FaaSTracker: Efficient Cross-Layer Provenance Tracking of Serverless Applications With Multi-Source Correlation

Qingyang Zeng, Lianjie Wu, Kaiyu Hou, Xue Leng, *Member, IEEE*, and Yan Chen, *Fellow, IEEE*

***Abstract*—Serverless computing, also known as Function-as-a-Service (FaaS), has gained popularity due to its flexibility, scalability, and transparent development. However, attacks against serverless are also increasing. Unfortunately, complex multi-layer FaaS architecture and frequently launched lightweight functions help attackers conceal their tracks. Specifically, 1) fully tracking the behavior of a function requires crossing multiple layers of FaaS. 2) Intrusive auditing components in functions affect function startup latency and performance. 3) Accurately provenance cross-layer function invocations require integrating data from multiple sources. In this paper, we propose FAASTRACKER, a cross-layer, non-intrusive, efficient provenance framework for accurately tracking user function behaviors in FaaS. FAASTRACKER tracks function behaviors across layers using a non-intrusive agent without any modifications to the function. In addition, it correlates data from multiple sources to construct a provenance graph of function workflows to locate attackers. We implement FAASTRACKER on the OpenFaaS platform and evaluate its performance using real-world serverless applications. Compared with state-of-the-art serverless provenance systems, FAASTRACKER provides a more accurate and complete view of provenance graphs and reduces 54.0% CPU and 48.9% memory resources.**

***Index Terms*—Serverless computing, function-as-a-service, provenance tracking, cross-layer.**

I. INTRODUCTION

AS AN emerging cloud programming paradigm, Serverless Computing, also known as function-as-a-service (FaaS), is widely used in modern web applications. In this paradigm, users only need to upload their code onto serverless platforms as functions, leaving everything else to the FaaS providers [1]. AWS [2], Azure [3], and Google [4] such providers take full responsibility for managing the infrastructure maintenance and resource provisioning [5]. A high degree of reliance on FaaS provider management brings convenience to users. However, it also entails delegating security to the serverless platform provider. Attackers continuously innovate to find new ways to penetrate serverless platforms. For example, a Denial-of-Wallet (DoW) attack [6] to consume normal users' resources. The behaviors of malicious users' functions will also affect normal functions [7], [8].

Provenance-based intrusion detection systems (PIDS) have gained attention from both the security industry and academia for their causality analysis capability [9]. PIDS effectively identifies the propagation paths of anomalous behaviors by constructing a causal dependency graph among system entities, such as processes, files, and network connections. Existing PIDS work [10], [11], [12], [13], [14] uses the host-based provenance graph to distinguish malicious behaviors. However, in serverless, there is a lack of complete and accurate provenance graphs to analyze the attacker's behavior. To build accurate provenance graphs on serverless, complete and correct data about the attacker's function invocations behavior is required. Thus, complete, efficient, and accurate provenance function behavior is crucial for FaaS providers to ensure the security of applications running on the platform.

Unfortunately, the characteristics of FaaS bring new problems to provenance. **First, multi-layer FaaS architecture leads to provenance across layers to ensure the graph is complete (P1).** FaaS architecture decouples into four layers: Operating System, Virtualization, Orchestration, and Coordination [15]. As shown in Figure 1, users invoke functions through the API gateway (①), and the invocations span four layers of the architecture. For each layer, FaaS providers use different technology stacks, increasing the platform's heterogeneity and the difficulty of tracing. **Second, the frequently launched lightweight function requires that provenance cannot affect function performance (P2).** As Figure 1 shows, functions first import the code in the handler and install the necessary dependencies in the runtime environment before executing. Due to the overhead of starting a new virtual execution environment, function execution thus incurs a cold start time [16]. Adding additional auditing components inside functions will increase the startup overhead, affecting function performance. **Third, attack function invocations occur across different layers of the FaaS architecture, demanding**

Received 10 April 2025; revised 2 October 2025 and 14 November 2025; accepted 15 November 2025. Date of publication 19 November 2025; date of current version 1 December 2025. This work was supported by the National Science Foundation under Grant CNS 2229454. The associate editor coordinating the review of this article and approving it for publication was Prof. Haibo Hu. *(Corresponding author: Xue Leng.)*

Qingyang Zeng and Lianjie Wu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: qyzeng@zju.edu.cn; 12221128@zju.edu.cn).

Kaiyu Hou is with Alibaba Cloud Computing, Hangzhou 100004, China (e-mail: kyhou@alibabacloud.com).

Xue Leng is with Hangzhou Institute of Technology, Xidian University, Hangzhou 311215, China (e-mail: lengxue@xidian.edu.cn).

Yan Chen is with the Department of Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: ychen@northwestern.edu).

Digital Object Identifier 10.1109/TIFS.2025.3634978

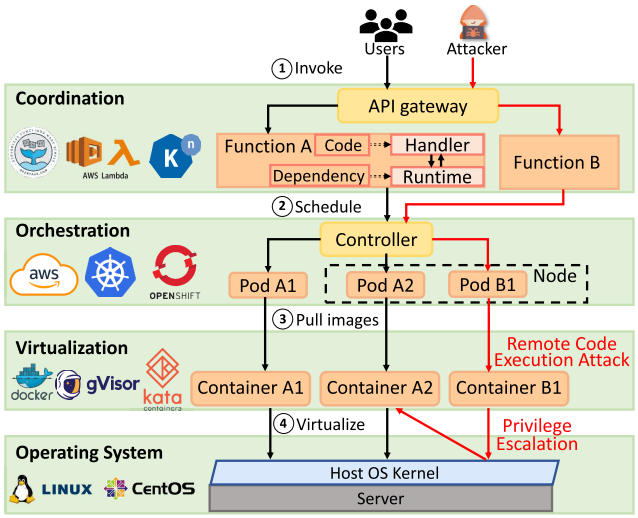


Fig. 1. Four-layered FaaS architecture. users interact only with the functions at the coordination layer. FaaS providers need to manage the four-layer FaaS architecture.

the integration of data from multiple layers for provenance (P3). Although the ephemeral, stateless nature of functions reduces the risk of long-term attacks [17], attackers can still exploit each layer of FaaS architecture to invoke functions to conduct attacks, e.g., DoW attack [6] at the Coordination layer, co-locate attack [18], [19] at the Orchestration layer. As Figure 1 shows, the attacker conducts a remote code execution attack [20] at the Virtualization layer and gets privilege escalation at the Operating System layer to control other users' functions. Integrating traces from a single layer or source cannot accurately provenance attacks. **Therefore, tracking in FaaS requires solving the problems of crossing the FaaS layer, achieving high performance without affecting function, and correlating data from multiple sources.**

However, existing work does not meet the above requirements. FaaS monitoring tools [21], [22], [23], [24] focus on function invocation latency, response status, and so on at the Coordination layer. Such tools provide a view of application performance but do not consider other layer data, e.g., low-level system events generated by functions. However, these low-level system events are useful for provenance attacks such as container escape [25]. Recent serverless provenance studies [26], [27] collect data inside functions and combine these logs to generate provenance graphs. Such intrusive provenance frameworks burden the startup overhead, increase function resource consumption, and generate inaccurate provenance graphs. Traditional provenance-based auditing works [28], [29], [30], [31], [32] leverage auditing tool e.g., ETW [33], Linux Audit [34] for investigating attacks. They only conduct provenance on a single machine not designed for distributed systems and can not locate user identity based on the function invocations.

This paper proposes FAASTRACKER, a cross-layer, non-intrusive, high-performance provenance framework for attack root cause analysis in FaaS with the following designs: **(i) FAASTRACKER designs a cross-layer and non-intrusive agent, FTracker Agent.** This agent attaches pre-defined kernel hooks to the different layers to perform cross-layer provenance. FTracker Agent leverages the privileged kernel space without intrusive functions to track behaviors universally from different layers. This non-intrusive agent avoids increasing cold start delay and resource consumption within functions. **(ii) FAASTRACKER uses *key-based map correlation* to connect cross-layer data.** FAASTRACKER designs an FTracker Server to receive cross-layer data from multiple sources. With the assistance of the key, the FTracker Server associates maps to construct cross-layer provenance graphs for accurate function invocations analysis. The function provenance graph is the foundation for PIDS work on the serverless, which provides data for research. In summary, our paper makes the following contributions:

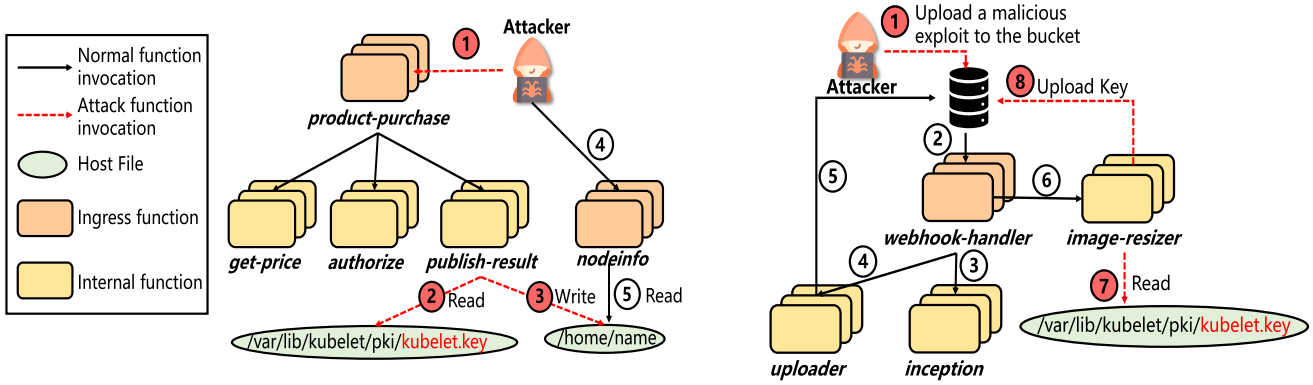
- We perform cross-layer attack invocations on the serverless application by exploiting vulnerabilities. However, existing work cannot accurately, efficiently, and completely provenance function behaviors in FaaS. (§ II)
- We design FAASTRACKER, a cross-layer, non-intrusive, high-performance provenance framework for function invocation analysis in FaaS. It offers an FTracker Agent to track function behaviors across layers efficiently and an FTracker Server to connect data from multiple sources to generate the provenance graph based on function invocations. (§ IV)
- We implement FAASTRACKER on the serverless platform OpenFaaS without any modifications to user functions. The platform does not need to modify the function runtime environment. (§ V)
- We evaluate FAASTRACKER both in functionality and runtime performance. Compared with state-of-the-art provenance work ALASTOR [26] and CLARION [35], FAASTRACKER generates more accurate and complete provenance graphs that show the cross-layer attack invocation. Compared with the FaaS provenance system ALASTOR, it also reduces 54.0% of node CPU and 48.9% of node memory resource consumption. (§ VI)

II. BACKGROUND AND MOTIVATION

This section begins with cross-layer attack invocations that occur in FaaS (§ II-A). Then, we summarize the limitations of existing work in FaaS provenance (§ II-B).

A. Attack Invocations in FaaS

While the expertise of FaaS providers managing the underlying infrastructure enhances security, attacks still occur on the platform [6], [18], [19]. Adversaries could conduct attacks through function invocations by exploiting vulnerabilities at each layer of the FaaS architecture. Serverless functions are invoked either synchronously or asynchronously. The synchronous invocation calls wait for the function to complete execution, and its result is immediately returned to the user [36]. Asynchronous invocations will be placed on an internal queue. The result will be returned to the user once the function has finished the asynchronous request [37]. Attacks occur in both synchronous and asynchronous serverless applications through function invocations.



(a) Cross-layer attacks based on application *Hello Retail!*

(b) Cross-layer attacks based on application *Image Pipeline*

Fig. 2. Cross layers attack examples. Red arrows denote the attack path. The attacker invokes functions to escape from the container and get the sensitive information.

1) Synchronous Invocation Attacks: Adversaries could conduct attacks through synchronous function invocations by exploiting vulnerabilities at each layer of the FaaS architecture. For example, the Linux kernel in the Operating System layer has the vulnerability CVE-2022-0847 [38] that could write read-only files. Unprivileged adversaries could use this flaw through function invocations to achieve container escape in the Virtualization layer. They could also leak confidential information about the Orchestration layer. We conduct this cross-layer attack on a serverless application *Hello, Retail!* [39]. It consists of 4 functions: *product-purchase*, *get-price*, *authorize*, and *publish-result* to purchase products. Users invoke the ingress function *product-purchase* to purchase products. The function *nodeinfo* is a function that obtains information (e.g., hostname, CPU, and Memory) about the node being scheduled. As Figure 2(a) shows, the internal function *publish-result* and the function *nodeinfo* are orchestrated to the same worker node that has the vulnerability CVE-2022-0847. The attacker invokes function *publish-result* through the ingress function (①). This function invocation would exploits the function *publish-result* to read the host file *kubelet.key* (②) and write to the read-only file */host/name* (③). Then the attacker invokes function *nodeinfo* as a normal user (④), but the attacker can get the *kubelet.key* of Kubernetes located in the Orchestration layer (⑤). At this point, the attacker has conducted a cross-layer attack from the virtualization layer to the Operating layer through synchronous function invocations and pretends to be a normal user to obtain confidential information.

2) Asynchronous Invocation Attacks: Adversaries performing asynchronous invocation attacks need to wait until the queue asynchronous function has been executed. We use an asynchronous application *Image Pipeline* [40] to conduct asynchronous invocation attacks. It consists of 4 functions: *webhook-handler*, *image-resizer*, *inception*, and *uploader*, of which function *image-resizer*, *inception* are asynchronous functions. By exploiting CVE-2021- 22555 [41], a heap overflow vulnerability in the Linux kernel, adversaries get the sensitive information. Figure 12(b) shows the cross-layer attack. The functions *inception* and *image-resizer* are asynchronous functions. The attacker uploads a compiled exploit file to the database (①). Event triggering has been added to this database. When a user uploads an image, function *webhook-handler* is invoked to process the image (②). It invokes function *inception* asynchronously to classify the image (③) and uploads the classification result back to the database (④, ⑤). Next, the function *webhook-handler* invokes function *image-resizer* asynchronously to resize the image (⑥). *image-resizer* downloads the file just uploaded by the attacker from the database and executes it. After obtaining the host file located at the Operating System layer, it uploads the file content back to the database (⑦, ⑧). After waiting for the asynchronous invocation to complete, the attacker obtains confidential information in the database.

These two types of attacks cross four layers in the serverless architecture. They leverage the Operating System layer vulnerability, invoke functions through the Coordination layer, escape from the container at the Virtualization layer, and leak the Orchestration layer sensitive information. Attacker function invocations pretend to be normal user function invocations. Identifying attack invocations and locating the attacker's identity is crucial to the platform's security.

B. Limitations in Existing Work

Unfortunately, existing work cannot efficiently, accurately, and completely provenance FaaS cross-layer attack invocations. We divide them into three categories.

Distributed monitoring contributes to monitoring and troubleshooting performance. eBPF-based monitoring work [42], [43], [44], [45] focuses on one-layer metrics, such as network metrics, which cannot be used directly for provenance attacks that require cross-layer tracing. Troubleshooting work [46], [47], [48] focuses on diagnosing performance to identify errors. DeepFlow [46] also uses eBPF technology to trace network flow for troubleshooting. However, the resulting error metrics, which do not include system events, are not suitable for root-cause analysis. FaaS monitoring tools [21], [22], [23], [24]only monitor function performance metrics at the Coordination layer, e.g., latency, response status, and errors. They do not collect low-level system events and correlate these events from multiple layers, which are important for provenance attacks in FaaS.

Serverless provenance traces suspicious events in serverless applications. ALASTOR [26] audit logs inside each function instance to build a global provenance graph of serverless applications. However, the intrusive tracing affects function performance, and incomplete cross-layer tracking leads to inaccurate provenance graphs. DisProTrack [27] combines the control flow graph (CFG) from the serverless application binaries, application logs, and system logs to build the provenance graph. However, it separately collects data from each container and host, which consumes a lot of resources. In addition, the graph does not correctly reflect the location of behaviors.

Traditional provenance focuses on reconstructing dependence graphs for Advanced and Persistent Threats (APTs). Data provenance techniques [13], [35], [49], [50], [51], [52] can be used in root cause analysis by tracking the graph backward, starting from a suspicious event. However, they only collect host events on a single machine and do not combine FaaS cross-layer traces from multiple sources, which cannot track function invocations across hosts.

To summarize, existing work has specific application track scenarios. Still, it cannot simultaneously provide a cross-layer, high-performance, multi-source correlation provenance framework for root cause analysis of attacks.

III. Threat Model

In this paper, we consider serverless applications running as functions in a public serverless platform. We assume that the cloud provider and FaaS platforms, e.g., Amazon Lambda are trusted, meaning that the provider will correctly deploy functions and will not attempt to collude with attackers. However, since each layer of FaaS architecture uses different technologies, each technology has vulnerabilities that can be exploited. We assume these vulnerabilities have not been patched. Attackers could exploit these vulnerabilities via function invocations.

Within this environment, the attackers have access to deploy their functions on the platform but have no access to other users' functions. A remote attacker can only access the platform through the function invocations. The goal of attackers is to exploit vulnerabilities in every layer to steal confidential information, and even control the whole cluster.

IV. System Design

In this section, we first propose the design goals of FaaSTracker and its overall architecture. Then, we elaborate on the two design aspects of FaaSTracker, namely, FTracker Agent and FTracker Server.

A. FaaSTracker *Overview*

The limitations of existing FaaS tools discussed in § II-B lead to the following high-level design goals for FaaSTracker.

Goal 1: Cross-Layer Tracking Approach. Function invocations across four layers of FaaS architecture. Events (e.g., invocation relationship, location of function, system call events) generated at each layer are crucial to provenance attack invocations completely. However, FaaS adopts different technology stacks at each layer. For example, open-source platforms [53], [54] apply Docker [55] at the Virtualization layer and Kubernetes [56] at the Orchestration layer. Designing tracking approaches for each technology is redundant. FaaSTracker should develop a universal cross-layer tracking method for each technology.

Goal 2: Non-Intrusive and High Performance. The intrusive tracking method installs the audit component in the container runtime. This increases the cold start latency when the function is initialized and drags down the service response when the function responds to the invocation. Thus, FaaSTracker should trace functions in a non-intrusive, efficient way without increasing latency and additional function resource consumption.

Goal 3: Multi-Source Correlation. Collected traces from different layers contain different information about functions. For example, the invocation relationship at the Coordination layer, the location of functions at the Orchestration layer, and the system events in containers at the Virtualization layer.FaaSTracker should associate four-layer traces in units of functions to perform attack invocation analysis.

Considering these design goals, we present the overall architecture of our framework in Figure 3. FaaSTracker consists of two high-level components: **FTracker Agent** and **FTracker Server**. The FTracker Agent is deployed in each worker node to track function behaviors across layers using Extended Berkeley Packet Filter (eBPF) hooks. In addition, the FTracker Agent is responsible for transmitting cross-layer data to the FTracker Server with high performance. The FTracker Server is responsible for correlating data from multiple sources and assembling them into detailed provenance graphs for attacks.

Workflow. Figure 3 describes the workflow of TraceFaaS. When users invoke functions through the API gateway, the workflow is set in motion (❶). Subsequently, the FTracker Agent tracks the function invocation from the Coordination, Orchestration, Virtualization, and Operating System layers (❷). The FTracker Server receives track data from distributed FTracker Agents from worker nodes (❸) and correlates them based on the key. Finally, the FTracker Server generates the cross-layer provenance graph of functions (❹).

B. FTracker Agent

FaaSTracker designs the FTracker Agent to facilitate cross-layer, non-intrusive, and high-performance tracking function behaviors in FaaS.**To achieve Goal 1, the FTracker Agent attaches eBPF hooks at different layers to track function behaviors in a unified way.**eBPF is an instruction set and an execution environment inside the Linux kernel [57]. It allows the safe execution of customized yet constrained functions inside the kernel at various locations [58], [59]. The FTracker Agent attaches kernel hooks at different layers regardless of the technology used. It tracks function behaviors from different layers through the eBPF map. The track data is stored in eBPF maps as keys and values. Different FaaS providers use different technologies (e.g., Azure and Google

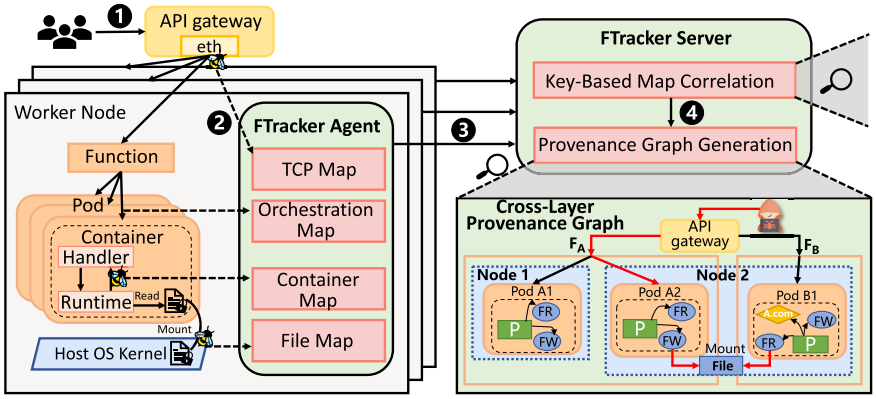


Fig. 3. Architecture overview of FAASTRACKER.

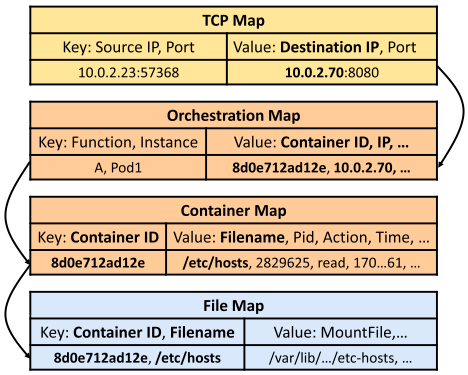


Fig. 4. Key-based maps correlation.

Functions use different API gateways), but as long as their operating system supports eBPF, they can use FTracker Agent to track function invocations. The modification to the FTracker Agent is the position of the hook in each layer.

As Figure 3 describes, the attached eBPF hooks collect metrics based on the characteristics of function calls and store them in different maps. The metrics stored in maps are shown in Figure 4. First, the hooks record the function calls through the API gateway in the TCP Map. The TCP Map records the TCP connection when users invoke the function at the Coordination layer. The IP and ports will be stored in the TCP map. Second, The invocation will be scheduled to an instance of the function by the controller at the Orchestration layer. The location information of the function (e.g., function instance IP, instance Container ID, function location worker node, and so on) is stored in the Orchestration map. Third, the hooks involve a series of events generated by the invoked function instance in the Container Map. These events occur in containers located at the virtualization layer. However, some files located in the container are mounted from the host. The operations on files in the container actually involve the host at the operating system layer. Fourth, the hooks record the real path of the file in the function instance in the File Map. FTracker Agent achieves cross-layer tracing by attaching hooks at different layers and storing cross-layer data at different maps.

To achieve Goal 2, the FTracker Agent adopts host-based deployment and optimizes the data collection process. The FTracker Agent is deployed on each worker node as a container without deploying extra components inside the function instance. This method reduces the cold start latency of functions. Since the FTracker Agent needs to attach hooks in the host kernel, it requires privileges to deploy. It is designed for the platform, so only the platform provider has the privilege to deploy it. Normal users cannot deploy the FTracker Agent privately, they can only get the traces from the FTracker Agent provided by the platform. FaaS providers do not need to make any modifications to user functions. The advantage of this non-intrusive framework is that it would not affect the performance of the function. However, since the FTracker Agent traces all functions on the worker node, it will consume node resources. We perform trace collection optimization to lower the tracking overhead.

Algorithm 1 Collected Traces Optimization

Input: C - collected data from eBPF Maps
Output: S - send data
1 c = a line of the collected data C, V = the set of container ID
2 D = the set of collected data from all containers, D_{id} = the set of collected data from a container
3 V = ∅, D = ∅, D_{id} = ∅
4 **for** *c in C* **do**
5 **if** *c.container_ID* ⊂ *V* **then**
6 get D_{id}
7 **if** $c \in C_{TCP}$ || $c \notin D_{id}$ **then**
8 D_{id} = D_{id} ∪ {c}
9 **if** $(c.time - D_{id}.time)_{min} > time$ **then**
10 S = compressData (D_{id})
11 sendData (S)
12 D_{id} = ∅
13 **else**
14 V = V ∪ c.container_ID
15 D_{id} = D_{id} ∪ {c}
16 D = D ∪ {D_{id}}

Algorithm 1 describes the traces collection optimization process. First, a line of the collected data will be processed from eBPF maps (line 4). If the container ID of this data is already in set V, it indicates that the sent data has already recorded events from this container (line 5). In that case, the data collection of this container will be extracted (line 6).

Next, it is determined whether the data entry already exists in the collection data for this container. Since duplicate container initial events are generated in the container from the container map, the data may already be stored in the collected data. If the data already exists, this entry will be skipped. Requests with identical inputs from the same user will be stored as different data because they come from the TCP map. If this data does not exist, the data will be added to the container data set (lines 7-8). Then, the timestamp of this data will be compared with the timestamp of the earliest data stored in the dataset (line 9). The time window for sending data can be selected. A longer time window means a lower frequency of sending data, which can reduce the consumption of system resources but will affect the real-time analysis of data. A short time window will frequently send data, occupying bandwidth and consuming resources. The one-minute time window is appropriate for sending complete invocation data. Since around 90% of the average interval between function invocations would not exceed one minute [60], one-minute time window can obtain relatively complete data nor consume excessive resources. If the time takes more than one minute, the container's data will be compressed using the engineering method gzip (lines 10). Other log reduction techniques [61], [62] can also be used to save resources. The compressed data will be sent to the FTracker Server, and the collected data for this container will be cleared. (lines 11-12). If the task takes less than one minute, the next data entry will be proceeded. If the container ID of this data entry is not in set V, the container ID and this data entry will be added to the corresponding dataset (lines 14-16).

C. FTracker Server

We design the FTracker Server to integrate information from multiple sources to generate comprehensive provenance graphs. **To achieve Goal 3, the FTracker Server assembles metrics layer by layer to construct the function invocation provenance graph.** As Figure 4 describes, the maps are associated with the key (function IP, container ID, and filename). First, when users invoke functions, the invocations will be stored in the TCP map. The invocation via IP address connects to the function instance that responds to the invocation. Second, the Orchestration map records the container ID of the function instance. The corresponding function instance binds the specific container IDs. Third, the system events generated by the function instance are collected in the Container map and associated with the container ID. Finally, container events related to the host are associated via the container ID and the filename in the File map.

Algorithm 2 outlines the data correlation from different maps and the provenance graph creation from cross-layer data. In the algorithm, the variable r represents a data entry read from the eBPF mapping, the variable s denotes a cross-layer data generated after correlation processing, and the variable g signifies the provenance graph of a particular layer. The algorithm first processes the data from the FTracker Agent, correlating data that comes from different maps. For TCP map data, the response function is located from the Orchestration map based on the IP address (line 6). The processed TCP data are added to the stored data (lines 7-8). For the response function, the container and file data are determined based on the container ID (line 9). The function information is added to the container and file data accordingly (lines 10-13). The processed data is stored with the same container ID (line 14). After correlating the data of the four maps, the data is stored (line 15). A cross-layer provenance graph is generated for the function instance after receiving a termination signal of the function instance (lines 16-17). The function provenance graph can be constructed in time. However, the FTracker Server builds the provenance graph for each function instance when it dies to build a complete and accurate provenance graph.

Algorithm 2 Build Cross-Layer Provenance Graph

```
   Input:  R - Received data from the FTracker Agent
   Output: S - stored data
           G - cross-layer provenance graph
1  r = a line of the received data from R, i in r_i represents
     maps (TCP, orchestration, container, file)
2  s = a line of the cross-layer data
3  g = a single layer provenance graph, i in g_i represents the
     layer (host, container, instance, function)
4  for r_t in R_t, r_c in R_c, r_f in R_f do
5      for r_o in R_o do
6          if r_t.DesIP == r_o.PodIP then
7              r_t.DesFunc = r_o.FuncLabel
8              s = s ∪ {r_t}
9          if r_c.container_ID == r_o.container_ID and
             r_f.container_ID == r_o.container_ID then
10             r_c.Instance = r_o.Podname
11             r_c.IP = r_o.PodIP
12             r_c.Func = r_o.FuncLabel
13             r_f.HostName = r_o.HostName
14             s = s ∪ {r_c} ∪ {r_f}
15         S = storeData(s)
16         if s.signal == SIGTERM then
17             buildGraph(s.container_ID)
18 Function buildGraph(container_ID)
19     g_c = (V, E), G = g_c ∪ g_i ∪ g_h ∪ g_f
20     V = ∅, E = ∅, Container_Data = ∅
21     fileSyscalls = {chdir, mount}
22     procSyscalls = {execve, read, write}
23     netSyscalls = {listen, connect}
24     for s in S do
25         if s.container_ID == container_ID then
26             Container_Data = Container_Data ∪ {s}
27     for c in Container_Data do
28         if c.Syscalls ∈ fileSyscalls then
29             V = V ∪ {c.file1}
30             E = E ∪ {c.file1 → c.file2, c.Syscalls}
31         if c.Syscalls ∈ procSyscalls then
32             V = V ∪ {c.pid}
33             E = E ∪ {c.pid → c.file, c.Syscalls}
34         if c.Syscalls ∈ netSyscalls then
35             V = V ∪ {c.sip : c.sport}
36             if c.DesFunc ∈ ∅ then
37                 E = E ∪ {c.pid → c.sip : c.sport, listen}
38             else
39                 E = E ∪ {c.sip : c.sport → c.DesFunc :
                     c.sport, connect}
40     g_i = g_i ∪ {g_c}
41     g_h = g_h ∪ {g_i}
42     g_f = g_f ∪ {g_h}
43     G = G ∪ {g_f}
```

Moreover, since each function instance has a short lifetime, building a provenance graph for each function instance does not take long.

The *buildGraph* process initializes the empty container graph with system objects as vertices and system calls as edges. The cross-layer provenance graph is composed of container, instance, host, and function provenance graphs (lines 19). The container subgraph is merged into the instance subgraph to which it belongs, the instance subgraph is merged into the node subgraph, and the function subgraph is composed of cross-node subgraphs. Function subgraphs combine with other components to construct a complete cross-layer provenance graph. The smallest granularity of the provenance graph is based on the container. The definition of the container graph is g_c = (V, E), where V represents the nodes and E represents edges. The nodes in the container graph are the system subjects (processes) and objects (files, network connections), while the edges represent the causal dependency events.

Based on the container ID, the FTracker Server gets this container's data from the stored data (lines 25-26). For each entry of the container data, nodes and edges are added to the container graph based on the system calls. The system objects (e.g., files, processes, sockets) are added as nodes in the graph. The edges connect nodes are labeled with corresponding system calls. (lines 27-39). Once the container provenance graph is completed, it will be added to the corresponding function instance provenance graph (line 40). Then, the function instance provenance graph will be added to the graph of the host where it is located (line 41). Next, the function instance provenance graph will be added to the corresponding function graph (line 42). In the end, the function provenance graph will be added to the cross-layer provenance graph containing the platform infrastructure (line 43). The FTracker Server generates the cross-layer provenance graph containing four layers, demonstrating cross-layer attack invocations in FaaS platforms.

V. Implementation

We choose Linux [34] as the operating system, Docker [55] as the virtualization layer technology, Kubernetes [63] as the infrastructure for the Orchestration layer, and OpenFaaS [53] as the serverless platform at the Coordination layer.

FTracker Agent is implemented in GO and C with about 1900 lines of code (LOC). FTracker Server contains about 400 LOC in Go and Python. We leverage eBPF's Kprobe and tracepoint hooks to collect cross-layer logs in the Linux kernel space. In the host kernel space, eBPF hooks get the container *cgroup_name* information, which contains the container ID. For each map, we use different hooks to trace function invocations and correlate them with the container ID. The eBPF hooks correspond to different syscalls. For the TCP map, the Kprobe hooks *inet_csk_listen_start* that catches listen syscalls and *tcp_connect* that catches connect syscalls are attached to trace connections. For the Container map, the tracepoint hooks *sys_enter_execve, sys_enter_read, sys_enter_write* and the Kprobe hook *do_sys_open, send_signal* are attached to record the container system events, including the terminal signal that ends the container process. For the File map, the tracepoint hook *sys_enter_chdir* and the Kprobe hook *do_mount* are attached to trace the file's real path.

We use BPF CO-RE [64] (Compile Once – Run Everywhere) to solve the problem that the eBPF programs need to be compiled on every node. FTracker Agent only needs to be compiled once and can be run on every node. The FTracker Agent and the FTracker Server are deployed as containers that can be orchestrated by Kubernetes. The FTracker Agent runs on every node, and the FTracker Server runs on the master node. We do not make any modifications to any layer of the serverless computing architecture, making our system compatible with any serverless platform.

VI. Evaluation

In this section, we conducted a comparative evaluation of FaaSTracker with state-of-the-art provenance work ALASTOR [26] and CLARION [35]. ALASTOR also aims to build the provenance graph on FaaS. It adds extra components inside the function to collect logs. CLARION is a host-based container-aware provenance tracking framework that distinguishes container and host behaviors. It is deployed on each host node and requires data to be collected separately on each host. However, it does not collect data in summary like FaaSTracker. The data of each host is not connected. Specifically, our evaluation seeks to answer the following four questions:

- **Q1:** Can FaaSTracker construct a cross-layer and complete provenance graph based on the synchronous and the asynchronous function invocation? (§ VI-B.1, § VI-B.4, Effectiveness)
- **Q2:** How much resource does FaaSTracker consume when trace functions? (§VI-C.1, Resource Consumption)
- **Q3:** Does FaaSTracker affect the function performance? (§ VI-C.2, Function Impact)
- **Q4:** How many logs does FaaSTracker generate? (§ VI-C.3, Space Overhead)

A. Experiment Setup

We consider several typical serverless applications to evaluate FaaSTracker, including (1) a commercial application *Hello, Retail!*, (2) a machine learning application *Image Pipeline*, and (3) a web application *bookinfo*. These applications have been widely used in recent serverless studies [26], [65], [66], [67].

Hello, Retail! [39] consists of 4 synchronous functions that implement various tasks such as purchase, query, authorization, and publish. The synchronous invocation invocations wait for the function to complete execution, and its result is immediately returned to the user. Since Hello, Retail! is all synchronous function invocations, the function execution time is short.*Hello, Retail!* is ideal for evaluating FaaSTracker performance of constructing provenance graphs from complex function behaviors (§ VI-B.1). Meanwhile, it is also suitable for evaluating the impact of FaaSTracker on function performance and log size since each function generates a large number of events (§ VI-C.2, § VI-C.3).

Image Pipeline [40] consists of 4 functions, 2 of which are asynchronous functions that use machine learning to classify and crop images. Asynchronous invocations will be placed on an internal queue. The result will be returned to the user once the function has handled the asynchronous request. Since the asynchronous function execution speed is slow, the new function request will not be executed immediately and will wait for the previous function request to complete. We use *Image Pipeline* to evaluate the asynchronous function invocation (§ VI-B.4).

bookinfo [68] consists of 4 synchronous functions that use the web page to display the details of books. It is easy to generate large burst web-based requests to evaluate the component performance of FAASTRACKER (§ VI-C.1).

1) Testbed Setup: We deployed FAASTRACKER on a 3-node Kubernetes cluster (v1.23.6) with OpenFaaS gateway (v0.26.3). The cluster used Docker (v23.0.1) as the container runtime. Each node has 8 × 2.20-GHz Intel Xeon E52620 CPUs and 64 GB of RAM. Ubuntu 20.04 is installed on the nodes, along with kernel versions 5.15.0 (master node), 5.13.0 (worker node1), and 5.8.0 (worker node2). We use different kernel versions on nodes to demonstrate that FAASTRACKER will not be affected by the kernel version.

B. Functionality Performance

To evaluate FAASTRACKER 's ability to construct the cross-layer attack provenance graph, we simulate two real attacks on serverless platforms based on the CVE-2022-0847 and CVE-2021-22555. These two attacks occur in synchronous and asynchronous function invocations as mentioned in § II-A.

1) Attack on Synchronous Function Invocations: Based on CVE-2022-0847, we performed a real attack on *Hello, Retail!*. Users purchase products by invoking *Hello, Retail!* functions. CVE-2022-0847 is a privilege escalation vulnerability that exists in Linux kernel versions 5.8 and later. By exploiting this vulnerability, the attacker can overwrite read-only files. With the capability *CAP_DAC_READ_SEARCH*, the attacker can even modify files on the host from a container, achieving container escape. We take the example depicted in Figure 2(a) in this evaluation. The attacker compiles an executable file exploiting the CVE-2022-0847 through a function invocation. As described in § II-A, this attack crosses four layers of the FaaS architecture, starting from the function invocation at the Coordination layer and exploiting the host file at the Operating System layer.

2) Attack Reconstruction With FAASTRACKER*:* The cross-layer provenance graph generated by FAASTRACKER is shown in Figure 5. It demonstrates how the attack crosses the four layers of serverless computing architecture. To simplify the provenance graph, we omit some metadata (e.g., timestamps, files) from the graph and use event sequence numbers to show the events chronologically. Red lines and red-colored components represent the attack path.

The attacker first invokes *Hello, Retail!* function chain through the API gateway at the Coordination layer (①). The invocation is sent to the ingress function *product-purchase* (②) at the Orchestration layer. An instance of this function responds to the request at the Virtualization layer, and it invokes internal function *product-publish* through the API gateway (③). These two steps cross three layers of the architecture. At the Orchestration layer, the function has different instances. Figure 5 accurately records the instance responding to the request. After responding to the request, the function instance generates events within the container at the Virtualization layer. Some files in the container are mounted by the host in the operating system. When function *product-publish* receives the invocation (④), it compiles the exploit file and executes it to escape from the container (⑤ -⑧). The new process reads the secret host file *kubelet.key* and writes to another host file */home/name* (⑨ -⑩) at the Operating System layer. The attacker pretends to be a normal user to invoke function *nodeinfo*(⑪). This function is used to get information about the node where the function is scheduled, including the node name, CPU, and memory. Since the container file */home/name* is mounted from the host, after modifying the host file with the key, even normal function invocations can access sensitive files on the host (⑫ -⑭).FAASTRACKER accurately generates an attack provenance graph based on the collected cross-layer data. Based on the function provenance graph, platform providers clearly find the abnormal cross-layer behavior and locate the attacker's function invocations request.

3) Attack Reconstruction With ALASTOR and CLARION: ALASTOR builds the local provenance graph in each function instance and connects them through the API gateway in the global provenance graph. However, as shown in Figure 6, it only collects logs inside the function at the Virtualization layer and function invocations from the API gateway at the Coordination layer. The logs it collected do not accurately reflect the actual location of the files. The global provenance graph it generated can not reflect the cross-layer operations (steps ⑨ -⑭) in Figure 5. It also can not discover that two functions *product-publish* and *nodeinfo* have an implicit relationship. This is very important to discover the real cause of the attack, and also to discover which files the attacker has tampered with. In addition, cross-layer behavior is very suspicious for attack detection, and ALASTOR does not reflect cross-layer behavior.FAASTRACKER shows exactly where files are located and what cross-layer behavior occurs when a function is invoked.

As shown in Figure 7, CLARION builds the provenance graph in each node. Although it can accurately distinguish the behavior of containers and hosts, it can not show the invocation of the function. CLARION can not match the IP to the function instance. It also cannot connect container behavior to function behavior. And since functions exist for a very short time, unmatched container events cannot be associated again. Since the CLARION provenance graph is not based on functions, it cannot reflect the function invocation sequence and the function relationship based on the attack. Therefore, it cannot accurately locate the function invocation from the attacker. However, FAASTRACKER clearly distinguishes between container and host behaviors while accurately associating function invocations with function instances.

4) Attack on Asynchronous Function Invocations: Based on CVE-2021-22555, we conducted an attack on *Image Pipeline* through asynchronous function calls. CVE-2021-22555 is a

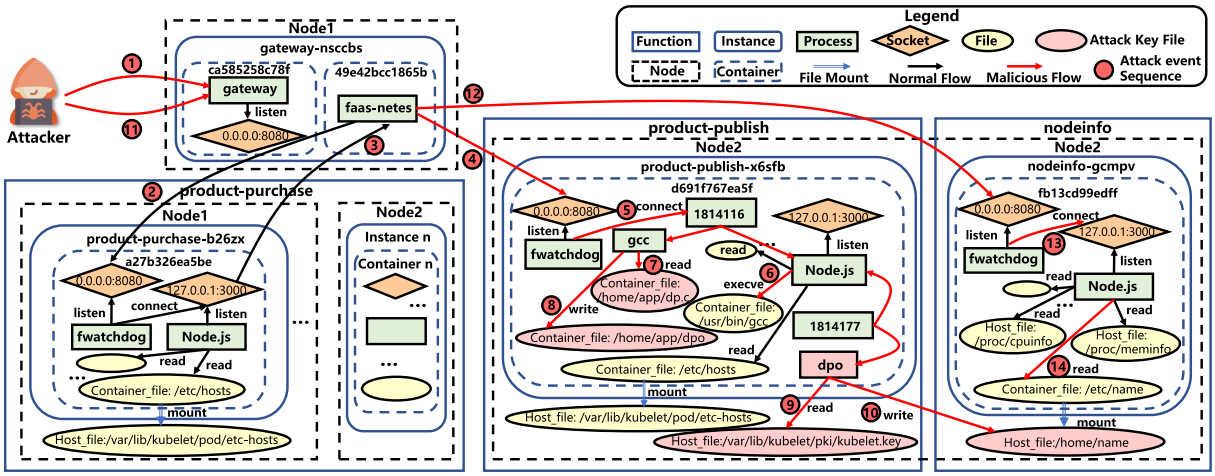


Fig. 5. Cross-layers provenance graph for synchronous function invocation generated by FAASTRACKER.

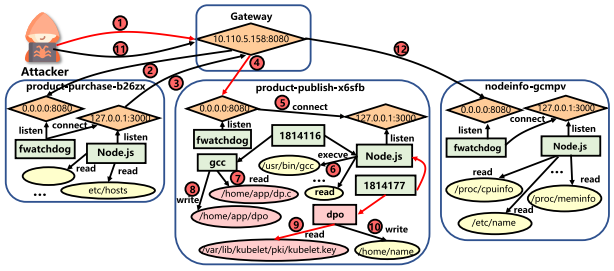


Fig. 6. Global provenance graph for synchronous function invocation generated by ALASTOR.

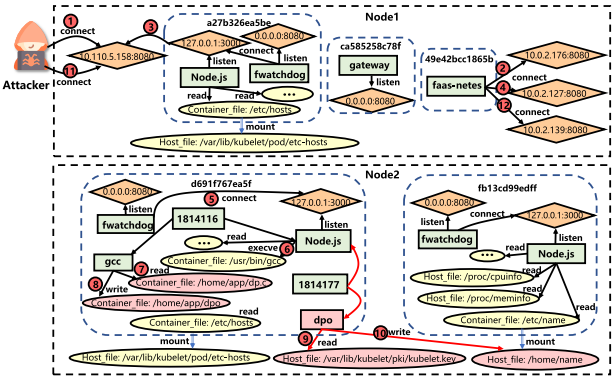


Fig. 7. Node provenance graph for synchronous function invocation generated by CLARION.

heap overflow vulnerability in the Linux kernel. It allows an attacker to gain privileges. The attacker performs cross-layer attacks by uploading malicious files to the database. As Figure 2(b) shows in § II-A, the attacker directly obtain the host's confidential files in the database.

5) Attack Reconstruction With FAASTRACKER*:* The cross-layer provenance graph of asynchronous function invocation is depicted in Figure 8. When the attacker uploads the exploit file to the database, the database will invoke the API gateway (① -②). The API gateway forwards the request to *webhook-handler* (③). When *webhook-handler* invokes *image-resizer* asynchronously, the request is serialized to the queue by API gateway via NATS and queue workers (④ -⑤). At a later time, the queue worker then dequeues and deserializes the request. The API gateway makes the request to the *image-resizer* (⑥-⑦). When *image-resizer* receives the request, it will download the exploit file from the database (⑧). Next, it executes this file to achieve container escape (⑨). After obtaining the host's secret file *kubelet.key*, it will upload the file to the database (⑩-⑪). The provenance graph reflects the queuing of requests during the asynchronous invocations and maps asynchronous requests to attack behaviors. Although the attack invocation is not directly invoked by the attacker, FAASTRACKER can still reflect the attack behavior. FaaS providers can use the provenance graph to find exploited files in the database and leaked confidential information.

6) Attack Reconstruction With ALASTOR and CLARION: ALASTOR does not consider asynchronous function invocations when building provenance graphs. This affects the construction of the global provenance graph. As shown in Figure 9, invocations can not be correctly matched to events on the global provenance graph since requests are not processed in real-time. The requests in Figure 8 (steps ⑤ - ⑦) can not display in Figure 9. ALASTOR is also unable to distinguish the location of the file accurately. It can observe abnormal code execution, but cannot accurately determine the attacked files and function invocations. Similar to synchronous function invocations, FAASTRACKER accurately displays cross-layer function behavior while accurately matching queued function invocations.

As shown in Figure 10, CLARION is still unable to recognize the function relationship. It can not reflect the asynchronous invocations between functions and associate the asynchronous function invocation with the container. The connection between the attack and the attacker cannot be found in Figure 10. The attacker can pretend to be a normal user uploading files, so it cannot accurately locate the attacker. FaaS providers cannot find the connection between the attacker and the attack through this provenance graph. But they can use

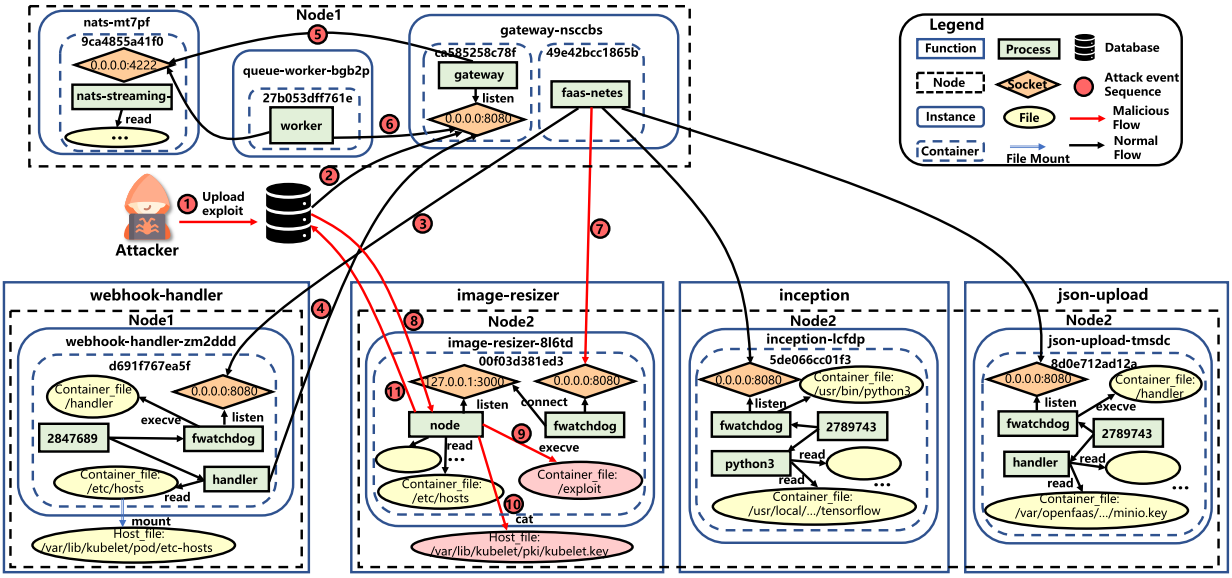


Fig. 8. Cross-layers provenance graph for asynchronous function invocation generated byFAASTRACKER.

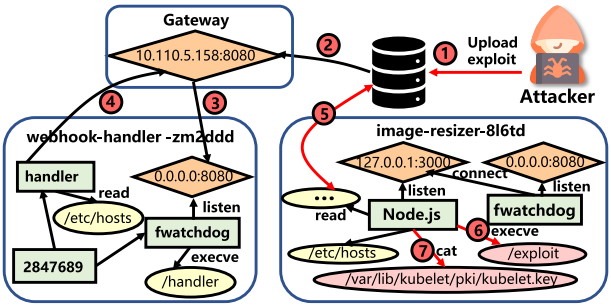


Fig. 9. Global provenance graph for asynchronous function invocation generated by ALASTOR.

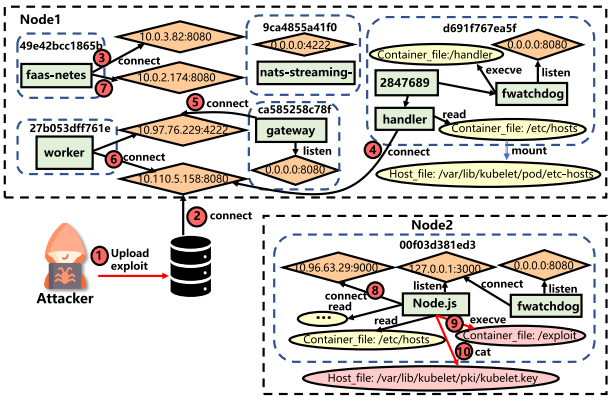
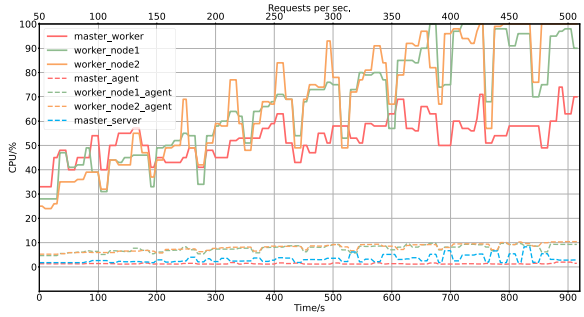


Fig. 10. Node provenance graph for asynchronous function invocation generated by CLARION.

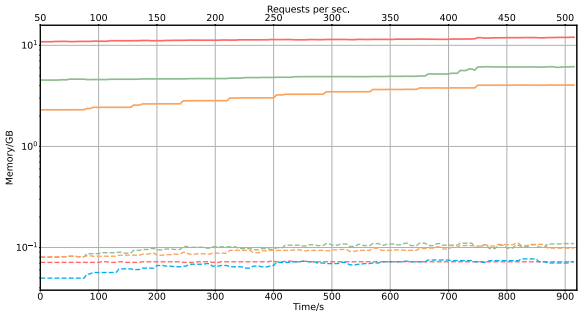
FAASTRACKER to accurately obtain the attacker's behavior in the container and host.

C. Runtime Performance

We measure the resource consumption of FAASTRACKER components under varying loads. Since CLARION is not designed for serverless platforms, it cannot be directly used in the distributed system, we only compare FAASTRACKER resource utilization with ALASTOR. First, we evaluate FAASTRACKER components' resource consumption under burst traffic. Second, we compare the impact of FAASTRACKER and ALASTOR on functions. Third, we compare the log size FAASTRACKER and ALASTOR under the same number of requests.

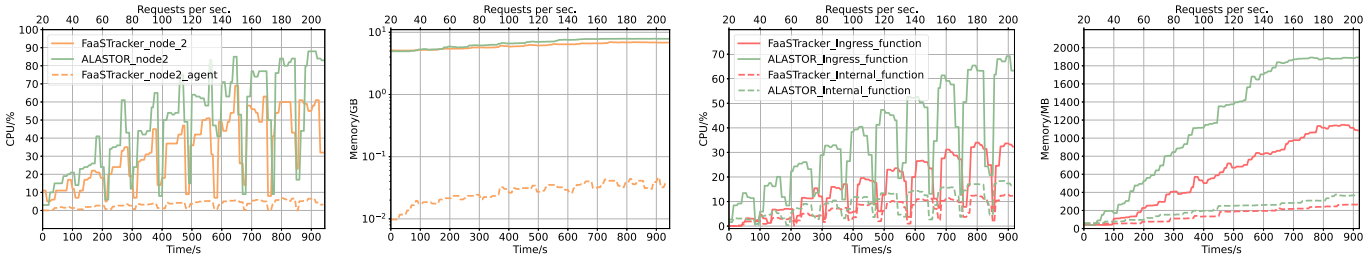


(a) Nodes and Components CPU Consumption



(b) Nodes and Components Memory Consumption

Fig. 11. Resource utilization of FaaSTracker Components. As the number of requests increases, the resources consumed by the node increase rapidly, but the resource consumption of FaaSTracke components increases slowly.



(a) Node CPU Consumption (b) Node Memory Consumption (c) Function CPU Consumption (d) Function Memory Consumption

Fig. 12. Resource utilization in FaaSTracker and ALASTOR. With the same requests per second, FaaSTracker consumes fewer resources than the ALASTOR.

1) FAASTRACKER *Components Resource Consumption:* We deploy the FTracker Agent at each node and the FTracker Server at the master node. Each node is deployed infrastructure pods (e.g., networking pods). The master node runs the Kubernetes control components, so its CPU and memory consumption are initially higher. The API gateway and other OpenFaaS components are located at the worker node1. Although we only evaluated FAASTRACKER on three nodes, the function is scalable.

We use the application *bookinfo* to evaluate the CPU and memory utilization of FAASTRACKER. The functions of *bookinfo* are scheduled to worker node1 and node2. Each function has a limit on CPU and memory consumption. Both functions and the API gateway have one instance at the beginning and will scale when their CPU consumption exceeds 80%. We use the HTTP load generator *hey* [69] to issue high burst external request loads of from 50 to 500 requests per second (rps).

As shown in Figure 11, the solid line represents the consumption of node resources, and the dotted line represents the consumption of FAASTRACKER component resources. As the number of requests increases, the CPU and memory resources consumed by the nodes increase rapidly. The FTracker Agent and FTracker Server resource consumption also increases but at a slower rate, and the resources they consume only account for a small part of the overall node resource consumption. When the CPU utilization on the node exceeds 95%, the highest CPU consumption of the FAASTRACKER agent on this node is only 10.4%. The highest CPU consumption of the FTracker Server is 8.3%. The increase in memory resource consumption is relatively slow. The maximum memory consumption of the FTracker Agent is 112 MB, and the FTracker server is 79 MB. Compared to the several gigabytes of memory consumption on the node, FAASTRACKER memory consumption is entirely acceptable.

2) Impact of FAASTRACKER *on Functions:* We first compare the function resource utilization of FAASTRACKER and ALASTOR under the same external requests and the same application *Hello Retail!*. Then we compare the function cold start time and response latency of FAASTRACKER and ALASTOR. FAASTRACKER directly deploys the original functions without making any changes. Functions do not require the installation of any additional components or dependencies. ALASTOR modifies each function instance by adding extra components. It installs a Linux diagnostic tool *strace* [70] to monitor processes and a database to store data inside the function. We deploy original and ALASTOR functions of *Hello Retail!* on the same node, worker node2. These two types of functions will be scaled when CPU resource consumption exceeds 80%. In FAASTRACKER, there is also an FTracker Agent on the worker node2.

As Figure 12(a) and Figure 12(b) show, although FAASTRACKER adds an agent on each node, FAASTRACKER consumes fewer node resources than ALASTOR under responding to the same number of requests. The FTracker Agent consumes no more than 6.1% of node CPU resources and 50 MB of Memory resources. FAASTRACKER reduces node CPU resource consumption by 30.2% and Memory resource consumption by 9.9% compared to ALASTOR. The reason FAASTRACKER reduces resource consumption is that functions consume fewer resources. ALASTOR performs auditing in function instances, functions consume more resources when facing the same number of requests. Figure 12(c) and Figure 12(b) show the functions' resource consumption. *Hello Retail!* consists of the ingress function and internal functions. As Figure 12(a) shows, users invoke ingress function *product-purchase*. The ingress function invokes internal functions to finish the users' tasks. All internal function traffic goes through the ingress function, so its resource consumption will be extremely large. Whether in the ingress function or the internal function, ALASTOR consumes more resources due to the audit of the inside the function. ALASTOR consumes 54.0% more function CPU resources and 48.9% function Memory resources than FAASTRACKER, which far exceeds the resources FTracker Agent consumes on the node. ALASTOR performs behavior tracing inside the function, so the greater the function resource consumption, the greater the function overhead. FAASTRACKER does not trace inside the function. The function overhead will not increase when functions consume more resources. The gap between the ingress functions of ALASTOR and FAASTRACKER is wider than the internal functions.

Figure 13 shows the average function cold start time and response latency. We repeatedly measured the cold start time for a function 100 times to get the average time. Compared to FAASTRACKER without modifying the function runtime environment, the intrusive tracing ALASTOR increases by 21.7% cold start time. We measure the time to receive a function response after sending an invocation request from a client (curl) on the local host machine under 100 requests per second. Since ALASTOR performs auditing inside the function when responding to invocations, it also increases the response latency by 11.9% compared to FAASTRACKER.

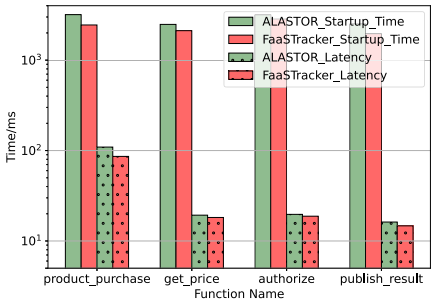


Fig. 13. Time overhead in FaaSTracker and ALASTOR. Both function cold start time and response latency are increased in ALASTOR compared to FaaSTracker.

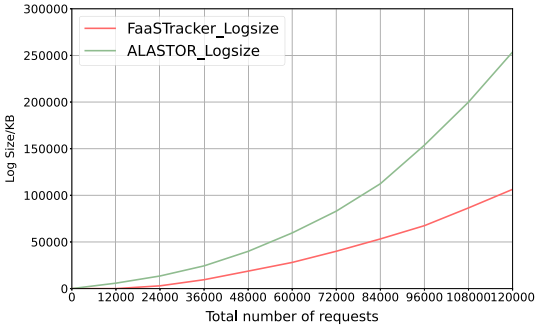


Fig. 14. Log size in FaaSTracker and ALASTOR. With the same requests per second, the total number of log sizes of FaaSTracker is less than ALASTOR.

3) Generated Log Size Comparison: FAASTRACKER optimizes tracing process to reduce the log size in § IV-B. While ALASTOR only extracts logs directly from all functions without optimization. We compare the size of logs in ALASTOR and FAASTRACKER. We generate the same number of requests for the same application *Hello Retail!*. Compared to the log collection in the ALASTOR paper [26], we evaluated with a higher number of requests. For ALASTOR, we collect the raw logs from function instances. Logs in function instances are collected using the strace tool that captures system calls. For FAASTRACKER, we directly count the size of the logs collected in the FTracker Server. Logs are collected across layers through different FTracker Agents and saved as raw data in JSON format. Figure 14 demonstrates the log growth for increasing requests to *Hello Retail!*.FAASTRACKER greatly reduces the log size compared with ALASTOR. When the total number of requests is 120000, the number of logs collected by ALASTOR is 247.7 MB, and FAASTRACKER is 105.1 MB which is 110.8 MB less than ALASTOR. Although the log size is reduced, critical information is not lost. As described in section VI-B,FAASTRACKER builds a more complete and accurate provenance graph than ALASTOR.

D. Applicability of FAASTRACKER

FaaS providers like AWS, Azure, and Google may use secure containers(e.g., Firecracker [71], Kata [72]) at the virtualization layer, which cannot use eBPF hooks at the host to trace secure container behaviors. However, FAASTRACKER provides a general cross-layer tracing framework. The virtualization layer data originally collected by eBPF can be replaced with the behavior data of the secure container. By correlating security container data with other layer data, FAASTRACKER can still build a cross-layer provenance graph. Tracing in the secure container is not the focus of the paper, but it can be used as a future research direction.

E. Security of FAASTRACKER

FAASTRACKER is designed for the FaaS platform provider. FTracker Agent is only deployed by the FaaS platform provider and runs on the host. We assume that the platform is trustworthy. Multi-tenants can obtain the function data provided by the platform, but do not have the privilege to deploy and enter FAASTRACKER. So, Multi-tenants do not have access to the kernel level. Besides, eBPF only has visibility to the kernel, and the FTracker Agent cannot manipulate kernel system calls.

F. Intrusion Detection Based on FAASTRACKER

The raw logs and function provenance graphs generated by FAASTRACKER can be ingested for threat detection(e.g., Denial-of-wallet [18], [19]). The function provenance graph is an important source for provenance-based intrusion detection. It reveals the relationship between system events in a function. Intrusion Detection is not the focus of our work. Since there is no intrusion detection work on FaaS, our work can serve as a basis for future research.

VII. CONCLUSION

In this paper, we present FAASTRACKER, a cross-layer, non-intrusive, and high-performance tracing framework for FaaS. First, to trace function invocations across the FaaS layer, we design the FTracker Agent with eBPF kernel hooks. The cross-layer hooks provide a unified tracing approach for different technologies. Second, with trace collection optimization, the FTracker Agent efficiently traces functions on the node with high performance. Non-intrusive tracing does not affect the function's runtime performance. Third, FAASTRACKER integrates traces from multiple sources to construct the cross-layer attack invocation provenance graph. FAASTRACKER accurately presents root cause analysis for cross-layer attacks on synchronous and asynchronous invocations of functions which is effective for PIDS. The overhead of FAASTRACKER is negligible. When the functions consume more than 90% of the CPU resources on the node, the FTracker Agent CPU resource consumption accounts for only 6.3% and the memory consumption is 91MB. In addition, compared to ALASTOR, FAASTRACKER reduces node CPU resources consumption by 54.0% and Memory resources consumption by 48.9%. Besides, compared to the intrusive tracing, FAASTRACKER reduces 21.7% cold start time and 11.9% function response latency.

ACKNOWLEDGMENT

Any opinions, recommendations, or findings are those of the authors and do not reflect the views of Alibaba Cloud Computing.

REFERENCES

[1] H. Yu et al., "RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 1, Apr. 2024, pp. 335–350.

[2] (2023). *AWS Lambda Introduction*. [Online]. Available: https://aws.amazon.com/?nc2=hlg

[3] Microsoft. (2023). *Azure Monitor Overview*. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/overview

[4] Google. (2023). *Google Cloud Monitoring Overview*. [Online]. Available: https://cloud.google.com/monitoring?hl=en

[5] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," *ACM Comput. Surveys*, vol. 54, no. 11s, pp. 1–36, Jan. 2022.

[6] E. Marin, D. Perino, and R. Di Pietro, "Serverless computing: A security perspective," *J. Cloud Comput.*, vol. 11, no. 1, pp. 1–12, Oct. 2022.

[7] H. M. Makrani et al., "Cloak & co-locate: Adversarial railroading of resource sharing-based attacks on the cloud," in *Proc. Int. Symp. Secure Private Execution Environ. Design (SEED)*, Sep. 2021, pp. 1–13.

[8] C. Fang et al., "Repttack: Exploiting cloud schedulers to guide co-location attacks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Apr. 2022.

[9] L. Wang et al., "Incorporating gradients to rules: Towards lightweight, adaptive provenance-based intrusion detection," 2024, *arXiv:2404.14720*.

[10] J. Zengy et al., "SHADEWATCHER: Recommendation-guided cyber threat analysis using system audit records," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 489–506.

[11] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "PROGRAPHER: An anomaly detection system based on provenance graph embedding," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 4355–4372.

[12] M. U. Rehman, H. Ahmadi, and W. U. Hassan, "Flash: A comprehensive approach to intrusion detection via provenance graph representation learning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2024, pp. 3552–3570.

[13] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan, "HOLMES: Real-time APT detection through correlation of suspicious information flows," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1137–1152.

[14] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1139–1155.

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

[16] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 386–400.

[17] (2023). *What is Serverless Security?*. [Online]. Available: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-serverless-security/

[18] 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.

[19] D. Kelly, F. G. Glavin, and E. Barrett, "Denial of wallet—Defining a looming threat to serverless computing," *J. Inf. Secur. Appl.*, vol. 60, Aug. 2021, Art. no. 102843.

[20] P. Liu et al., "Understanding the security risks of Docker hub," in *Proc. 25th Eur. Symp. Res. Comput. Secur. Comput. Security*, Guildford, U.K. Cham, Switzerland: Springer, Sep. 2020, pp. 257–276.

[21] Amazon. (2023). *AWS X-Ray Introduction*. [Online]. Available: https://aws.amazon.com/xray/?nc1=hls

[22] Datadog. (2023). *Monitor AWS Lambda Functions With Datadog*. Accessed: Oct. 16, 2023. [Online]. Available: https://www.datadoghq.com/

[23] IOpipe. (Oct. 2023). *Gartner Names IOpipe Cool Vendor in Performance Analysis*. [Online]. Available: https://www.iopipe.com/

[24] Dashbird. (2023). *Agentless Monitoring and Error Detection for Serverless*. [Online]. Available: https://dashbird.io/

[25] Q. Zeng, M. Kavousi, Y. Luo, L. Jin, and Y. Chen, "Full-stack vulnerability analysis of the cloud-native platform," *Comput. Secur.*, vol. 129, Jun. 2023, Art. no. 103173.

[26] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the provenance of serverless intrusions," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 2443–2460.

[27] U. Satapathy, R. Thakur, S. Chattopadhyay, and S. Chakraborty, "DisProTrack: Distributed provenance tracking over serverless applications," in *Proc. IEEE Conf. Comput. Commun.*, May 2023, pp. 1–10.

[28] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1172–1189.

[29] A. Bates, D. Tian, R. B. K. Butler, and T. Moyer, "Trustworthy whole-system provenance for the Linux kernel," in *Proc. USENIX Conf. Secur. Symp. (SEC)*, Austin, TX, USA, Aug. 2015, pp. 319–334.

[30] L. Yu et al., "ALchemist: Fusing application and audit logs for precise attack provenance without instrumentation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021.

[31] Q. Wang et al., "You are what you do: Hunting stealthy malware via data provenance analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[32] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "Mpi: Multiple perspective attack investigation with semantic aware execution partitioning," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1111–1128.

[33] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, Dec. 2015, pp. 401–410.

[34] Linux. (Oct. 2023). *The Linux Audit Daemon*. [Online]. Available: https://linux.die.net/man/8/auditd

[35] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran, "CLARION: Sound and clear provenance tracking for microservice deployments," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 3989–4006.

[36] (May 2024). *Comparing Lambda Invocation Modes*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/invocation-modes.html

[37] (2024). *Asynchronous Invocation*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html

[38] (2023). *National Vulnerability Database*. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2022-0847

[39] *Hello Retail! Application*, SimonEismann, Würzburg, Germany, 2021.

[40] *Image Pipeline Application*, viveksyngh, Chennai, India, 2021.

[41] (2023). *National Vulnerability Database*. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2021-22555

[42] T. Weng, W. Yang, G. Yu, P. Chen, J. Cui, and C. Zhang, "Kmon: An in-kernel transparent monitoring system for microservice systems with eBPF," in *Proc. IEEE/ACM International Workshop Cloud Intelligence (CloudIntelligence)*, May 2021, pp. 25–30.

[43] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2020, pp. 697–702.

[44] C. Lee, R. Yoshitani, and T. Hirotsu, "Enhancing packet tracing of microservices in container overlay networks using eBPF," in *Proc. 17th Asian Internet Eng. Conf.*, Dec. 2022, pp. 53–61.

[45] J. Levin and T. A. Benson, "ViperProbe: Rethinking microservice observability with eBPF," in *Proc. IEEE 9th Int. Conf. Cloud Netw. (CloudNet)*, Nov. 2020, pp. 1–8.

[46] J. Shen et al., "Network-centric distributed tracing with DeepFlow: Troubleshooting your microservices in zero code," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 420–437.

[47] E. Ates et al., "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2019, pp. 165–170.

[48] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *ACM Trans. Comput. Syst. (TOCS)*, vol. 41, pp. 1–28, Jun. 2016.

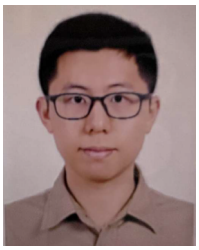
[49] M. N. Hossain et al., "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *Proc. USENIX Conf. Secur. Symp. (SEC)*, Vancouver, BC, Canada, Aug. 2017, pp. 487–504. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hossain

[50] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "POIROT: Aligning attack behavior with kernel audit records for cyber threat hunting," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 1795–1812.

[51] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[52] R. Sekar, H. Kimm, and R. Aich, "EAudit: A fast, scalable and deployable audit data collection system," in *Proc. IEEE Symp. Secur. Privacy*, May 2024, pp. 3571–3589.
[53] OpenFaaS. (2023). *OpenFaaS–Serverless Functions Made Simple*. [Online]. Available: https://docs.openfaas.com/
[54] Knative. (2023). *Knative is an Open-Source Enterprise-Level Solution to Build Serverless and Event Driven Applications*. [Online]. Available: https://knative.dev/docs/
[55] Docker. (2023). *Make Better, Secure Software From the Start*. [Online]. Available: https://www.docker.com/
[56] Kubernetes. (2023). *Kubernetes Introduction*. [Online]. Available: https://kubernetes.io/
[57] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Junior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Comput. Surveys*, vol. 53, no. 1, pp. 1–36, Jan. 2021.
[58] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating distributed protocols with eBPF," in *Proc. NSDI*, Apr. 2023, pp. 1391–1407. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/zhou
[59] Y. He et al., "Cross container attacks: The bewildered eBPF on clouds," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 5971–5988.
[60] 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.
[61] M. N. Hossain et al., "Dependence-preserving data compaction for scalable forensic analysis," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1723–1740.
[62] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: high-fidelity attack investigation via transparent multi-layer log analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.
[63] Kubernetes. (2023). *Kubernetes Inc*. [Online]. Available: https://kubernetes.io/
[64] (2020). *BPF CO-RE (Compile Once–Run Everywhere)*. [Online]. Available: https://nakryiko.com/posts/bpf-portability-and-co-re/
[65] K. Hou, S. Lin, Y. Chen, and V. Yegneswaran, "QFaaS: Accelerating and securing serverless cloud networks with QUIC," in *Proc. 13th Symp. Cloud Comput.*, Nov. 2022, pp. 240–256.
[66] Q. Zeng, K. Hou, X. Leng, and Y. Chen, "DirectFaaS: A clean-slate network architecture for efficient serverless chain communications," in *Proc. ACM Web Conf.*, May 2024, pp. 2759–2767.
[67] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in *Proc. Web Conf.*, Apr. 2020, pp. 939–950.
[68] Istio. (2023). *Bookinfo Application*. [Online]. Available: https://istio.io/latest/docs/examples/bookinfo/
[69] rakyll. (2020). *Hey Introduction*. [Online]. Available: https://github.com/rakyll/hey
[70] (Jun. 2024). *Strace Introduction*. [Online]. Available: https://github.com/strace/strace
[71] Firecracker. (2023). *Secure and Fast MicroVMs for Serverless Computing*. [Online]. Available: https://firecracker-microvm.github.io/
[72] Aliyun. (2025). *Kata Container Introduction*. [Online]. Available: https://katacontainers.io/

Qingyang Zeng received the B.E. degree from North China Electric Power University, Beijing, China, in 2020. She is currently pursuing the Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China. Her research interests include serverless, microservices, and cloud security.



Lianjie Wu received the B.E. degree from Shanghai University, Shanghai, China, in 2022. He is currently pursuing the Ph.D. degree with Zhejiang University, Hangzhou, China. His current research interests include cloud security and container security.

Kaiyu Hou received the Ph.D. degree in computer science from Northwestern University, IL, USA, in 2022. He is currently a Senior Engineer with Alibaba Cloud Computing. His research interests include cloud networks, networked systems, and network protocols.

Xue Leng (Member, IEEE) received the Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2020. She is currently a Jingying Associate Professor with Hangzhou Institute of Technology, Xidian University, Hangzhou, China. Her research interests include security enhancement and performance optimization of cloud systems.

Yan Chen (Fellow, IEEE) received the Ph.D. degree in computer science from the University of California at Berkeley, Berkeley, CA, USA, in 2003. He is currently a Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA. Based on Google Scholar, his articles have been cited over 7000 times, and his H-index is 34. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. He won the Department of Energy (DoE) Early CAREER Award in 2005, the Department of Defense (DoD) Young Investigator Award in 2007, and the Best Paper nomination in ACM SIGCOMM 2010.