sjst.scst.edu.ly

Surman Journal for Science and Technology
ISSN: Online (2790-5721) - Print (2790-5713)

مجلة صرمان للعلوم والتقنية
Vol 8, No.2. Jun – Dec. 2026
Pages: 247 ~ 259

Timing Side-Channel Analysis of Cold Start Behavior in OpenFaaS Environments

Retaj Alwan Hassan*, Nuha Omran Abokhdair, Rawad Massoud Salman

Faculty of Science, University of Zawia, Alzawiya - Libya

pg5252018022@zu.edu.ly

ABSTRACT

Serverless computing has seen rapid adoption in recent years. Platforms like AWS Lambda and Google Cloud Functions allow developers to deploy code without managing servers. However, the cold start problem—the delay when a function is invoked after being idle—remains a challenge. Most research treats cold starts as a performance issue. In this paper, we examine them from a security perspective. Specifically, we ask whether the timing difference between cold and warm starts can be used as a side-channel to infer information about a FaaS environment. Two experiments were conducted on OpenFaaS deployed on Kubernetes. In the first, response times were measured under cold and warm start conditions across 50 trials. Cold starts averaged 1415.4ms, warm starts 91.8ms. Cold starts were detected with 84% accuracy ($p < 0.000001$). In the second experiment, two functions were deployed in the same Pod—a victim and a CPU-intensive target—and the victim's response time was measured under idle and loaded conditions. Response times rose from 58.3ms to 113.8ms under load, yielding 90% detection ($p < 0.000001$). The results suggest that, under the tested OpenFaaS configuration, container-level isolation may not fully mitigate timing side-channel leakage. These findings indicate the potential feasibility of such attacks, though further validation under production environments is warranted.

Keywords: *serverless computing, cold start, timing side-channel attack, OpenFaaS, Kubernetes, FaaS security*.

Submitted: 10/06/2026 *Accepted: 14/07/2026*

INTRODUCTION

Serverless computing, and specifically the Function-as-a-Service (FaaS) model, has changed how applications are built. Developers write functions, upload them, and the platform handles scaling, provisioning, and billing. This convenience comes with new security challenges that are not yet fully understood.

One of these challenges is the cold start phenomenon. When a function is invoked after a period of inactivity, the platform must initialize a new container, load the runtime, and prepare the execution environment. This process can take anywhere from a few hundred milliseconds to over two seconds. In contrast, a warm function—one that was recently invoked—responds almost immediately. The difference between these two states is large and measurable.

Most existing work on cold starts focuses on performance. Researchers have proposed keep-alive policies, predictive warming, and image sharing techniques to reduce cold start latency [1]. However, there is a noticeable gap: very few studies ask whether this timing difference can be exploited for malicious purposes. If an attacker can measure response times from outside the

platform, they might be able to tell whether a function is cold or warm. From that, they could infer usage patterns, detect idle periods, or even sense the presence of other tenants on the same infrastructure.

This idea is not far-fetched. Timing side-channel attacks have been demonstrated in production FaaS environments. Zhao et al. showed that cache-based side-channel attacks work in Google Cloud Run, extracting cryptographic secrets from a victim container despite noise from other tenants [2]. Shao et al. developed a methodology for constructing co-location attacks on serverless clouds and demonstrated them on Microsoft Azure Functions [3]. Pathade et al. recently classified cold start exploitation as a distinct function-level vulnerability in the FaaS attack surface [4].

What is missing is work that directly tests whether cold start timing can be exploited as a side-channel in an open-source FaaS platform. This is the gap the present study aims to address.

OpenFaaS was chosen because it is open-source, runs on Kubernetes, and has been used in prior academic security research including provenance tracking and auditing [5], [6]. Using an open-source platform provides visibility into container lifecycle events that proprietary platforms hide.

The research questions:

1. Can cold start events be reliably detected through remote timing measurements alone?
2. Can the activity of a co-located function be detected through timing side-channel observations?

BACKGROUND AND RELATED WORK

A. Cold Start in Serverless Computing

Cold start latency is one of the most studied problems in serverless computing. It occurs because an idle function's container is deallocated to save resources. When a new request arrives, the platform must provision a fresh container, load the function image, initialize the runtime, and execute any startup code. This process takes time.

Research on cold start has largely focused on measurement and mitigation. Golec et al. provided a systematic review of cold start latency in serverless computing, categorizing mitigation strategies and identifying trends in the research landscape [1]. Other approaches include snapshot-based restoration and predictive warming. These studies treat cold starts as a performance problem. They aim to make cold starts faster, not to understand what information they might leak. Alzayat et al. demonstrated that container reuse to avoid cold starts has security implications, as bugs in function implementations or third-party libraries can leak private data between invocations [7]. Marin et al. provided a comprehensive security analysis of the serverless computing paradigm, identifying the shared responsibility model between cloud providers and users as a key source of ambiguity in security enforcement [8]. Escaleira et al. conducted a systematic review of security mechanisms proposed for serverless platforms, finding that while many defenses target application-level threats, comparatively few address infrastructure-level vulnerabilities such as cold start timing or cross-tenant resource contention [9].

B. Timing Side-Channel Attacks

Timing side-channel attacks work by measuring how long a system takes to respond and correlating that timing with internal state. They do not require breaching system defenses. The attacker simply observes.

In FaaS environments, these attacks have been proven feasible. Zhao et al. demonstrated that last-level cache side-channel attacks can extract ECDSA nonce bits from a co-located victim container in Google Cloud Run in approximately 19 seconds on average [2]. The same research group extended these findings by demonstrating a complete attack chain—from achieving co-location with a target function to exfiltrating sensitive data—establishing that cache side-channel attacks are not merely feasible but practical in modern public cloud environments [10]. This work is particularly relevant because it shows that FaaS side-channel attacks succeed despite production noise and limited attack windows.

C. Co-Location Risks

Serverless platforms are multi-tenant. Functions from different users may execute on the same physical machine or container orchestration node. This sharing creates potential side-channels through resource contention.

Shao et al. presented a comprehensive methodology for constructing co-location attacks on serverless clouds [3]. They identified exploitable features in scheduling algorithms and demonstrated instance co-location on both open-source infrastructures and Microsoft Azure Functions. Their work also proposed a scheduler-level mitigation called Double-Dip. The Kumo simulator further demonstrated how scheduling decisions directly affect the success of co-location attacks in serverless environments [11]. Zhao et al. further demonstrated the practicality of co-location attacks on public cloud FaaS platforms, showing that attackers can achieve function co-residency with targeted victims and exploit shared microarchitectural resources to extract sensitive information [12].

D. Security Research on OpenFaaS

OpenFaaS has been used as a research platform in several security studies. Datta et al. built ALASTOR, a provenance-based auditing framework for investigating serverless intrusions, on OpenFaaS [5]. The same research group further extended this work by developing techniques for detecting compromised functions in FaaS environments, demonstrating that behavioral anomalies at the function level can be identified through provenance analysis [6]. These prior uses validate OpenFaaS as a suitable platform for security experiments.

E. Research Gap

The existing literature either treats cold starts as a performance issue to be solved, or studies side-channel attacks through mechanisms other than cold start timing. No published work was found that directly tests whether cold start latency can be exploited as a timing side-channel in an open-source FaaS environment. This paper aims to fill that gap.

METHODOLOGY

A. Environment Setup

All experiments were conducted on a local machine with the following software stack: Windows 10, Docker Desktop version 29.4.1, KinD (Kubernetes in Docker) version 1.27.3,

OpenFaaS installed via Helm on Kubernetes, and Python 3.14.4 with Flask, requests, and statistical libraries. Anyone with a machine that runs Docker can reproduce these experiments.

Both functions—the victim and the target—were deployed in the same Kubernetes Pod, inside the openfaas-fn namespace. The attack tool communicated with them through kubectl port-forward.

B. Victim Function Design

A minimal victim function was essential. If the function performed complex operations—reading files, querying databases, processing images—it would not be possible to separate timing variations caused by the platform from those caused by the function itself. A simple Flask application was written that performs exactly one action: sleep for ten milliseconds, then return a response.

The ten-millisecond delay is constant. Any extra time measured in the response cannot come from the function itself. It must be the platform—either a cold start initialization penalty, or CPU contention from a co-located function.

The function was containerized with Docker, pushed to Docker Hub, and deployed on Kubernetes as a Deployment with an associated Service.

C. Experiment 1: Cold Start Detection

1) Objective

The first research question asked whether cold start events can be distinguished from warm invocations through external timing measurements alone. The cold start penalty is well documented as a performance problem; this experiment investigated whether it also functions as an information leak observable from outside the platform.

2) Design Rationale

The procedure was designed to isolate cold start as the single independent variable. By deleting the Pod before each trial, Kubernetes was forced to create a replacement, guaranteeing a genuine cold start—new container, fresh runtime, everything starting from scratch.

The one-second wait before the first request was determined empirically. Sending the request immediately would result in connection errors because the Pod would not be ready. Waiting too long would allow the Pod to complete initialization, eliminating the cold start effect. One second proved to be the optimal interval: short enough to catch the Pod during initialization, long enough to avoid failures.

The first request after the wait captures the cold start measurement. The second request, sent immediately after, captures a warm start—same function, same network conditions, but now the container is active and Flask is ready. The difference between these two values represents the cold start penalty.

Fifty trials were conducted. This number provides sufficient data for a stable average and a valid t-test, while remaining practical to execute. Failed requests—Pod not ready, connection dropped—were discarded and the trial was retried to ensure clean data.

3) Procedure

The flowchart in Fig. 1 illustrates the full process. Each trial followed these steps:

1. **Force Cold Start.** Delete the Pod hosting the victim function using kubectl delete pod. Kubernetes automatically creates a replacement.
2. **Wait for Initialization.** Pause for one second while the new Pod begins its initialization sequence—pulling the container image, starting the Python interpreter, and loading Flask.
3. **Measure Cold Start.** Send an HTTP GET request and record the round-trip time using time.perf_counter_ns().
4. **Measure Warm Start.** Send a second HTTP GET request immediately after the first one returns.
5. **Store the Pair.** If both requests succeed, save the two measurements as a (cold, warm) pair. If either fails, discard and retry.
6. **Repeat.** Continue until 50 successful trials are collected.

4) Flowchart

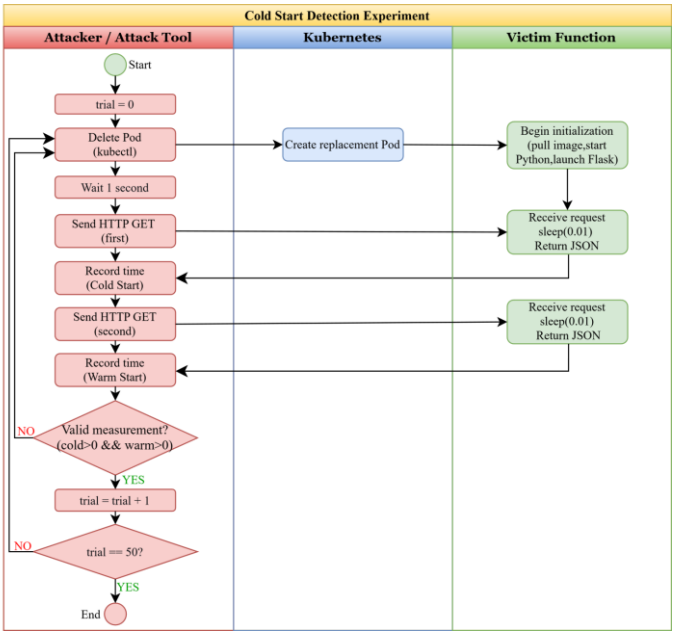

Figure. 1. *Swimlane activity diagram showing the interaction among the attacker, Kubernetes, and the victim function during one trial of the cold start detection experiment.*

D. Experiment 2: Co-Tenant Detection

1) Objective

The second research question asked whether the activity of a co-located function can be detected through timing measurements of a victim function. This extends the investigation from platform-level events—cold starts—to tenant-level activity, which represents a more subtle and potentially more dangerous side-channel.

2) Design Rationale

A target function was written specifically to create CPU pressure. It performs 20 million mathematical operations—square roots and sine calculations—per request. This workload is

deliberately CPU-bound rather than I/O-bound. Reading files or waiting for network responses would not create the kind of processor contention needed for this experiment.

Both functions were deployed in the same Kubernetes Pod. This configuration was intentionally selected to evaluate the upper bound of timing leakage under worst-case resource sharing conditions. By forcing both containers to share the same CPU and memory allocation, the setup creates a scenario where interference is maximized—representing the most pessimistic case for isolation. The same-Pod deployment does not claim to represent the default behavior of OpenFaaS or Kubernetes in production, where schedulers typically distribute functions across distinct Pods. Rather, it serves as a controlled experimental condition to determine whether timing side-channels can, in principle, arise from resource contention in a FaaS environment. If leakage cannot be detected even under this extreme scenario, it may suggest that such channels are unlikely to manifest under weaker conditions. Conversely, if leakage is detected here, these findings would motivate further investigation under realistic deployment topologies to assess the practical significance of the observed effects.

The experiment was divided into two phases. The baseline phase—30 measurements with the target idle—establishes what "normal" looks like for the victim function. The attack phase—50 trials flooding the target with 1000 requests—shows what happens when the shared CPU is under contention.

During each attack trial, the victim was measured 10 times and only the maximum value was kept. The maximum was chosen over the average because it reflects the attacker's perspective. An attacker does not need every measurement to show an anomaly—one unusually slow response is enough to leak information. The average might hide brief moments of contention; the maximum reveals them.

3) Procedure

Fig. 2 illustrates the two-phase procedure.

Phase 1: Baseline Measurement

1. Ensure the target function is idle.
2. Send an HTTP GET request to the victim function and measure the response time.
3. Record the value.
4. Repeat until 30 baseline measurements are collected.

Phase 2: Attack Measurement

1. Send 1000 requests to the target function using a background thread. This forces the target to saturate the shared CPU.
2. While the target is under load, send 10 HTTP GET requests to the victim function.
3. Record the maximum of these 10 response times.
4. Store this maximum as one attack measurement.
5. Wait one second before the next trial to allow the system to stabilize.
6. Repeat until 50 attack measurements are collected.

sjst.scst.edu.ly
Surman Journal for Science and Technology
ISSN: Online (2790-5721) - Print (2790-5713)
مجلة صرمان للعلوم والتقنية
Vol 8, No.2. Jun – Dec. 2026
Pages: 247 ~ 259

4) Flowchart

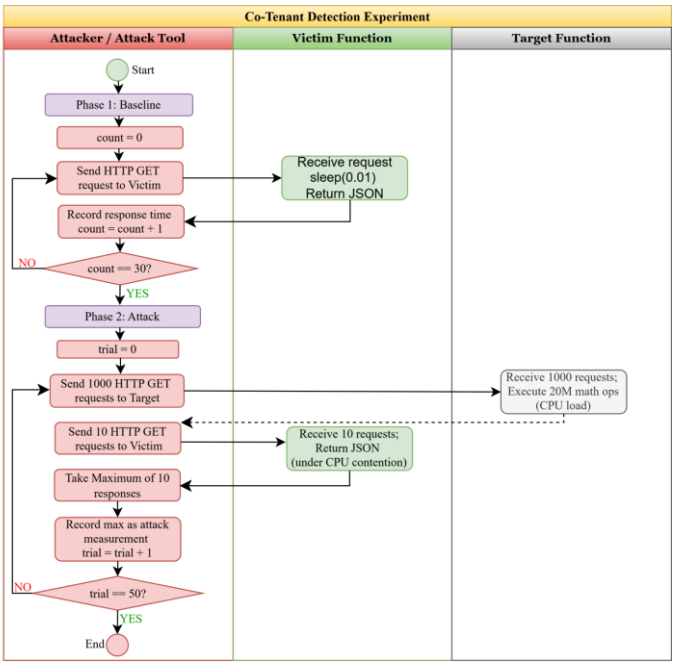

Figure. 2. *Swimlane activity diagram showing the two-phase procedure of the co-tenant detection experiment.*

E. Statistical Analysis

For both experiments, the core analytical task was the same: compare two sets of measurements and determine whether the difference between them is statistically significant.

For Experiment 1 (Cold Start Detection), a paired t-test was applied, because each cold start measurement was paired with a warm start measurement from the same trial—same Pod, same network conditions. For Experiment 2 (Co-Tenant Detection), an independent samples t-test was used, as the baseline and attack measurements were collected in separate phases under different conditions. A p-value below 0.05 is considered statistically significant; values below 0.001 indicate strong evidence that the difference is real.

Detection thresholds were also calculated. For the cold start experiment, the threshold was set at $\mu\_warm + 3\sigma\_warm$. Any response time exceeding this value was classified as a cold start. For the co-tenant experiment, the threshold was set at $\mu\_baseline + 2\sigma\_baseline$. Two standard deviations were used rather than three because the effect size was smaller and a more sensitive threshold was required.

Finally, detection rates were computed as the proportion of trials correctly identified. This percentage reflects how useful the side-channel would be to an actual attacker.

RESULTS

A. Experiment 1: Cold Start Detection

Table I summarizes the results from 50 trials of cold and warm start measurements.

sjst.scst.edu.ly
Surman Journal for Science and Technology
ISSN: Online (2790-5721) - Print (2790-5713)
مجلة صرمان للعلوم والتقنية
Vol 8, No.2. Jun – Dec. 2026
Pages: 247 ~ 259

TABLE I. COLD VS. WARM START RESPONSE TIMES

Metric	Cold	Warm
Mean	1415.4ms	91.8ms
Std Dev	783.8ms	22.7ms
Min	61.4ms	32.5ms
Max	2171.2ms	123.4ms

The average cold start took 1415.4ms, compared to 91.8ms for a warm start—a difference of 1323.6ms. Cold start times showed substantial variability (SD 783.8ms), reflecting the inherent unpredictability of container startup. Warm start times were tightly clustered (SD 22.7ms), indicating consistent performance once the function is active.

A paired t-test confirmed the difference is statistically significant: $t = 11.97$, $p < 0.000001$. A detection threshold was calculated as $\mu_warm + 3\sigma_warm = 159.9$ms. Any response time above this threshold was classified as a cold start. Using this rule, 42 out of 50 cold starts were correctly identified, yielding an 84% detection rate. The eight missed detections occurred when the new Pod initialized unusually quickly, with response times between 61.4ms and 123.0ms.

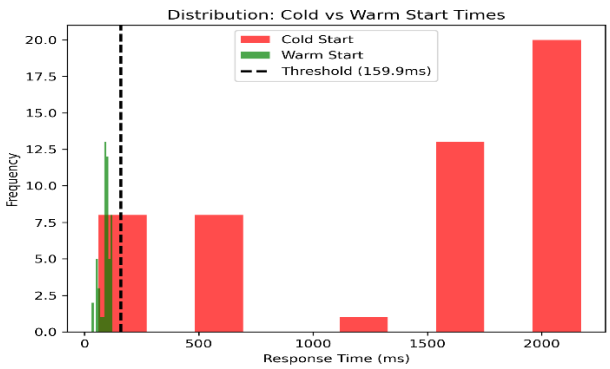

Figure. 3. *Histogram comparing cold start and warm start response time distributions.*

Fig. 3 illustrates the cold start and warm start response time distributions. The red bars represent cold start response times, while the green bars represent warm start response times. The black dashed line marks the detection threshold ($\mu_warm + 3\sigma_warm = 159.9$ms). The two distributions show minimal overlap, indicating that cold and warm starts can be reliably distinguished by response time alone. The wide spread of cold start times (61.4–2171.2ms) reflects the inherent variability of container initialization, whereas the tight clustering of warm start times (32.5–123.4ms) demonstrates consistent performance once the function is active.

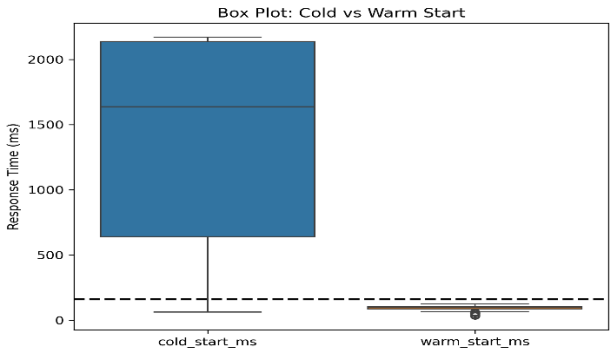

Figure. 4. *Box plot comparing cold and warm start response time distributions.*

Fig. 4 illustrates the cold start and warm start response time distributions. The cold start box shows a wide interquartile range with several outliers, reflecting high variability in container startup times. The warm start box is compact with no outliers, indicating consistent and predictable response times once the function is active.

B. Experiment 2: Co-Tenant Detection

Table II summarizes the results from the co-tenant detection experiment.

TABLE II. BASELINE VS. UNDER ATTACK RESPONSE TIMES

Metric	Baseline	Under Attack
Mean	58.3ms	113.8ms
Std Dev	10.9ms	32.6ms
Min	32.0ms	70.4ms
Max	73.8ms	213.5ms

Under CPU load from the target function, the victim's response time increased by 95%, from 58.3ms to 113.8ms. The mean difference was 55.5ms. While smaller than the cold start difference, this increase was consistent across trials.

A t-test confirmed statistical significance: $t = -9.01$, $p < 0.000001$. For this experiment, a detection threshold of $\mu_baseline + 2\sigma_baseline = 80.1$ms was used. Two standard deviations were used rather than three because the effect size was smaller and a tighter threshold was needed for adequate sensitivity. With this threshold, 45 out of 50 attack trials were correctly identified, yielding 90% detection.

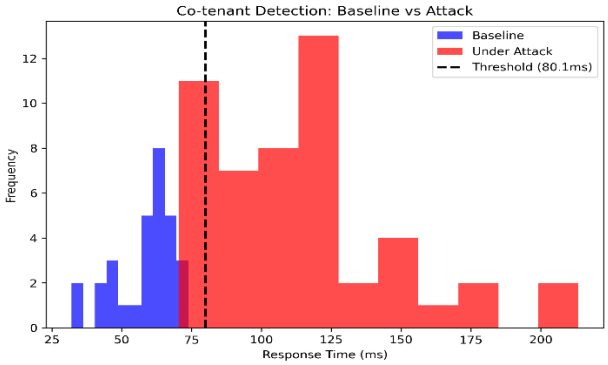

Figure. 5. *Histogram comparing baseline and under-attack response time distributions.*

Fig. 5 illustrates the baseline and under-attack response time distributions. The blue bars represent baseline measurements (mean 58.3ms, target function idle). The red bars represent measurements taken under CPU contention (mean 113.8ms, target function under load). The black dashed line marks the detection threshold ($\mu_baseline + 2\sigma_baseline = 80.1$ms). Although some overlap exists between the two distributions, the under-attack measurements are visibly shifted to the right, indicating the impact of co-tenant CPU load on victim function response time.

Surman Journal for Science and Technology
ISSN: Online (2790-5721) - Print (2790-5713)
sjst.scst.edu.ly
مجلة صرمان للعلوم والتقنية
Vol 8, No.2. Jun – Dec. 2026
Pages: 247 ~ 259

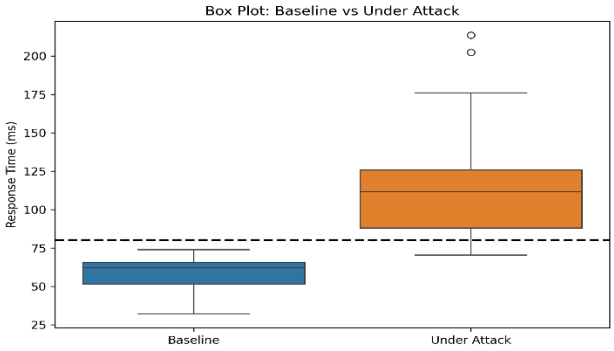

Figure. 6. *Box plot comparing baseline and under-attack response time distributions.*

Fig. 6 illustrates the baseline and under-attack response time distributions. The under-attack box is shifted upward and shows greater spread than the baseline box. The increased median and wider interquartile range under attack conditions reflect the additional latency and variability introduced by CPU contention from the co-located target function.

C. Classification Performance

To further characterize the effectiveness of the timing side-channel, the detection results for both experiments are summarized as simple classification tables in Table III. A True Positive (TP) represents a correctly identified cold start (Experiment 1) or co-tenant activity (Experiment 2). A False Negative (FN) represents a missed detection. Since the detection threshold is applied to individual response time measurements known to be cold starts or attack trials, the notion of a False Positive (FP) does not arise in the traditional sense. The table therefore reports TP, FN, and the total number of trials for each experiment.

TABLE III. CLASSIFICATION PERFORMANCE SUMMARY

Metric	Experiment 1 (Cold Start)	Experiment 2 (Co-Tenant)
Total Trials	50	50
True Positives (TP)	42	45
False Negatives (FN)	8	5
Detection Rate (TP/Total)	84%	90%

As shown in Table III, Experiment 1 correctly identified 42 out of 50 cold starts, with 8 missed detections occurring when the new Pod initialized quickly enough to produce response times below the 159.9ms threshold. Experiment 2 correctly identified 45 out of 50 co-tenant activity trials, with 5 missed detections where CPU contention was insufficient to raise the victim's response time above the 80.1ms threshold.

DISCUSSION

A. Interpretation of Results

The results from both experiments, supported by the visual evidence in Figs. 3-6, provide evidence consistent with the feasibility of timing side-channel observations in the tested FaaS environment. However, these findings should be interpreted as preliminary, as they are derived from a controlled, single-node deployment and may not generalize to all production settings. The cold start experiment produced a particularly large timing difference—over one second—making cold start events trivially detectable from outside the system. Even with the natural variability of container startup, 84% accuracy is reliable enough to be useful to an attacker.

The co-tenant experiment showed a timing difference of 55.5ms—substantially smaller than the cold start difference, reflecting the subtler nature of resource contention. By taking multiple measurements per trial and recording the maximum, a detection rate of 90% was achieved with strong statistical significance ($t = -9.01$, $p < 0.000001$).

These results align with Zhao et al.'s finding that FaaS side-channel attacks work despite production noise [2], and with Shao et al.'s demonstration that serverless schedulers have exploitable features for co-location attacks [3].

B. Security Implications

If these findings were to generalize to production environments, an external observer who can invoke a function and measure response times could potentially:

1. Determine when the function was last used, because a cold start indicates a period of inactivity.
2. Map usage patterns over time—when the system is busy and when it is idle.
3. Detect whether other tenants are active on the same infrastructure.
4. Choose the best time to launch other attacks, when monitoring is likely to be minimal.

Notably, these observations would not require breaching the platform; they rely solely on invoking functions and measuring response times—actions available to any authorized user of a FaaS platform.

C. Limitations

This study has several limitations that should be explicitly acknowledged.

First, all experiments were conducted on a single local machine using KinD (Kubernetes in Docker). This environment, while providing full control over container lifecycle events, does not replicate the scale, network complexity, and hardware diversity of production cloud platforms such as AWS Lambda, Google Cloud Run, or Azure Functions. The timing measurements reported in this paper reflect the behavior of a local, single-node Kubernetes cluster and should be interpreted accordingly.

Second, and importantly, the results presented here constitute a proof-of-concept. The primary contribution of this work is demonstrating the existence and exploitability of a timing side-channel through cold start latency in a FaaS environment. This paper does not assert that the attack succeeds with identical effectiveness on commercial serverless platforms. Factors such as proprietary scheduling algorithms, multi-tenant isolation mechanisms, and hardware heterogeneity in production clouds may significantly affect the feasibility and detection rates of the proposed attack.

Third, the victim function employs a fixed 10ms workload. Real-world functions exhibit variable execution times depending on input data, backend services, and resource requirements. This variability could either mask or amplify the timing differences observed in this study.

Fourth, while 50 trials per experiment provide strong statistical significance, larger sample sizes conducted across diverse conditions may reveal additional patterns or edge cases not captured here.

Finally, the target function in Experiment 2 was deliberately designed to be computationally intensive to simulate a worst-case co-tenancy scenario. In practice, the resource consumption patterns of real co-located functions may produce different contention effects than those observed in this study.

D. Potential Defenses

Several countermeasures could reduce the effectiveness of these attacks:

1. **Timing jitter:** Adding small random delays to responses would make cold start timing harder to distinguish from normal variability.
2. **Keep-alive improvements:** Keeping functions warm for longer periods would reduce cold start frequency, shrinking the attack window.
3. **Scheduler hardening:** Mechanisms like the Double-Dip scheduler proposed by Shao et al. [3] could make co-location harder to achieve.
4. **Request pattern monitoring:** Repeated timing measurement requests might be detectable by analyzing invocation patterns.

CONCLUSION AND FUTURE WORK

This paper investigated whether cold start timing can be exploited as a side-channel in FaaS environments. Based on two experiments conducted on OpenFaaS deployed on Kubernetes, the results indicate that measurable timing differences exist between cold and warm starts (over one second on average), with a detection rate of 84% under the tested conditions. Similarly, co-tenant CPU activity was associated with an observable timing shift in the victim function's response time, achieving a 90% detection rate in this experimental setup. Both findings were statistically significant at $p < 0.000001$.

These results suggest that, under the specific configuration tested, container isolation mechanisms may not fully eliminate timing side-channel leakage. The findings indicate the potential feasibility of such attacks, though they do not confirm that these observations would hold in all production deployments. If validated more broadly, the only capability needed to exploit this side-channel would be the ability to invoke functions and measure response times.

Several directions for future work emerge from this study:

1. Replicate the experiments on multi-node Kubernetes clusters and on commercial platforms such as AWS Lambda or Google Cloud Run to assess the generalizability of these findings.
2. Explore whether timing side-channels can reveal more detailed information, such as function type, input data characteristics, or code structure.
3. Develop and evaluate potential countermeasures, including response timing jitter, intelligent keep-alive policies, and scheduler-level isolation mechanisms.

Investigate whether machine learning models can detect timing side-channel attacks by recognizing anomalous measurement patterns in invocation logs.

REFERENCES

[1] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions," *ACM Comput. Surv.*, vol. 57, no. 3, 2024, doi: 10.1145/3700875.

[2] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud," *Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS*, vol. 2, pp. 582–600, 2024, doi: 10.1145/3620665.3640403.

[3] W. Shao *et al.*, "Bit of a Close Talker: A Practical Guide to Serverless Cloud Co-Location Attacks," 2026, [Online]. Available: http://arxiv.org/abs/2512.10361

[4] C. Pathade, V. Dhimam, S. Ahmad, and I. Lareb, "Serverless AI Security: Attack Surface Analysis and Runtime Protection Mechanisms for FaaS-Based Machine Learning," 2026, [Online]. Available: http://arxiv.org/abs/2601.11664

[5] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the Provenance of Serverless Intrusions," *Proc. 31st USENIX Secur. Symp. Secur. 2022*, pp. 2443–2460, 2022.

[6] L. Ben-Shimol *et al.*, "Detection of compromised functions in a serverless cloud environment," *Comput. Secur.*, vol. 150, 2025, doi: 10.1016/j.cose.2024.104261.

[7] M. Alzayat, J. Mace, P. Druschel, and D. Garg, "Groundhog: Efficient Request Isolation in FaaS," *Proc. 18th Eur. Conf. Comput. Syst. EuroSys 2023*, pp. 398–415, 2023, doi: 10.1145/3552326.3567503.

[8] E. Marin, D. Perino, and R. Di Pietro, "Serverless computing: a security perspective," *J. Cloud Comput.*, vol. 11, no. 1, 2022, doi: 10.1186/s13677-022-00347-w.

[9] P. Escaleira, V. A. Cunha, J. P. Barraca, D. Gomes, and R. L. Aguiar, "A systematic review on security mechanisms for serverless computing," *Cluster Comput.*, vol. 28, no. 7, 2025, doi: 10.1007/s10586-025-05371-4.

[10] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "From Colocation to Exfiltration: Practical Cache Side-Channel Attacks in the Modern Public Cloud," *IEEE Micro*, vol. 45, no. 4, pp. 95–102, 2025, doi: 10.1109/MM.2025.3574715.

[11] W. Shao, K. Khasawneh, S. Rafatirad, H. Homayoun, and C. Fang, "Kumo: A Security-Focused Serverless Cloud Simulator," 2026, [Online]. Available: http://arxiv.org/abs/2603.19787

[12] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, *Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS*, vol. 1, no. 1. Association for Computing Machinery, 2024. doi: 10.1145/3617232.3624867.