Shared Resource Entanglement Attacks against Serverless Computing

Yuwen Cui*, Mingkui Wei‡, Zhuo Lu†, Yao Liu*

*Department of Computer †Department of Electrical Engineering, University of South Florida

‡Department of Cyber Security Engineering, George Mason University

Email: ycui@usf.edu, yliu@cse.usf.edu, mwei2@gmu.edu, zhuolu@usf.edu

***Abstract*—Serverless computing has emerged as a popular paradigm for building and deploying applications in the cloud. At the heart of this paradigm lies the concept of serverless functions, which have gained significant popularity due to their simplicity, scalability, and cost-efficiency. However, running serverless services on the same physical machine can introduce security threats and risks, primarily when identical serverless functions belong to different tenants. In this paper, we investigate the resource consumption of serverless functions and propose a shared resource entanglement (SRE) threat model. The SRE threat model enables adversaries to infer sensitive data from a victim's serverless function by intercepting and monitoring the host kernel interacting data patterns and the data load and write rate patterns. The experiment results demonstrate that adversaries can identify patterns in the data processing of co-resident serverless functions using the SRE threat model. Also, this vulnerability could lead to the leakage of sensitive data in the Kubernetes platform.**

***Index Terms*—serverless functions, shared resource entanglement (SRE), co-resident**

I. Introduction

Serverless functions, often called Function-as-a-Service (FaaS), allow developers to write modular and event-driven code that can be executed in a highly scalable and managed environment. With serverless functions, developers can build applications as a collection of individual functions, each responsible for a specific task, and seamlessly integrate them to create complex workflows [1]–[4]. As shown in Fig. 1, a cluster node is depicted with the capability to configure multiple serverless functions. An API gateway is set up to enable web or mobile clients to access each serverless function within a cluster node. Each cluster node can accommodate several pods, and each pod can host one or multiple serverless functions. In serverless platforms, a pod is the smallest deployable unit that represents a single instance of a process or a set of tightly coupled processes running together on a single host. Therefore, a significant security concern about serverless functions is the co-residency threat, which allows the attacker to know that its serverless function runs on the same physical hardware as that of a target user.

The shared nature of underlying resources among co-resident serverless functions enables attackers to acquire sensitive data from a serverless function created by another user. When sensitive data is processed by a function belonging to another user and persists in shared memory or cache, there is a potential risk of the attacker's function gaining access to or inferring this data. This paper investigates the potential vulnerabilities of data

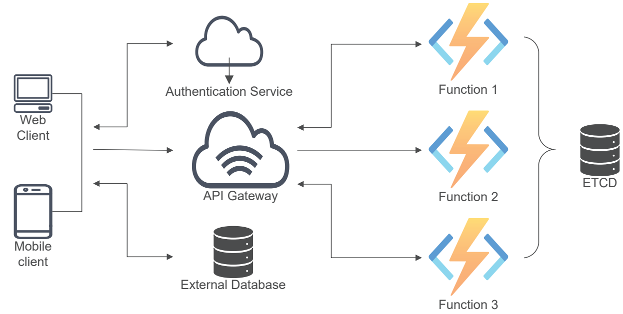


Fig. 1. The serverless computing paradigm alleviates the complexities associated with developing cloud-native applications by providing a high-level programming abstraction known as Function-as-a-Service (FaaS).

leakage resulting from the interference between co-resident serverless functions. By investigating the interference between co-resident serverless functions, we discovered that the shared memory resources of a serverless function can be affected by another. Several previous studies have concentrated on the side channel attacks in cloud virtual machines [5]–[7]. These attacks involve exploiting unintended information leakage through observable resource usage patterns and behaviors of co-resident containers running on the same virtual machine within a cloud environment. However, the state-of-the-art serverless computing platforms typically implement isolation mechanisms to mitigate side-channel attacks, including Flush+flush [8] and Flush+reload [9]. As a result, the current isolation strategies employed by serverless computing platforms make side-channel attacks more challenging to execute. To overcome the challenge of the current isolation strategies in serverless computing platforms, we introduce the concept of Shared Resource Entanglement (SRE), which focuses on the host kernel interacting data pattern of memory information [10] and the data load/write timing patterns where multiple functions, running concurrently on the same underlying infrastructure, become interconnected and mutually influence each other due to resource sharing.

SRE attacks exploit Linux's memory-based pseudo-file systems, which provide access to kernel-related information and configurations. When multiple container tenants share the same kernel, incomplete kernel implementations can expose system-wide memory usage data. By analyzing memory consumption patterns of co-resident serverless functions with varying inputs, attackers can infer the data being processed by the victim function. Additionally, the co-resident serverless functions may exhibit different timing delays while processing the same

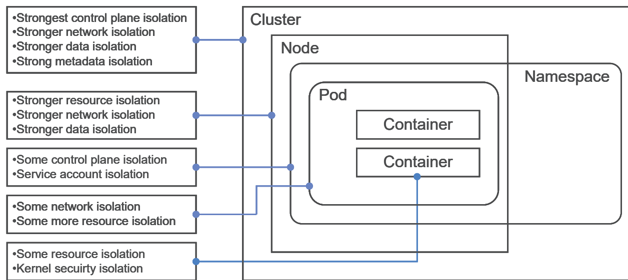


Fig. 2. Isolation of each layer in Kubernetes [11].

data due to hardware optimization policies on a physical machine. These timing delay patterns can also be used to analyze potential data input in co-resident serverless functions. Consequently, serverless functions impact each other's resource availability and performance, creating unique entanglement patterns. This entanglement arises when the resource usage patterns, performance characteristics, or resource demands of one function influence the behavior and performance of other co-resident functions.

To investigate the presence of SRE and assess the effectiveness of the proposed SRE threat model in Kubernetes environments, we conducted experiments using both a locally structured Kubernetes environment and a real-world serverless Kubernetes platform known as Google Kubernetes Engine [12]. The results indicate that accuracy can reach 100% under low-noise conditions in both Google Kubernetes Engine and local Kubernetes environments. When the SRE attack operates under heavy noise conditions, accuracy depends on the margin for discerning noise patterns. With an appropriate margin and noise recognition algorithm, the attacker can enhance accuracy to over 90%. In summary, the main contributions of this research can be summarized as follows:

- We investigate the co-residency attacks in serverless platforms and explore the phenomenon of shared resource interference among serverless functions within the Kubernetes serverless platform.
- We identify the patterns of memory-based pseudo file systems that occur among co-resident serverless functions and then develop a threat model that showcases the identification of relevant SRE patterns.
- We identify the patterns of shared resource load and write time rate that occur among co-resident serverless functions and then develop a threat model that showcases the identification of relevant SRE patterns.

The remaining sections of this paper are structured as follows. Section II introduces the co-residency attacks against serverless computing. Section III outlines our threat model. Sections IV and V detail the experiment results. Section VI discusses defense methods and Section concludes this paper.

II. Background and Motivations

In this section, we will provide a brief overview of the background concerning vulnerabilities in serverless computing and discuss the motivation behind our research.

A. side-channel Attacks

Side-channel attacks exploit physical information leakage, such as cache hits/misses, power consumption, and timing data [7]–[9], [13]–[15]. While prior research on side-channel attacks across various contexts highlights unique challenges in serverless environments, the multi-tenant nature of serverless platforms heightens security risks. This is due to multiple users or organizations sharing the same infrastructure, including physical hosts, VMs, and containers.

Paper [10] and [6] investigate side channel attacks in virtualized environments and discuss the potential dangers associated with co-residency attacks. Although not specific to serverless computing, it provides insights into the exploitation of shared resources and covert channels between multi-tenancy container cloud services, which can be relevant in the context of serverless platforms.

Co-residency attacks entail deploying malicious functions on the same physical host or virtual machine as a target function. Through co-residency, attackers can exploit shared resources and timing variations to infer sensitive information, including cryptographic keys, secret parameters, or confidential data processed by the targeted function [16]–[18]. To increase security, serverless platforms implement various isolation mechanisms to ensure resource separation between functions. As illustrated in Fig. 2, each layer in the Kubernetes project is isolated. However, concerns persist regarding the security implications associated with co-residency.

Anil et al. [19] have successfully implemented a fast co-residency detection method using memory bus hardware. This hardware enables the transmission and reception of bits to detect memory bus contention through shared resources and timing variations in AWS Lambda. Additionally, Alexander and Prateek [20] design and implement an auto-scaling locality-aware load balancer for FaaS platforms. Although their work is not specifically focused on co-residency attacks in serverless platforms, it offers valuable insights into leveraging function performance characteristics within multi-tenancy serverless services. These insights can be relevant in the context of co-residency attacks.

Jens and Mathias [5] conduct a timing-based attack to detect co-resident Virtual Machines (VMs). The proposed timing-based attack can identify the software version in the co-resident VMs by memory deduplication. Paper [6] adapts the network traffic timing delay to detect whether an adversary is a co-resident of its target. The network traffic timing delay is derived from traffic analysis, enabling a malicious co-resident VM to inject a watermark signature into the network flow of the target instance [21].

B. Co-residency Attacks

As demonstrated in paper [6], the network traffic timing delay is relatively straightforward to design and exhibits broader applicability. Consequently, the paper also investigates the co-residency attack in the context of timing-based side-channel attacks. Experimental findings showcase the feasibility and

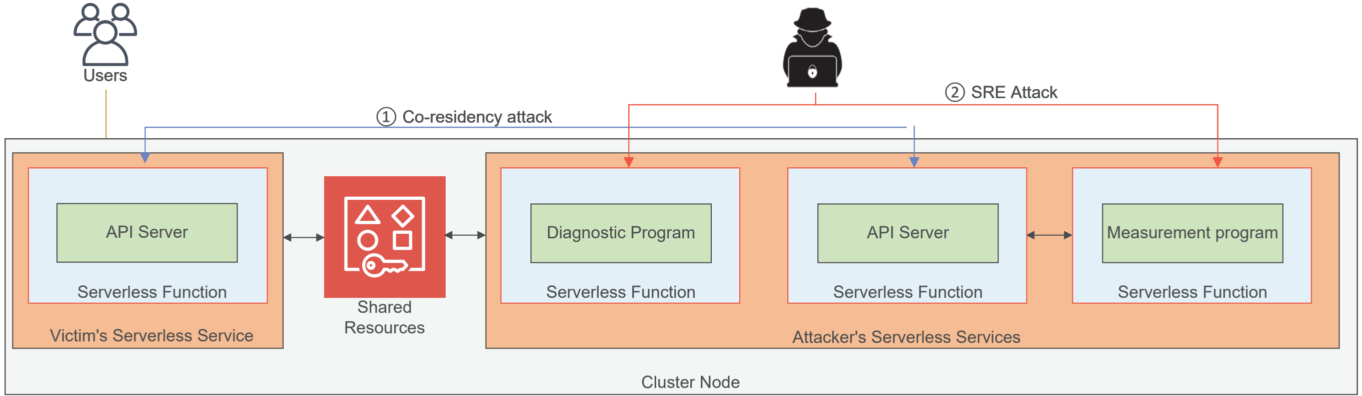


Fig. 3. The architecture of the SRE threat model involves the victim and attacker running relatively isolated serverless functions within a co-located cluster node.

high efficiency of timing-based side-channel attacks for co-residency attacks on serverless platforms.

Unlike previous works, this study primarily focuses on analyzing the mutual interference of free memory and data read/write timing delays in co-resident serverless functions processing identical input data across multiple datasets. By recording these values, we identify patterns that adversaries can use to predict the victim function's input, potentially leading to data leakage. The goal is to determine if specific characteristics emerge during data processing. The discovery of MemFree and timing delay patterns, and their application to the proposed SRE threat model, will be discussed in Section III.

III. Attack Method

This section will demonstrate our attack strategies in serverless computing platforms and the details of the proposed SRE-based attacks against a co-located serverless function that belongs to a victim.

A. Assumption

This work is motivated by the presence of system-level covert channel attacks in serverless computing infrastructures [6], [19]. We go beyond the proposed co-residency threat model and imagine a serverless function in traditional timing channel attacks. The traditional timing channel attacks are typically used to extract sensitive information from a victim's system by monitoring cache behavior and exploiting timing differences in memory access. Leveraging these traditional timing channels, we analyze the data load/write timing delay and memory interacting information based on the host kernel to disclose the data processed within the serverless function.

In the proposed threat model, we make the assumption that the attacker does not intervene in the victim's behaviors. As depicted in Fig. 3, the attacker must initiate a co-residency attack before launching the SRE attack. The objective is to colocate the attacker's serverless function on the same physical machine as the targeted victim's serverless function for an extended period of time. The SRE attack can be executed if the attacker successfully achieves co-location of the serverless functions on the same physical machine. During the SRE attack phase, we assume the attacker runs the same serverless function as the victim. This is facilitated by serverless platforms, which often utilize the same container image from the container hub for identical serverless functions. To optimize memory usage, an overlay filesystem is employed when creating the container image, resulting in shared pages in the page cache among co-resident containers. Through experimentation, it has been observed that running the same serverless function as the victim enables efficient detection of memory consumption information patterns and timing delays in data load/write for the co-resident serverless functions. In the adversary's serverless function, a memory consumption information measurement program and a memory data load/write timing delay measurement program have been employed to collect and monitor shared resource entanglement patterns of memory consumption and memory data load/write timing delay information. During the runtime of measurement programs, a large number of requests will be sent to the attacker's API Server. Furthermore, we assume that the victim's instances are processing extensive data, such as images and videos, allowing the adversary to detect memory consumption information from these instances within a few seconds. Extensive data like images and videos usually consume lots of memory space and spend time to load and write during the data processing phase.

B. Attack Overview

The goal of the co-residency attacks is to establish that containers deployed on serverless platforms remain susceptible to co-residency attacks, irrespective of the pseudo file system. In our proposed SRE threat model, the attacker utilizes this attack to determine if their serverless function is executing on a co-resident cluster node alongside the victim's serverless functions. Subsequently, the attacker can analyze the different sensitivities of the co-resident serverless functions to factors such as architecture, system load, choice of orchestrator, and adversarial activities. In this paper, our primary focus lies on the SRE attack rather than the detection of co-resident serverless functions.

The proposed SRE threat model in this paper includes memory consumption measurement and memory data load/write

```
$ cat /proc/meminfo
MemTotal:       16390360 kB
MemFree:        13921716 kB
MemAvailable:   14771400 kB
Buffers:           98204 kB
Cached:           967640 kB
SwapCached:            0 kB
Active:           430760 kB
Inactive:        1783208 kB
Active(anon):       1104 kB
Inactive(anon):  1148480 kB
Active(file):     429656 kB
Inactive(file):   634728 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:               524 kB
Writeback:             0 kB
AnonPages:       1148180 kB
Mapped:           391844 kB
Shmem:              1428 kB
```

Fig. 4. The overview of */proc/meminfo* file.

timing delay measurement, and the diagnostic program is primarily responsible for intercepting and monitoring the changes and patterns in shared resources during the execution of serverless functions. The diagnostic program requires the attacker to provide input data similar to the victim's serverless function, allowing for the recording of application runtime and data processing characteristics. By leveraging the attacker's input data during the attack, it becomes possible to infer the data processed by the victim's serverless function. In the following sections, we will provide a detailed explanation of the working principles and attack procedures within the SRE threat model.

C. Measurement Program

1) The Memory Utilization Entanglement Patterns: Containers, such as Docker, provide an isolated runtime environment for applications. When running a container, the */proc/meminfo* file within the container represents the memory statistics of the container itself. It gives insights into the memory consumption of the containerized application, regardless of the underlying host system. This allows containerized applications to have visibility into their memory usage, helping with resource management and optimization [10], [17], [22]. In a serverless environment, the */proc/meminfo* file may not be directly accessible or relevant to the developers. Since serverless functions are ephemeral and stateless, the memory management is handled by the platform provider, and developers typically don't have direct control over memory allocation or monitoring. However, we found that developers can deploy malicious containers that run the serverless function to get permission to access the */proc/meminfo* file.

As shown in Fig. 4, the */proc/meminfo* file is a virtual file that contains detailed information about memory usage, including both physical and virtual memory. It provides insights into various memory-related statistics, including total available memory, free memory, used memory, and memory utilization by specific system components. System administrators and developers can leverage this information to monitor and optimize memory usage. For instance, the MemTotal value represents the overall physical memory capacity of the system, which typically remains constant unless there are changes in the system's memory configuration. Therefore, MemTotal is not a suitable characteristic for initiating the fingerprint of SRE. The MemFree value represents the total amount of memory that is currently unused by any process. It dynamically changes as processes allocate or release memory. During data processing, the MemFree value can fluctuate as memory is allocated and deallocated by processes [23]. Additionally, the value of MemFree in a container can be affected by the memory usage of other containers residing on the same physical machine.

When data processing starts in a container, memory usage increases as the processes allocate memory to store and manipulate the data. As memory is allocated, the value of MemFree decreases, indicating a reduction in available free memory. Typically, the MemFree value exhibits variation across three levels: around the baseline case, bit 1, and bit 0 [24]. The containers on a physical machine can allocate varying amounts of memory to represent bits 1 and 0, resulting in an immediate drop in the MemFree field within the */proc/meminfo* file. The value of MemFree in a container can be influenced by the memory usage of other co-resident container processes. As such, we propose to analyze the behavior of co-resident serverless functions based on the MemFree value.

The incomplete implementation of namespaces in the kernel can cause the leakage problem [10]. To identify the leaked information from the co-located serverless function, we trace the memory allocation and consumption through */proc/meminfo* file. */proc/meminfo* file lists a large amount of valuable information about the system's memory usage, including the total amount of usable memory, the amount of physical RAM left unused by the system, and the total amount of dirty memory. The representation of different amounts of bits 1 and 0 in memory will change the MemFree value in */proc/meminfo* file. Due to the different number of bits 0 and 1 when storing different data in memory, the resulting variations in the MemFree value also exhibit distinct patterns. To ensure reliable entanglement between the victim's serverless function and the adversary's serverless function, we design a REST API in a serverless function that includes the interfaces of *TASK_WRITE*, *TASK_LOAD*, *UPDATE_TASK*, and *DELETE_TASK*.

In our threat model, we mainly focus on the interfaces of *TASK_WRITE* and *TASK_LOAD*. The *TASK_WRITE* API interface is used to simulate the victim's use of a serverless function for processing large files. The proposed attack model primarily analyzes the interference of memory consumption and timing delays of memory data load/write between the victim's serverless function and the co-located identical serverless function on the adversary side. This analysis aims to determine if the adversary and victim are using the same data,

potentially leading to the leakage of the victim's sensitive data. Throughout this process, the adversary relies on the victim's serverless function's memory writing processes during execution. In real-world scenarios, this experiment can simulate the processing of large files.

Algorithm 1 The memory consumption entanglement detection

Mem_{free}: the current MemFree value from */proc/meminfo* file.

$M_{recording[]}$: the free memory size recording list.

S: the current state of serverless function, True means running, False means stop.

BLOCK_SIZE: the memory block size to load.

DIFFERENCE: the difference value before the memory used and after the memory used.

```
1: Mem_average ← M_recording.sum()/M_recording.length()
2: S_task ← True
3: while S_task is True do
4:     for i=0 to BLOCK_SIZE do
5:         if Mem_cur_average − Mem_old_average ≥ DIFFERENCE then
6:             Set Hit the data pattern
7:         else
8:             Set Miss the data pattern
9:         end if
10:    end for
11:    if No data processing then
12:        S_task ← False
13:    else
14:        Recording the MemFree value from /proc/meminfo file
15:    end if
16: end while
```

Unlike the *TASK_WRITE* API interface, the *TASK_LOAD* API interface runs only on the adversary side serverless function. The *TASK_LOAD* API allows the adversary to monitor memory changes within the adversary's container by accessing the */proc/meminfo* file. Algorithm 1 provides the details of the SRE attack using the */proc/meminfo* file.

The value of DIFFERENCE is a constant that acts as a threshold for distinguishing different patterns. The attacker can conduct experiments by executing the serverless function with different input data to obtain the appropriate value for DIFFERENCE. As shown in Algorithm 1, we can detect the memory changes through the subtraction of current $M_{average}$ and the last state of $M_{average}$. If the memory changes reach the constant value DIFFERENCE, the program will set a hit for the visited block, rather than a miss.

2) The Timing Entanglement Patterns: Memory load/write time delay is another crucial metric in the SRE threat model. In order to enhance the performance of serverless computing, memory deduplication [25] techniques are commonly applied during data processing. It is due to the presence of memory deduplication that load/write time delays occur when co-resident serverless functions execute identical data, leading to mutual interference. The memory load/write timing delay measurement method is primarily used to detect time delays during data load and write operations. If co-resident serverless functions share physical memory, it can result in variations in the time delays between them. In this paper, we write a C program that retrieves the hardware time using the $rdtsc$ instruction. We record the timing patterns as $Pattern_{\Delta t} = T_{completing} - T_{starting} \pm T_{\Delta Noise}$. For a serverless service, we determine the data load/write time by calculating the difference between the data input time ($T_{starting}$) and process completion time. Additionally, we calculate the program boot time and the running start time to analyze program loading patterns and deduce the noise patterns.

IV. Evaluation Setup

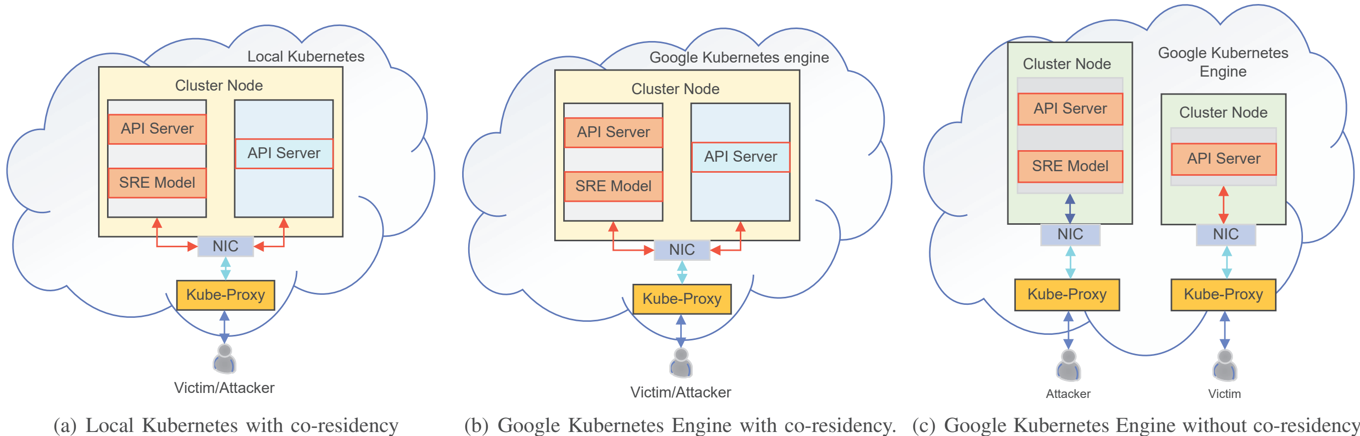
The experimental setup involves two Flask-based serverless functions written in Python, alongside a C-based time measurement program. Users can execute these functions via HTTP/HTTPS links. The first function, "API Server," handles extensive word load and write processes, while the second focuses on image processing. These distinct functions were chosen for their varied memory utilization and operations. The simplicity of the programs ensures no risk of user data leakage or interference with other serverless functions.

TABLE I
The testbeds architecture configuration

Name	Local Kubernetes		Google Kubernetes Engine
	Master Node	Worker Nodes	
OS	Ubuntu 22.04.1 LTS Kernel 5.19.0-42-generic	Ubuntu 22.04.1 LTS Kernel 5.19.0-42-generic	Linux 5.15.107+
CPU	Intel Core i5-6500	Intel Core i7-10700K	Intel Xeon
RAM	8G	32G	12G
Kubernetes	v1.22.0	v1.22.0	v1.23.5-gke.1200

To assess architectural sensitivity, we conducted the attack on a quiescent setup where only the adversary and the victim were active in a local Kubernetes environment. This setup allowed us to demonstrate the feasibility of the attack while ensuring that the measurements were unaffected by external interference. Subsequently, we deploy the attacker on the Google Kubernetes Engine platform to analyze the impact of the victim's serverless functions on the attack's success. Similar to real-world cloud environments, we examined the effect of co-resident load on the system's performance and its influence on the attack's efficacy. In addition to the victim and adversary, we assessed the attack's success in scenarios where the victim's serverless functions exerted high demands on memory and network bandwidth.

As shown in Fig. 5, we evaluate our approach in several different testbeds. The first was a local Kubernetes that contained a master node and a worker node in different workstations. Table I presents the configuration details of the environment variables used in this experiment. For the proposed SRE attack model, we conducted tests using both a local Kubernetes environment and a real-world Google Kubernetes Engine platform. In the offline environment, a Kubernetes platform consisting of a master node and a worker node was set up. For the online



(a) Local Kubernetes with co-residency (b) Google Kubernetes Engine with co-residency. (c) Google Kubernetes Engine without co-residency.

Fig. 5. Testbed toplogies in evaluation.

testing, the Kubernetes environment was created directly via Google Cloud to evaluate the proposed SRE attack model.

Local Kubernetes: The local testing environment for Kubernetes consists of two hosts, with one serving as the master node and the other configured as the worker node. Within the worker node, we have established separate pods for the victim's API server and the attacker's API server. Additionally, we have deployed the measurement program and diagnostic program of the attacker within the worker node. The victim and the attacker access their respective serverless function interfaces, running on the worker node, through kube-proxy.

To validate the feasibility of the SRE threat model, we performed experimental tests in the local Kubernetes testing environment under three conditions: no interference from other serverless functions, interference only with the victim's serverless function, and a noisy environment with multiple concurrent serverless functions. In the interference-free experiment, we solely executed the attacker's serverless functions while monitoring changes in the free memory value and memory data load/write time delay. Subsequently, we executed both the victim's and the attacker's serverless functions on the local worker node to investigate the information regarding shared resource entanglement. In the noisy environment experiment, apart from running multiple serverless functions identical to the victim's function, we also deployed multiple serverless functions of different types. This created a scenario with significant noise and interference from various concurrent serverless functions. These two sets of experiments aimed to evaluate the SRE threat model in both controlled and realistic environments, providing insights into the performance, resource utilization, and interference effects of the victim function under different conditions.

For the configuration of the worker node in the local Kubernetes environment, we utilized the default resource allocation strategy. In the configuration files for all serverless functions, except for service loading and port settings, we rely on the system's default resource allocation and scheduling scheme.

Google Kubernetes Engine: To further validate the feasibility of the proposed SRE threat model, we conducted verification through the Google Kubernetes Engine environment. The setup in the Google Kubernetes Engine environment was similar to the local Kubernetes environment which relies on the system's default resource allocation and scheduling.

In the Google Kubernetes Engine platform, we deploy our cluster node in the cloud server which is located in the region of **us-central1-c** For ease of serverless function management and operational convenience of the experimental platform, we opted to create a standard cluster.

In the Google Kubernetes Engine testing environment, we conducted experiments not only on co-resident serverless functions but also on separate cluster nodes for the victim and attacker's serverless functions. By running the functions on different cluster nodes, we aimed to compare the performance of the SRE attacks in terms of the accuracy of data inference. Through this validation process in the Google Kubernetes Engine environment, we aimed to provide more comprehensive evidence of the SRE threat model's viability and effectiveness.

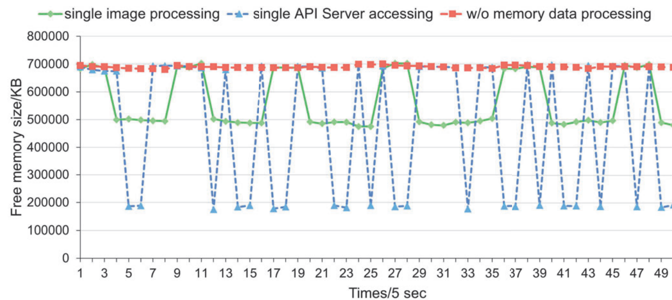
V. Evaluation Result

In this section, we will assess the effectiveness of our proposed SRE threat model in various scenarios outlined in Section IV.

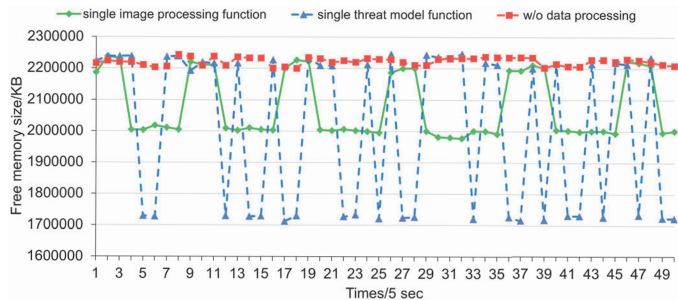
A. The analysis of memory utilization entanglement patterns

Fig. 6 presents the test results for different operating environments in both the local Kubernetes and online Google Kubernetes Engine environments. In the experiment, we recorded the *MemFree* value from the */proc/meminfo* file in the container every 5 seconds and observed its variations based on different experimental scenarios. Fig. 6(a) illustrates the statistical results of the *MemFree* value in the local Kubernetes environment. In the local Kubernetes experiments, we observed the following scenarios: the free memory status without any memory data processing, the free memory status when a single API server runs and performs data processing, and the free memory status during the execution of a single serverless function based on image processing.

In Fig. 6(a), the red dashed line represents the statistical results of free memory in the container when the serverless function is not performing any data processing. From the



(a) Local Kubernetes.



(b) Commercial Google Kubernetes Engine.

Fig. 6. Memory utilization entanglement evaluation in various scenarios.

results, it can be observed that the free memory size remains stable when no memory usage is involved. The blue dashed line and the solid orange line represent the statistical results of free memory in the API server and the image denoising serverless function when data processing is performed. In the image-denoising serverless function, we let the serverless function process around 200MB images parallel. Comparing the blue dashed line and the solid orange line with the red dashed line, it is evident that when data is written into memory, the MemFree value experiences a significant change. By comparing the variations in MemFree value during data processing between image processing and the API server, it is noticeable that different applications and data exhibit specific patterns of change. For example, in our experiments, the image processing tasks mainly involved continuous data, and the memory usage patterns for the same image data were generally consistent.

Fig. 6(b) displays the experimental results obtained from the Google commercial platform, Google Kubernetes Engine. Similar to the local Kubernetes testing environment, the online Google Kubernetes Engine platform was used with the same inputs to verify the free memory status without memory data processing, the free memory status when a single API server runs and performs data processing, and the free memory status during the execution of a single serverless function based on image processing. By comparing the variations in free memory without memory data processing between the local Kubernetes environment and the online Google Kubernetes Engine platform, it is evident that the online Google Kubernetes Engine platform is less stable. This is because the cluster nodes in the online Google Kubernetes Engine platform serve numerous users and accommodate a variety of serverless functions, introducing additional noise and causing more noticeable fluctuations in free memory within the platform's containers. However, by comparing the variations in free memory during the execution of serverless functions between the local Kubernetes environment and the online Google Kubernetes Engine platform, it can be observed that the noise generated by other users' serverless functions does not significantly impact the results obtained from the proposed SRE threat model. This demonstrates the feasibility and reliability of the SRE threat model, despite the presence of noise from other user functions in the online Google Kubernetes Engine platform.

To demonstrate that the SRE threat model remains effective even in environments with significant amounts of noise, we conducted tests by designing complex noisy environments. During the testing process, we ensured that both the local Kubernetes and online Google Kubernetes Engine platforms had the same client-side access to the API server, thereby ensuring consistent noise characteristics. Fig. 7 illustrates the data results obtained from running multiple API servers accessing the local Kubernetes and online Google Kubernetes Engine platforms. The blue line represents the experimental results from the local Kubernetes platform. From the data, it can be observed that when there are numerous serverless functions executing data processing tasks in the runtime environment, it becomes challenging to analyze the data characteristics of the victim's execution solely based on the values of free memory. However, through analyzing the data results from the online Google Kubernetes Engine platform, it can be observed that the proposed SRE threat model remains effective. In the data graph of the online Google Kubernetes Engine platform, green markers represent normal data, red markers represent data points where memory data anomalies occur despite no memory usage, and purple markers represent anomalies caused by data being written into memory but affected by noise. By comparing the data from the online Google Kubernetes Engine platform with that of the local Kubernetes platform, it is evident that the noise in the online Google Kubernetes Engine platform has minimal impact on the SRE threat model. In light of these data differences, through extensive observation, we found that in the online Google Kubernetes Engine environment, when the cluster node resources become overloaded or when numerous pods simultaneously consume a significant amount of resources, the Kubernetes controller crashes the pods within the current cluster node and migrates them to other cluster

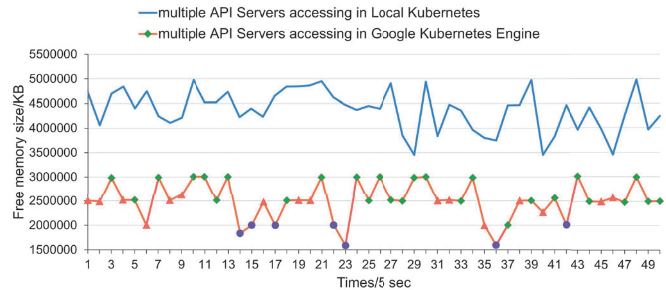
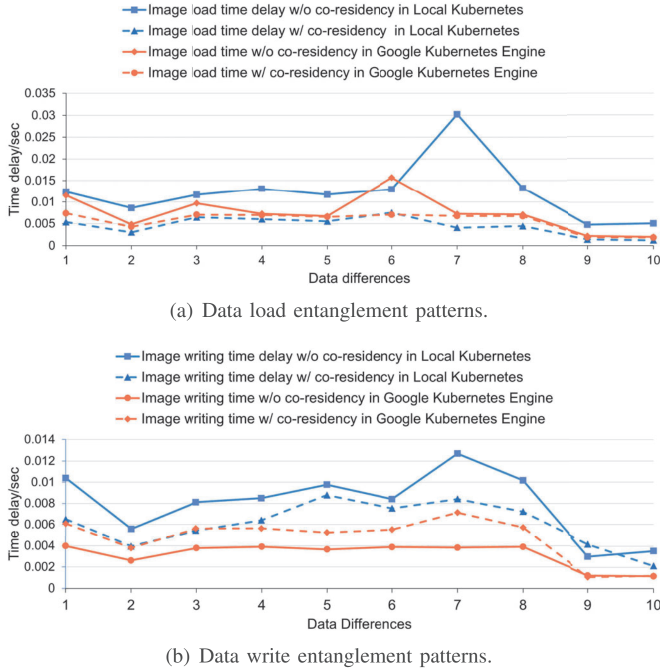


Fig. 7. Noisy testbed by multiple API servers accessing in Local Kubernetes and Google Kubernetes Engine.



(a) Data load entanglement patterns.

(b) Data write entanglement patterns.

Fig. 8. The time entanglement for data read and write in different data input.

nodes. Undoubtedly, the load balancing strategy of Kubernetes enhances the effectiveness of our SRE threat model during attacks, as some noisy serverless functions are migrated to other cluster nodes.

The experimental results demonstrate that analyzing the statistical behavior of free memory can reveal whether the user and attacker are utilizing the same serverless function and processing the same data. Different serverless functions and input data exhibit distinct characteristics that result in specific ranges, including the intervals and durations of free memory variations. Consequently, the attacker can infer the data information being processed by the victim through analysis and estimation.

B. The analysis of the timing entanglement patterns

Section V-A demonstrates that variations in free memory can lead to the leakage of users' sensitive data. However, one limitation is that the accuracy decreases significantly when the input data occupies a small amount of memory. To address this, we introduce the timing delay method to compensate for the shortcomings of analyzing free memory content. This chapter presents experiments to demonstrate the potential risks of user's sensitive data leakage caused by timing delays.

Fig. 8 illustrates the time delay information for reading and storing images during the execution of a serverless function for image processing. In the experiment, ten images of different sizes and contents were selected to observe whether different types of data inputs would render the SRE threat model ineffective. Fig. 8(a) presents the time overhead for image loading on both the local Kubernetes platform and the online Google Kubernetes Engine. By comparing the time delay of processing individual images with and without co-resident serverless functions performing image processing, it can be observed that when the same serverless function processes the same data on the same cluster node, the data loading time is greater compared to scenarios without co-resident functions and different data inputs. In other words, if both the adversary and victim employ the same serverless function for processing identical data inputs, the data loading time would be smaller than in situations without co-resident serverless functions or with different data inputs.

Fig. 8(b) displays the time delay for data writing during the image storage process of the image processing serverless function. In the experiment, we conducted tests with and without co-resident serverless functions on both local Kubernetes and online Google Kubernetes Engine platforms. The results revealed that when there is a co-resident serverless function involved in the data storage process, the time delay is slightly longer, and this phenomenon is more pronounced in the Google Kubernetes Engine environment.

Upon analysis, it was found that the repetition of the victim's serverless function processing the same data every 10 seconds in the co-resident scenario leads to the function's increased utilization of the system's output resources, resulting in a prolonged time delay for data storage. However, from the perspective of our SRE threat model, this situation is advantageous for predicting the victim's input data. Overall, the presence of a co-resident serverless function during data storage introduces a slight delay, particularly in the Google Kubernetes Engine environment. This delay can be leveraged by our SRE threat model to better analyze and predict the victim's input data.

VI. Defense Discussion

A. The memory utilization entanglement

To prevent memory utilization entanglement from exposing kernel and physical device parameters, thereby enabling containers and other applications to query a wide range of data, the most efficient method is to implement complete container isolation in serverless platforms. However, complete container isolation significantly impacts serverless platform performance. Considering performance concerns in serverless platforms, an effective solution is to prevent containers from directly accessing the $/proc$ filesystem, thereby mitigating the risk of memory utilization entanglement. If a serverless function requires memory information from the $/proc/meminfo$ file, the user should declare this permission in the YAML file. Additionally, containers use a layered file system, so mounting frequently used pseudo filesystems as read-only, where possible, limits attackers' ability to access sensitive files. This ensures that write operations within the container do not impact the host machine or other containers.

B. The timing entanglement

To defend against the timing delay entanglement in serverless platforms, several strategies can be implemented to obscure timing behavior and make it harder for attackers to exploit. One approach is to vary the execution path of the code.

This can be achieved by randomizing the order of operations or inserting dummy operations, thereby making the execution time less predictable. Additionally, the host machine can leverage hardware features to block precise timing measurements. For example, enabling CPU features that ensure constant-time execution for certain instructions can be effective. This prevents attackers from using instructions like $rdtsc$ and $rdtscp$ to directly obtain timing information from physical hardware. These combined techniques can significantly enhance the security of serverless platforms against timing attacks.

VII. Conclusion

We introduce a potential data leakage threat model in serverless platforms by detecting the shared resource interference between the co-resident serverless functions. This threat model includes two parts: a measurement program and a diagnostic program. The measurement program is utilized to intercept and monitor the system changes induced by the execution of co-resident serverless functions, specifically caused by the preemption of shared resources. On the other hand, the diagnostic program aims to infer the victim's input data by analyzing the characteristic information obtained from the shared resource entanglement. This inference can potentially result in the leakage of the victim's sensitive data. The experimental results confirm that co-resident serverless functions experience mutual interference in the utilization of shared resources. For specific data sets, adversaries can infer the victim's input data by observing the resource changes during the processing of different data, potentially leading to the leakage of sensitive data.

Acknowledgment

This work at University of South Florida(USF) was supported in part by the National Science Foundation (NSF) under grants 2315353 and 2404741. We thank USF for providing the necessary resources and support throughout this research.

References

[1] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, 2019.

[2] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A review of serverless use cases and their characteristics," *arXiv preprint arXiv:2008.11110*, 2020.

[3] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE software*, vol. 38, no. 1, pp. 25–31, 2020.

[4] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–34, 2022.

[5] J. Lindemann and M. Fischer, "A memory-deduplication side-channel attack to detect applications in co-resident virtual machines," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 183–192.

[6] S. Shringarputale, P. McDaniel, K. Butler, and T. La Porta, "Co-residency attacks on containers are real," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2020, pp. 53–66.

[7] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 990–1003.

[8] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 2016, pp. 279–299.

[9] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[10] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "Containerleaks: Emerging security threats of information leakages in container clouds," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 237–248.

[11] T. Allclair and M. Kaczorowski, "Exploring container security: Isolation at different layers of the kubernetes stack." Google Cloud Platform Blog, 2018. [Online]. Available: https://cloud.google.com/blog/products/gcp/exploring-container-security-isolation-at-different-layers-of-the-kubernetes-stack

[12] "Google kubernetes engine (gke)." [Online]. Available: https://cloud.google.com/kubernetes-engine

[13] F.-X. Standaert, "Introduction to side-channel attacks," *Secure integrated circuits and systems*, pp. 27–42, 2010.

[14] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel, "Malicious co-residency on the cloud: Attacks and defense," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[15] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24, New York, NY, USA, 2024, p. 582–600. [Online]. Available: https://doi.org/10.1145/3620665.3640403

[16] 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 *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2627–2641.

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

[18] Z. N. Zhao, A. Morrison, C. W. Fletcher, and Torrellas, "Everywhere all at once: Co-location attacks on public cloud faas," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24, New York, NY, USA, 2024, p. 133–149. [Online]. Available: https://doi.org/10.1145/3617232.3624867

[19] A. Yelam, S. Subbareddy, K. Ganesan, S. Savage, and A. Mirian, "Coresident evil: Covert communication in the cloud with lambdas," in *Proceedings of the Web Conference 2021*, 2021, pp. 1005–1016.

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

[21] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler, "Detecting co-residency with active traffic analysis techniques," in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, 2012, pp. 1–12.

[22] J. Zhang, C. Shen, and G. Qu, "Mex+ sync: Software covert channels exploiting mutual exclusion and synchronization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[23] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli, "Kernel-level tracing for detecting stegomalware and covert channels in linux environments," *Computer Networks*, vol. 191, p. 108010, 2021.

[24] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "A study on the security implications of information leakages in container clouds," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 174–191, 2018.

[25] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella, "Memory deduplication for serverless computing with medes," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 714–729.