additional privileges, and hardware-level contention cannot be monitored or controlled by current OS tools, like cgroups. Third, attackers do not need timing information, and the attack can begin immediately after co-location placement. Finally, attackers can take advantage of the free plans offered by most platforms. A few accounts are enough to deplete the victim's available credit, pushing them into debt.

DoW Mitigation. With the increasing focus on the serverless billing model, services such as billing alerts and resource restrictions are now available on most serverless platforms [76]. However, these services are ineffective against mitigating DoW attacks. First, when an account accumulates a large number of charges in a short time due to a DoW attack, the platform will freeze the account or notify the victim. This turns the DoW attack into a common DoS attack, causing temporary unavailability of the service before the user can react. Second, even with billing alerts, customers may still be overcharged for a brief period because of improper configuration [4]. Third, if the attacker adopts a more strategic approach, the attack will proceed gradually. Unlike traditional DoS attacks, a DoW attack does not require total service denial. Instead, the user's bill will slowly rise, while the quality of service will decline gradually. Therefore, to mitigate DoW attacks, it is recommended to use Clover for real-time detection and promptly disable suspicious functions [43].

Potential limitations. Clover has several potential limitations. First, as an additional dynamically loaded library, Clover increases function cold-start latency, although this overhead is substantially smaller than that introduced by commonly used libraries such as NumPy. Second, we use a ring buffer to cache metrics. This design effectively bounds the memory footprint, but our current prototype does not include an automatic metric retrieval mechanism. As a result, under very high request rates, newly collected metrics may overwrite older buffer entries if they are not retrieved in time. Third, Clover assumes that the platform will integrate our APIs into the provided runtime or a similar wrapper. However, in rare deployment scenarios, tenants may provide fully custom images that bypass these managed layers. In such instances, the platform may not be able to invoke our APIs transparently. It should be noted that this limitation signifies a general boundary of application-layer observability mechanisms [77], [78] rather than being a constraint specific to Clover.

Extensions beyond the conference version. This manuscript substantially extends our conference version [79] with new designs and an expanded evaluation. First, we present a more rigorous billing-cost formulation that clarifies the pay-as-you-go model and pinpoints which components of the billing pipeline are stressed by DoW attacks. Second, we identify a new practical challenge, namely detecting DoW attacks under heterogeneous, multi-modal workloads. Third, we generalize workload verification from a single-distribution assumption to a multi-distribution detection model by incorporating mixture modeling and robust covariance estimation. Fourth, we substantially expand the evaluation with additional baselines, a broader set of functions, more tests, and richer metrics.

Metric selection. We do not implement per-function metric selection for four reasons. First, the cost is impractical. It would require many additional invocations because the PMU event space is large (roughly 200 candidates), while only a small number of hardware counters (typically fewer than 10) can be enabled at once. Second, profiling is challenging to make reliable. During these extra invocations, it is difficult for the platform to construct comprehensive and representative profiling workloads for each function, which can undermine metric selection. Third, modern platforms maintain a large and continuously evolving function catalog. In this setting, per-function profiling would introduce high operational expense and engineering complexity. Finally, the marginal benefit is limited. Our chosen metric set already captures both DoW-relevant signatures and general workload behavior and achieves strong end-to-end detection accuracy, so per-function customization is unlikely to justify the added complexity.

IX. Related Work

Covert channels and memory DoS attacks. Covert channels have long been an important topic in the security community [32], [33], [34], [35], [36], [37]. These channels exploit shared resources in multi-tenant platforms that cannot

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

be fully isolated through hardware virtualization. Information is subtly transmitted between tenants through performance degradation. Memory DoS attacks exacerbate this congestion. Zhang et al. [43] performed memory DoS attacks on VMs and used the two-sample Kolmogorov-Smirnov test [80] for detection. Li et al. [42] proposed three memory DoS detection methods with higher specificity, shorter detection delays, and negligible performance overhead. Compared to prior research, our work focuses on DoW attacks in serverless computing, which are more destructive and present greater detection challenges.

Denial-of-Wallet attacks and recent studies. FineLame [67] was the first to show that a carefully crafted malicious payload can exhaust a victim's computational resources. Following this, Kelly et al. [25] and Marin et al. [26] introduced the notion of Denial-of-Wallet attacks. Recent studies have proposed several traffic-based external DoW detection techniques. For example, DoWNet [81] proposed aggregating function invocation counts per hour into a grayscale image to identify DoW attacks. Ben-Shimol et al. [82] established an ontology and knowledge graph based on logs and proposed a three-layer security solution for DoW attacks. FODWNN-DoWAD [83] introduced a neural network-based method for external DoW detection, including a feature selection algorithm, a DoW attack detection algorithm, and a hyperparameter tuning method. With the exception of FineLame [67], however, these works focused on external DoW attacks. In contrast, Clover targets internal DoW attacks that inflate per-request function execution through resource contention.

Comparison with related work. Internal DoW attacks manifest as anomalous patterns in multivariate performance traces. Depending on how anomalies are modeled, existing detection approaches mainly fall into two classes: *sequence-based* and *distribution-based* methods. Sequence-based methods model metrics as multivariate time series. They detect anomalies by learning normal temporal dynamics. Recent work uses expressive models such as modern TCNs [69] and transformer-based detectors [70]. Their main limitation is an implicit continuity assumption that abnormal states must persist long enough to be separable from short-lived fluctuations. This assumption mismatches internal DoW, where attacks may occur intermittently between consecutive requests. Distribution-based methods treat each invocation as an independent sample. For example, FineLame [67] clusters per-request resource profiles to identify outliers. However, these methods largely rely on absolute metric magnitudes. This makes it difficult to distinguish contention-induced slowdowns from benign workload variation. Clover distinguishes itself by proposing workload verification. It learns the metric–duration relationship and flags internal DoW attacks when execution time becomes inconsistent with the inferred workload correlation.

X. Ethical Considerations

With the exception of Section III, all of our attack experiments are conducted on local servers. For the DoW attacks in Section III, we have obtained permission from the relevant cloud providers to carry out the tests. Under their guidance, we performed the attacks during off-peak hours. The victim and attacker accounts are separate, both of which we control. To minimize the duration of the experiment and reduce the potential for external impact, we select the simplest Hello-World application. In comparison to earlier work [31] using the same technique, our experiment has a smaller impact. We deployed fewer functions than [31] and designed our code to limit each memory bus lockdown to a few seconds.

We reported our findings to the relevant cloud providers (i.e., Amazon, Google, Microsoft, and Alibaba) through their respective security disclosure portals. Some providers acknowledged this issue as a security vulnerability and considered our disclosure informative. However, all providers concluded that there is no effective fix at present and decided not to track this bug in a short time.

XI. Conclusion

In this paper, we conduct a thorough analysis of the Denial-of-Wallet attack, a variant of the Denial-of-Service attack specifically targeting serverless platforms. We execute a real-world DoW attack on commercial serverless platforms, and our results demonstrate that such an attack leads to a 37-fold increase in payment costs for the victim's functions. We develop Clover as a practical system designed for detecting real-world DoW attacks on serverless platforms. Clover utilizes a fast and efficient agent to gather performance metrics and execution durations of serverless functions. Furthermore, it introduces two accurate DoW detection models based on the concept of workload verification. Our evaluation shows that the data collection overhead of Clover is under 3.2%, while its DoW detection accuracy reaches 92.7%, even under the most challenging scenarios. We believe our findings can inspire further research on economic security risks in cloud computing. In future work, we plan to extend Clover to detect a wider range of performance-degrading attacks in serverless environments.

Acknowledgment

The authors sincerely thank the editors and the anonymous reviewers for their constructive comments and valuable suggestions, which have helped improve this manuscript substantially. They also acknowledge the contributions of Yuzhuo Tan and XiAo Jiang to this work.

References

[1] D. Senie et al., *Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing*, document RFC 2827, 2000.

[2] S. Yu, W. Zhou, R. Doss, and W. Jia, "Traceback of DDoS attacks using entropy variations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 3, pp. 412–425, Mar. 2011.

[3] J. Wu et al., *Source Address Validation Improvement (SAVI) Framework*, document RFC 7039, 2013.

[4] *We Burnt $72K Testing Firebase + Cloud Run and Almost Went Bankrupt*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.codeisgo.com/post/we-burnt-72k-testing-firebase-cloud-run-and-almost-went-bankrupt-2020121301/

[5] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Computer*, vol. 40, no. 9, pp. 103–105, Sep. 2007.

[6] *AWS Lambda Pricing*. Accessed: Apr. 1, 2026. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[7] *Google Cloud Function Pricing*. Accessed: Apr. 1, 2026. [Online]. Available: https://cloud.google.com/functions/pricing

[8] *Microsoft Azure Functions Pricing*. Accessed: Apr. 1, 2026. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/

[9] *Alibaba Cloud Function Compute Pricing*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.alibabacloud.com/help/en/doc-detail/54301.htm

[10] *Jaeger*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.jaegertracing.io/

[11] *Fluentd*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.fluentd.org/

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

[12] *Intel PCM*. Accessed: Apr. 1, 2026. [Online]. Available: https://github.com/opcm/pcm.git

[13] *Perf*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.brendangregg.com/perf.html

[14] *AWS Lambda*. Accessed: Apr. 1, 2026. [Online]. Available: https://aws.amazon.com/lambda/

[15] *Google Cloud Functions*. Accessed: Apr. 1, 2026. [Online]. Available: https://cloud.google.com/functions/

[16] *Microsoft Azure Functions*. Accessed: Apr. 1, 2026. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[17] *Alibaba Function Compute*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.alibabacloud.com/product/function-compute

[18] *Intel 64 Manuals*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[19] *AMD64 Architecture Programmer's Manual*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf

[20] H. Awwad, C. Lin, R. Ward, and M. Shahrad, "Estimating the carbon footprint of serverless functions on a public cloud platform," in *Proc. 3rd Workshop Serverless Syst., Appl. Methodologies*, Mar. 2025, pp. 12–20.

[21] M. M. H. Bechtel and H. Yun, "Cache bank-aware denial-of-service attacks on multicore ARM processors," in *Proc. RTAS*, May 2023, pp. 198–208.

[22] T. Schirmer, N. Japke, S. Greten, T. Pfandzelter, and D. Bermbach, "The night shift: Understanding performance variability of cloud serverless platforms," in *Proc. 1st Workshop Serverless Syst., Appl. Methodol.*, May 2023, pp. 27–33.

[23] D. Lambion, R. Schmitz, R. Cordingly, N. Heydari, and W. Lloyd, "Characterizing X86 and ARM serverless performance variation: A natural language processing case study," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, Jul. 2022, pp. 69–75.

[24] J. Wen, Z. Chen, F. Sarro, and S. Wang, "Unveiling overlooked performance variance in serverless computing," *Empirical Softw. Eng.*, vol. 30, no. 2, pp. 1–26, Mar. 2025, Art. no. 59.

[25] D. Kelly et al., "Denial of wallet- defining a looming threat to serverless computing," *J. Inf. Secur. Appl.*, vol. 60, Aug. 2021, Art. no. 102843.

[26] E. Marin, D. Perino, and R. Di Pietro, "Serverless computing: A security perspective," *J. Cloud Comput.*, vol. 11, no. 1, p. 69, Oct. 2022.

[27] *Edmunds Going Serverless on AWS*. Accessed: Apr. 1, 2026. [Online]. Available: https://aws.amazon.com/cn/solutions/case-studies/edmunds-serverless/

[28] *Optimizing Enterprise Economics With Serverless Architectures*. Accessed: Apr. 1, 2026. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/optimizing-enterprise-economics-with-serverless/optimizing-enterprise-economics-with-serverless.html

[29] W. Shao et al., "Bit of a close talker: A practical guide to serverless cloud co-location attacks," in *Proc. NDSS*, 2026, pp. 1–18.

[30] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Everywhere all at once: Co-location attacks on public cloud FaaS," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2024, pp. 133–149.

[31] A. Yelam, S. Subbareddy, K. Ganesan, S. Savage, and A. Mirian, "CoResident evil: Covert communication in the cloud with lambdas," in *Proc. Web Conf.*, Apr. 2021, pp. 1005–1016.

[32] M. Tan, J. Wan, Z. Zhou, and Z. Li, "Invisible probe: Timing attacks with PCIe congestion side-channel," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 322–338.

[33] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[34] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622.

[35] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 990–1003.

[36] P. Chen, L. Li, and Z. Yang, "Cross-VM and cross-processor covert channels exploiting processor idle power management," in *Proc. Secur.*, 2021, pp. 733–750.

[37] J. Haj-Yahya et al., "IChannels: Exploiting current management mechanisms to create covert channels in modern processors," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 985–998.

[38] *Sustaining Free Compute in a Hostile ENV*. Accessed: Apr. 1, 2026. [Online]. Available: https://www.koyeb.com/blog/sustaining-free-compute-in-a-hostile-environment

[39] A. Joosen et al., "How does it function? Characterizing long-term trends in production serverless workloads," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2023, pp. 443–458.

[40] N. Nasiri et al., "In-production characterization of an open source serverless platform and new scaling strategies," in *Proc. EuroSys*, 2026, pp. 1–19.

[41] (2006). *Marissa Mayer at Web 2.0*. [Online]. Available: https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html

[42] Z. Li, T. Sen, H. Shen, and M. C. Chuah, "A study on the impact of memory DoS attacks on cloud applications and exploring real-time detection schemes," *IEEE/ACM Trans. Netw.*, vol. 30, no. 4, pp. 1644–1658, Aug. 2022.

[43] T. Zhang, Y. Zhang, and R. B. Lee, "DoS attacks on your memory in cloud," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 253–265.

[44] X. Yang et al., "Computer performance microscopy with shim," *SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 170–184, 2015.

[45] J. Audibert et al., "Usad: Unsupervised anomaly detection on multivariate time series," in *Proc. KDD*, 2020, pp. 3395–3404.

[46] S. Fu, S. Gupta, R. Mittal, and S. Ratnasamy, "On the use of ML for blackbox system performance prediction," in *Proc. NSDI*, 2021, pp. 763–784.

[47] H. Ren et al., "Time-series anomaly detection service at Microsoft," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 3009–3017.

[48] A. P. Nowak, A. Yasin, A. Mendelson, and W. Zwaenepoel, "Establishing a base of trust with performance counters for enterprise workloads," in *Proc. ATC*, 2015, pp. 541–548.

[49] S. Srikanthan, S. Dwarkadas, and K. Shen, "Data sharing or resource contention: Toward performance transparency on multicore systems," in *Proc. ATC*, 2015, pp. 529–540.

[50] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., architectures, protocols Comput. Commun.*, Jul. 2020, pp. 270–282.

[51] J. Li, J. Qian, and H. Guan, "A holistic model for performance prediction and optimization on NUMA-based virtualized systems," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2019, pp. 352–360.

[52] Y. Wu, A. Chen, and L. T. X. Phan, "Zeno: Diagnosing performance problems with temporal provenance," in *Proc. NSDI*, 2019, pp. 395–420.

[53] D. Mvondo et al., "OFC: An opportunistic caching system for FaaS platforms," in *Proc. 16th Eur. Conf. Comput. Syst.*, Apr. 2021, pp. 228–244.

[54] Y. Zhang et al., "Faster and cheaper serverless computing on harvested resources," in *Proc. 28th Symp. Operating Syst. Princ.*, Oct. 2021, pp. 724–739.

[55] *I. Perfmon-Events*. Accessed: Apr. 1, 2026. [Online]. Available: https://perfmon-events.intel.com/index.html?pltfrm=haswellserver.html

[56] *Linux Manual Pages Perfeventopen(2)*. Accessed: Apr. 1, 2026. [Online]. Available: https://man7.org/linux/man-pages/man2/perfeventopen.2.html

[57] *AWS Lambda Ruby Runtime*. Accessed: Apr. 1, 2026. [Online]. Available: https://github.com/aws/aws-lambda-ruby-runtime-interface-client

[58] *AWS Lambda Nodejs Runtime*. Accessed: Apr. 1, 2026. [Online]. Available: https://github.com/aws/aws-lambda-nodejs-runtime-interface-client

[59] *AWS Lambda Python Runtime*. Accessed: Apr. 1, 2026. [Online]. Available: https://github.com/aws/aws-lambda-python-runtime-interface-client

[60] V. Harsh, W. Zhou, S. Ashok, R. N. Mysore, B. Godfrey, and S. Banerjee, "Murphy: Performance diagnosis of distributed cloud applications," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 438–451.

[61] S. Ashok, V. Harsh, B. Godfrey, R. Mittal, S. Parthasarathy, and L. Shwartz, "TraceWeaver: Distributed request tracing for microservices without application modification," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 828–842.

[62] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. ATC*, 2020, pp. 205–218.

[63] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. ATC*, 2018, pp. 133–145.

[64] *BayesianGaussianMixture*. Accessed: Apr. 1, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.mixture.BayesianGaussianMixture.html

[65] *MinCovDet*. Accessed: Apr. 1, 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.covariance.MinCovDet.html

[66] *Kubernetes Documentation-Configuration Best Practices*. Accessed: Apr. 1, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/overview/

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.

[67] H. M. Demoulin, I. Pedisich, N. Vasilakis, V. Liu, B. T. Loo, and L. T. X. Phan, "Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame," in *Proc. ATC*, 2019, pp. 693–708.
[68] D. Liu et al., "MicroHECL: High-efficient root cause localization in large-scale microservice systems," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng., Softw. Eng. Pract.*, May 2021, pp. 338–347.
[69] D. Luo et al., "Moderntcn: A modern pure convolution structure for general time series analysis," in *Proc. ICLR*, 2024, pp. 1–43.
[70] Y. Feng, W. Zhang, Y. Fu, W. Jiang, J. Zhu, and W. Ren, "SensitiveHUE: Multivariate time series anomaly detection by enhancing the sensitivity to normal patterns," in *Proc. 30th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, Aug. 2024, pp. 782–793.
[71] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A serverless benchmark suite for function-as-a-service computing," in *Proc. 22nd Int. Middleware Conf.*, Nov. 2021, pp. 64–78.
[72] *Knative*. Accessed: Apr. 1, 2026. [Online]. Available: https://knative.dev/docs/
[73] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *Proc. Secur.*, 2015, pp. 913–928.
[74] 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 *Proc. 16th ACM Conf. Comput. Commun. Secur.*, Nov. 2009, pp. 199–212.
[75] X. Zhang, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud," in *Proc. Secur.*, 2015, pp. 929–944.
[76] *AWS Cost Anomaly Detection*. Accessed: Apr. 1, 2026. [Online]. Available: https://aws.amazon.com/aws-cost-management/aws-cost-anomaly-detection/
[77] *Using Lambda Insights*. Accessed: Apr. 1, 2026. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Lambda-Insights.html
[78] *C. monitoring Azure Functions*. Accessed: Apr. 1, 2026. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/configure-monitoring
[79] J. Shen, H. Zhang, Y. Geng, J. Li, J. Wang, and M. Xu, "Gringotts: Fast and accurate internal denial-of-wallet detection for serverless computing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 2627–2641, doi: 10.1145/3548606.3560629.
[80] F. J. Massey, "The Kolmogorov–Smirnov test for goodness of fit," *J. Amer. Stat. Assoc.*, vol. 46, no. 253, pp. 68–78, Mar. 1951.
[81] D. Kelly, F. G. Glavin, and E. Barrett, "DoWNet—Classification of denial-of-wallet attacks on serverless application traffic," *J. Cybersecurity*, vol. 10, no. 1, p. 004, Jan. 2024.
[82] L. Ben-Shimol et al., "Observability and incident response in managed serverless environments using ontology-based log monitoring," *IEEE Trans. Cloud Comput.*, vol. 13, no. 4, pp. 1161–1176, Oct. 2025.
[83] P. Renukadevi, S. Amaran, A. Vikram, T. Prabhakara Rao, and M. K. Ishak, "Enhancing cybersecurity through fusion of optimization with deep wavelet neural networks on denial of wallet attack detection in serverless computing," *IEEE Access*, vol. 13, pp. 47111–47122, 2025.

Junxian Shen received the B.Eng. degree from the Department of Computer Science and Technology, Tsinghua University, and the Ph.D. degree from the Institute for Network Sciences and Cyberspace, Tsinghua University. He is currently an Assistant Professor with the School of Computer Science and Engineering, South China University of Technology, China. His research interests include networked system management and observability.

Han Zhang (Member, IEEE) received the B.E. degree in computer science from Jilin University and the Ph.D. degree in computer science from Tsinghua University. He is currently an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. He is broadly interested in network systems and security, with an emphasis on wide area networks, internet, and the cloud. His research strives to make these systems smart and secure by bridging networking and security with observability, data science, machine learning, formal verifications, artificial intelligence, and distributed computing.

Weiwei Lin (Senior Member, IEEE) received the B.S. and M.S. degrees from Nanchang University in 2001 and 2004, respectively, and the Ph.D. degree in computer application from South China University of Technology in 2007. He has been a Visiting Scholar at Clemson University from 2016 to 2017. Currently, he is a Professor with the School of Computer Science and Engineering, South China University of Technology. He has published more than 200 papers in refereed journals and conference proceedings. His research interests include distributed systems, cloud computing, and AI application technologies. He is a Distinguished Member of CCF. He has been a reviewer of many international journals, including ICML, NeurIPS, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTIONS ON COMPUTERS, and IEEE TRANSACTIONS ON MOBILE COMPUTING.

Yantao Geng received the B.S. degree from the Department of Automation, Tsinghua University, China. He is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests focus on monitoring and fault diagnosis of distributed applications.

Jiawei Li received the B.E. degree in software engineering from Wuhan University of Technology and the Ph.D. degree in cyberspace security from Tsinghua University. Her research focuses on machine learning and security.

Jilong Wang (Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, in 2000. He is currently a Professor with Tsinghua University. His research interests include network measurement, location-oriented networks, SDN systems, and network security.

Mingwei Xu (Senior Member, IEEE) received the B.S. and Ph.D. degrees from Tsinghua University, Beijing, China. He is currently a Full Professor with the Department of Computer Science, Tsinghua University. He is a winner of the National Science Foundation for Distinguished Young Scholars of China.

Authorized licensed use limited to: Tsinghua University. Downloaded on October 15,2026 at 12:54:53 UTC from IEEE Xplore. Restrictions apply.