Computing (2021) 103:1025–1056
https://doi.org/10.1007/s00607-021-00924-y

SURVEY ARTICLE

Secure FaaS orchestration in the fog: how far are we?

Alessandro Bocci[1] · **Stefano Forti**[1] · **Gian-Luigi Ferrari**[1] · **Antonio Brogi**[1]

Received: 4 November 2020 / Accepted: 11 February 2021 / Published online: 9 March 2021

Abstract
Function-as-a-Service (FaaS) allows developers to define, orchestrate and run modular event-based pieces of code on virtualised resources, without the burden of managing the underlying infrastructure nor the life-cycle of such pieces of code. Indeed, FaaS providers offer resource auto-provisioning, auto-scaling and pay-per-use billing at no costs for idle time. This makes it easy to scale running code and it represents an effective and increasingly adopted way to deliver software. This article aims at offering an overview of the existing literature in the field of next-gen FaaS from three different perspectives: (i) the *definition* of FaaS orchestrations, (ii) the *execution* of FaaS orchestrations in Fog computing environments, and (iii) the *security* of FaaS orchestrations. Our analysis identify trends and gaps in the literature, paving the way to further research on securing FaaS orchestrations in Fog computing landscapes.

Keywords FaaS · Serverless · Fog Computing · Security · Cloud computing

Mathematics Subject Classification 68-02 · 68N19 · 68M14

1 Introduction

Function-as-a-Service (FaaS) is evolving microservice-based software architectures into function-based ones [1,2], as an instance of the *serverless* paradigm in Cloud settings [3]. Indeed, FaaS applications can be obtained by suitably orchestrating stateless, event-triggered functions running on virtualised Cloud infrastructures, with no need for programmers to set up nor to directly operate the deployment stack. One one hand, from a programmer's perspective, the FaaS paradigm permits focussing only on the *business logic* of an application implemented by composing functions as simple, well defined building blocks and by *decoupling* the writing of application

✉ Alessandro Bocci
alessandro.bocci@phd.unipi.it

[1] Department of Computer Science, University of Pisa, Pisa, Italy

code from the (automated) *management* of the deployment servers. On the other hand, form a provider's perspective, the FaaS paradigm requires more fine-grained resource allocation strategies to enable on-demand and faster provisioning and scaling, based on containers, as well as to feature pay-as-you-go billing for each running function instance.

Amazon launched serverless computing with AWS Lambda in 2015 [4], primarily intended for Cloud computing settings. Meanwhile, *Fog computing* [5,6] emerged to exploit processing, storage and networking resources along the Cloud-IoT continuum—from personal devices, through network switches, to Cloud datacentres. The main goal of the Fog is to support the stringent Quality of Service (QoS) requirements of next-gen IoT applications as well as to improve the Quality of Experience (QoE) of the existing ones [7–9]. Indeed, the Fog permits processing—where it is best-suited along the Cloud-IoT continuum—the huge amounts of data that the Internet of Things (IoT) is producing daily so to enact prompter responses to sensed events and to improve contextual data insights.

The idea of applying the serverless paradigm in the Fog looks appealing and promising [10]. Event-based serverless functions are naturally suited to define computation on IoT data, containerisation technologies are lightweight and can run on most edge devices, and on-demand execution of functions can lead to improved usage of resource-constrained devices, closer to the edge of the Internet [11–13]. However, the adoption of the FaaS paradigm in Fog scenarios poses the challenge of preserving and enforcing security constraints. Indeed, the Fog comes along with new distinctive security threats to be faced [14]. Indeed, Fog infrastructures will have to deal with the many issues related to the physical vulnerability of accessible edge devices, reducing the available Trust Computing Base (TCB), viz. the set of hardware and software which can be considered trustable in a system. Factually, edge devices could be easily hacked, broken or even stolen by malicious users and can only offer a limited set of security countermeasures [15]. Moreover, serverless platforms should provide isolation among users and accurate accounting for billing purposes [1], which might be non-trivial to ensure in highly pervasive Fog infrastructures with hundreds of nodes and, possibly, of service providers.

In this context, *how to realise secure FaaS orchestration in Fog computing* is still a largely open research problem, being security monitoring and enforcement more complex to achieve than in traditional Cloud settings due to the aforementioned characteristics of the FaaS and Fog paradigm [16].

The goal of this survey is to offer an overview of the existing literature, so to identify open challenges to tackle the above research problem. To accomplish this objective, we organised the review of existing literature and platforms under the following three main perspectives:

P1. languages, models and methodologies to *define FaaS orchestrations*,
P2. platforms, techniques and methodologies to *execute FaaS orchestrations in the Fog*, and
P3. techniques and methodologies to *secure FaaS orchestrations* both statically and at runtime.

The rest of this article is organised as follows. After describing some commercial and open source FaaS platforms to outline their main characteristics under P1–P3 (Sect. 2), we describe the methodology followed to realise our survey and to define the *corpus* of the reviewed scientific literature (Sect. 3). Then, we analyse it under the aforementioned perspectives P1 (Sect. 4), P2 (Sect. 5) and P3 (Sect. 6). Finally, by identifying research gaps at the intersections of those perspectives, we point to some specific open problems and future research challenges for securing the execution of FaaS orchestrations in the Fog (Sect. 7).

2 FaaS platforms

In this section, we overview the main features of commercial and open source FaaS platforms which are currently used in the software industry to implement serverless systems. To better capture and depict the current FaaS world, it is worth mentioning and briefly illustrating the existing platforms being the ones where FaaS has been initially started, developed and put in production environments. For such reason, we decided to review some of the most commonly used FaaS platforms in this Section, while focussing the rest of the article on the state of the art in the research literature. The selection was performed starting from the most prominent FaaS platforms, and refined by relying upon existing surveys[1] comparing commercial and open source FaaS platforms, and finally by using web search engines and GitHub. This section gives an overview of existing FaaS platforms recapitulating on their main characteristics and the trends that are ongoing in the serverless settings, already considering the three perspectives analysed later on this survey, viz. function orchestration definition, executing orchestration in the Fog and security of FaaS orchestration.

AWS Lambda [4]—It is the FaaS platform offered by Amazon. Being integrated with the Amazon Web Services (AWS) suite, it permits to natively run functions—called *Lambda functions*—written in a set of commonly used programming languages (viz. Java, Go, PowerShell, Node.js, C♯, Python and Ruby). Besides, it permits defining specific triggers to launch functions and to automatically manage the allocated resources to execute. It features pay-per-use billing, with precision at the level of hundreds of milliseconds.
Workflow-based orchestrations of distributed applications can be specified with AWS Step Functions [21], that allows designing workflows by defining a set of states and guarded transactions between them. States can be tasks or language constructs (viz. if-then-else, parallel execution and maps) that modify the execution flow. Tasks represent single units of work and they can be Lambda functions, AWS services or activities. Activities are worker services implemented and hosted by the users and featuring an AWS resource address obtainable through AWS console, SDK or API. Workflows have an initial state and a final state, and every state must declare its successor, and whether it represents a successful or failed execution in case it is an end state. Asynchronous tasks are not supported natively. By

[1] For more details and comparisons of existing FaaS platforms, we refer readers to the surveys by Scheuner and Leitner [17], Lopez et al. [18], Wang et al. [19] and Yussupov et al. [20].

default, a failure in a task causes the failure of the whole workflow execution. It is possible to specify a retry policy for a state to manage faulty executions. Standard and express workflows are considered. Standard workflows can be used for long-running, durable, and auditable workflows, while express workflows are suitable for high-volume, event-processing workloads. The given workflow language is available in a visual graph-based version as well as in JSON format.

Moreover, Amazon permits running Lambda functions on Fog nodes using AWS IoT Greengrass [22]. Greengrass extends Cloud capabilities to local edge devices that connect to IoT devices. It is possible to deploy a Lambda function on such local devices. User configurations enable setting up memory limits for running functions and to selecting the lifecycle type of such functions. Similarly to AWS Step Functions workflows, two types of lifecycles are offered for GreenGrass functions: *on-demand* for short-lived functions that are stopped after the execution, and *long-lived* functions that run continuously.

From a security perspective, Amazon promotes a *shared responsibility model*, dividing the security *of* the Cloud and *in* the Cloud. Security *of* the Cloud is Amazon's responsibility and it comprehends the infrastructure protection, in terms of hardware security and the layer of software that concerns storage, network and computation. Security *in* the Cloud is the client's responsibility and it comprehends sensitivity of data, company's requirements, and applicable laws and regulations. To assist clients in performing identity and access management, Amazon supplies an identity and access management service that helps an administrator in securely controlling access to AWS resources.

Azure functions [23]—It is the FaaS platform by Microsoft, featuring a billing strategy analogous to the one of AWS Lambda and integrating with the Azure Cloud services. Azure Functions supports various commonly used programming languages (viz. C♯, Javascript, F♯, Java, PowerShell, Python, Typescript). The use of triggers to execute functions is similar to the one of AWS Lambda. Differently from AWS, Azure Functions feature pay-per-use billing with a precision of seconds and allows monthly subscriptions.

Durable Functions [24] enables writing stateful functions and, in particular, *orchestrator* functions. Orchestrator functions can be used to compose stateless and stateful functions. The Durable Functions platform is written in C♯ and it features the same expressive power of the Microsoft flagship language. An orchestrator function inputs a *context*, that is an object used to call the orchestrated functions. The name of the function called and its arguments are passed to the context that will return the final result. The functions orchestrated are called synchronously or asynchronously, with the possibility to wait for the end of parallel executions by setting up suitable barriers. From an orchestrator function, it is possible to call other orchestrator functions as well. Under the hood, Durable Function extensively relies upon message passing through asynchronous queues.

Microsoft enables deploying Azure Functions on Fog nodes by using Azure IoT Edge [25]. Azure IoT Edge is a collection of services which enables extending Azure Cloud resources with edge nodes and IoT devices, creating a Cloud-IoT computing continuum. Deploying Azure Functions to edge devices requires functions to be containerised using Docker, having their images published in the Azure

registry. Azure Functions that meet these requirements can be then deployed to available Fog nodes directly through the Azure portal. The monitoring and the results of a function can be checked using the IoT Hub, the component of the Azure Cloud that manages IoT Edge devices.
On the security perspective, similarly to AWS Lambda, Azure Functions guarantees platform security by protecting functions from other clients, updating virtual machines and runtime software, and encrypting every communication between services. They supply a service to monitor activities, logging analytics to individuate attacks, and a service to manage authentication and authorization. In their comparison among existing FaaS platforms, Wang et al. [19] raised a warning on a potential security vulnerability of Azure Functions since the platform allows different tenants to share the same Virtual Machine for hosting their FaaS, what might represent a stepping stone for cross-function side-channel attacks.
Cloud functions [26]—It is the FaaS platform offered by Google. Similarly to AWS Lambda and Azure Functions, it supports fewer programming languages (viz. Node.js, Python, Go, and Java). Similarly to Amazon and Microsoft, Cloud Functions use triggers to execute functions. Billing is pay-per-use with precision at the level of hundreds of milliseconds.
Google does not provide developers with specific languages and tools to compose serverless functions on its FaaS platform. Naturally, any programming language of choice can be used by application developers to manually orchestrate HTTP requests to function end-points without specific support for FaaS orchestration constructs.
Google Cloud Functions security is oriented to the access control, split across identity-based and network-based access control. The identity-based access control is granted on a per-function basis via Cloud access management to allow for control over developer operations or function invocations. In the network-based access control, access is controlled by specifying network settings for individual functions. This allows for more control over the network ingress and egress, i.e. to and from the functions.

Apache OpenWhisk [27]—It is an open-source FaaS platform, initially created by IBM and, later on, maintained by the Apache Foundation. OpenWhisk allows developers to write serverless functions—called *Actions*—that can be dynamically scheduled and run in response to associated events—via *Triggers*—from external sources or from HTTP requests.
The full platform could be too resource-demanding to be executed on Fog nodes. To get around this limitation and use the same codebase of the main project, OpenWhisk developers reduced the components of the full version to a smaller architecture that is more suited to be placed on Fog nodes. They call it *Lean* OpenWhisk,[2] and the architecture on a node is reduced only to the Controller (responsible for load balancing) and the Invoker (responsible for executing serverless functions) of the full version, with a lean load-balancer to manage the handling of functions.

[2] Lean OpenWhisk https://github.com/kpavel/incubator-openwhisk/tree/lean.

OpenWhisk delegates completely to the platform administrator the duty to secure the target infrastructure and to the functions operators the security of applications.
IBM cloud functions [28]—It is the FaaS platform offered by IBM, based on OpenWhisk. Functions can be naturally integrated with IBM Cloud services. Support is provided for several programming languages (viz. Javascript, Python, Swift, PHP, Go, Java, Ruby, .Net Core) and are accepted as well user-supplied executable files to be run as containers starting from public Docker images. The execution model of the functions is the OpenWhisk one. IBM Cloud Functions features pay-per-use billing time with precision at the level of seconds.
IBM Cloud Functions offers IBM Composer [29], which extends OpenWhisk built-in function composition capabilities with conditional branching, error handling (try-catch, retry), loop constructs, parallel execution, asynchronous invocation, and map. Indeed, OpenWhisk by itself only allows pipelining functions. The compositions are expressed in Javascript or Python languages and compiled into JSON, ready to be deployed to the OpenWhisk platform. The library mimics the control flow of an imperative program language, treating functions as statements. Redis key-value storage can be exploited to perform stateful computations, such as in the case of parallel function executions to temporarily store results as soon as they are ready.
IBM Cloud Functions does not currently support execution on Fog nodes. The only serverless support on the Edge IBM supplies is to run web service functions, that are part of Cloud IBM Internet Services, mainly oriented to support web-based applications.
Concerning security, IBM follows the same policy of the other commercial FaaS platforms, even if they do not have a specific documentation entry related to the FaaS service. Generally speaking, IBM guarantees infrastructure security leaving application-related aspects to their clients.
Kubernetes-based platforms—Many existing platforms are built on top of Kubernetes and use it as their container orchestration engine for serverless functions. They run on private, public or hybrid Clouds and, being container-based, they can support any programming language of choice. The use of Kubernetes as orchestration engine for containers that will run serverless functions implies the possibility to run them on compatible Fog nodes. Open source platforms are expected to provide more flexibility and control compared to the commercial ones at the price of having more responsibility in terms of security.

Fission [30]—It is an open source community-based serverless framework that permits writing functions in every language, mapping them to incoming HTTP requests or other events. It allows orchestrating Fission functions with Fission Workflows, still at an early stage of development, by expressing suitable YAML workflows as sequences of tasks to be executed. Tasks can be functions, HTTP requests to a service or control flow constructs. The constructs include conditional branching and many loop variants. The *foreach* construct features for parallel execution over different parts of the input.
Kubeless [31]—It is a Kubernetes-native serverless framework, open source and community maintained, that allows deploying functions without having to

worry about the underlying infrastructure. It uses Custom Resource Definitions (CRDs) to extend the Kubernetes API, which allows developers to interact with functions as if they were native Kubernetes objects. Runtimes and languages needed to deploy a function can be specified by the users.
Knative [32]—It is a FaaS platform that leaves higher-level concepts (viz. API, CLIs, tooling) to a vendors to integrate with other platforms. It is open source and community-based. As Kubeless, it uses CRDs to extends the Kubernetes API. It can be integrated with *Istio*, a service that provides, authentication, authorisation and encryption of communication.
OpenFaaS [33]—It is an open source serverless platform maintained by OpenFaaS Ltd. It permits, through a command-line interface, packing functions into Docker containers. In each container, a Web server acts as an entry point to the container and allows invoking the hosted function.
Nuclio [34]—It is a serverless platform focused on data, I/O, and compute-intensive workloads, maintained by Iguazio Ltd. It provides GPU execution and easy integration with data science libraries and tools. It is the only one of this group that proposes both a vanilla open source version and an enterprise version with extended features, viz. platform hosted on the maintainers' Cloud and it is provided support for authentication and authorisation.

This overview of the FaaS platform outlines the state of the art of production-ready platforms concerning the three perspectives mentioned in the Introduction. As aforementioned, the rest of this survey is focused on the state of the art in the research literature.

3 Setting the stage

In this section, we describe the criteria used to sort out the research articles included in this survey for the three different perspectives P1–P3 identified in the Introduction. Throughout our work, we relied upon the material available in the leading scientific research libraries (i.e. IEEE Xplore Digital Library, Wiley Online Library, ACM Digital Library, Web of Science) by searching them through their search engines and Google Scholar. To fully capture the advances in the field, both journal and conference articles were collected during the search phase. Finally, references in selected articles were also considered to find more works to review.

During this phase, we adopted the following search and selection criteria:

P1 Languages, models and methodologies to *define FaaS orchestrations*.
Main search criteria: (FaaS $\vee$ serverless) $\wedge$ (composition $\vee$ orchestration) $\wedge$ (language $\vee$ model).
Inclusion criteria: We included proposals on languages, formal methods and models that permit defining and combining single serverless functions, in order to orchestrate them into more complex applications.
Exclusion criteria: We excluded those works lacking the possibility to define orchestrations of serverless functions, proposals that rely on FaaS as an enabling

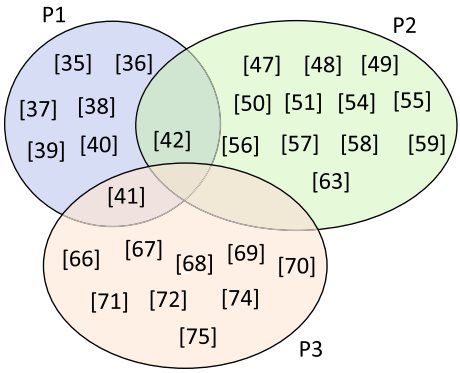

Fig. 1 Articles surveyed and classified P1—defining FaaS orchestrations, P2—executing FaaS orchestrations in the Fog and P3—securing FaaS orchestrations

technology but do not directly propose research advances on FaaS, and comparisons or surveys on existing serverless platforms as well.

P2 Platforms, techniques and methodologies to *execute FaaS orchestrations in the Fog*.
Main search criteria: (Fog $\vee$ Edge) $\wedge$ (FaaS $\vee$ serverless) $\wedge$ (placement $\vee$ scheduling $\vee$ execution).
Inclusion criteria: We included proposals on platforms, approaches and frameworks that contain methodologies or strategies to support the execution of applications emerging from serverless orchestrations in Fog scenarios[3].
Exclusion criteria: We excluded those works lacking novel orchestration strategies or capable of running FaaS only on single nodes. We also excluded works that are using existing serverless platforms as use cases without discussing orchestration details, nor research contributions under the considered perspective.

P3 Techniques and methodologies to *secure FaaS orchestrations* both statically and at runtime.
Main search criteria: (FaaS $\vee$ serverless) $\wedge$ security.
Inclusion criteria: We included proposals on novel tools, methodologies and approaches to secure the execution of serverless functions and their input/output data.
Exclusion criteria: We excluded works discussing security analyses without proposing any specific tool, methodology or approach to secure serverless execution.

From the selected corpus we excluded also patents, posters and publicly archived theses. Throughout this initial phase, we collected and analysed over 80 scientific articles, which were then reduced to 29 by applying the aforementioned inclusion and exclusion criteria. Figure 1 shows the articles found under the three perspectives P1–P3, and highlights the very few works that we identified at their intersections.

[3] Fog computing implies exploiting a continuity of Cloud, Edge and IoT resources, while Edge computing only focuses on exploiting local edge devices. However, the terms *Fog* and *Edge* are oftentimes used interchangeably in the literature. For this reason, we also considered and surveyed articles including the Edge keyword.

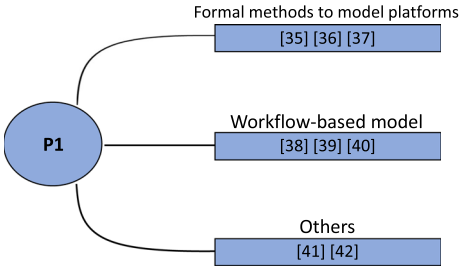

Fig. 2 Categorisation of articles in P1—defining FaaS orchestrations

It is worth noting that the intersection between P2 and P3 is empty since none of these works targets both security aspects of FaaS orchestration and FaaS orchestration in Fog settings.

4 Defining FaaS orchestrations

This section surveys languages and methodologies proposed to *define FaaS orchestration*, i.e. to programmatically combine simple well-defined serverless functions into more complex applications. The maturing of such languages and methodologies is essential for easily implementing new FaaS-based applications, and for the development and functioning of next-gen FaaS platforms.

4.1 Literature review

Figure 2 sketches the categorisation we will use to describe the articles surveyed in this section.

The works by Baldini et al. [35], Jangda et al. [36] and Gabbrielli et al. [37] introduce formal techniques for modelling and analysing existing FaaS orchestration platforms, with the two-fold objective of designing next-gen FaaS systems and of verifying properties of FaaS orchestrations.

Among the first authors highlighting the importance of devising new and safe languages to enable FaaS orchestration, Baldini et al. [35] present the *serverless trilemma*, a set of constraints that identifies the need of a careful evaluation of trade-offs in function orchestration. Those trade-offs mainly concern the fact that (i) functions should be considered as black boxes, (ii) function orchestration should preserve a substitution principle[4] with respect to synchronous invocation, and (iii) function invocations should not be double-billed. To support their analysis, after formally formulating the aforementioned trilemma, the authors present a model of the reactive core of OpenWhisk, and an OpenWhisk implementation of a solution to the trilemma, considering sequential compositions only. The considered reactive core of OpenWhisk is based on events (*triggers*) which represent a class of topics in an abstract message queue. Functions (*Actions*), have a unique name, their specific source codes, and always input

[4] From the point of view of an invoking client, the synchronous invocation to a single function or to a set of orchestrated functions should be indistinguishable.

and output dictionaries. A serverless orchestration is then represented as a set of *rules*, associating a specific topic of a trigger to a function. Each rule has an *enabled* status bit that determines if a message of a trigger results in an invocation of the function. The two operations to compose functions are `invoke`, used to run a function on a given input, and `fire`, used to create a new message for the event queue related to a trigger. When a new message is created, all the rules associated with that trigger are checked, and the enabled rules start the function invocations with the message as input. This mechanism is completely asynchronous and only permits composing sequences of functions based on events.

Jangda et al. [36] aim at enabling programmers to reason on the correctness of the serverless functions and FaaS orchestrations they implement, by helping them in better understanding and dealing with *low-level* operational details of FaaS platforms. In this context, they open sourced[5] the *Serverless Programming Language* (SPL). SPL is a domain-specific language based on a operational semantics, that models a serverless platform accounting failures, concurrency, function restarts, and instance reuse happening under the hood of such platform and considering interactions with external services. Communication is via message passing, modelling events via the "receive event" and "return result" primitives. Overall, the SPL language for function orchestration features three main primitives: simple function invocation, sequential execution of functions passing the output of one to another, and application of a function to a tuple modifying only the first element of the tuple while leaving the others unchanged. Those primitives permit expressing only simple orchestrations of serverless functions, without loops nor conditionals, which, however, have been implemented in the experimental prototype of SPL. Last, to make the input/output format uniform, a sub-language for JSON transformations is used. Experimental results are promising and improve over OpenWhisk performance.

Gabbrielli et al. [37] propose their *Serverless Kernel Calculus* (SKC), a formalism to model event-based function orchestration which combines features of λ-calculus and of π-calculus to describe functions and their communication, respectively. The basic model is then extended with stateful computation relying on message queues or databases, and event-based function compositions. Features typical of λ-calculus are used to model (recursive) function declarations, while communication is based on *futures* that represent the return value asynchronous function invocations. The event-based programming paradigm, typical of FaaS systems, is then modelled by encapsulating events in user-defined *handler* functions, which cannot directly invoke one another. Last but not, least, external services are considered as potential sources of events triggering function executions. An evaluation of the expressiveness of SKC in capturing all relevant features of FaaS orchestrations is performed over a real use case from a user registration system implemented in AWS Lambda.

The works proposed by López et al. [38], Eismann et al. [39] and Yang et al. [40], exploit the definition of FaaS orchestration expressed as workflows to model FaaS applications or to perform evaluations on cost or performance.

López et al. [38] propose *Triggerflow*, an architecture for serverless workflows. Such a proposal aims at supporting heterogeneous workflows while exploiting both

[5] Available at: https://plasma-umass.org/foundations-of-serverless/home/.

the reactive design of the serverless paradigm and good performance for high-volume workloads. The architecture of Triggerflow is based on an *Event-Condition-Action* model in which *triggers* (active rules) define which function (i.e. action) must be launched in response to Events, or to Conditions evaluated over one or more Events. In such a model, workflows are represented as a finite state machine, where the states are serverless functions and the transitions are labelled by triggers. The set of initial states are functions linked to initial events, the set of final states are functions linked to termination events. A trigger is a 4-tuple composed by (i) an Event, the atomic piece of information that drives flows in applications, (ii) a Context, the state of the trigger during its lifetime, (iii) a Condition, the active rules that filters events to decide if they match in order to launch the corresponding functions, and (iv) an Action, one or more function launched in response to matching Conditions. Experimental assessments are performed using AWS Lambda and IBM Cloud Functions showing that Triggerflow can support high-volume event processing workloads and transparently optimise workflows of scientific applications.

Eismann et al. [39] combine learning techniques (to predict function execution times and output parameters distributions) and Monte Carlo simulations (to predict costs) in order to compare and optimise the cost of executing FaaS orchestration workflows. Experimental evaluation, showing a 96.2% prediction accuracy, is performed over five real function orchestrations over the Google Cloud Function infrastructure.

Still employing a workflow-based modelling of FaaS applications, Yang et al. [40] propose EdgeBench, a benchmark to evaluate FaaS applications performances on the Fog. Workflows can be declared by specifying the business logic of each function, the data storage or transfer backends it needs, and its target execution tier (viz. IoT, Edge or Cloud). Performance evaluation is prototyped in OpenFaaS and it is based on metrics measured both at the function level (e.g. runtime latency, storage read/write latency, and the function communication latency) and at the workflow-level (e.g. CPU usage, network throughput). A benchmark is proposed for two representative applications (i.e. video analytics and IoT-hub management).

From a different perspective, Gerasimov [41] proposes *Anzer*, a domain-specific language that enables type-checking on function compositions. Anzer enables explicit and type-safe FaaS compositions by relying upon a type system that features basic types (integers, strings, boolean) along with the possibility to include user-defined types. A *Maybe* constructor is also featured by the language to allow for optional data types and polymorphism is supported, based on a well-defined notion of sub-typing. Type checking is performed on the description of the input function compositions, and compositions are considered safe if and only function inputs and outputs are coherently described. Besides, Anzer offers a mechanism to recognise failed computations based on monads, while it does not account for the presence of external services nor for events generated externally from the function compositions. The current prototype release[6] of the Anzer platform is still under testing, relies upon Apache OpenWhisk and only supports the Go language.

Finally, Persson and Angelsmark [42] focus on Fog settings, falling in the intersection of perspective P1, discussed in this section and P2 discussed in Sect. 5. Here we

[6] Available at https://github.com/tariel-x/anzer.

briefly illustrate the expressive characteristics of [42] to orchestrate serverless functions. The approach extends an actor-based framework in a serverless manner, where actors can be written as serverless functions. The Python-based declarative language CalvinScript is used in [42] to implement IoT applications capable of expressing (also through a graphical representation) the data-flow between functions and the placement of functions on specific nodes. The control flow of an application is handled by writing specific functions that handle the flow, e.g. to express a conditional branch a function is needed that receives input data, tests a condition and passes the data to another function based on such test. To illustrate the framework some IoT application examples are detailed and it is described an overall comparison with FaaS platforms, such as AWS Lambda and OpenWhisk.

4.2 Summary at a Glance

Table 1 resumes the main mechanisms offered by the surveyed articles. For each surveyed approach, the columns in Table 1 indicate whether it features

- Support for *basic programming constructs* (viz. sequential composition, conditional branching, iteration and parallel composition) which allow creating more complex applications using serverless functions as building blocks,
- The possibility to exploit a *direct trigger* (e.g. an HTTP request) to invoke a specific serverless function,
- The possibility to exploit a *trigger by publish/subscribe* events to automatically run a function whenever such events happen,
- The possibility to define *recursive functions*, that is, to recursively invoke a function at runtime,
- The availability of *type checking* mechanisms verifying the correctness of composition of functions calls based on their input and output data types.

It is worth noting that the sequential construct is available in all surveyed approaches. Other basic constructs—viz. sequential composition, conditional branching, iteration and parallel composition—are featured by most of the approaches in the literature [36–40,42]. The usage of direct triggering is easier to implement although less flexible from a programmer's perspective and it is featured by [36–38,40]. In contrast, the triggering by a subscribed event is featured only by [35,38]. Recursion is generally discouraged by FaaS platform providers as it might lead to long execution times of functions and repeated function calls, which actually cause paying for idle time, but it is nevertheless allowed by [35–38]. Type checking on function compositions is only studied by [41], while other solutions consider dictionaries as the only input/output types and do not perform code analyses.

To improve the definition of FaaS orchestration, a wider usage of the function triggering by subscribed events can help at decoupling functions from the caller, by easing next-gen application development. For instance, this is especially important in the field of IoT applications where the publish/subscribe mechanism is well-established and functions run upon triggers from cyber-physical systems. By extending their support to recursion, FaaS platforms could avoid launching new function instances (e.g. via tail recursion optimisation) and unnecessary billing and resource-wasting behaviour,

Table 1 Defining FaaS orchestration: a bird's-eye view

	Sequential composition	Conditional Branching	Iteration	Parallel composition	Recursive functions	Direct trigger	Publish/subscribe trigger	Type checking
Baldini et al. [35]	✓				✓		✓	
SPL [36]	✓	✓	✓	✓	✓	✓		
SKC [37]	✓	✓	✓	✓	✓	✓		
Triggerflow [38]	✓	✓	✓	✓	✓	✓	✓	
Eismann et al. [39]	✓	✓	✓	✓				
EdgeBench [40]	✓	✓	✓	✓	✓	✓		
Anzer [41]	✓							✓
CalvinScript [42]	✓	✓	✓	✓				

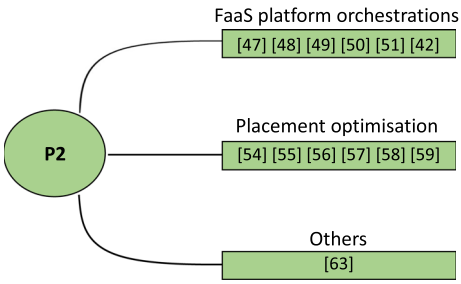

Fig. 3 Categorisation of articles in P2—executing FaaS orchestrations in the Fog

while allowing programmers to naturally express recursive functions. Type checking is also an interesting line to follow to reduce errors by early detection of wrong invocations. This permits avoiding run-time errors, by easing application testing and maintenance.

Finally, as an alternative to orchestration languages and workflows employed by most of the surveyed approaches, Calvin [42] exploits an actor-based model to compose services and serverless functions into applications. Such models might be interesting to further investigate in FaaS settings as they have recently been employed to describe AI [43] applications—including annotations on their hardware requirements—and parallel computing patterns [44].

5 Executing FaaS orchestrations in the Fog

This section surveys existing approaches proposed to *execute FaaS orchestrations in the Fog*, i.e. to place, deploy, run orchestrated serverless functions along the Cloud-IoT continuum, suitably closer to their end-users, or to IoT data sources, which trigger them or wait for processed results. Research in this field is still emerging[7] but it will be utterly important to achieve context- and QoS-awareness of FaaS applications running on next-gen networking infrastructures.

5.1 Literature review

Figure 3 sketches the categorisation we will use to describe the articles surveyed in this section.

The works proposed by Cheng et al. [47], Baresi and Mendonça [48] and Baresi et al. [49], Cicconetti et al. [50], Mortazavi et al. [51] and Persson and Angelsmark [42] are based on FaaS platform orchestration in the Fog. They propose novel architectures or adapt well-established ones to the Fog settings, in order to define and analyse the serverless characteristics useful to next-gen application development.

[7] There are several works that aim at supporting the serverless paradigm in the Fog proposing solutions or techniques involving single nodes and tackling issues like intra-node communication, constrained resources and function isolation [13,45,46]. Those works are excluded from the survey as per the exclusion criteria detailed in Sect. 3.

Cheng et al. [47] proposed *Fog Function*. It is based on a programming model for data-intensive IoT applications, which exploits data-aware context discovery to enable data-driven orchestration of serverless functions. The unit of computation is called a Fog Function and is denoted by a unique identifier, a dockerised function, its required inputs, and (possibly) its expected outputs, a geo-scope to suitably select input data, an execution priority, and specific Service Level Objectives (SLOs) to be met at runtime (e.g. latency, results accuracy). A centralised management node enables the whole orchestration system by relying on two main components, viz. the *Orchestrator* and the *Discovery* components. Besides, all involved Fog nodes run two distributed software agents, viz. a *Broker* and a *Worker*. On one hand, Brokers collect contextual data coming from the IoT, intermediate processing results and data on node resource availability, which are then used by the Discovery component to obtain a global system view. On the other hand, Workers execute functions when suitably triggered by the Orchestrator. Indeed, the Orchestrator receives Fog Functions from clients, subscribes to the Discovery and exploits it to identify and process, via a suitable *Worker*, the received input data. Last, the Orchestrator can migrate tasks from one node to another within the managed Cloud-IoT continuum.

Based on several uses cases, the works by Baresi and Mendonça [48] and Baresi et al. [49] identify key characteristics to adapt the FaaS paradigm to Fog scenarios and propose a 3-tiered self-managing platform for Multi-access Edge Computing (MEC) infrastructures, respectively. The characteristics identified in *SEP* [48] include (i) low-latency computation offloading to reduce response times, (ii) inter-platform collaboration to achieve prompt and balanced resource allocation, (iii) latency optimisation to reduce communication overhead, (iv) opportunistic data analysis to achieve faster processing of IoT inputs, (v) geo-aware edge nodes coordination via publish-subscribe mechanisms, (vi) stateful application partitioning into functions to reduce computation times and resource requirements. Via an OpenWhisk prototype, they demonstrate the need to further reduce latencies and computing overhead of existing FaaS systems, by possibly exploiting hierarchical orchestration mechanisms. Indeed, later on, Baresi et al. [49] propose a 3-tiered hierarchical framework—*PAPS*—for decentralised self-management of function containers in MEC scenarios. First, based on latency-aware clustering, a centralised network *supervisor* clusters the whole infrastructure into communities. Then, each community elects a *Leader* node which attempts avoiding SLO (viz. response and maximum execution times) violations by optimally placing containers onto its community nodes. Finally, *single node* management handles containers scaling to deal with workload variations and to ensure locally meeting the set SLOs. Simulations in PeerSim [52] show promising results of PAPS in optimising the considered SLOs according to the proposed 3-tiered self-management architecture.

Another work focusing on enabling FaaS in MEC architectures is carried out by Cicconetti et al. [50], by proposing that all involved edge nodes (viz. MEC hosts) run a serverless platform. Such a distributed platform interacts with a centralised *MEC Orchestrator* to retrieve—upon client requests—the functions to be executed and on which host execute them depending on the client's context. The mapping from client nodes to functions and MEC hosts is maintained by an *optimiser* component. Three different optimisation strategies to determine and update the mapping are proposed, i.e. *static* (unchanged over time), *centralised* (based on global network knowledge at

the orchestrator) and *decentralised* (based on distributed local function dispatchers). Experimental assessment of the three strategies results in the *static* strategy showing the worst performance in terms of latency and the best in terms of bandwidth usage, having client requests always mapped to the same MEC host. On the other hand, the centralised and the decentralised strategies outperform the static one in terms of latency but show way higher bandwidth usage due to communication overhead from the orchestrator/dispatcher.

Mortazavi et al. [51] rely on a Fog computing platform based on a multi-tier Cloud-IoT infrastructure. In this context, they propose *CloudPath* a platform to support the execution of FaaS applications in Cloud-IoT scenarios. CloudPath nodes feature a container runtime and a module that can dynamically deploy and remove applications, according to users' preferences, system policies and node resources. Example system policies might require a function to run on a specific level of the IoT-to-Cloud path hierarchy, or on any higher layer towards the Cloud. The current prototype permits choosing among three main layers (e.g. Edge, Core, Cloud) in the hierarchy by suitably labelling the functions, and to set latency requirements. Experimental assessments on CloudPath show that response times lower by a factor of 10 with respect to using a single public Cloud.

By extending an actor-based approach for serverless computing, Persson and Angelsmark [42] extend their prototype IoT platform, Calvin [53], with FaaS capabilities. Calvin permits defining distributed actor as unit components to process data, and to make those actors interact into graph-based workflows. The authors extend Calvin with IoT-based functions called *kappas* which can be triggered by Calvin actors. Such functions can have input and output (e.g. face recognition), no input nor output (e.g. reaction to sensed events, periodical updates), only input (e.g. storage), or only output (e.g. temperature reading). Analysing and comparing their proposal with existing platforms—e.g AWS Lambda and Apache OpenWhisk—the authors found similar basic principles, with main differences highlighted in the possibility to write stateful *kappas* and fewer resources needed to run the Calvin framework, being suitable for Fog settings.

Differently from previous works Pinto et al. [54], Das et al. [55], Aske and Zhao [56], Cho et al. [57], Cicconetti et al. [58] and Rausch et al. [59] propose approaches to optimise the placement of serverless functions in the Fog, by relying on monitored metrics (e.g. latency, billing) or on available hardware resources to make informed decisions.

Pinto et al. [54] aim at dynamically deciding whether to run a serverless function within the local Edge network or on Cloud servers, based on historical data on the function execution times. An edge proxy is in charge of making such a decision, and three heuristics (viz. greedy, upper confidence bounds and Bayesian upper confidence bounds) have been proposed to do so. Besides, the proposed approach is capable of handling failures (e.g. Internet disconnection) by forcing computation to happen on local edge resources.

On the same line of work, Das et al. [55] present a framework to dynamically decide where to execute tasks in Fog infrastructures, trying to optimise execution time and function billing costs. To this end, a regression-based prediction model is trained on three representative single-function applications—viz. image resizing, face detection

and speech-to-text—to estimate compute, network transfer and storage latencies, and costs. The optimisation framework runs on edge nodes, directly receiving input from IoT data sources. A decision engine extracts the input characteristics and leverages the prediction model to decide on whether to execute a function in the Cloud or locally (still using Cloud resources for storage purposes). Experimental assessments show that the prediction model only incurs in 6% absolute error when estimating execution times, thus improving by 3 orders of magnitude the featured latency with respect to Edge-only computation, which takes longer to queue up and process incoming requests.

Aske and Zhao [56] propose *MPSC*, a serverless monitoring and scheduling system to assist programmers in selecting a serverless provider, based on average execution time, affinity constraints, and costs. After monitoring the providers' performance in real-time, the proposed framework recommends where to execute functions also accounting for user-defined scheduling policies written in Python. The framework exposes a REST API to be used by the FaaS applications to load function requirements and scheduling policies and to trigger function execution. Experimental results—with AWS Lambda, IBM Bluemix and Apache OpenWhisk—show 200% faster round-trip execution times when compared to execution on a single provider and, in general, greater flexibility in using Edge computing in continuity with Cloud resources to adaptively improve application QoS.

Aiming at reducing latencies and avoiding exceeding maximum response times, Cho et al. [57] present *RACER*, a solution to distribute FaaS workload over hierarchically organised Fog infrastructures. In the considered scenario, edge devices produce function tasks to be placed on Fog infrastructure nodes to run. The proposed solution solves the placement problem employing a (relaxed version of the) *token bucket* algorithm [60], considering task constraints. The RACER *Controller* checks the admissibility of tasks for the whole network while RACER *Agents* are in charge of managing individual edge regions, by employing reinforcement learning to optimise workload distribution and response times, notifying to the controller local decisions. Simulations show that RACER outperforms both static workload distribution and *Area* [61], a state-of-the-art algorithm for workload distribution, for what concerns the achieved response times. Besides RACER incurs in decision times which are two orders of magnitude lower than those of Area.

Cicconetti et al. [58] propose an architecture to realise serverless computing at the Edge in an Software-Defined Networking (SDN) scenario where network routers can assign function execution to edge devices based on an arbitrary cost (e.g. latency, bandwidth usage, energy consumption). The network state and the edge resource availability is monitored and updated by SDN controllers. Three different placement strategies are proposed: (i) *Least-Impedance* (LI), always selecting the minimum cost node, (ii) *Random-Proportional* (RP), selecting a node with a probability proportional to its cost to avoid overloading always the same nodes, and (iii) *Round-Robin* (RR) fairly combining cost and workload considerations. Both (ii) and (iii) achieve long-term fairness in allocating FaaS, while (iii) also achieves short-term fairness. Simulations with OpenWhisk show that the proposed RP and RR strategies can efficiently handle fast-changing workload and network conditions, whilst LI performs worse than static allocation.

More recently, based on their model to annotate function requirements [62], Rausch et al. [59] present *Skippy*, a container-based scheduler to optimise the placement of serverless functions in the Fog. Skippy relies on multi-criteria optimisation and extends the default scheduling logic of Kubernetes by introducing four Fog-oriented scheduling constraints: (i) proximity to container image registries, (ii) proximity to data producers, (iii) available node resources and (iv) Edge or Cloud locality. Such constraints can be weighted differently to adapt the scheduling behaviour so to achieve different objectives (e.g. minimising execution times, bandwidth usage, costs or resource exploitation). Skippy dynamically and distributedly collects information on node resources, while it inputs the target network topology as a graph annotated with (static) bandwidth values. Experimental assessment with OpenFaaS on a small-scale testbed shows that Skippy enables locality-sensitive function placement, with higher data throughput and reduced bandwidth usage.

Finally, Bermbach et al. [63] are the sole to take an infrastructure provider's perspective. They advocate the need for a distributed auction-based strategy to allocate FaaS to (Cloud or Edge) providers' nodes. In this scenario, programmers submit functions to target nodes along with a storage and processing bid. By accepting or rejecting those bids, target nodes can decide whether to store function executables and, afterwards, whether they can actually run them depending on their currently available resources. Rejected requests are then pushed to the next node on the path towards the Cloud. Preliminary simulation results show that this approach can increase providers' revenues in Fog serverless scenarios.

5.2 Summary at a Glance

Table 2 offers an overview of the reviewed articles, showing which elements the orchestration strategy consider. For each surveyed approach, the columns of Table 2 indicate if the proposed orchestration strategy features

- *Latency-awareness*, aiming at reducing the latency of application responses,
- *Resource-awareness*, trying to optimise the hardware resources of the nodes that execute serverless functions,
- Allowing a *custom strategy*, which permits platform users to express their own strategy to orchestrate functions in the Fog,
- *Cost-awareness*, aiming at reducing the costs by the platform clients,
- *Data-awareness*, permitting to exploit what IoT data represent and where are produced, in order to improve functions placement and scaling,
- *Bandwidth-awareness*, trying to reduce the overall bandwidth usage of the network,
- *Failure-awareness*, being able to recognise and to try to overcome possible runtime failures, e.g. nodes that became unreachable.

It is worth mentioning that the latency of applications (considered in [48–51,55–59]) and the resource optimisation of nodes (considered in [47–49,51,56–59,63]) are taken into account the most in the surveyed literature. [42,51,56,63] also allow specifying user-defined strategies. [55,56,59,63] take into account the costs, an important element in the FaaS settings. Only [47,50,59] are oriented to data-awareness, which represents

Table 2 Overview of elements considered in the reviewed orchestration strategies

Reference	Latency-awareness	Resource-awareness	Custom strategy	Cost-awareness	Data-awareness	Bandwidth-awareness	Failure-awareness
Fog Function [47]	✓	✓			✓		
SEP [48]	✓	✓			✓		
PAPS [49]	✓	✓					
Cicconetti et al. [50]	✓				✓		
CloudPath [51]	✓	✓	✓				
Kappa [42]			✓				
Pinto et al. [54]						✓	✓
Das et al. [55]	✓			✓			
MPSC. [56]	✓	✓	✓	✓			
RACER. [57]	✓	✓					
Cicconetti et al. [58]	✓	✓				✓	
Skippy [59]	✓	✓		✓	✓	✓	
Bermbach et al. [63]		✓	✓	✓			

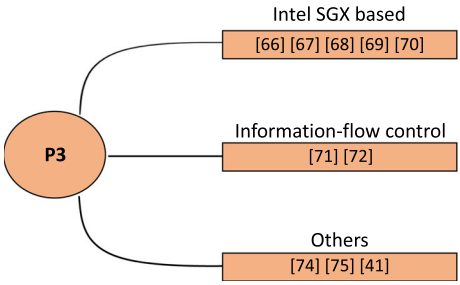

Fig. 4 Categorisation of articles in P3—securing FaaS orchestrations

a key element in Fog applications being generally centred on IoT data extraction and analysis. Finally, [50,54,59] consider the bandwidth and the possibility to be temporally isolated from the Cloud, due to network failures.

To enhance the execution of FaaS orchestration in the Fog, solutions aimed at reducing bandwidth, which is among the objectives of the Fog paradigm, deserve further investigation. Another aspect worth further investigation concerns topology-aware strategies like strategies capable of grouping Fog nodes to achieve faster workload processing and to coordinate better on overall resource usage. It is also important to detect and resolve failures at run-time, especially in Fog settings where the connection to devices or their limited resources can make nodes unexpectedly unreachable or unable to execute functions [64].

Additionally, preliminary work exploiting the FaaS paradigm in Osmotic Computing [65] shows that it would be of interest to devise orchestration strategies that consider peculiarities of such an emerging paradigm, e.g. availability of functions in different versions, with different requirements, to permit adaptation when no eligible placement can be found that meet all initially set requirements. Finally, it would be interesting to define a set of benchmark applications and infrastructures to evaluate and compare the performance of FaaS orchestration strategies in Fog scenarios. A starting point towards this direction could be the recent work by Yang et al. [40], discussed in Sect. 4.

6 Securing FaaS orchestrations

This section surveys approaches proposed to secure the execution of FaaS orchestrations, i.e. to enforce data and billing integrity, confidentiality and availability through specialised hardware and/or code analyses techniques. The evolution and adoption of such approaches are crucial to increase programmers' and end-users' trust in serverless systems, especially by enabling secure FaaS on next-gen Cloud-IoT infrastructures.

6.1 Literature review

Figure 4 sketches the categorisation we will use to describe the articles surveyed in this section.

In the scope of this section, most of the surveyed works (i.e. [66–70]) focus on the exploitation of specialised hardware resources, the Intel SGX, that permit creating enclaves[8] to protect the memory and the execution of functions from superusers of certain machines. Intel SGX permits creating a Trust Computing Base (TCB) in order to protect the execution from possibly malicious FaaS providers. Almost all of these works try to protect data confidentiality. Only [70] do not address data integrity as a protected asset. Finally, all works but [67] address function integrity, protecting either the chain, the memory or the code of serverless functions.

First, Trach et al. [66] present *Clemmys*, a secure platform for serverless computing implemented by relying on Apache OpenWhisk. A shielded execution framework is built on top of Intel SGX and used to run unmodified applications, by securely handling the FaaS platform gateway and the workers that execute functions. Function data are protected by implementing the encryption of incoming traffic on the gateway and by distributing encryption keys to the workers through a key management system. Communication is performed by a specific message format that contains information about the name of the invoked function and information about the order of execution of a function chain, viz. a function composition, so that any function can realise when chain execution has been tampered.

Still based on OpenWhisk, Alder et al. [67] propose *S-FaaS*, an architecture to provide strong security and accountability guarantees in FaaS settings tying to guarantee a correct estimation of the resources used to both the FaaS provider and the function provider. The architecture proposed includes a Key Distribution Enclave (KDE) and worker enclaves running on the FaaS platform. The KDE is responsible for distributing the keys to the workers, for publishing them to clients and for attesting such keys and each worker enclave. To activate a function, clients encrypt the input data with the published key together with the hash of the function to invoke. Then, the KDE attests the working enclave before passing the incoming data to the correct target function. Finally, such a function processes the data and returns the encrypted output, also generating a receipt for the client to attest that its execution completed. The enclave setup for function execution also enables a resource management mechanism that measures metrics of the execution and uses keys distributed by the KDE to sign such measurements, permitting to attest the resource consumption.

Brenner and Kapitza [68] propose a design for a secure FaaS architecture based on Javascript (JS) isolation capabilities. To achieve that, the authors use recent JS engines to guarantee isolation of functions. The use of a specific interpreted language permits to make assumptions to perform optimisations, i.e. bundle libraries with function code to augment isolation, store a single interpreter in the enclave and reuse of runtime contexts. The authors propose a general architecture and two related specific instances based on different JS engines (SecureDuk and Google Secure V8), being the first more lightweight and the latter more performant. The general architecture has in the enclave a connection manager to communicate with clients, a JS interpreter and sandboxes to execute functions. Every function is loaded from external storage and attested using the Intel SGX. The encryption schema uses two keys, one to for the communication with

[8] An enclave provides hardware-level isolate code and region of memory, unreachable from any process outside the enclave, even high privilege ones.

clients and one to protect data on external storage. The two different implementations with the JS engines are used to validate and test the design in different settings.

Gjerdrum et al. [69] present *Diggi*, a FaaS framework to execute secure distributed functions while maintaining a small TCB. The framework runtime is split into two parts, one considered trusted and one considered untrusted. Each physical host runs a daemon process that executes the untrusted runtime shared by functions, all featuring distinct instances of the trusted runtime within their respective enclaves. On one hand, the trusted runtime manages thread scheduling, message passing, state preservation, key distribution, and key attestation. On the other hand, the untrusted runtime handles deployment, lifecycle management, configuration, inter-node messaging, networking, and filesystem interaction. Functions must implement specific callbacks and can register custom ones, in order to be managed by the runtime. Applications may be composed by multiple functions that are compiled, signed, and deployed according to well-defined specification files formatted in JSON. To manage deployment, attestation and keys distribution is expected the presence of a trusted component (*trusted root*), that has to run inside a trusted security domain, in order to run a protocol to perform multi-attestation and key distribution to the functions.

Finally, Qiang et al. [70] propose *Se-Lambda*, a secure serverless computing framework that resorts to the Web Assembly sandboxed environment besides the Intel SGX. The architecture of the framework is formed by modules on the API Gateway and by a two-way sandbox for every function, that depends on the enclaves to be considered trusted. The modules on the API Gateway handle the security checks on every client request and attest the sandbox of the functions that should execute such request. The two-way protection provided by the sandbox consists of protecting functions data by using SGX enclaves protects from one side, and to protect the host runtime of the Cloud provider using WebAssembly sandboxed environment from the other. To overcome the overhead of creating and destructing enclaves and sandboxes, the lifecycle of such components is modified to permits to dynamically load functions into the sandboxes reusing initialised enclaves.

Differently from previous proposals, Alpernas et al. [71] and Datta et al. [72] rely on information flow control [73] techniques to identify information leaks or unsafe access to data or functions. Those techniques are generally based on a security lattice, i.e. a graph that defines a partial order on well-defined data security levels, usually from low to high. Data and stakeholders involved in the system are tagged with such levels to enforce policies that try to prevent low-level stakeholders from accessing or modifying high-level data.

On one hand, Alpernas et al. [71] present *Trapeze*, an approach for dynamic information flow control in serverless systems with the objective to protect data confidentiality by external attackers or by misconfigured applications, by enforcing the property of termination-sensitive non-interference (TSNI). Input and output of each function is monitored to tracks the flow of information and enforces a security policy. A trusted authentication gateway tags with security labels all external input and output channels and use them to build the security lattice. Labels are assigned at runtime to every serverless function activation based on the event that triggers the function, e.g an HTTP request has the label of the client that performs the request. The access to storage data or communication channels is permitted by functions with the label at least of the

same level on the security lattice. When a function creates or updates a record in the store, the record inherits the function's label. Instead, when it reads from the store, the function only observes values whose labels are below or equal to its own label. From the function's perspective, the store behaves as if it did not contain any data that the function may not observe. To implement such storage access semantic are employed *faceted storage*, meaning that each storage record can contain several values with different security labels. The Trapeze prototype implementation is currently compatible with AWS Lambda and Apache OpenWhisk.

On the other hand, Datta et al. [72] propose *Valve*, an approach for transparent dynamic information flow of serverless functions aiming to protect leaks of data due to malicious or buggy functions. The solution works in a setting where every communication passes through a centralised gateway, in which is placed a Valve controller. Functions are executed in containers, in which are placed entities called Valve agents. The agents keep track of every communication of the functions by proxying network requests issued from within the containers, learning the information flow, labelling it with *taints* and sending them to the controller. When operating in enforcement mode, the agents authorise or deny requests according to the security policy they must guarantee. During enforcement mode, the agents perform post-invocation garbage collection by deleting modified files, in order to avoid leaks of data from container reuse. The controller runs as a service besides the functions of the platform, performing four tasks: (i) it checks the information flow of the application, based on the taints received from the agents, (ii) it generates a default security policy based on such accumulated knowledge, (iii) it incorporates rules specified by developers and other configuration into the security policy, (iv) it manages the enforcement mode by sending the security policy to the agents. To represent the policies, the authors proposed a specification grammar, where each policy is defined by a set of functions with ingress and egress rules with a condition on the taints to allow or deny data exchange. A comparison of Valve against OpenFaaS with Trapeze [71] shows that Valve can mitigate common attacks, while assisting developers in auditing the information flows of their applications.

From a language-based perspective, Boucher et al. [74] propose a design for serverless platforms that run user-submitted microservices—in the form of functions—within shared processes. The objectives of this work are to improve latency of serverless applications and to increase the security of serverless functions providing strong isolation within the processes. The solution is guided by language-based isolation with defence in depth. To achieve their goals, the authors leverage on Rust, a type-safe compiled programming language that uses a lightweight runtime similar to the C language. Only functions written in Rust are admitted in the platform and language mechanism are exploited to guarantee the isolation of functions without damaging the performances. The aspect that Rust does not provide is the possibility that a function fully occupied the CPU and a malicious or a buggy function can stall all the processes. To avoid this issue, authors prevents the use of some system calls that can block the execution. To reduce the submission of unsafe code, there is a whitelisting of Rust libraries considered safe to be used. If the application compiles it is proven memory-safe and if it links then it depends only on trusted libraries. Then, the deployment server produces a shared object file, which the provider then distributes to each compute node on which it might run.

Recently, Gadepalli et al. [75] proposed *Sledge*, a serverless framework for single edge nodes, based on the WebAssembly runtime (i.e. a binary instruction format for a stack-based VMs). Sledger employs lightweight function isolation and allows executing multiple untrusted modules in the same process, while guaranteeing memory safety and control-flow integrity. An *ad-hoc* compiler from LLVM bytecode to WebAssembly enables four different configurations for runtime memory access bound checks. This enables Sledge to leverage heterogeneous hardware and software capabilities for bounds check management, while also allowing edge providers to identify trade-offs between application performance and the cost of checking memory accesses. Sledge also implements software-based safety checks on function pointer invocations to ensure control-flow integrity by checking that modules only invoke functions they can access, with valid input types.

Last, but not least, it is worth mentioning the work by Gerasimov [41], already introduced in Sect. 4. The type system introduced by this approach can naturally prevent application developer mistakes that can lead to security vulnerabilities. It can be used to protect the integrity of function chaining, preventing to have a type error between two consecutive functions.

6.2 Summary at a Glance

Table 3 shows the main aspects emerged by analysing the proposed solutions. For each approach, the columns in Table 3 are divided into three groups

- *Protected asset*, which includes
  - *Data confidentiality* and *data integrity* that are the most classical assets of information security,
  - *Function integrity* is intended as code, local memory or order in a function chain,
  - *Billing*, which is intended as the measurement by the platform that will cause a cost for the clients.
- *Causes* considered that threat such assets, divided in
  - *FaaS provider*, the entity that manages the platform or a malicious employee within such entity,
  - *External attacker*, an unknown party that tries to violate the system,
  - *Application provider*, who manages the FaaS application running on the platform,
  - *Function developers mistakes* either in code functions or platform configurations.
- the *Protection technique* employed, individuated in
  - *Function sandbox*, that is used to isolate the functions by means of specialised hardware or runtime support,
  - *Remote attestation*, which consists in acknowledging securely clients or other functions,

Table 3 Overview of security aspects

Reference	Protected Asset				Causes				Protection Technique					
	Data Confidentiality	Data Integrity	Function Integrity	Billing	FaaS Provider attack	External attack	App (function) Provider	App developer Mistake	Function sandbox	Remote Attestation	Key management system	Information flow control	Defence in depth	Function Type System
Clemmys [66]	✓	✓	✓		✓	✓	✓		✓	✓	✓			
S-FaaS [67]	✓	✓		✓	✓	✓	✓		✓	✓	✓			
Brenner and Kapitza [68]	✓	✓	✓		✓				✓	✓	✓			
Diggi [69]	✓	✓	✓		✓	✓			✓	✓	✓			
Se-Lambda [70]	✓		✓		✓	✓			✓	✓				
Trapeze [71]	✓					✓		✓				✓		
Valve [72]	✓						✓	✓				✓		
Boucher et al. [74]			✓			✓		✓					✓	
Sledger [75]			✓			✓		✓	✓					✓
Gerasimov [41]			✓					✓						✓

- *Key management systems*, that protect and distribute encryption keys,
- *Information flow control*, employed to recognise information leaks during the communication flow,
- *Defence in depth*, which is the strategy to stratify a system defence around the assets,
- *Function type system*, intend as typing input and output of functions to avoid type errors in a chain.

It is worth mentioning that the main focus is on the protection of confidentiality of data provided to serverless functions and on the results of their computation, indeed, all the surveyed solutions but [41,74] are defending such asset. Data integrity is less considered, only half of the works surveyed take it in account [66–69]. The integrity of functions is also an important aspect of the serverless scenario that has been targeted by [41,66,68–70,74,75]. Correct billing is an asset considered only by [67], even if costs are always pinpointed a key characteristic of the FaaS model. Most the time, the FaaS provider is considered untrusted (i.e. [66–70]) and it is considered dangerous almost as an external attacker (i.e. [66,68–71,74,75]). The application provider can also be considered malicious respect to the users of the application [66,67,72]. Functions flaws and platforms misconfigurations are another possible cause of security problems, [41,71,72,74,75] try to employ methods to prevent them. For this reason, there is a preference for techniques relying on specialised hardware to sandbox and attest functions or to manage the encryption keys, like those featured by [66–69], while only [75] employs runtime support sandboxing. Information flow control approaches are only employed by [71,72], while defence-in-depth is only proposed by [74]. Finally, [41,75] are the sole proposals including function type safety.

The vast majority of the reviewed works employ runtime monitoring techniques and have an impact on applications performances. Having static analysis techniques that can assist the monitoring can be helpful to improve security while reducing the burden of the runtime. In settings where the FaaS provider can be considered trusted, information flow techniques can be useful.

7 Conclusions

This article surveyed methodologies and techniques to achieve the secure execution of FaaS orchestrations in Fog computing. Particularly, we considered and analysed exiting solutions:

P1. to *define* FaaS orchestrations, focussing on the composition of serverless functions into complex applications and considering linguistics aspects that will be key in easing the development and functioning of next-gen FaaS system (Sect. 4),
P2. to *execute* FaaS orchestrations in the Fog, focussing on methodologies to place, deploy, run orchestrated serverless functions along the Cloud-IoT so to enable serverless computing closer to IoT devices and to end-users, thus improving the overall QoS and QoE of running FaaS applications (Sect. 5), and
P3. to *secure* FaaS orchestrations, focussing on enforcing and protecting data integrity, confidentiality and functions integrity, thus securing serverless platforms and run-

ning FaaS applications and improving developers' and users' trust in next-gen FaaS systems (Sect. 6).

We gave an overview of recent relevant research activities under P1–P3, aiming at identifying open challenges at their intersection, i.e. aiming at realising secure FaaS orchestration in Fog settings. To the best of our knowledge, there are currently no proposals that fall in such intersection. We now analyse possible future directions that can be explored to further investigate these emerging topics, by first considering pairwise intersections of P1–P3.

P1 ∩ P2 (*Defining FaaS orchestrations ∩ Executing FaaS orchestrations in the Fog*)

- *Orchestration-aware Execution* All approaches reviewed under P2 focus on executing one function at a time, lacking the possibility to exploit information on the behaviour of an application composed of a set of orchestrated functions. Particularly, in the Fog, orchestration-aware approaches can enable better function placement to FaaS-enabled computing nodes along the Cloud-IoT continuum, thus improving overall resource management and QoS of running applications, e.g. by selecting nodes more suited for certain types of function workload. Besides, exploiting data-flow information can further improve decision-making on function deployment, e.g. by placing functions according to data-locality principles to avoid unnecessary data transfers between "consecutive" functions, or between IoT data sources and data processing functions.
- *Definition and Execution of context- and QoS-aware orchestrations* Only few works [43,59] are focussing on methodologies or approaches to defining context or QoS requirements of both single functions and function orchestrations. Defining annotations or specification languages to allow developers specifying contextual constraints (e.g. hardware, software requirements, node affinity) and QoS constraints (e.g. latency, bandwidth) on FaaS orchestrations, could improve the execution of FaaS-based applications in the Fog. Recent works in Cloud and Fog scenarios [76], consider these aspects in the context- and QoS-aware placement of multi-service applications, and it would be of interest to adapt/extend those methodologies also to handle FaaS orchestrations.

P1 ∩ P3 (*Defining FaaS orchestrations ∩ Securing FaaS orchestrations*)

- *Definition of Security requirements* As per our analysis, there are no proposals of definitions of *security requirements* for: (i) FaaS orchestrations (i.e. defining security policies for an application composed by a set of serverless functions), (ii) single functions (e.g. requiring the availability of certain security countermeasures or white-listing execution only on certain nodes), and (iii) data flowing across functions (e.g. using partially ordered security levels to avoid disclosure of data). Enabling the possibility to specify —analyse, and enforce— such security requirements also in FaaS landscapes (as done, for instance, in microservice-based architectures [14,77]) will move a step towards supporting multi-level security strategies in next-gen FaaS systems along Fog infrastructures.

– *Static analyses of FaaS orchestrations* The definition of the aforementioned security requirements can also enable static analyses of FaaS orchestrations to early detect errors and security flaws both in FaaS orchestrations and in their placement, lightening the burden on runtime support. This is especially relevant in settings where hardware resources are limited (i.e. closer to the network edge), and the footprint of (runtime and infrastructure) monitoring can impact on application performance. Another interesting line in static analyses includes devising more sophisticated static type checking techniques of functions orchestrations, which were only preliminarily studied in [41].

P2 ∩ P3 (*Executing FaaS orchestrations in the Fog* ∩ *Securing FaaS orchestrations*)

– *Secure Execution of FaaS orchestrations in the Fog* Most of the reviewed security approaches for FaaS rely on specialised hardware, which is difficult to achieve in Fog settings due to the presence of many heterogeneous, general-purpose or resource-constrained devices. To securely execute FaaS orchestrations in the Fog, serverless runtime support needs to be aware of the security requirements of both single functions and orchestrations, and their data and data flows, respectively. To this end, a possible interesting direction consists of supporting and verifying dynamically, at runtime, the aforementioned requirements, using them to assist orchestration and execution of serverless applications in the Fog. Such runtime support could make use and extend consolidated information flow control techniques [73] as only proposed by [71,72], and as successfully exploited in other fields, such as programming languages [78], web applications [79,80], and databases [81].

Finally, pursuing the main ambition of our survey, we conclude by pointing to future lines of work that fall at the intersection between *all* the three considered perspectives P1–P3:

P1 ∩ P2 ∩ P3 (*Secure FaaS Orchestration in the Fog*)

– *Secure placement of FaaS Orchestration in the Fog* To enable the secure deployment of FaaS orchestrations to Fog infrastructures, it would be interesting to *jointly* consider the (hardware, software and network QoS) requirements of FaaS orchestrations and their security requirements. Security requirements can concern needed (hardware, software and organisational) countermeasures (as in [14]), and policies to secure data flows and external service interactions. To this end, information-flow security permit checking FaaS orchestration for information leaks, by labelling functions with suitable security types. A compatible labelling of computing nodes can then drive function placement in a security-aware manner, while also considering other metrics. Finally, the adaptive deployment of different versions of a serverless function depending on contextual information, as envisioned by Osmotic Computing, is also an interesting line to investigate.
– *Fog platforms for FaaS orchestrations* Realising a platform to securely manage the Monitor-Analyse-Plan-Execute cycle of FaaS orchestrations in Cloud-IoT settings—while monitoring security, QoS and billing—is an interesting open

research challenge. Such a platform will need to be natively multi-tenant and capable of offering suitable views and management tools to all involved stakeholders, i.e. application operators, FaaS service providers and infrastructure providers. All these impose handling security in a holistic and extensible manner so to forbid malicious behaviours from all parties, and to preserve confidentiality and integrity of both data and code. Overall, the availability of new FaaS platforms will both enable a plethora of next-gen IoT applications and create a new flexible market for serverless computing in Fog settings, allowing application operators to specify and have enforced at runtime their target functional and non-functional SLOs.

Acknowledgements This work has been partly supported by the project "*DECLWARE: Declarative methodologies of application design and deployment*"(PRA_2018_66), funded by University of Pisa, Italy, and by the project "*GIÒ: a Fog computing testbed for research & education*", funded by the Department of Computer Science of the University of Pisa, Italy.

Funding Open access funding provided by Università di Pisa within the CRUI-CARE Agreement.

References

1. Baldini I, Castro PC, Chang KS, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A, Suter P (2017) Serverless computing: current trends and open problems. In: Chaudhary S, Somani G, Buyya R (eds) Research advances in cloud computing. Springer, Berlin
2. Jonas E, Schleier-Smith J, Sreekanti V, Tsai C, Khandelwal A, Pu Q, Shankar V, Carreira J, Krauth K, Yadwadkar NJ, Gonzalez JE, Popa RA, Stoica I, Patterson DA (2019) Cloud programming simplified: A berkeley view on serverless computing. CoRR
3. Eyk EV, Toader L, Talluri S, Versluis L, Uta A, Iosup A (2018) Serverless is more: from paas to present cloud computing. IEEE Internet Comput 22:8–17
4. AWS Lambda Releases. https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html. Accessed Jan 2021
5. Bonomi F, Milito R, Natarajan P, Zhu J (2014) Fog computing: a platform for internet of things and analytics. In: Bessis N, Dobre C (eds) Big data and internet of things: A roadmap for smart environments. Springer, Berlin
6. Habibi P, Farhoudi M, Kazemian S, Khorsandi S, Leon-Garcia A (2020) Fog computing: a comprehensive architectural survey. IEEE Access 8:69105–69133
7. Mahmud R, Srirama SN, Ramamohanarao K, Buyya R (2019) Quality of experience (QoE)-aware placement of applications in fog computing environments. J. Parallel Distributed Comput 132:190–203
8. Guerrero C, Lera I, Juiz C (2019) Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. Future Gener Comput Syst 97:131–144
9. Brogi A, Forti S, Ibrahim A (2018) Optimising QoS-assurance, resource usage and cost of fog application deployments. In: CLOSER (Selected Papers)
10. Großmann M, Ioannidis C, Le DT (2019) Applicability of serverless computing in fog computing environments for iot scenarios. In: UCC companion, association for computing machinery

11. Raghavendra MS, Chawla P (2018) A review on container-based lightweight virtualization for fog computing. In: ICRITO
12. von Leon D, Miori L, Sanin J, El Ioini N, Helmer S, Pahl C (2019) A lightweight container middleware for edge cloud architectures. Fog and edge computing
13. Pfandzelter T, Bermbach D (2020) tinyfaas: A lightweight faas platform for edge environments. In: ICFC
14. Forti S, Ferrari G-L, Brogi A (2020) Secure cloud-edge deployments, with trust. Future Gener Comput Syst 102:775–788
15. Ni J, Zhang K, Lin X, Shen X (2017) Securing fog computing for internet of things applications: challenges and solutions. IEEE Commun Surv Tutor 20:601–628
16. Vaquero LM, Cuadrado F, Elkhatib Y, Bernal-Bernabe J, Srirama SN, Zhani MF (2019) Research challenges in nextgen service orchestration. Future Gener Comput Syst 90:20–38
17. Scheuner J, Leitner P (2020) Function-as-a-service performance evaluation: A multivocal literature review. J Syst Softw 170:110708
18. López PG, Artigas MS, París G, Pons DB, Ollobarren ÁR, Pinto DA (2018) Comparison of faas orchestration systems. In: UCC Companion
19. Wang L, Li M, Zhang Y, Ristenpart T, Swift M (2018) Peeking behind the curtains of serverless platforms. In: USENIX ATC
20. Yussupov V, Soldani J, Breitenbücher U, Brogi A, Leymann F (2020) Faasten your decisions: Classification framework and technology review of function-as-a-service platforms. CoRR
21. AWS Step Functions. https://docs.aws.amazon.com/step-functions/index.html. Accessed Jan 2021
22. AWS IoT Greengrass. https://aws.amazon.com/it/greengrass/. Accessed Jan 2021
23. Microsoft Azure Functions. https://azure.microsoft.com/services/functions/. Accessed Jan 2021
24. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/. Accessed Jan 2021
25. Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge/. Accessed Jan 2021
26. Google Cloud Functions. https://cloud.google.com/functions. Accessed Jan 2021
27. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed Jan 2021
28. IBM Cloud Functions. https://cloud.ibm.com/functions/. Accessed Jan 2021
29. IBM Composer. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer. Accessed Jan 2021
30. Fission. https://fission.io/. Accessed Jan 2021
31. Kubeless. https://kubeless.io/
32. Knative. https://knative.dev/. Accessed Jan 2021
33. OpenFaaS. https://www.openfaas.com/. Accessed Jan 2021
34. Nuclio. https://nuclio.io/. Accessed Jan 2021
35. Baldini I, Cheng P, Fink SJ, Mitchell N, Muthusamy V, Rabbah R, Suter P, Tardieu O (2017) The serverless trilemma: function composition for serverless computing. In: Onward!
36. Jangda A, Pinckney D, Brun Y, Guha A (2019) Formal foundations of serverless computing. In: Proceedings of the ACM on programming languages, 3(OOPSLA)
37. Gabbrielli M, Giallorenzo S, Lanese I, Montesi F, Peressotti M, Zingaro SP (2019) No more, no less—a formal model for serverless computing. In: COORDINATION
38. López PG, Arjona A, Sampé J, Slominski A, Villard L (2020) Triggerflow: trigger-based orchestration of serverless workflows. In: DEBS
39. Eismann S, Grohmann J, Eyk EV, Herbst N, Kounev S (2020) Predicting the costs of serverless workflows. In: ICPE
40. Yang Q, Jin R, Gandhi N, Ge X, Khouzani HA, Zhao M (2020) Edgebench: a workflow-based benchmark for edge computing. CoRR
41. Gerasimov N (2019) The DSL for composing functions for FaaS platform. In: SEIM-2019, p 13
42. Persson P, Angelsmark O (2017) Kappa: serverless iot deployment. In: WOSC@Middleware
43. Moritz P, Nishihara R, Wang S, Tumanov A, Liaw R, Liang E, Elibol M, Yang Z, Paul W, Jordan MI, Stoica I (2018) Ray: a distributed framework for emerging AI applications. In: OSDI
44. Rinaldi L, Torquati M, De Sensi D, Mencagli G, Danelutto M (2020) Improving the performance of actors on multi-cores with parallel patterns. Int J Parallel Program 40:692–712
45. Gadepalli PK, Peach G, Cherkasova L, Aitken R, Parmer G (2019) Challenges and opportunities for efficient serverless computing at the edge. In: SRDS
46. Hall A, Ramachandran U (2019) An execution model for serverless functions at the edge. In: IoTDI

47. Cheng B, Fürst J, Solmaz G, Sanada T (2019) Fog function: Serverless fog computing for data intensive iot services. In: SCC
48. Baresi L, Mendonça DF (2019) Towards a serverless platform for edge computing. In: ICFC
49. Baresi L, Mendonça DF, Quattrocchi G (2019) PAPS: A framework for decentralized self-management at the edge. In: ICSOC
50. Cicconetti C, Conti M, Passarella A, Sabella D (2020) Toward distributed computing environments with serverless solutions in edge systems. IEEE Commun Mag 58:40–46
51. Mortazavi SH, Salehe M, Gomes CS, Phillips C, de Lara E (2017) Cloudpath: a multi-tier cloud computing framework. In: SEC
52. Montresor A, Jelasity M (2009) Peersim: A scalable p2p simulator. In P2P, pp 99–100
53. Angelsmark O, Persson P (2016) Requirement-based deployment of applications in calvin. In: InterOSS@IoT
54. Pinto D, Dias JP, Ferreira HS (2018) Dynamic allocation of serverless functions in IoT environments. In: EUC
55. Das A, Imai S, Wittie MP, Patterson S (2020) Performance optimization for edge-cloud serverless platforms via dynamic task placement. In: CoRR
56. Aske A, Zhao X (2018) Supporting multi-provider serverless computing on the edge. In: ICPP
57. Cho C, Shin S, Jeon H, Yoon S (2020) Qos-aware workload distribution in hierarchical edge clouds: A reinforcement learning approach. IEEE Access
58. Cicconetti C, Conti M, Passarella A (2020) A decentralized framework for serverless edge computing in the internet of things. IEEE Trans Netw Serv Manag
59. Rausch T, Rashed A, Dustdar S (2021) Optimized container scheduling for data-intensive serverless edge computing. Future Gener Comput Syst 114:259–271
60. Stojcev MK (2005) Alberto leon-garcia, indra widjaja, communication networks: fundamental concepts and key architectures, second edition, Mcgraw Hill Higher Education, Boston, 2004, ISBN 0-07-119848-2. hardcover, pp 900, plus XXVII. Microelectron. Reliab
61. Fan Q, Ansari N (2018) Application aware workload allocation for edge computing-based IoT. IEEE Internet Things J 5:2146–2153
62. Rausch T, Hummer W, Muthusamy V, Rashed A, Dustdar S (2019) Towards a serverless platform for edge AI. In: HotEdge
63. Bermbach D, Maghsudi S, Hasenburg J, Pfandzelter T (2020) Towards auction-based function placement in serverless fog platforms. In: ICFC
64. Forti S, Gaglianese M, Brogi A (2021) Lightweight self-organising distributed monitoring of Fog infrastructures. Future Gener Comput Syst 114:605–618
65. Buzachis A, Fazio M, Celesti A, Villari M (2019) Osmotic flow deployment leveraging faas capabilities. In: IDCS
66. Trach B, Oleksenko O, Gregor F, Bhatotia P, Fetzer C (2019) Clemmys: towards secure remote execution in FaaS. In: SYSTOR
67. Alder F, Asokan N, Kurnikov A, Paverd A, Steiner M (2019) S-faas: Trustworthy and accountable function-as-a-service using intel SGX. In: CCSW@CCS. ACM, pp 185–199
68. Brenner S, Kapitza R (2019) Trust more, serverless. In: Hershcovitch M, Goel A, Morrison A, (eds) SYSTOR
69. Gjerdrum AT, Johansen HD, Brenna L, Johansen D (2019) Diggi: A secure framework for hosting native cloud functions with minimal trust. In: (TPS-ISA), pp 18–27
70. Qiang W, Dong Z, Jin H (2018) Se-lambda: Securing privacy-sensitive serverless applications using SGX enclave. In: SecureComm 2018, Part I
71. Alpernas K, Flanagan C, Fouladi S, Ryzhyk L, Sagiv M, Schmitz T, Winstein K (2018) Secure serverless computing using dynamic information flow control. ACM Program. Lang., no. OOPSLA, Proc
72. Datta P, Kumar P, Morris T, Grace M, Rahmati A, Bates A (2020) Valve: securing function workflows on serverless computing platforms. In: WWW '20
73. Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE J Sel Areas Commun 21:5–19
74. Boucher S, Kalia A, Andersen DG, Kaminsky M (2018) Putting the "micro" back in microservice. In: USENIX ATC
75. Gadepalli PK, McBride S, Peach G, Cherkasova L, Parmer G (2020) Sledge: a serverless-first, light-weight wasm runtime for the edge. In: Middleware

76. Brogi A, Forti S, Guerrero C, Lera I (2020) How to place your apps in the fog: state of the art and open challenges. Softw Pract Exp 50:719–740
77. Mann ZÁ (2020) Secure software placement and configuration. Future Gener Comput Syst 110:243–253
78. Hedin D, Birgisson A, Bello L, Sabelfeld A (2014) Jsflow: tracking information flow in javascript and its APIs. In: SAC
79. Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo, Securing web application code by static analysis and runtime protection. In: WWW, 2004
80. Hedin D, Sabelfeld A (2015) Web application security using JSFlow. In: SYNASC
81. Guarnieri M, Balliu M, Schoepe D, Basin DA, Sabelfeld A (2019) Information-flow control for database-backed applications. In: EuroS&P