# ALASTOR: Reconstructing the Provenance of Serverless Intrusions

Pubali Datta
*University of Illinois at Urbana-Champaign*

Isaac Polinsky
*North Carolina State University*

Muhammad Adil Inam
*University of Illinois at Urbana-Champaign*

Adam Bates
*University of Illinois at Urbana-Champaign*

William Enck
*North Carolina State University*

## Abstract

Serverless computing has freed developers from the burden of managing their own platform and infrastructure, allowing them to rapidly prototype and deploy applications. Despite its surging popularity, however, serverless raises a number of concerning security implications. Among them is the difficulty of investigating intrusions – by decomposing traditional applications into ephemeral re-entrant functions, serverless has enabled attackers to conceal their activities within legitimate workflows, and even prevent root cause analysis by abusing warm container reuse policies to break causal paths. Unfortunately, neither traditional approaches to system auditing nor commercial serverless security products provide the transparency needed to accurately track these novel threats.

In this work, we propose ALASTOR, a provenance-based auditing framework that enables precise tracing of suspicious events in serverless applications. ALASTOR records function activity at both system and application layers to capture a holistic picture of each function instances' behavior. It then aggregates provenance from different functions at a central repository within the serverless platform, stitching it together to produce a global data provenance graph of complex function workflows. ALASTOR is both function and language-agnostic, and can easily be integrated into existing serverless platforms with minimal modification. We implement ALASTOR for the OpenFaaS platform and evaluate its performance using the well-established Nordstrom *Hello,Retail!* application, discovering in the process that ALASTOR imposes manageable overheads (13.74%), in exchange for significantly improved forensic capabilities as compared to commercially-available monitoring tools. To our knowledge, ALASTOR is the first auditing framework specifically designed to satisfy the operational requirements of serverless platforms.

## 1 Introduction

Serverless Computing, the newest offering in the cloud computing ecosystem, has become an attractive solution for many companies including Netflix, T-Mobile and Zillow [20]. Also known as Function-as-a-Service (FaaS), serverless allows large companies to conveniently auto-scale to massive loads while also allowing small start-ups to "scale-to-zero" and avoid the costs of idle servers [40]. Consequently, the serverless market growth was estimated from $1.88 billion in 2016 to $7.72 billion by 2022, an annual growth rate of 32.7% [64]. Serverless platforms free web-developers from all hardware and software stack management tasks, allowing them to rapidly prototype applications [99] as interdependent set of small task-specific functions. They can even make use of existing function logic, made available through public markets [2, 73], closed-source license agreements [46, 53], or third-party libraries [44, 45].

At first glance, serverless computing would seem to significantly raise the bar for would-be attackers; after-all, the provider-managed platform is likely to be correctly configured, the small footprint of each function lends itself to stringent access control restrictions, and the stateless and short-lived nature of function execution seems to eliminate the possibility of persistent compromise. Unfortunately, industry security researchers quickly discovered that this was not the case [58, 61]. One major oversight in the serverless security model is the ubiquitous practice of caching recently-invoked functions in memory to improve performance. Known as "warm container reuse," this optimization grants attackers the ability to establish quasi-persistence when a vulnerability is discovered, violating the isolation of individual function invocations. The container reuse problem is exacerbated by poor security policy and configuration [58], enabling attackers to move laterally through function workflows. As a result of these capabilities, vulnerabilities can be exploited just as easily in serverless applications as they can in traditional servers.

Given that a variety of attack strategies are feasible on serverless, is the full range of traditional defensive solutions also available? Traditional approaches (e.g., Linux Audit [95]) lack visibility into the platform and are thus oblivious to key serverless semantics such as function instances, platform APIs, and container reuse. The same is true of state-of-the-art

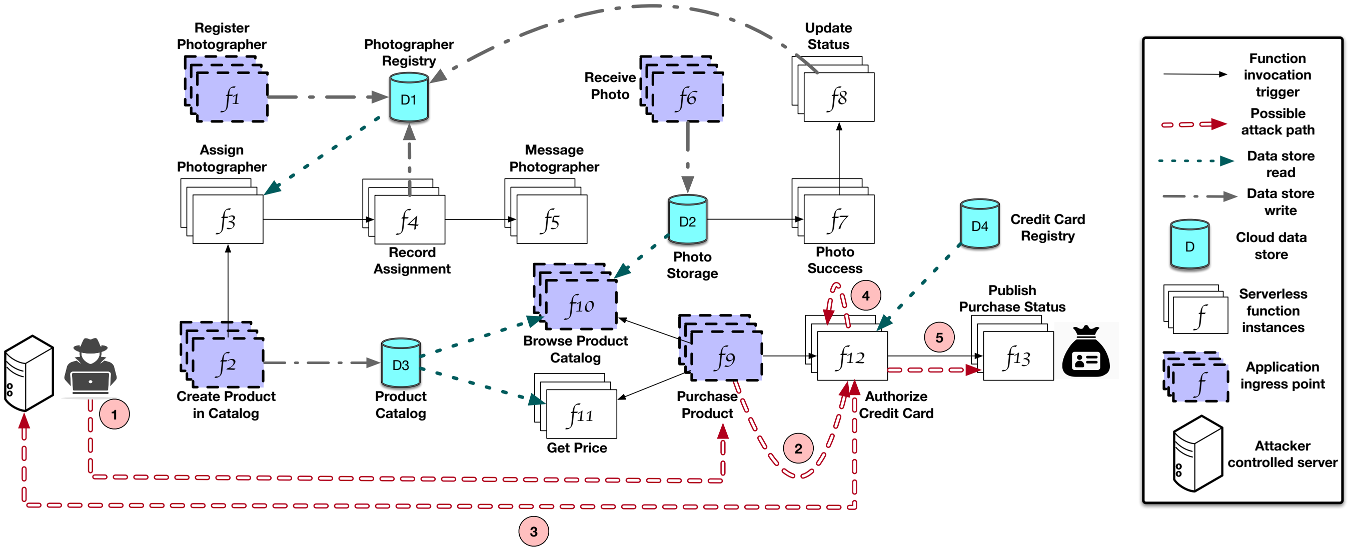


Figure 1: Architecture of *Hello,Retail!*, created by Nordstrom, which here is annotated to demonstrate possible attack strategies against serverless applications. Red arrows denote the attack path used in our case study in Section 8. This attack leverages both container reuse and function workflows to exfiltrate data.

provenance-based auditing systems, many of which aggressively filter terminated system activities (e.g., [62,63]) and are thus in conflict with the notion of stateless ephemeral functions. Worse yet, current serverless-specific industry solutions are also lacking. Platforms offer limited support for execution tracing, error reporting, resource utilization, and function monitoring [3,6,69,70], but these services provide only a restricted view of the application and are often limited by strict usage limits [4]. Such limits discourage developers from logging vital forensic evidence, contradicting the attack investigation philosophy of "constant vigilance" [52]. Encouragingly, there is a range of third party observability tools [13, 84, 85, 90] that offer features like distributed tracing and cost analysis, but these services are limited to certain language runtimes and primarily target function-level protections. Unfortunately, we are not aware of any service (or combination of services) that provides holistic visibility into system-, network-, and platform-layer interactions, all of which are necessary in order to accurately trace serverless attacks.

In this paper, we seek to resolve these challenges through considering the following questions. (1) *Serverless provenance:* What are the specific agents, entities, and activities that must be monitored to accurately reconstruct the provenance of serverless applications? (2) *Universal auditing for serverless:* How can forensic evidence at various levels of the serverless stack be integrated to facilitate effective threat investigation? In answering these questions, we propose ALASTOR, a serverless provenance framework that collects information at different levels of the platform stack at finer granularity than existing solutions. In ALASTOR, function instances (i.e., containers) are monitored both at the system level and the application level (e.g., monitoring HTTP requests) and collected information are reconciled to provide a holistic picture of happenings inside the instance. The global provenance builder service in ALASTOR collates information collected at different function instances at a central reservoir within the platform. This service then stitches the pieces of information to derive a holistic provenance graph that succinctly explains the interconnection and causation relationship between all the components of a serverless application. ALASTOR is function agnostic and can easily be integrated into existing serverless platforms with minimal modification. We implement the ALASTOR framework on top of the OpenFaaS serverless platform. In this paper, we make the following contributions.

- We present ALASTOR, a function-agnostic provenance framework for attack investigation in serverless environments. ALASTOR differs from traditional provenance tools in that ALASTOR keeps track of both dead and live entities in the environment.
- We implement ALASTOR [1] on top of the open source serverless platform OpenFaaS and measure its performance overhead compared to the vanilla OpenFaaS environment. We discover that ALASTOR imposes only 13.74% overhead.
- We conduct a serverless intrusion case study on the well-known *Hello,Retail!* application in which we compare ALASTOR to Epsagon [13], a state-of-the-practice commercial serverless tracing tool. We demonstrate ALASTOR's superior capabilities by reconstructing the attack path in full detail, making it easy to diagnose the intrusion and determine its impacts.

[1] Our code and data are available for download at https://bitbucket.org/sts-lab/alastor/

## 2 Background and Motivation

Serverless computing liberates customers from the burdens of managing their own software stack; virtual machine provisioning, patches and upgrades to the operating system, load balancing, and auto-scaling of requests are instead the responsibility of the cloud provider. Serverless adopts a pay-per-use model where customers are billed according to their CPU, memory, and network usage only for the duration of their function executions, significantly reducing application deployment costs [16]. Serverless enables rapid prototyping of applications by allowing developers to implement the business logic as a set of small reentrant functions which chain together into workflows that perform high-level tasks. To protect the platform, from malicious or compromised applications, functions are executed in isolated containers ("function instances") that are intended to provide a sterile environment for each stateless and short-lived function invocation. For example, Figure 1 depicts the *Hello,Retail!* serverless e-commerce application.

**Attacks on Serverless:** While serverless applications are subject to traditional web application vulnerabilities [96] including event injection [29,56,58,61,78] and vulnerabilities in library and platform code [5,9,43,47], industry practitioners claim that ephemeral serverless functions make exploiting such vulnerabilities more difficult in practice [49]. However, despite the fact that a function's lifecycle typically spans just milliseconds of time, attackers have found new ways to exploit serverless functions. Figure 1 includes a depiction of an exploitation where synthetic vulnerabilities are injected into the *Hello,Retail!* application for the purposes of illustration. While we leave the details of the attack to our case study discussion in Section 8, we mention it here to motivate two key attack strategies often administered by attackers:

- *Exploiting Container Reuse:* Even though the functions are supposed to be stateless, the cost of setting up an entire runtime environment for each function execution has encouraged container reuse. That is, "warm" containers are cached and reused for future invocations of the same function within a pre-configured timeout window [17,72,97]. Opaque platform policies and scheduling algorithm details obscure this practice, making it difficult for customers to account for such issues during application development. Attackers have discovered that warm container reuse can be exploited to achieve persistence by writing malware or toolkits to an in-memory partition (e.g., `/tmp`), then forcing the compromised function instance to remain in the cache [58]. Step ④ in Figure 1 exploits container reuse.

- *Exfiltration through Function Workflows:* Many attacks depend on the ability to exfiltrate stolen data [58]. Unfortunately, simply restricting functions' network access is an ineffective deterrent; attackers have developed methods of laundering stolen data through downstream authorized functions and legitimate platform APIs in order to reach the open Internet [61,78]. They can leverage legitimate function transitions to move laterally through the application [58,61,78]. Moreover, the complexity of serverless access control policies leads to increased chance of misconfigurations [41,74], thus creating greater opportunity for attackers. Step ⑤ in Figure 1 exfiltrates data over an authorized workflow.

In addition to the above attack scenarios, the abundance of third party functions [2,44–46,53,73] creates an additional attack surface, exposing serverless applications to untrusted and potentially malicious function code.

**Limitations of Existing Approaches:** Data provenance techniques are used in operating systems to parse system-level audit logs (e.g., Windows ETW [68], Linux Audit [1]) into a causal graph that describes the dependencies between system subjects (e.g., processes) and system objects (e.g., files, network sockets). Data provenance techniques can be used in root cause analysis by tracing the graph backwards starting from a suspicious event. Moreover, a forward tracing query can help in understanding the consequences of the attack. Thus, provenance graphs are useful for investigating cyber attacks [60]. However, most system-level provenance techniques are confined to the events happening inside a single machine, while the nature of attacks on serverless platforms calls for distributed tracing and auditing mechanisms.

While a specialized auditing framework for serverless has not yet emerged, growing evidence of attacks against serverless platforms has led to the release of various tools to improve runtime observability into serverless applications. Cloud providers offer execution tracing, error reporting, alerts, and resource usage breakdowns [3,6,69,70]. However, these techniques tend to focus on individual functions in isolation and are often impeded by strict usage limits [4]. A range of third party observability tools [13,84,85,90], offer additional features but are limited to certain language runtimes and platforms. Security mechanisms within these products are mostly geared towards function-level protection and do not consider more complex multi-function attack paths. When these products offer distributed tracing (e.g., [13]), they provide only an opaque view of function execution that does not explain system-level interactions, which is likely too coarse-grained to diagnose and investigate attacks. Moreover, these tools do not consider intra-container interactions (i.e., container reuse), which is necessary to prevent cross-invocation attacks.

## 3 Threat Model & Assumptions

This work considers attacks against a serverless application running in a third party public compute cloud platform, e.g., Amazon Lambda. We assume that the cloud provider and platform infrastructure are trusted, meaning that the provider will correctly deploy functions and will not attempt to collude with attackers. Based on the popularity of Amazon Lambda and other similar FaaS platforms, we assume that the

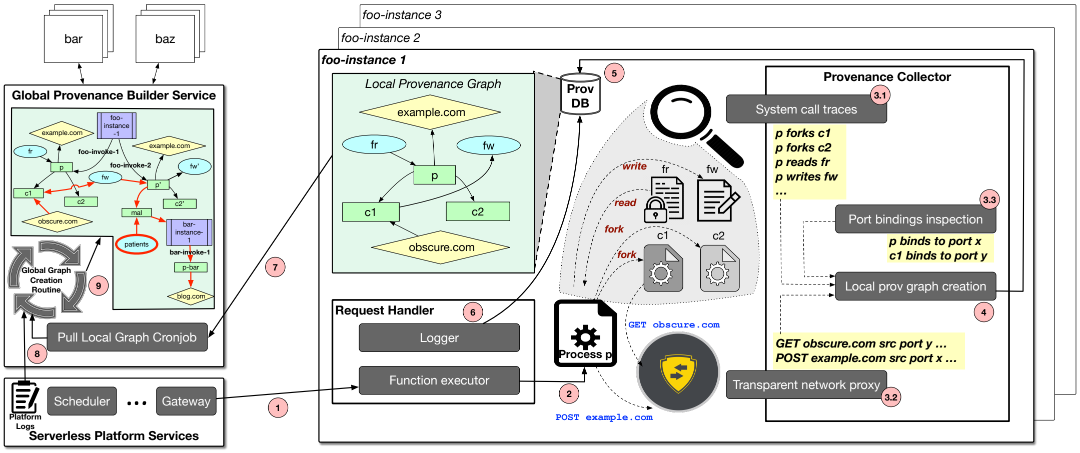


Figure 2: An overview of the ALASTOR architecture and the provenance collection workflow.

provider is offering container-based serverless environments, not language-based environments. Container-based platforms are popular because they do not restrict customers to certain languages or runtime environments, but this model also raises increased security concerns because functions are not (necessarily) written in type/memory-safe languages and they can interact with the underlying host. Just as customers may run potentially-buggy function code, we also admit the possibility that customer's access control policy configurations, which restrict the permissions of individual functions, are misconfigured or insufficient. This is because there is already ample evidence that role-based access controls in clouds (e.g., Amazon IAM) are often incorrectly configured [41, 74] or are overly permissive, allowing the attacker to traverse legitimate function workflows to advance their goals [58, 61, 78].

Within this environment, we consider a serverless cloud application that is the target of a sophisticated remote attacker with the primary goal of data exfiltration. To do so, the attacker can leverage a variety of traditional techniques and procedures that are known problems on traditional computers (e.g., [7]), including binary exploitation, command injection, downloading and executing penetration testing tools etc. *In addition*, they are also able to employ the aforementioned serverless-specific attack techniques to achieve persistence and exfiltrate data within this environment. The attacker can use the compromised functions to execute any permissible system flow to exfiltrate data, including transmission to the external network, writing to persistent storage somewhere in the cloud, or even writing to ephemeral storage inside the function container for later retrieval. While the attacker effectively has free rein within compromised function instances, we assume that the attacker does not have administrative access to the victim customer's account.[2] Thus, they are unable to launch their own functions, modify container images, or tamper with access control policies.

We make the following additional assumptions about this environment. Based on the architectures used by popular serverless platforms today, we also assume the presence of an API gateway in the cloud platform to handle external requests originating from the public Internet. We further make the assumption that all serverless functions are invoked through the use of REST API calls or other forms of Remote Procedure Calls (event triggers, asynchronous callbacks). This assumption is appropriate because web and API serving are the most popular use cases in the serverless paradigm [57]. Like most auditing literature (e.g., [48, 65, 71, 77, 100]), we assume that our event tracing mechanisms and the event logs are correct at the time of their use. Because the cloud platform is trusted and the attacker does not have administrative access to the customer account, it is reasonable to assume that secure storage for these logs exists. Under a more aggressive threat model, log integrity could be verified through the use of tamper-evident cryptographic protocols (e.g., [59, 75, 76]). Finally, we do not consider cloud side channels (e.g., cross-tenant side channels [80]) in this paper.

## 4 Overview

Given the state of serverless security, it is clear that there is a pressing need for improved auditing and transparency. However, deploying provenance-based auditing techniques in the serverless domain poses several notable challenges:

- *Auditing ephemeral activities.* Serverless functions are short-lived. Forensic analysis is not typically designed to support auditing system entities that no longer exist or do not effect the present state of the system. For example, in the pioneering LogGC paper [63], Lee et al. deem such events

[2] While exceptions have been observed [58], granting individual functions access to the platform API keys is an egregious misconfiguration of access control policy that is unlikely to occur. It is important to note, however, that this means the attacker we consider is less capable than a true insider attacker.

as "garbage" and propose removing them from the log. A variety of auditing frameworks similarly prune repeated events [100] or causal paths [51]. State in serverless architectures is extremely limited; applying such techniques to serverless risks destroying vital evidence of attack activities.

- *Replication of vulnerable programs.* A vulnerability in one function implicates the security of many functions and containers that may be replicated across many instances on different physical machines. Devising methods to trace a vulnerability back to candidate sources and assess its potential impact on the broader application is not straightforward.
- *Unwieldy auditing costs.* Serverless customers pay only for the resources they truly need. That said, serverless is far from optimal in terms of its footprint in the audit log — because function infrastructure is in a constant state of re-launch and teardown, there are actually many more events associated with a function invocation than would be expected from a typical web request. In other words, we would expect the overheads associated with serverless auditing to be *even worse* than traditional server infrastructure.

**Our Approach:** While auditing serverless is challenging, a key design aspect of serverless platforms can be leveraged to ease the challenges — *execution partitioning* [62], an essential precursor to causal analysis, is innate in serverless. In execution partitioning, long-lived processes are subdivided into autonomous units of work, allowing an investigator to trace from a process output to the associated process input without following irrelevant inputs (i.e., false provenance). Due to the event-driven nature of serverless, low-level system events can be reliably bound to an event-trigger. Furthermore, high fan-out processes are scarce due to ephemeral functions. Therefore, it is not necessary to perform additional execution partitioning because process activities are already short-lived and will be associated with little false provenance, if any.

Based on these observations, we propose ALASTOR, a serverless provenance framework that transparently collects information at every function instance (i.e., container) both at the system level and the application level, then reconciles them to provide a holistic picture of happenings inside the instance. ALASTOR aggregates information from different functions at a central reservoir within the platform and encodes the discovered causal dependencies into a global data provenance graph to enable serverless attack investigation.

**ALASTOR:** Figure 2 describes two main components of ALASTOR: the provenance collector and the global provenance builder service. The provenance collector resides inside each function instance and collects both system-level and application-level activities initiated by the executing function. The global provenance builder service communicates to the containers through the underlying container runtime and fetches local provenance data stored in the function instances. This service then stitches the pieces of information to derive a holistic provenance graph that succinctly explains the interconnection and causation relationship between all the components of a serverless application.

**Provenance Collection Workflow:** The request handler inside a function container starts listening for incoming requests when the container is deployed on the platform. The provenance collection workflow is set in motion when the function executor inside the request handler receives an invocation request (①). The executor then forks a process (②) that executes the function logic. The Provenance Collector collects a trace of the process, file and network system calls (3.1) invoked by the forked process and its descendants. The Provenance Collector also intercepts the function's network traffic through the use of a transparent network proxy (3.2), and reconciles this higher level information with the lower level system events through inspecting the port bindings (3.3). This combined information is parsed into a local provenance graph by the Provenance Collector (④) and is stored in a local database within the container (⑤). The local provenance graph is appended with application level metadata (e.g., request ID, function execution duration, request and response headers and body) emitted from the request handler logger (⑥).

The global provenance builder service runs on the control plane of the serverless platform, using a cronjob that pulls the local provenance graphs from all function instances (⑦). The global graph creation routine runs within the global provenance builder service and collates the local graphs and some metadata collected from the logs of the platform services (e.g., container ID, container state) (⑧) into a global graph that defines the behavior of a serverless application (⑨).

# 5 ALASTOR Design

## 5.1 Provenance collector

Provenance Collector, the primary component in ALASTOR architecture, performs four tasks described below.

**System Call Tracing:** The Provenance Collector's system call tracing mechanism keeps track of the process, file and network related system calls using the *strace* utility. The tracing mechanism generates system-level event trace as shown at step 3.1 in Figure 2. System call tracing is critical to provenance collection because an attacker's malicious activities are recorded in the set of system calls invoked by them. For example, cross-invocation interference achieved through container reuse [58] involves calling file-related system calls `open`, `read` and `write` with same set of files as input parameters, across invocations. This creates an explicit data flow from one function execution to another and this data flow can be captured by inspecting the system call trace.

**Network Profiling:** While system call tracing provides visibility into low-level network activity, IP-level information is not sufficiently descriptive because application components are regularly replicated and migrated. To address this, we additionally profile the REST API calls and http network requests that form the basis of serverless communications [57]. To observe API usage and http requests, ALASTOR deploys a transparent network proxy in each container. This network proxy monitors all incoming and outgoing network requests (step (3.2) in Figure 2) originating from the container. To address the matter of encrypted traffic, ALASTOR's provenance collector contains an HTTPS proxy (*mitmproxy* [32]), which is a common technique in enterprise environments for monitoring security-sensitive network flows [36]. The network profiling component not only allows ALASTOR to reconstruct function workflows, but is also useful in identifying malware downloads or sensitive data exfiltration to remote servers.

**Process to Network Request Association:** While the network proxy provides insight into the higher level network activities of the function, it is unable to associate a network request to the specific system process that initiated the request. To address this limitation, ALASTOR uses the *ss* utility to map processes to port numbers (*pid* → *port*) of TCP sockets they use (step (3.3) in Figure 2) and computes the source port of originating network requests (*port* → *request*) using *mitmproxy*. This information is combined to compute the set of requests sent by each process (*pid* → *request*) in the container.

**Local Provenance Graph Creation:** In this phase, the provenance collector encodes the collected provenance into a provenance graph. The vertices in this graph are the system subjects (processes) and objects (files, network connections) observed in the function instance while the edges represent the causal dependency events. A causal dependency event can be a system call invocation or a REST API call or network request. The edges are annotated with a timestamp of the event and the type of event (e.g., read, open, GET etc.).

**Algorithm 1** Local Provenance Graph Creation

**Inputs:** Request handler log *r.log*; *process.trace* files; network proxy output *net.log*
**Output:** Local provenance graph *G*

```
 1: $G := (V,E)$
 2: $containerId$ := Read container ID from /proc/self/cgroup
 3: $V := \{containerId\}$
 4: $E := \emptyset$
 5: $\textsc{Map}(\langle invreqId; pid\rangle)$ := Compute $InvocationRequestID$ to $PID$ mappings from
    $r.log$
 6: for each $\langle invreqId; pid\rangle \in \textsc{Map}(\langle invreqId; pid\rangle)$ do
 7:     $V := V \cup \{pid\}$
 8:     $E := E \cup \{(containerId \rightarrow pid, invreqId)\}$
 9: end for
10: for each $\langle pid; httpReq\rangle$ computed from $net.log$ do
11:     $V := V \cup \{pid\}$        ▷ $httpReq$ is a two-tuple $(\langle ipaddr : port\rangle, restOp)$ where
                                       $restOp \in$ {get, post, put, delete}
12:     $V := V \cup \{\langle ipaddr : port\rangle\}$
13:     $E := E \cup \{(pid \rightarrow \langle ipaddr : port\rangle, restOp)\}$
14: end for
15: $procSyscalls$ := {execve,fork,clone}
16: $netSyscalls$ := {bind,listen,connect,accept,sendto,rcvfrom}
17: $writeParams$ := {O_WRONLY,O_RDWR,O_APPEND,O_CREAT,O_TRUNC}
18: for each $(v \in V \mid v.type == process)$ do
19:     $\textsc{getSystemCalls}(v.pid)$
20: end for

21: function $\textsc{getSystemCalls}(pid)$
22:     for each $(sysCall \in pid.trace \mid sysCall.retVal \geqslant 0)$ do
23:         if $sysCall \in procSyscalls$ then
24:             $child.pid$ := Compute child process ID from
                $sysCall.params$ and $sysCall.retVal$
25:             $V := V \cup \{child.pid\}$
26:             $E := E \cup \{(pid \rightarrow child.pid, sysCall)\}$
27:             $\textsc{getSystemCalls}(child.pid)$
28:         end if
29:         if $sysCall \in netSyscalls$ then
30:             $\langle ipaddr : port\rangle$ := Compute from $sysCall.params$
31:             $V := V \cup \langle ipaddr : port\rangle$
32:             $E := E \cup \{(pid \rightarrow \langle ipaddr : port\rangle, sysCall)\}$
33:         end if
34:         if $sysCall ==$ open then
35:             $file$ := Compute file name from $sysCall.params$
36:             $V := V \cup \{file\}$
37:             if $sysCall.params \cap writeParams \neq \emptyset$ then
38:                 $E := E \cup \{(pid \rightarrow file, write)\}$
39:             else if O_RDONLY $\in sysCall.params$ then
40:                 $E := E \cup \{(file \rightarrow pid, read)\}$
41:             end if
42:         end if
43:     end for
44: end function
```

Algorithm 1 outlines the local provenance graph creation routine used by the Provenance Collector. The algorithm takes as input the request handler logs, system call traces of processes and process to network requests mappings as computed above. The algorithm initializes the graph with a single container vertex representing the function instance and an empty set of edges (lines 1-4). The container ID can be read from the `proc` directory within the container (e.g.,`/proc/self/cgroup` file in Docker container runtime). Next, the request handler logs are parsed to identify processes associated with individual incoming function invocation requests and these process nodes are added in the graph (lines 5-9). The network profiling provenance is encoded as IP address vertices connected to respective process nodes (lines 10-14). Then the GETSYSTEMCALLS(P) routine is called on each of the process vertices currently present in the graph (lines 18-20). This routine (lines 21-44) parses the *strace* output for the process *p*. The system calls invoked by process *p* are examined to eliminate failed system calls (failed system calls may have negative return values) (line 22). This routine also (lines 23-28) computes the progeny of *p* by tracking the *fork*, *execve* and *clone* system calls and recursively calls GETSYSTEMCALLS on the child processes. The process nodes (*p* and its descendants) are included in the graph and are connected with edges directed from the parent to the children nodes. Next, the system call parameters are processed (lines 29-42) to compute the system objects (e.g., files, sockets) affected by the successful system calls. These objects are added as nodes in the graph and the edges connecting process nodes to system objects it accessed are labeled with corresponding system calls. The direction of the edge depends on the direction of the data flow. For example, an edge labeled `read` will be directed to the process from the file object (line 40), and a `write` edge will be in the opposite direction (i.e., from the process to the file being written) (line 38). In each function instance, a local provenance graph is created and stored until it is fetched from the *Global Provenance Builder Service*.

## 5.2 Global Provenance Builder Service

The global provenance builder service works alongside other system services within the serverless platform. This service implements one of the unique features of serverless provenance collection technique: *dead-provenance*. Dead-provenance deals with keeping track of the objects not existing in the environment anymore (i.e., dead containers) and computing how the events induced in the past by dead objects causally influence live objects. Dead-provenance is essential in serverless environments to detect attacks described in Section 2, in contrast to traditional monolithic system provenance [63]. The global provenance builder service performs its operations in the following two phases.

**Information collection from the platform and the instances:** The global provenance builder service queries the platform services to learn about the deployed functions, running containers and their descriptions (e.g., whether the containers are in *initialized*, *running* or *terminated* states). Platforms generally assign identifiers to every object in the ecosystem. The unique identifiers for every function and every container enables this service to trace flows across function-instances that may or may not be active within the same timespan, or might have been terminated before. Unique identifiers (normally assigned by the image registries) for container images are useful for tracing vulnerabilities discovered in one forensics investigation in other uses of the same image.

Most serverless platforms' designs allow every function instance to be uniquely addressable within the overlay network that connects them. Using these network addresses of the instances, the global provenance builder service runs a periodic routine that fetches local provenance graphs from each of the function containers with the help of the underlying container runtime apis. By design, when a container's health starts to deteriorate (e.g., the root process in the container receives a SIGTERM or SIGKILL signal), the container runs a *pre-termination* routine that sends the remaining local provenance information to the global provenance builder service that has not been yet fetched by the service, ensuring the completeness of the global provenance graph created in the next step.

**Global provenance graph creation:** The global provenance graph algorithm (Algorithm 2) takes as input the set of local provenance graphs generated from Algorithm 1, and the serverless platform logs (gateway logs, DNS server logs etc.). The algorithm initializes the global graph with vertices corresponding to various platform services (lines 1-4). Then all the edges and vertices from the local graphs are added to the global graph (lines 6-9). The platform logs record the IP addresses assigned to the different containers. A container ID to IP address map is created from these logs (line 10). The platform gateway routes and logs every request to a function whether the request is intra-platform or externally originated. Finally, from the gateway logs and the map constructed in line 10, the flow of requests among containers is computed, and the corresponding edges labelled with request ID and timestamp are added to the global provenance graph(lines 11-14).

**Algorithm 2** Global Provenance Graph Creation

```
Inputs: Set of local provenance graphs $\{lg_{containerId}\}$; $platform.log$ files
Output: Global provenance graph $G$

1: $G := (V, E)$
2: $gateway :=$ Read gateway IP from $platform.log$
3: $platform\_dns :=$ Read DNS server IP from $platform.log$
4: $V := \{gateway, platform\_dns\}$
5: $E := \emptyset$
6: for each $lg_{containerId}$ do
7:     $V := V \cup \{lg_{containerId}.V\}$
8:     $E := E \cup \{lg_{containerId}.E\}$
9: end for
10: $\text{MAP}(\langle IP; containerId \rangle) :=$ Compute IP $assigned$ to $containerId$ from $platform.log$ for each container
11: for each $reqId$ logged by the gateway in $platform.log$ do
12:     $E := E \cup \{(gateway \rightarrow containerId, reqId)\} \mid MAP[destIP_{reqID}] == containerId$
13:     $E := E \cup \{(containerId \rightarrow gateway, reqId)\} \mid MAP[srcIP_{reqID}] == containerId$
14: end for
```

When pre-termination provenance arrives from a terminating container, the subgraph corresponding to that container is appended with the new information and the corresponding container vertex is labeled *terminated*. In this way ALASTOR preserves dead provenance in the global provenance graph. As described in Figure 2, a complete multi-function attack path is visible in the global provenance graph which is not discoverable when the local graphs are inspected in isolation.

# 6 Implementation

We implemented ALASTOR in Go within OpenFaaS,[3] an open-source serverless platform. OpenFaaS is compatible with several container orchestration platform backends, including Kubernetes and Docker Swarm. We deployed ALASTOR with OpenFaaS over Kubernetes. We primarily modified two components of OpenFaaS, the *watchdog* and *of-watchdog*, representing an addition of approximately 75 lines of Go code (excluding build scripts, comments and blank lines). These components act as request handlers in the function container that receive and process incoming requests to the function. Our changes are transparent to the function and do not interfere with the functionalities of these components.

Our modifications are aimed towards instrumenting the request handler in each container to proxy network requests and track system calls as part of ALASTOR's Provenance Collector. We used the Mitmproxy [32] python library to proxy HTTP and HTTPS requests generated by the functions destined towards addresses external to the Kubernetes cluster. The intra-cluster traffic is monitored using system logs emitted from Kubernetes apiserver, OpenFaaS gateway, and the instrumented function-container. The system call tracing mechanism is implemented using the *strace* utility. The

[3] https://www.openfaas.com/

Global Provenance Builder Service is implemented as an independent cron job that runs at the Kubernetes control plane, without requiring any change in the underlying platform.

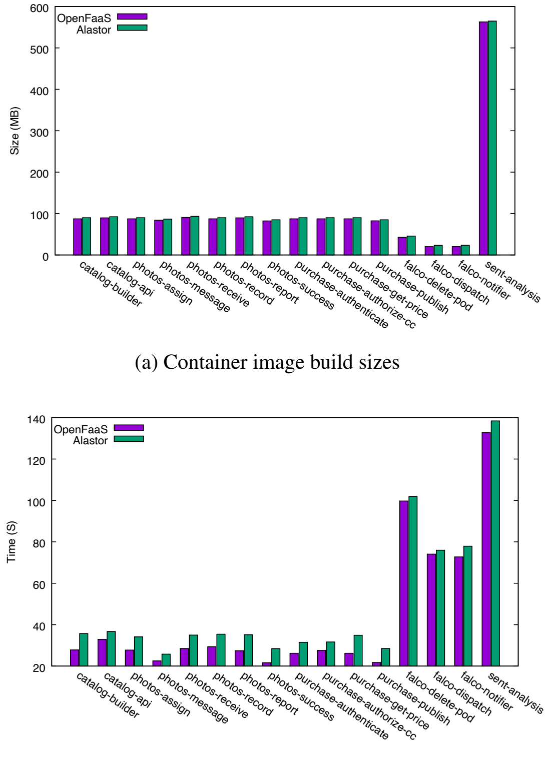
## 7 Performance Evaluation

In this section, we evaluate the performance overhead of ALASTOR. We deployed ALASTOR on a server-class machine with 64-core Intel(R) Xeon(R) CPU E5-2683v4 @2.10GHz and 132 GB memory running CentOS Linux 7 (Core) 64 bit OS. Our experiments used the Docker version 20.10.3 as the container runtime, Kubernetes version 1.18.8 for orchestration, and OpenFaaS with of-watchdog version 0.8.1. We configured the Kubernetes cluster as a single node cluster with both the control plane services and user deployed workloads running on the same node (i.e., the server-class machine). All Docker images required for the following experiments were pre-pulled in order to minimize the effects of external networking variations. We compared ALASTOR's performance against the standard OpenFaaS (Vanilla). Since Epsagon does not support the OpenFaaS platform, we were unable to include it in our performance experiments.

**Application workloads:** Our primary test application is *Hello,Retail!*, [4] which has been extensively used in recent serverless literature [19, 33, 81]. Shown in Figure 1, *Hello,Retail!* consists of 13 functions, 5 of which are publicly accessible and 8 are internal (can only be accessed by other functions). There are also 4 data stores used by different functions. We make use of Alpernas et al.'s fork of *Hello,Retail!* [19], which replaces calls to the AWS-specific components (e.g., S3) with calls to open-source alternatives (e.g., sql-datastore), and make minimal further modifications so that the application could run on the OpenFaaS framework deployed on a Kubernetes cluster.

We also make use of two additional applications, a threat response engine (`falco` [14]) and a sentiment analysis app (`sent-analysis`) [15]. The Falco app [38] consists of 3 functions: `dispatch` ingests alerts from a threat detection engine, then invokes either `notifier` for publishing alerts to an administrator channel (NOTICE level alerts) or a `delete-pod` function that deletes the offending Kubernetes pod (WARNING level alerts). The sentiment analysis application consists of a single function that takes text as input, computes a sentiment of the text, and returns the result. This corpus of applications will allow us to characterize ALASTOR's performance for different workload profiles.

**Build and Orchestration Performance:** The pre-deployment costs of building and storing container images are shown in Figures 3.[5] The build sizes and times are averaged across 30 iterations for each function. The small increase in image size over Vanilla OpenFaaS is due to the additional ALASTOR code that is compiled into the OpenFaaS `of-watchdog` binary and the installation of the tracing mechanism and proxy certificates. Build time overhead can be attributed to installing system call tracing and proxy libraries during image building. These are *one-time* costs incurred when building an image for the first time.



(a) Container image build sizes

(b) Container image build times

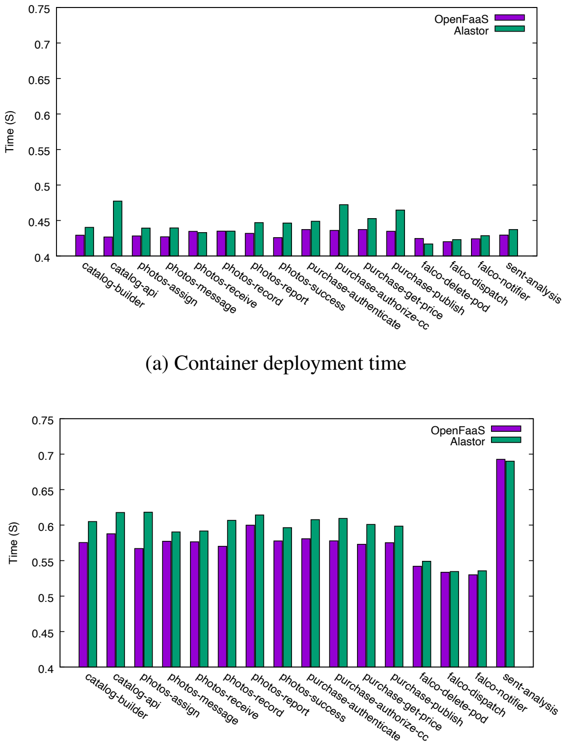
Figure 3: Container image build performance comparison.

Overheads for orchestration performance are shown in Figure 4. Figure 4a reports the function deployment time, which is the time required for the docker runtime to transition a function instance from the "Container Creating" state to the "Ready" state. Figure 4b reports the function teardown time, which is the time required to transition the container from the "Running" state to the "Terminated" state. Both figures show the average overhead across 50 iterations of each function. Averaging across all functions, ALASTOR deployment and teardown overheads are 3.2% and 3.6%, respectively.

**Runtime Performance:** Of particular importance is the impact of ALASTOR on function response latency, which we measure as the time taken to receive a function response after sending an invocation request from a client (`curl`) on the local host machine. Figure 5 shows the response latencies for each function, averaged over 500 repetitions. To improve readability, the figure splits the y-scale such that the *photos-receive* and *sent-analysis* functions with larger latencies can be shown in the top half. Across the *Hello,Retail!* application,

[4] https://github.com/Nordstrom/hello-retail

[5] Function names are shortened in the figures for brevity.



(a) Container deployment time

(b) Container teardown time

Figure 4: Orchestration performance for functions with ALASTOR as compared to vanilla OpenFaaS.

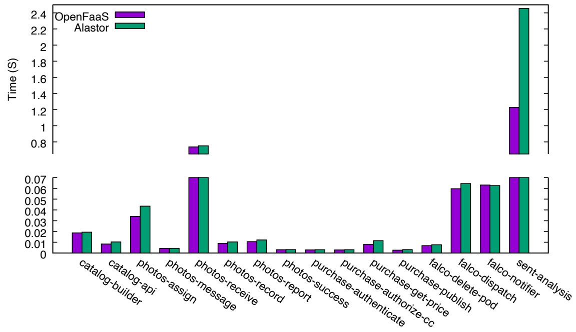
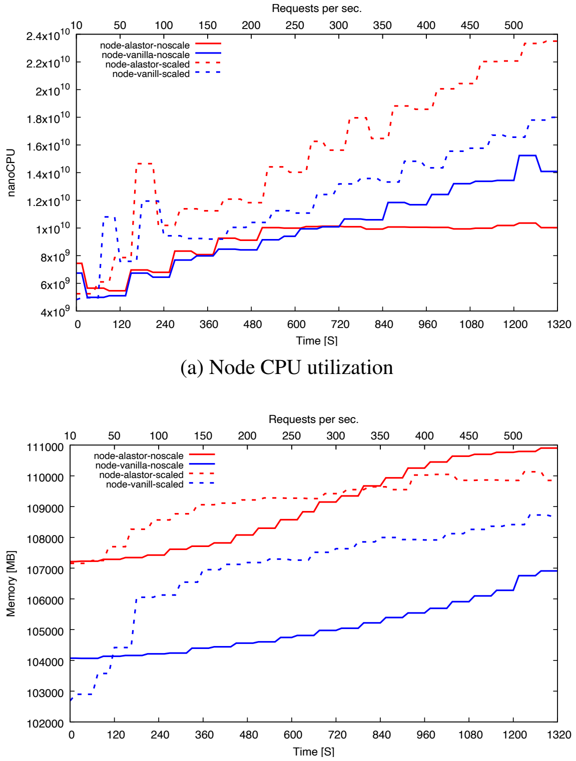


Figure 5: Response latencies of functions with ALASTOR as compared to vanilla OpenFaaS.

ALASTOR incurs modest overheads averaging 13.74%, and across the `falco` application, 6.22%. In the worst case, for the `sent-analysis` function ALASTOR imposed 99.8% (1.2 sec) overhead; we attribute this to the I/O heavy nature of this function, which performs many system calls on a large text blob. These results may indicate that ALASTOR is better-suited to event-driven applications (e.g., *Hello,Retail!*, Falco) that do not perform extensive file I/O.

**CPU and Memory utilization:** Next, we measure the resource utilization overheads of ALASTOR under varying loads.



(a) Node CPU utilization

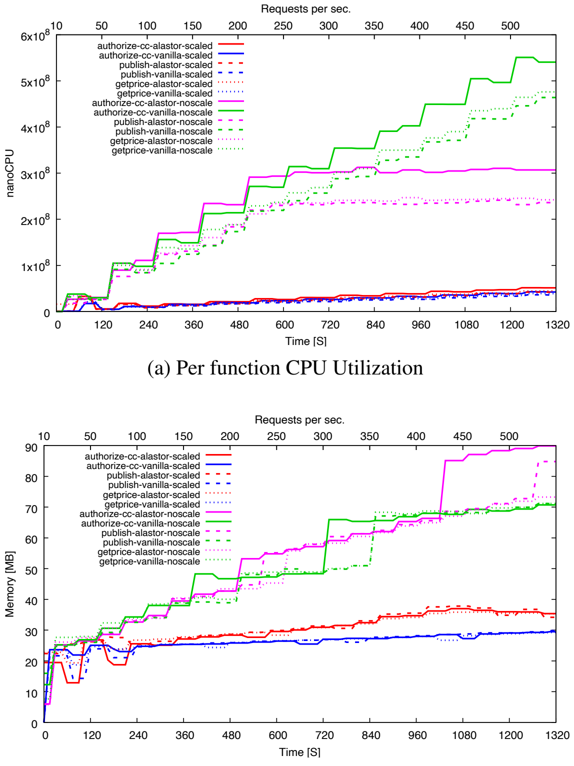
(b) Node memory Utilization

Figure 6: Node CPU and memory utilization executing *Hello,Retail!* Product Purchase workflow with ALASTOR as compared to vanilla OpenFaaS, with and without function scaling enabled.

To do so, we make use of *Hello,Retail!*'s *product-purchase* workflow and use an HTTP load generator (`hey`) to issue increasingly high request loads ranging from 10 to 500 requests per second (rps). Results for each load are reported over 120 second trials. We also repeat this experiment in two cluster configurations: where only a single container per function is used (*noscale*) and where containers can scale up to 50 per function (*scaled*). We measure utilization with Kubernetes Metrics Server,[6] in which 1 nanoCPU is defined as 1 billionth of a cpu-core.

Figure 6 shows our results, with per-function CPU and memory utilization reported in Figure 7. In Figure 6a, the CPU utilization overhead for ALASTOR grows at a gradual constant rate in the *scaled* configuration, with peaks denoting the creation of new containers to cater to the increased request load. In *noscale*, the total CPU utilization is lower because the server's additional cores are not in use. In this case, ALASTOR cannot handle beyond 200 rps and starts queueing and dropping requests. The vanilla setup also struggles and starts dropping requests at this time, although managing to make progress at a much slower rate. In Figure 6b we observe

[6] https://github.com/kubernetes-sigs/metrics-server



(a) Per function CPU Utilization

(b) Per function memory utilization

Figure 7: Per function CPU and memory utilization executing *Hello,Retail!* Product Purchase workflow with ALASTOR as compared to vanilla OpenFaaS, with and without function scaling enabled.

that ALASTOR imposes almost constant memory overhead throughout. For ALASTOR in the *noscale* configuration, the memory footprint surpasses the *scaled* footprint at 300 rps due to increasing queue of unprocessed in-flight requests. While the CPU and memory overheads of ALASTOR are significant, we observe that ALASTOR is able to support at least up to 500 rps on our test server; in fact, we were unable to fully saturate the ALASTOR-enabled set-up using our test apparatus.

In Figure 7 we report the average CPU and memory utilization across all running containers for a function in the scaled configuration. This diagram shows a similar trend to Figure 6. With function scaling enabled (red and blue lines), both the CPU and memory utilization growth for ALASTOR are modest and stable with increasing request load.

**Disk and Network Utilization:** Storing and managing massive logs is a key problem in system auditing. In ALASTOR's case, logs impose overheads related to both network transmission (to the global provenance builder service) and long-term disk storage. Figure 8 reports on the log growth in *Hello,Retail!* for an increasing number of requests to the *product-purchase* workflow under four configurations: the *raw* log, the log after filtering using Hossain et al.'s Source Dependency Preserving Reduction (DPR) system [51], a *tar* compressed version of the log, and finally a log that was first filtered with DPR and then compressed (*DPR+tar*). DPR is a state-of-the-art log reduction technique that selectively deleted log events that are not necessary to correctly identifying an object's ancestors in the graph. It can be observed that all configurations grow linearly with the number of requests; this is because the small and well-formed nature of function behaviors lead to highly deterministic provenance graphs. As a result, the log compression and reduction techniques are highly effective at eliminating the redundancies of typical system execution. Ultimately, we conclude that ALASTOR behaves similarly to other system auditing frameworks – while it produces a huge amount of data in raw form, these costs can be effectively mitigated through a combination of log filtering and data compression techniques. In fact, if we considered a request load of 30 million per month on Amazon AWS, the *DPR+tar* would impose just ~6.57 GB of storage per month. Following the standard tier pricing ($0.023/GB per month) of AWS S3 storage [10], this would lead to a cost of just $0.45 over a 3 month period and $5.44 over a 3 year period.

Closely related to disk utilization is the complexity of ALASTOR's provenance graphs; if the graphs for individual function or workflows are too large, they will be difficult to interpret. Table 1 reports on graph complexity for three end-to-end workflow invocations in *Hello,Retail!*: a benign

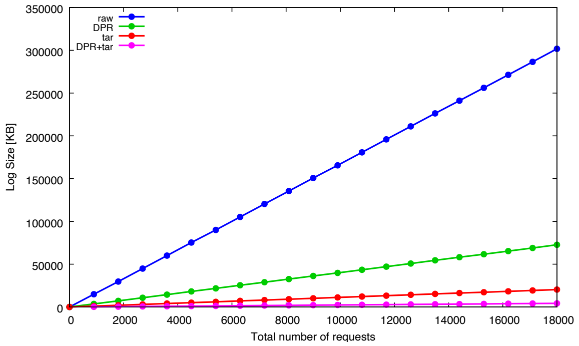


Figure 8: Total size of logs generated while executing *Hello,Retail!* Product Purchase workflow with ALASTOR.

| Workflows | #Node | #Edge | Local graphs | | | |
|---|---|---|---|---|---|---|
| | | | purchase | get-price | authorize-cc | publish |
| Benign | 69 | 80 | N18 E28 | N13 E13 | N21 E14 | N13 E13 |
| Attack1 | 76 | 117 | N19 E36 | N13 E13 | N26 E34 | N13 E13 |
| Attack2 | 49 | 65 | N18 E29 | N13 E13 | - | N13 E13 |

Table 1: Complexity of ALASTOR request graphs in 3 different *Hello,Retail!* scenarios. The second and third column denotes the size of the global provenance graph for a single request and the local graph sizes of the constituent functions are shown in the next columns.

invocation of *product-purchase*, and the two attack scenarios described in Section 8. It can be observed that the Attack1 scenario, which abuses warm container reuse to link multiple workflow invocations together, results in a large graph. Regardless, all scenarios produce succinct graph representations that are orders of magnitude smaller than typical whole-system provenance graphs (e.g., [25]). We attribute this to the ephemeral and event-driven nature of serverless computing; because execution is short-lived and well-defined, the typical problems of graph complexity and dependency explosion [62] do not arise for ALASTOR.

**Total Cost of Operation:** Based on these results, we can calculate the total cost of operating ALASTOR on an Amazon Lambda application based on published cost models [11]. To demonstrate, we consider "*product-purchase-get-price*" and "*sentiment-analysis*," the functions that incurred the highest response overheads. To mirror a request load similar to an example quoted by Amazon, we choose a constant load of 10 rps, which translates to 25.92 million requests per month. Amazon costs are based on monthly request charges and monthly compute charges. Billable requests are charged $0.2 per million and computed as $(25920000 - 1000000)$, resulting in a fee of $4.98 for either function regardless of whether ALASTOR is active. Compute charges, measured in GigaByte-Seconds (GB-S), are calculated using the formula $(num\_requests * exec\_time\_per\_request * memory\_tier/1024 - 400000)$, where Amazon exampts the first 400k GB-S from fees and then charges $0.00001667 per GB-S. The execution time for *get-price* is 0.0081s / 0.0115s with/without ALASTOR, while the time for *sentiment-analysis* is 1.228s / 2.454s. Both functions fit in the lowest memory tier (*mem_tier*) of 128 MB. Because the *get-price* function does not exceed 400k GB-S, the total compute charges are $0 with or without ALASTOR. For *sentiment-analysis*, the compute charges exceed 400k and roughly double when ALASTOR is deployed, jumping from $66.21 to $130.85.[7] From these results, we can conclude that in the typical case the hosting cost of ALASTOR will be proportional to the overhead ALASTOR imposes on execution time. Costs double for *sentiment-analysis* where observed overheads were 99.8%, whereas for *Hello,Retail!* costs will increase roughly by 13.74%.

# 8 Security Analysis

We now demonstrate the security benefits of ALASTOR as compared to commercially-available serverless tools.

## 8.1 Defending *Hello,Retail!*

We begin by considering attacks against *Hello,Retail!* using the setup described in Section 7. We also configure *Hello,Retail!* to make use of Epsagon [13], a commercial serverless monitoring tool. Epsagon monitors a serverless application and generates traces in the form of a graph that encodes function invocations in each end-to-end workflow execution within the application. Unfortunately, we were unable to run Epsagon on OpenFaaS, so we ported our modified version of *Hello,Retail!* back to AWS Lambda and replicated the attack scenarios on both setups. We compare the insights provided by Epsagon traces as compared to ALASTOR's provenance graph in order to answer the following research questions: does the output of each system indicate that the application has deviated from its normal behavior? (**RQ1**); does the output of each system provide an explanation of the attacker's actions within the application? (**RQ2**), and can the output of each system be used to diagnose the serverless-specific attack techniques? (**RQ3**).

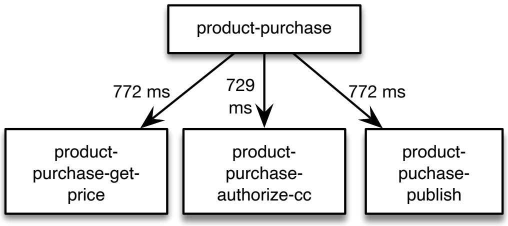


Figure 9: Epsagon provides the above visualization of the malicious workflow trace in *Hello,Retail!*. Rectangles represent function invocations and edges mark invocation requests with response latencies. The visualization is identical regardless of whether the attack behavior is present.

### 8.1.1 Data Exfiltration Attack

We first consider the the attack scenario depicted in Figure 1. Here, an attacker seeks to abuse the commonplace misconfiguration of cloud databases [94] to retrieve customer credit card data, which is stored in the *Credit Card Registry* ($D_4$). In this case study, functions $f_9$ and $f_{12}$ were modified with a backdoor that simulates a remote code execution vulnerability enabling the attacker to download attack tools from an attacker controlled server. The backdoor in $f_9$ is triggered by supplying a "malicious" key in the request body. This key is passed to $f_{12}$ to trigger the phases of the attack. If the key-value pair is not present in the request body, handling of the request proceeds as normal. While this attack trigger is synthetic, it is identical to an actual command injection vulnerability from the perspective of the tracing mechanisms.

In the first phase of the attack, the attacker accesses the publicly accessible *Purchase Product* function $f_9$ (Fig. 1, ①) and then pivots to the *Authorize Credit Card* function $f_{12}$ (②), where they download the attack scripts from an attacker controlled server (③). Once the download completes, the attacker initiates phase two in a follow-up request, once again pivoting to $f_{12}$ through $f_9$, but in this phase the attacker exploits container reuse to execute the attack scripts downloaded in

[7] We confirmed these results with the Lambda Pricing Calculator [12].

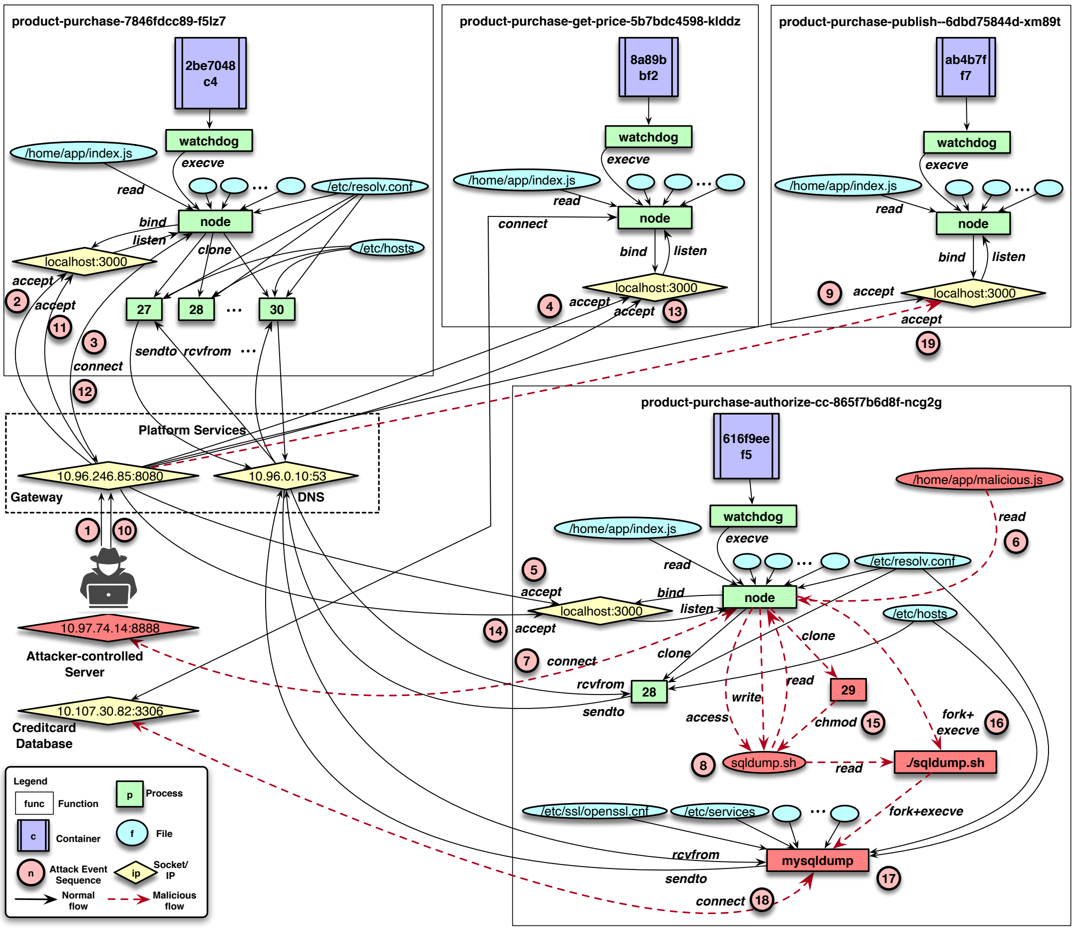


Figure 10: Global provenance graph for product-purchase attack workflow generated by ALASTOR. Some metadata is suppressed from the graph for simplicity.

the previous phase (④). Through executing the previously downloaded attack scripts, the attacker retrieves a data dump of $D_4$ and then sends the sensitive information back to $f_9$ in an HTTP response. Finally, following the normal workflow, $f_9$ forwards the response to $f_{13}$ which in turn sends it back to the attacker or may publish the sensitive information to public domain (⑤). Note that this attack scenario leverages the two key serverless-specific attack strategies, exploiting container reuse and exfiltration through function workflows.

**Attack Investigation with Epsason:** Epsagon creates one graph for each request made to the publicly accessible function *product-purchase* ($f_9$) as shown in Figure 9. Unfortunately, the structure is identical regardless of whether a malicious or legitimate request is issued. However, Epsagon also records additional metadata for each function request and execution that are not present in the trace visualization. In order to provide a fairer comparison between the two tools, we attempt to provide an integrated visualization of all of the relevant trace information captured by Epsagon. Figure 11 demonstrates Epsagon's view of a serverless application following 3 requests: one benign and two malicious requests that constitute the attack. We omit additional irrelevant metadata (headers and body) in order to reduce clutter.

Inspecting Figure 11, it can be seen that Epsagon provides relevant metadata about function usage including environment runtime details, whether the container started executing in cold or warm state, memory utilization, and the duration of the function execution. Using this information, we can observe an unusually large execution time (about two orders of magnitude higher than the benign scenario) at step ⑭ in Figure 11 due to the ongoing retrieval of the entire contents of $D_4$. This marks a deviation from normal behavior, thereby satisfying **RQ1**, but it is a weak threat indicator because the reason for this anomaly is not clear. Without additional supporting evidence, the attack appears to be a performance bug as opposed to a sophisticated intrusion attempt. With regards to **RQ2**, although Epsagon records a variety of function metadata, it does not record connections to additional application components including the credit-card database ($D_4$), nor does it log the connection to the attacker controlled remote server.

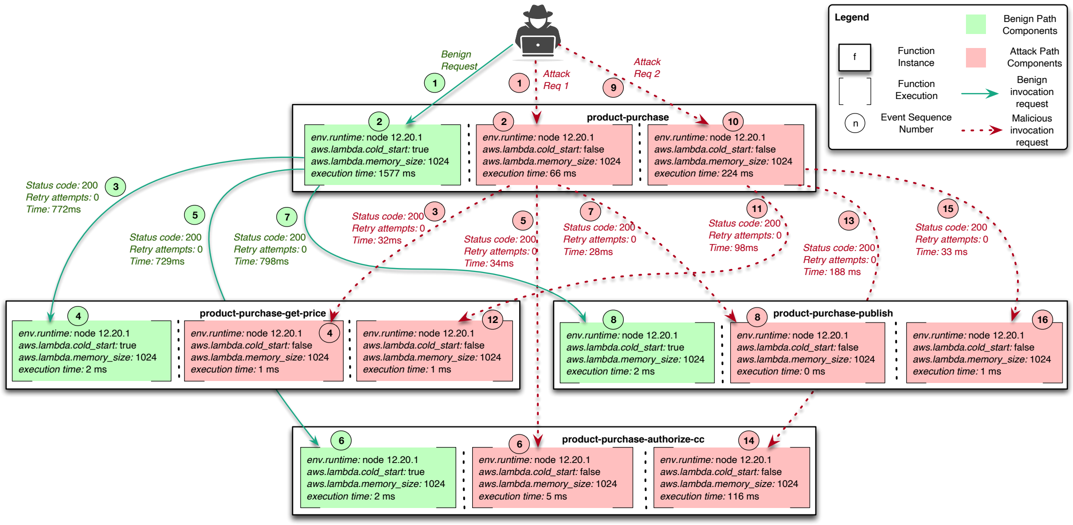

Legend
Benign Path Components
f
Function Instance
Attack Path Components
Function Execution
Benign invocation request
n
Event Sequence Number
Malicious invocation request
Benign Request
1
Attack Req 1
1
Attack Req 2
9
product-purchase
2
env.runtime: node 12.20.1
aws.lambda.cold_start: true
aws.lambda.memory_size: 1024
execution time: 1577 ms
2
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 66 ms
10
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 224 ms
Status code: 200
Retry attempts: 0
Time: 772ms
3
5
Status code: 200
Retry attempts: 0
Time: 729ms
7
Status code: 200
Retry attempts: 0
Time: 798ms
Status code: 200
Retry attempts: 0
Time: 32ms
3
5
Status code: 200
Retry attempts: 0
Time: 34ms
7
Status code: 200
Retry attempts: 0
Time: 28ms
11
Status code: 200
Retry attempts: 0
Time: 98ms
13
Status code: 200
Retry attempts: 0
Time: 188 ms
15
Status code: 200
Retry attempts: 0
Time: 33 ms
product-purchase-get-price
4
env.runtime: node 12.20.1
aws.lambda.cold_start: true
aws.lambda.memory_size: 1024
execution time: 2 ms
4
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 1 ms
12
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 1 ms
product-purchase-publish
8
env.runtime: node 12.20.1
aws.lambda.cold_start: true
aws.lambda.memory_size: 1024
execution time: 2 ms
8
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 0 ms
16
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 1 ms
product-purchase-authorize-cc
6
env.runtime: node 12.20.1
aws.lambda.cold_start: true
aws.lambda.memory_size: 1024
execution time: 2 ms
6
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 5 ms
14
env.runtime: node 12.20.1
aws.lambda.cold_start: false
aws.lambda.memory_size: 1024
execution time: 116 ms

Figure 11: We consolidate all attack traces and additional metadata captured by Epsagon in this diagram. For the purpose of clear understanding of the flow, we used green color for normal flow path components and red for the attack path components.

Thus, Epsagon is unable to reconstruct the complete attack path. Finally, in consideration of **RQ3**, Epsagon does track one relevant attribute, *cold_start*, which indicates container reuse in steps ②,④,⑥, ⑧,⑩,⑫,⑭ and ⑯. However, container reuse alone is not an indicator of compromise; it is in fact a common occurrence in any application. Epsagon is unable to detect *data sharing* between repeated uses of the same function instance, which is more suspicious. Therefore, Epsagon traces do not enable serverless attack detection.

**Attack Investigation with ALASTOR:** The global provenance graph generated by ALASTOR is shown in Figure 10. To simplify the provenance graph, we omit some metadata (e.g., timestamps, duration of operations, request and response headers and body) from the graph and use event sequence numbers to show the events chronologically. Furthermore, we only show the events related to the attack requests (① and ⑩). Red dotted lines and red colored components are used where attack path deviates from normal execution ( ***RQ1***).

The attacker makes their first request (①) to *product-purchase* through the OpenFaaS gateway and this request is accepted within the function instance (②). After processing the incoming request, *product-purchase* invokes *product-purchase-get-price* through the gateway (③-④). This function's responsibility is to retrieve the price of a product from the Product Database (i.e., 10.107.30.82 : 3306). Next, *product-purchase* makes another request to *product-purchase-authorize-cc* (⑤) to authorize the credit card transaction. However, this function has a backdoor embedded (i.e.,`/home/app/malicious.js`) which is triggered by the "malicious" key in the request body (⑥). Alongside the normal operations of the function, this backdoor vulnerability also opens a connection to the attacker controlled server (⑦) and downloads a malicious script `sqldump.sh` (⑧). In the next step, *product-purchase* invokes *product-purchase-publish* to publish the status of the transaction in public domain (⑨) and also returns this in a response message to the attacker.

Now the attacker begins the next phase of the attack by making another request to *product-purchase* (⑩-⑪) and after retracing the steps in first phase (⑫-⑭), we arrive to the same *product-purchase-authorize-cc* instance. Since the attacker makes the two attack requests in quick succession, the warm container reuse policy in serverless platforms ensures that the second request is scheduled to the same container and the downloaded attack tools (i.e.,`sqldump.sh`) are available for use to the backdoor process. Next, the permissions of the attack script is modified (⑮) and the script is run (⑯) to fork a new `mysqldump` process (⑰). This process connects to the Creditcard Database (⑱) and retrieves a database dump. The attacker may continue sending multiple spurious requests to the gateway to keep the container warm and to allow the `mysqldump` process finish the data retrieval. The attacker can exfiltrate the data either by sending it back through a response message or by publishing it to a public domain using *product-purchase-publish* (⑲). As demonstrated in Figure 10, ALASTOR sheds light onto the specific actions of the attacker in this intrusion (***RQ2***). It also diagnoses the serverless-specific techniques (***RQ3***) of cross-invocation dependencies (`sqldump` written in ⑤, executed in ⑭) and exfiltration over legitimate flows (data theft at ⑱ is routed to the public domain via request to *product-purchase-publish* at ⑲).

### 8.1.2 Business Logic Manipulation Detection

We briefly discuss an additional attack scenario for *Hello,Retail!* entailing *business logic manipulation*. When applications are developed in serverless architectures, an ap-

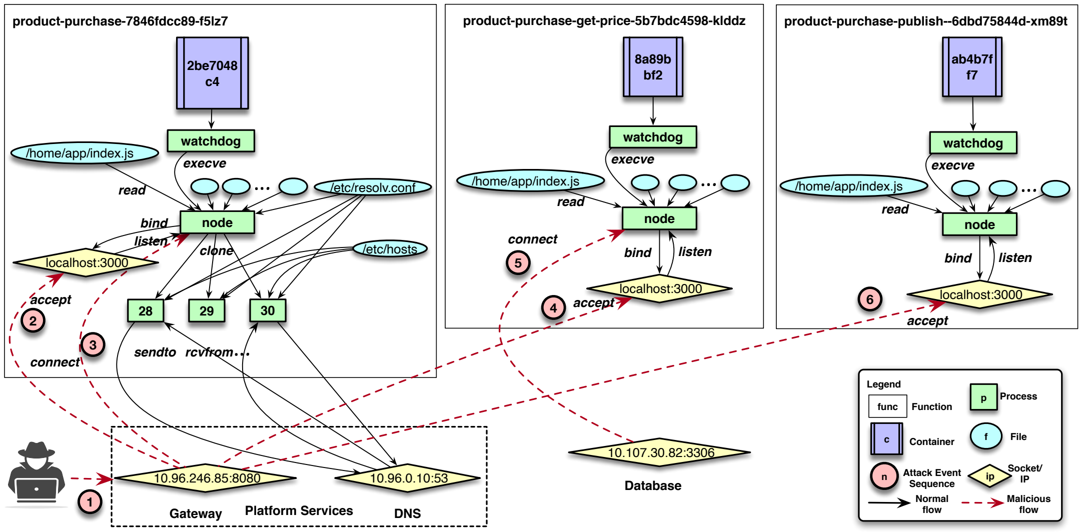


Figure 12: Global provenance graph for a business logic manipulation attack workflow.

plication's business logic is broken across many single purpose functions that interact with each other to accomplish a task. As a result, the order in which these functions execute is paramount to the correct execution of a workflow, and changes in the flow-order can lead to severe consequences, such as broken authentication. We implemented this scenario in *Hello,Retail!* where the workflow-path is redirected based on a malicious field embedded in the incoming request. This synthetic vulnerability bypasses the product-purchase-authorize-cc function in the product-purchase workflow to successfully make an unauthorized purchase. The resulting provenance graph captured by ALASTOR can be found in Figure 12. In this graph, it can be seen clearly that the attacker was able to manipulate the function workflow to bypass the *product-purchase-authorize-cc* function, underscoring ALASTOR's general usefulness in threat investigation. Because the comparative benefits between ALASTOR and Epsagon do not change, we omit the results for Epsagon.

### 8.2 Generality of ALASTOR Auditing

We consider two additional applications, `falco` and `sent-analysis`, discussed in detail in Section 7. `falco`'s pod-deletion workflow consists of two functions – `dispatch` ingest alerts from a threat detection engine, and `delete-pod-fn` deletes a malicious pod for critical alerts. One possible scenario entails an attacker employing logic manipulation to bypass `dispatch` and launch a DoS attack with `delete-pod-fn`. However, as we show for *Hello,Retail!*, ALASTOR could effectively trace this attack. For `sent-analysis`, one possible scenario is for an attacker to attempt to exfiltrate sensitive training data from the function, which would require system calls and network requests that are also visible to ALASTOR. As a final measure of the generality of our approach, Figure 13 reports on the observed system call distribution of the benign behaviors of our workload applications from our experiments. The ubiquity of syscall-based data processing and API-based network communication assures that ALASTOR is well-positioned to explain any suspicious activity.

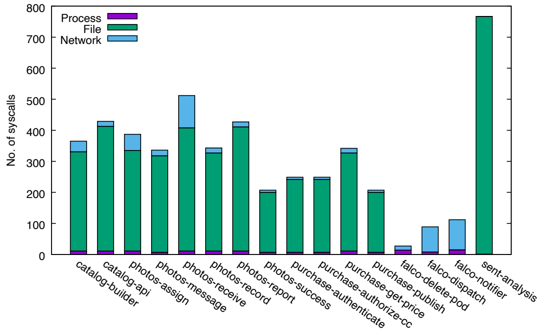


Figure 13: System call distributions (by category) of functions traced with ALASTOR.

### 8.3 Intrusion Detection with ALASTOR

Like most log data, ALASTOR can also facilitate security analytics. To demonstrate, we trained an implementation[8] of Du et al.'s DeepLog system [35], a deep learning based anomaly detection system, on ALASTOR traces. We chose DeepLog because it operates on unstructured, free-text log entries, providing an easy way to measure the discriminatory capabilities of ALASTOR's telemetry data. We use an HTTP

[8] https://github.com/nailo2c/deeplog

|  | P | N |
|---|---|---|
| P | 140 | 10 |
| N | 5 | 112 |

(a) Confusion Matrix

| Metric | Values |
|---|---|
| Accuracy | 0.944 |
| Precision | 0.966 |
| Recall | 0.933 |
| F1 | 0.949 |

(b) metrics

Figure 14: DeepLog performance on ALASTOR traces.

workload generator *'hey'*[9] to generate benign and attack traces of *Hello,Retail!* activity from the *product purchase* workflow on OpenFaaS. The training dataset consisted of benign requests issued at 10 requests per second for 180 seconds. Test data was generated by issuing benign or malicious requests at 10 requests per second for 30 seconds. The malicious requests represent the Data Exfiltration attack scenario discussed in Section 8.1.1 (Figure 10). We use the Spell [34] log parser to preprocess the collected logs following the steps described in [35]. We then split the preprocessed logs into sequences of 200 millisecond duration. Our test data contained 150 attack sequences and 117 benign sequences. Default parameters were used for the model, e.g., we trained for 35 epochs.

Our results are summarized in Figure 14. False negative sequences may result from few events in the complete attack timeline that is in common with benign traffic. Similarly, false positives may be a result of infrequent benign activity related to log transmission or container health monitoring and maintenance by the orchestration platform. Encouragingly, we observe an F1 score of 0.949, providing promising preliminary evidence that ALASTOR telemetry would also prove useful as a general purpose information source for serverless intrusion detection. While we were not able to extract sufficient log output from Epsagon to compare these results, it is very likely that Epsagon would struggle to detect the attack due to miniscule amount of attack evidence (i.e., longer execution time) contained in the coarse-grained trace. In contrast, ALASTOR traces contain fine-grained event sequences and document significant structural changes in application behavior during the attack.

## 9 Related work

**Serverless Auditing and Tracing:** Limited visibility into a serverless application is a recurring problem in modern cloud computing, leading to various tools ranging from performance monitoring and analysis solutions [3, 6, 31, 69, 70] to tracing and observability tools [13, 84, 85, 90]. However, these solutions collect metrics and perform tracing only at a macro level, and are insufficient to provide a fine-grained complete picture of the lower level system events within a serverless application, essential for attack investigation. Existing distributed system tracing tools, such as Dtrace [27], Dapper [93], X-trace [39], MagPie [24], Fay [37], and Pivot-Tracing [66] are not designed to account for aspects of the serverless architecture and require instrumentation of the application. Conversely, lprof [103] and Stitch [102] are able to profile distributed applications without instrumentation; however, lprof requires static binary analysis, and Stitch requires structured application log messages with object identifiers for tracing. Moreover, these tools only capture correlations rather than causality and thus are not appropriate for serverless forensics. Hong et al. [50] proposed a conceptual design of a threat intelligence system for serverless frameworks. However, it is based on only application level logs and is unsuitable for root cause analysis in forensics investigation. To our knowledge, ALASTOR is the first causality analysis framework to satisfy the operational requirements of serverless platforms.

**Serverless Security:** Prior work has demonstrated unique attack opportunities in serverless platforms, including event injection attacks [29, 56, 78] and data exfiltration [58, 61]. Further, serverless access control misconfigurations enable attackers to steal sensitive information [41, 74] and launch denial-of-service (or *denial-of-wallet*) attacks by exhausting allocated resource limits and increasing bills [8, 101]. Baldini et al. [23] concluded that the lack of function isolation is a major problem in popular cloud platforms. Wang et al. [98] performed a range of studies on metrics like scalability, cold-start latency, and instance lifetime and they reported arbitrary code execution bugs in Azure Functions making the platform vulnerable to side-channel attacks.

In face of these attacks, various tools have been proposed to improve function security before deployment and during runtime [21, 82, 83, 86–89, 91]. Prior work has also proposed improved isolation using Intel SGX to build secure containers [22] and secure cloud functions [18, 26, 79]. Beyond isolation, prior work has considered flow control (e.g., Trapeze [19], Valve [33], SecLambda [55]) and other access control models (e.g., Will.IAM [81]) to thwart serverless attacks by denying suspicious access requests according to strict flow and access control policies. Orchestration frameworks have also been enhanced [28] with security policy support for serverless applications. These security tools are proactive precautions, while ALASTOR strengthens security posture through improved threat response. Formal modeling of serverless platforms [42, 54], and semi-automated troubleshooting based on log data [67] may also improve serverless security.

**System Auditing in Cloud Environments:** It is especially interesting to compare our system to Chen et al.'s CLARION [30], a technique for precise auditing of container clusters. While both systems can be used to audit serverless functions running in containerized environments, the challenges addressed by these systems are quite different. CLARION is a host-based agent that provides a solution for correctly auditing multiple namespaced environments without ambiguity,

[9] https://github.com/rakyll/hey

while ALASTOR is a container-based agent and thus does not need to disambiguate namespaces. Although is could be used to audit serverless applications, CLARION does not produce separate graphs for multiple requests to the same function instance and also would struggle to trace workflows across multiple container servers in the cloud. In effect, ALASTOR leverages the unique constraints of the FaaS environment – ephemeral computation, remote procedure calls, and (mostly) isolated container environments – to produce an optimized solution to serverless auditing.

## 10 Discussion & Conclusion

In this work, we have demonstrated methods of layered provenance-based auditing of serverless infrastructure that enable precision tracing of attacks on serverless applications. We now conclude by considering potential limitations of ALASTOR as well as opportunities for future work.

**Time Synchronization of Global Graph:** Time synchronization among different functions is resolved by preserving the ordering of rest API calls. While miniscule time drift may be possible between two system calls recorded in different function instances, preserving the happens-before relationships between API requests ensures that the causal links between functions' system graphs are correct.

**Expensive I/O bound applications:** Serverless functions normally work on small pieces of data with execution times on the order of milliseconds. If an application included long-running I/O-intensive functions, it may be necessary to modify ALASTOR to reduce the granularity of I/O logging.

**Applicability of ALASTOR to other Serverless Platforms:** Systems like Cloudflare, Fastly, and Faasm that provide per-request isolation can benefit from ALASTOR as a general purpose auditing tool. Faasm [92] enables memory isolation of executed functions using WebAssembly, while permitting memory sharing among functions to enable efficient data processing. In Faasm, ALASTOR could be used to investigate possible disclosures of sensitive data (e.g., medical data). It may also be possible to extend ALASTOR to audit access to memory regions through introspection of memory system calls. Moreover, ALASTOR logs can be ingested by other monitoring tools for threat detection and other analyses.

**Threat Model:** ALASTOR's design assumes that an attacker can compromise a function instance but has limited administrative access to the victim's account. This assumption is reasonable if application RBAC policies are properly configured. However, an attacker that fully compromises the container may be able to prevent future logs from reaching the global provenance builder service, or could erase log entries. One possible response to such attacks would be to extend ALASTOR to generate cryptographic log commitments to detect log integrity violations (e.g., [75]) and increase the frequency of log transmission to bound the amount of unprotected log accessible to the attacker.

## Acknowledgements

This work was supported in part by NSF CNS 17-50024, NSF CNS 19-55228 and NSF CNS 20-55127. The views expressed are those of the authors only.

## References

[1] Linux audit daemon. https://linux.die.net/man/8/auditd.

[2] AWS Serverless Application Repository. https://aws.amazon.com/serverless/serverlessrepo/, 2019.

[3] AWS X-Ray: Analyze and debug production, distributed applications. https://aws.amazon.com/xray/, 2019.

[4] CloudWatch Logs Limits. https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/cloudwatch_limits_cwl.html, 2019.

[5] CVE-2019-5736: runc container breakout. https://www.openwall.com/lists/oss-security/2019/02/11/2, 2019.

[6] Google Cloud: Viewing Monitored Metrics. https://cloud.google.com/functions/docs/monitoring/metrics, 2019.

[7] MITRE ATT&CK. https://attack.mitre.org, 2019.

[8] New Attack Vector - Serverless Crypto Mining. https://www.puresec.io/blog/new-attack-vector-serverless-crypto-mining, 2019.

[9] ReDoS Vulnerability in "AWS-Lambda-Multipart-Parser" Node Package. https://www.puresec.io/blog/redos-vulnerability-in-aws-lambda-multipart-parser-node-package, 2019.

[10] AWS s3 pricing. https://aws.amazon.com/s3/pricing/, 2021.

[11] Aws lambda pricing. https://aws.amazon.com/lambda/pricing/, 2021.

[12] Aws lambda pricing calculator. https://s3.amazonaws.com/lambda-tools/pricing-calculator.html, 2021.

[13] Epsagon: Monitoring and troubleshooting for serverless applications. https://epsagon.com/, 2021.

[14] Kubernetes response engine powered by openfaas. https://github.com/developer-guy/falco-the-kubernetes-response-engine-using-openfaas-functions, 2021.

[15] Sentiment analysis. https://github.com/openfaas/faas/tree/master/sample-functions/SentimentAnalysis, 2021.

[16] G. Adzic and R. Chatley. Serverless computing: Economic and architectural impact. In *the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[17] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference (ATC)*, 2018.

[18] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW)*, 2019.

[19] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein. Secure serverless computing using dynamic information flow control. In *2018 ACM SIGPLAN Conference on Prog. Lang. (OOPSLA)*, 2018.

[20] Amazon AWS Lambda. AWS Lambda Customer Case Studies. https://aws.amazon.com/lambda/resources/customer-case-studies/, 2020.

[21] Aqua. Aqua Cloud Native Security Platform. https://www.aquasec.com/products/aqua-container-security-platform/, 2019.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[23] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore. 2017.

[24] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[25] A. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security Symposium*, 2015.

[26] S. Brenner and R. Kapitza. Trust more, serverless. In *ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[27] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference (ATC)*, 2004.

[28] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, et al. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 2020.

[29] Check Point Software. A Deep Dive into Serverless Attacks, SLS-1: Event Injection. https://www.protego.io/a-deep-dive-into-serverless-attacks-sls-1-event-injection/, 2019.

[30] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran. CLARION: Sound and clear provenance tracking for microservice deployments. In *USENIX Security Symposium*, 2021.

[31] R. Cordingly, H. Yu, V. Hoang, Z. Sadeghi, D. Foster, D. Perez, R. Hatchett, and W. Lloyd. The serverless application analytics framework: Enabling design trade-off evaluation for serverless software. In *International Workshop on Serverless Computing (WoSC)*, 2020.

[32] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors. mitmproxy 4.0: A free and open source interactive HTTPS proxy, 2010.

[33] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, , and A. Bates. Valve: Securing function workflows on serverless computing platforms. In *The Web Conference (WWW)*, 2020.

[34] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *IEEE International Conference on Data Mining (ICDM)*, 2016.

[35] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[36] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. The Security Impact of HTTPS Interception. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[37] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 2012.

[38] Falco. The falco project. https://falco.org/, 2021.

[39] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *USENIX conference on Networked Systems Design & Implementation (NSDI)*, 2007.

[40] Francois Lagier. Serverless: The Ideal Choice For Startups? (CloudForecast Case Study). https://www.serverless.com/blog/serverless-for-startups, 2020.

[41] Frederik Willaert. AWS Lambda Container Lifetime and Config Refresh. https://www.linkedin.com/pulse/aws-lambda-container-lifetime-config-refresh-frederik-willaert/, 2019.

[42] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro. No more, no less. In *Coordination Models and Languages*, Springer International, 2019.

[43] Gathering weak npm credentials. https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md, 2019.

[44] Github. serverless-image-processor. https://github.com/Mercateo/serverless-image-processor, 2019.

[45] The Amazon API Gateway Serverless Developer Portal. https://github.com/awslabs/aws-api-gateway-developer-portal, 2019.

[46] Google Cloud. Serverless Pixel Tracking Architecture. https://cloud.google.com/solutions/serverless-pixel-tracking, 2019.

[47] Guy Podjarny. Securing Serverless – by Breaking in. https://www.infoq.com/presentations/serverless-security-2018, 2020.

[48] W. U. Hassan, M. Noureddine, P. Datta, and A. Bates. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2020.

[49] Hillel Sollow. Top 4 Reasons Why Serverless Is Secure. https://blog.checkpoint.com/2020/07/13/top-4-reasons-why-serverless-is-secure/, 2020.

[50] S. Hong, A. Srivastava, W. Shambrook, and T. Dumitras. Go serverless: Securing cloud via serverless design patterns. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2018.

[51] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security Symposium*, 2018.

[52] Jack Danahy. Capital One Breach Highlights. https://blog.alertlogic.com/capital-one-breach-highlights-importance-of-constant-vigilance/, 2019.

[53] Jacopo Tagliabue. (Server)less is more. https://medium.com/tooso/server-less-is-more-98d4275c37ae, 2019.

[54] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. In *ACM Program. Lang. (OOPSLA)*, 2019.

[55] D. S. Jegan, L. Wang, S. Bhagat, T. Ristenpart, and M. Swift. Guarding serverless applications with seclambda. *arXiv preprint arXiv:2011.05322*, 2020.

[56] Jeremy Daly. Event Injection: Protecting your Serverless Applications. https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/, 2020.

[57] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[58] R. Jones. Gone in 60 Milliseconds: Intrusion and Exfiltration in Server-less Architectures. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds, 2019.

[59] V. Karande, E. Bauman, Z. Lin, and L. Khan. SGX-Log: Securing System Logs With SGX. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[60] S. T. King and P. M. Chen. Backtracking intrusions. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[61] A. Krug and G. Jones. Hacking serverless runtimes: Profiling AWS Lambda, Azure Functions, And more. https://www.blackhat.com/us-17/briefings/schedule/#hacking-serverless-runtimes-profiling-aws-lambda-azure-functions-and-more-6434, 2019.

[62] K. H. Lee, X. Zhang, and D. Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[63] K. H. Lee, X. Zhang, and D. Xu. LogGC: Garbage Collecting Audit Log. In *ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2013.

[64] P. Leitner, E. Wittern, J. Spillner, and W. Hummer. A mixed-method empirical study of function-as-a-service software development in industrial practice. *Journal of Systems and Software*, 2019.

[65] 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 *USENIX Security Symposium*, 2017.

[66] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[67] J. Manner, S. Kolb, and G. Wirtz. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems*, 2019.

[68] Microsoft. Windows Event tracing. https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.

[69] Microsoft. Azure application insights. https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview, 2019.

[70] Microsoft. Azure Monitor: Full observability into your applications, infrastructure, and network. https://azure.microsoft.com/en-us/services/monitor/, 2019.

[71] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan. Holmes: Real-time apt detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy*, 2019.

[72] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX Annual Technical Conference (ATC)*, 2018.

[73] OpenFaaS. OpenFaaS Function Store. https://github.com/openfaas/store, 2019.

[74] Ory Segal. Securing Serverless: Attacking an AWS Account via a Lambda Function. https://www.darkreading.com/cloud/securing-serverless-attacking-an-aws-account-via-a-lambda-function/a/d-id/1333047, 2019.

[75] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2020.

[76] R. Paccagnella, K. Liao, D. Tian, and A. Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[77] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[78] PureSec. Hacking a Serverless Application: Demo. https://www.youtube.com/watch?v=TcN7wHuroVw, 2019.

[79] W. Qiang, Z. Dong, and H. Jin. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. *Security and Privacy in Communication Networks*, Springer International Publishing, 2018.

[80] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009

[81] A. Sankaran, P. Datta, and A. Bates. Workflow integration alleviates identity and access management in serverless computing. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[82] FunctionShield. https://www.puresec.io/function-shield, 2019.

[83] Intrinsic: Software security, re-invented. https://intrinsic.com/, 2019.

[84] IO|pipe. https://www.iopipe.com/, 2019.

[85] Monitor serverless applications. https://dashbird.io/, 2019.

[86] Protego Serverless Runtime Security. https://www.protego.io/platform/elastic-defense/, 2019.

[87] Serverless Security for AWS Lambda, Azure Functions, and Google Cloud Functions. https://www.twistlock.com/solutions/serverless-security-aws-lambda-azure-google-cloud/, 2019.

[88] Snyk. https://snyk.io/, 2019.

[89] Sysdig Secure. https://sysdig.com/products/secure/, 2019.

[90] Thundra: Quickly pinpoint problems in serverless. https://www.thundra.io/, 2019.

[91] Vandium-node. https://github.com/vandium-io/vandium-node, 2019.

[92] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference (ATC)*, 2020.

[93] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Google, Inc, 2010.

[94] Suresh Kasinathan. You're One Misconfiguration Away from a Cloud-Based Data Breach). https://www.darkreading.com/cloud/youre-one-misconfiguration-away-from-a-cloud-based-data-breach/a/d-id/1337464, 2020.

[95] SUSE LINUXAG. Linux Audit-Subsystem Design Documentation for Linux Kernel 2.6, v0.1. Available at http://uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf, 2004.

[96] OWASP Serverless Top 10. https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project, 2019.

[97] T. Wagner. Understanding Container Reuse. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2019.

[98] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *USENIX Annual Technical Conference (ATC)*, 2018.

[99] Xavier Bruhiere. Lambda functions for rapid prototyping. https://developer.ibm.com/articles/cl-lambda-functions-rapid-prototyping/, 2019.

[100] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[101] Yan Cui. Many-faced threats to Serverless security. https://hackernoon.com/many-faced-threats-to-serverless-security-519e94d19dba, 2021.

[102] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[103] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.