Placing FaaS in the Fog, Securely

Alessandro **Bocci**^{1}, Stefano **Forti**^{1}, Gian-Luigi **Ferrari**^{1} and Antonio **Brogi**^{1}

^{1}*University of Pisa, Pisa, Italy*

Abstract

Placing FaaS applications onto Fog infrastructures is an open problem presenting various challenges. It requires considering hardware and software requirements of single functions as well as Quality of Service requirements of the overall application. In this article, we propose a declarative methodology to address the placement of FaaS applications onto Fog infrastructures, supported by a running prototype. Our methodology considers hardware and software requirements, and latency constraints on function-function and function-service interactions. Particular attention is given to information flow security constraints and trust relations among the involved stakeholders, to rank eligible output placements. A lifelike motivating example from augmented reality is used to showcase the prototype.

Keywords

Information-Flow Security, Function-as-a-Service, FaaS, Fog

1. Introduction

The *Function-as-a-Service* (FaaS) model, started up by Amazon with AWS Lambda [1] in 2015, was primarily intended for Cloud settings. This brought an evolution in how software is organised and deployed in the Cloud [2]. Meanwhile, *Fog computing* [3, 4] emerged to exploit processing, storage and networking along the Cloud-IoT continuum – from personal devices, through network switches, to Cloud datacentres. The Fog is designed to support the stringent Quality of Service (QoS) requirements (e.g. on latency, security) of next-gen IoT applications as well as to improve the Quality of Experience (QoE) of the existing ones [5, 6, 7]. Indeed, Fog Computing 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 enable prompter and insightful responses to sensed events.

The FaaS paradigm supports the orchestration of stateless, event-triggered functions, abstracting from the target deployment stack, and possibly interacting with external services (e.g. databases, ML engines, geo-localisation engines) [8, 9]. Event-based serverless functions are naturally suited to define computation on IoT data, containerisation technologies are lightweight and seamlessly deployable on Fog nodes, and on-demand execution of functions can lead to improved usage of resource-constrained devices [10, 11, 12]. The natural combination of FaaS and Fog computing brings with it various challenges of both foundational and technological nature. In particular, a key issue concerns the definition of novel and effective *strategies to place serverless functions over the pervasive Fog computing nodes*. The placement problem, i.e.

CEUR Workshop Proceedings (CEUR-WS.org)

finding the mappings from functions to Fog nodes, requires to consider *hardware, software and QoS requirements* of single functions as well as their composition into complex applications, and requirements on function *bindings with external services*. As other application placement problems in the Fog, deciding on the above is NP-hard, thus worst-case exp-time [13].

Speaking of QoS, FaaS placement requires constraining the function-function and function-service *latencies*, which are crucial to support latency-sensitive applications [14]. Furthermore, guaranteeing a high level of *security* is an important QoS aspect to consider [15, 16, 17] as a security flaw in a single function (or in its interactions with external services) may induce an overall security flaw of the function composition. Placing FaaS to Fog nodes therefore needs to consider different security contexts, the security level of data flowing between functions, and *trust* relations among application operators, and infrastructure and external service providers. All these are utterly important to avoid data leaks, e.g. by placing functions handling sensitive data onto insecure or untrusted nodes, or by binding a function to a service managed by an untrusted provider [18].

This article builds over our previous work on secure and trust-aware multiservice application placement in the Fog [17]. Our approach permits to express security requirements of IoT applications, as well as infrastructure security capabilities taking into account a trust model of the involved entities. An explainable assessment of the security level of the possible deployments is automatically derived.In this article, we showcase an extension of [17] over a lifelike motivating example. The extension includes three main directions:

(*a*) The definition of a declarative model to represent function requirements including security policies, and trust relations,

(*b*) The (formal) logical definition of the function placement problem, which exploits information-flow security [19] to determine functions' security contexts, and commutative semirings to model and propagate trust [20], and

(*c*) A prototype[1] of the above, FaaS2Fog, implemented by relying upon an algebraic extension of Prolog, to determine eligible placements (i.e. mappings from functions to Fog nodes) and function-service bindings (i.e. mappings from functions' service requirements to actual service instances).

We consider a simple orchestration mechanism where FaaS services are chained together as sequences of functions. We call it *FaaS chains*, and they represent a first step towards placing more complex FaaS orchestrations (e.g. those of OpenWhisk [21]). FaaS2Fog employs information-flow security [19] to avoid information leaks to untrusted/insecure nodes or services, and checks all hardware, software and latency chains requirements.

Our approach features two desirable properties to support the placement of FaaS in the Fog. First, it is *declarative*, hence more concise, easier to understand and maintain compared with procedural solutions, and it offers high flexibility and extensibility, to accommodate the ever-changing needs of Fog scenarios. Second, it is intrinsically *explainable* as it derives proofs for input queries by relying on Prolog resolution engines, and it can be easily extended to justify *why* a certain placement decision was made, in the spirit of explainable AI [22].

The rest of this article is organised as follows. After describing a lifelike motivating example from augmented reality (Sect. 2), we illustrate FaaS2Fog methodology (Sect. 3), used to solve

---

[1]Open sourced and freely available at: https://github.com/di-unipi-socc/FaaS2Fog.

and discuss the motivating example (Sect. 4). After presenting closely related work (Sect. 5), we conclude by pointing to some directions for future work (Sect. 6).

2. Motivating Example

This section illustrates a lifelike motivating example to highlight the challenges related to the FaaS chain placement problem. Consider an Augmented Reality (AR) application trying to avoid people gatherings in commercial areas of a city centre, so to contrast the spreading of Covid-19, by *load-balancing* people presence across similar shops. When users point with their smartphones the entrance of a shop they are about to enter, the application renders over the video 3D information about the number of people inside and its overall capacity. If a high risk of gathering is detected, the application suggests alternative similar locations to the users by giving them 3D AR directions, along with incentives (e.g. discounts, special offers) to move away from the gathering area. Such an application is implemented by a FaaS chain (Fig. 1):

1. f_{Login}, which authenticates the user into the service,
2. f_{Shop}, which recognises the shop entrance in the framed images, so to discriminate among physically close activities,
3. f_{Geo}, which identifies the shop by its image and coordinates via an external map service, also retrieving all shop information,
4. f_{Gather}, which computes the risk of gathering (based on crowd-sensed information), individuates alternative shops and incentives to move there,
5. f_{AR}, which renders AR information over the framed video footage and sends it back to the mobile client.

For each function in the chain, Fig. 1 lists the software requirements in terms of required runtime support, and the hardware requirements. For instance, function f_{Shop} needs Python3 and NumPy to be available on the deployment node, along with 2 GB of RAM, and 4 vCPUs at 1.5 GHz at least.
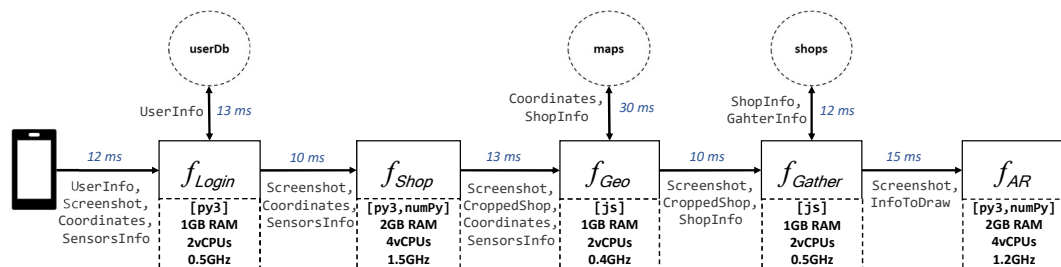
Figure 1: Example FaaS chain.

Each function connects to the type of external service(s) it will need to bind with at runtime (represented as dashed circles). As an example, function f_{Login} needs to bind to an instance of the userDB service. Function-function and function-service interactions (represented as arrows) are annotated with the parameters they exchange and with the maximum end-to-end latency they tolerate between their deployment nodes. For instance, f_{Login} passes to f_{Shop} a Screenshot of the framed shop, with its geographical Coordinates, and SensorsInfo from the client smartphone. Such interaction tolerates a latency of at most 10 ms.

To prevent sensible data from leaking, chain input parameters are labelled with suitable security types, as those in the lattice of Fig. 2, modelling the security level pertaining to certain sensible data from top (i.e. secret) to low (i.e. public).

Those security labels propagate with data along the function chain (and across external services), determining a security context for each function (and service), based on the data it will process. For instance, f_{Login} inputs UserInfo, Screenshot, Coordinates, SensorsInfo can be labelled by the tuple (top, low, low, low), assigning to such function the highest type among those, i.e. top. Se-

top
|
medium
|
low

Figure 2: Example security lattice.

curity types propagate to the outputs (Screenshot, Coordinates, SensorsInfo) with the labels (low, low, low), which will be used in turn to determine the (low) security context of f_{Shop}. Finally, we assume that chain triggers come from event generators – invoking suitable orchestrator API – that are connected to infrastructure nodes. In our example, the event generator is the client application, which we assume to connect to the closest available node. Functions in each chain are executed sequentially, and the final result of the computation is returned to the client application.

Fig. 3 sketches the target (city centre) Fog infrastructure, upon which the chains described above will be placed and run. Nodes are owned and managed by different providers, the information we need are their software and hardware capabilities, the services they host, and their security countermeasures, expressed as per the taxonomy of Fig. 4 (as in [17]). The countermeasures will be used to specify (*i*) the security policies in force and (*ii*) to assign security labels to nodes.

Service Id	Service Type	Service Provider
myUsers	userDB	appOp
cMaps	maps	cloudProvider
openM	maps	openS
myShop	shops.	appOp

cloudNode
(cloudProvider)
[all]
∞GB, ∞CPUs, ∞GHz
[all]

cMaps
(maps)

30 ms

ispRouter
(telco)
[py3,js,numPy]
4GB, 4vCPUs, 2GHz
[antiTamp,pubKeyE]

myUsers
(userDB)

5 ms   *7 ms*   *8 ms*   *6 ms*

myShop
(shops)

switch
(university)
[py3,js]
2GB, 2vCPUs, 2GHz
[pubKeyE]

southAntenna
(telco)
[py3,numPy]
2GB, 4vCPUs, 1.5GHz
[pubKeyE]

12 ms

northAntenna
(telco)
[js]
2GB, 2vCPUs, 1.5GHz
[antiTamp,pubKeyE]

private1
(privateCitizen1)
[js]
1GB, 2vCPUs, 2.5GHz
[]

openM
(maps)

5 ms   *7 ms*   *7 ms*

labServer
(university)
[py3,js,numPy]
4GB, 4vCPUs, 2GHz
[antiTamp,pubKeyE]

7 ms

officeServer
(university)
[py3]
1GB, 2vCPUs, 1GHz
[]

private2
(privateCitizen2)
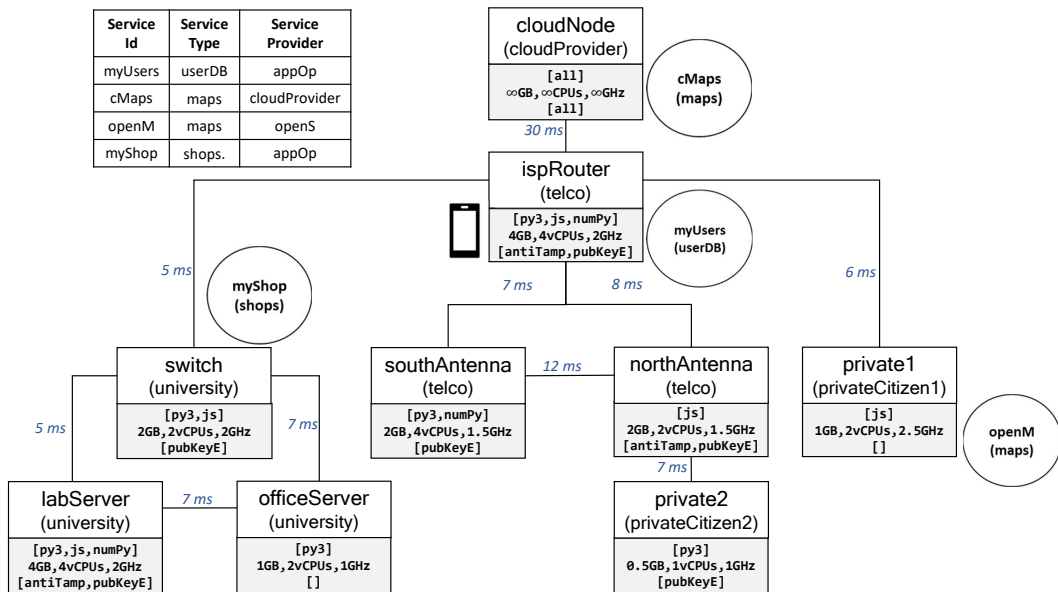[py3]
0.5GB, 1vCPUs, 1GHz
[pubKeyE]

Figure 3: Example Fog infrastructure.

As an example, node switch, owned by university, supports Python and Javascript code, and

features 2 GB of RAM, 2 vCPUs at 2 GHz, and public key encryption as security countermeasure. Services instances are managed by their providers that detail information about service types and deployment nodes. For instance, service `cMaps` provided by `cloudProvider` is of type `maps` and is deployed to node `cloudNode`. Links between nodes are annotated with their end-to-end latency, e.g. the latency between `ispRouter` and `northAntenna` is 8 ms.
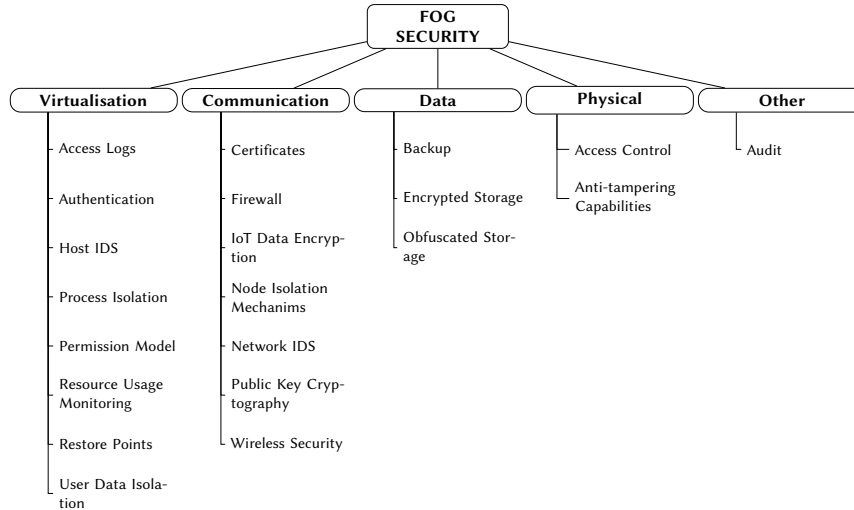
Figure 4: Security capabilities for Fog computing.

Deploying the chain in Fig. 1 onto infrastructure of Fig. 3 requires satisfying all functions' software, hardware and latency requirements, and determining a suitable binding from functions to the service instances they need. Furthermore, function labels and node labels should be compatible, i.e. a node can host a function only if it features an equal or higher security label. As an example, the application operator can establish that a node featuring *public key encryption* without *anti-tampering* is labelled as `medium` security. This means that the node can only support functions labelled either `low` or `medium`. Note that this is required to avoid leaking data from a higher to a lower security context.

The problem we aim at solving in the above scenario is as follows:

> *Is there any eligible placement and service binding of* `chainGath` *over the available Fog infrastructure, such that all software, hardware, latency, external service, security requirements are met, and data leaks are avoided? Can trust relations among involved stakeholders help in evaluating the quality of eligible placements?*

In Sect. 4 we will show how our methodology and prototype FaaS2Fog can be used to answer the above questions.

3. Methodology

In this section, we describe our methodology to place FaaS chains to Fog infrastructure. We illustrate both the modelling (Sect. 3.1) and the declarative solution (Sect. 3.2) of the problem,

showing the main Prolog[2] facts where clarifications are needed.

3.1. Modelling Applications, Infrastructures and Trust

Declaring FaaS Chains. Application operators declare the requirements of a function `F` they will chain into FaaS applications using

```
functionReqs(F, SoftwareRequirements, HardwareRequirements, ServiceTypeRequirements).
```

Such requirements include `SoftwareRequirements` and `HardwareRequirements`, as well as `Service-TypeRequirements` used to bind at run time service instances to be invoked by the function.

The function behaviour of a function `F` is defined by a relation between security types of input data (`Inputs`), output data (`Outputs`) and data exchanged with external services (`Service-Interactions`). The declaration of function behaviours is

```
functionBehaviour(F, Inputs, ServiceInteractions, Outputs):- ...
```

We assume that such information can be defined manually, or obtained by exploiting static analyses and type systems (see e.g. [23, 24]).

A Function chain `FChain` managed by `Operator` is declared as

```
functionChain(FChain, Operator, EventTrigger, Functions, Latencies).
```

where are included the list of `Functions` forming the chain, the FaaS `Operator`, the `EventTrigger` associated with the chain, and the maximum tolerated `Latencies` from the trigger to the first chain function, and between consecutive chained functions. Chain triggers are denoted by pairs containing the trigger source and the list of security labels of its parameters (e.g. `[top,low,low,low]`). Those security labels will match the input types of the first function and propagate along the chain according to declared function behaviours. Chained functions are denoted by pairs containing a function identifier and a list of actual service instances each function will bind to. Bindings either specify a certain service instance identifier or are left unbound, meaning that any service of the required type can satisfy the requirement.

Application operators also declare a lattice of the security types to define their functions and function chains, as the one in Fig. 2, using

```
g_lattice_higherThan(Higher, Lower).
```

to represent each ordered pair of the lattice with the `Higher` and the `Lower` security types.

Declaring Fog Infrastructures. Infrastructure node are declared as

```
node(NodeId, ProvId, SecCaps, SWCaps, HWCaps).
```

including the `NodeId` (e.g. its IP address) and the security, software and hardware capabilities featured by the node (viz. `SecCaps`, `SWCaps` and `HWCaps`), which enable FaaS2Fog to check the latter against function requirements. Security capabilities are lists of literals from Fig. 4.

End-to-end links between nodes (`NId1` and `NId2`) are associated with the average `Latency` in milliseconds experienced between the nodes, declared as

---

[2]A Prolog program is a finite set of *clauses* of the form: `a :- b1, ... , bn.` stating that a holds when b1 ∧ ⋯ ∧ bn holds, where n≥0 and a, b1, ..., bn are atomic literals. Clauses with empty condition are also called *facts*. Prolog variables begin with upper-case letters, lists are denoted by square brackets, and negation by `\+`. Conventionally, a Prolog predicate `pred` with N arguments is indicated as `pred/N`.

```
g_link(NId1, NId2, Latency).
```

External services, running on available nodes, are declared by service providers as

```
service(ServiceId, ServiceProvider, ServiceType, NodeId).
```

where are specified the `ServiceId`, the identifier of the `ServiceProvider`, the `ServiceType`, and the `NodeId` of the service host node.

The declaration of an event generator `EG` is

```
eventGenerator(EG, EventType, NodeId).
```

and it includes the `EventType` of generated triggers it can generate as well as the `NodeId` of the node with which it is connected. It is worth noting that data about infrastructure capabilities, services and event generators can be obtained by exploiting available monitoring tools targeting Fog settings (e.g. FogMon [25], or one of the tools surveyed in [26]).

Declaring Security Policies. The security context of nodes and external services is determined according to the security policies specified by application operators. Node and service labelling is performed as per the same security types of functions parameters to guarantee that types can be checked against each other and to enforce that functions are placed onto nodes and interacts with services that feature at least their security type. Node labelling is specified by the declaration

```
assignNodeLabel(Node, SecurityContextLabel) :- ...
```

meaning that the `SecurityContextLabel` is assigned to the `Node` if all conditions of the right hand-side of the rule hold. For example, a node is labelled `top` only if such node features both anti-tampering and public key encryption among its security capabilities. Analogously, external services are labelled by defining the security policies based on the available service information.

Declaring Trust Opinions. All involved stakeholders (i.e. application operators, infrastructure providers, service providers) can declare trust opinions towards each other. Following [20], trust relations are modelled as pairs $(T, C) \in [0, 1] \times [0, 1]$ where T represents a level of trust (the higher the better) and C the confidence in (i.e. the quality of) such value T, based on trust monitoring data. By employing a dialect of Prolog, viz. α-Problog [27], trust opinions are declared as facts annotated with (T, C), in the form

```
(T,C)::trustOpinion(Stakeholder1, Stakeholder2).
```

where `Stakeholder1` declares her trust level and confidence toward `Stakeholder2`.

3.2. Placing FaaS Chains onto Fog Infrastructures

In this section, we briefly illustrate how `FaaS2Fog` determines FaaS application placements, based on the modelling described in the previous section. Fig. 5 lists the `faas2fog/2` predicate, giving an overview of the overall functioning of our prototype. After retrieving a function chain by its identifier `CId` (line 2), `faas2fog/2` retrieves also an event generator associated to the chain, if any (line 3). Then, after propagating security types along the FaaS chain to determine the security contexts needed by each function (line 4), `faas2fog/2` determines an eligible FaaS chain `Placement` satisfying all set requirements (line 5).

```
1  faas2fog(CId, Placement):-
2      functionChain(CId, AppOp, (GeneratorId,TriggerTypes), FList, LatReqs),
3      eventGenerator(GeneratorId, GeneratorNode),
4      typePropagation(TriggerTypes, FList, TFList),
5      mapping(AppOp, TFList, LatReqs, GeneratorNode, Placement).
```

Figure 5: The `faas2fog/2` predicate.

FaaS2Fog carries out three main steps to determine eligible placement of FaaS application:

1. Propagating Security Types. First, the `typePropagation/3` predicate listed in Fig. 6 performs security type propagation along the chain to be placed. By recursively scanning the list of chained functions, `typePropagation/3` instantiates the security types of all (input, output, external) parameters using the `functionBehaviour/4` predicate, declared by the application operator for each function (line 3). The current function `F` is then assigned the highest security type `FType` among the security labels of the parameters it handles determined by `highestType/2` (lines 4–6). The third argument of `typePropagation/3` accumulates this new information for all functions. Newly determined output types are used as input types for the next function in the chain, and so on so forth.

```
1  typePropagation(_, [], []).
2  typePropagation(InTypes,[(F,FServices)|FunctionList],
                     [(F,FServices,FType)|TypedFunctionList]):-
3      functionBehaviour(F, InTypes, InteractionsTypes, OutTypes),
4      append(InTypes, InteractionsTypes, TempTypes),
       append(TempTypes, OutTypes, AllTypes), sort(AllTypes, AllTypesSorted),
5      highestType(AllTypesSorted,FType),
6      typePropagation(OutTypes, FunctionList, TypedFunctionList).
```

Figure 6: The `typePropagation/3` predicate.

2. Resource- & QoS-aware Placement. After this step, the `mapping/7` (Fig. 7) scans the list of typed functions `TFList` and determines an eligible placement for each of them, until the list is empty (line 1). For each function `F` to be placed, `mapping/7` non-deterministically selects a candidate node `N` (line 3) and checks the latency constraint between `N` and the node of the previously allocated function[3] (line 4). Then, software and hardware requirements of `F` are checked against `N` capabilities (lines 5–6). Notably, the predicate `hwReqsOK/5` also checks whether cumulative hardware requirements allow placing `F` onto `N`, i.e. whether, due to the allocation of previous functions `OldAllocHW`, the current capacity of `N` can still host `F` as well. If this is possible, the hardware allocation is updated by summing the hardware required by `F` to the allocated hardware at `N` into `AllocHW`. Consequently, the `compatibleNodeType/2` predicate checks whether the security context of `N` (determined by the `assignNodeLabel/2` provided by the application operator) is compatible with the security context required by the function. Finally, the `bindServices/6` predicate determines if bindings to external services comply to security and latency constraints of the function to be placed. Note that when a binding requirement is left unbound, `bindServices/6` can determine eligible services (if any) that satisfy the requirements. Each placement and binding for `F` is accumulated in the last parameter of `mapping/7`, in tuples like `on(F,N,FBinding)`. Such result will be returned by `faas2fog/2`.

3. Trust Propagation. FaaS2Fog also considers trust relations by using the semiring-based

---

[3]In case of the first function, `mapping/7` checks the latency from the event generator.

```
1    mapping(_, [], [], _, AllocHW, AllocHW, []).
2    mapping(AppOp, [(F,FServices,FType)|TFList], [ReqLatency|LatReqs],
            PreviousNode, OldAllocHW, NewAllocHW, [on(F,N,FBinding)|P]):-
3       getNode(N, SWCaps, HWCaps),
4       link(PreviousNode, N, FeatLatency), FeatLatency =< ReqLatency,
5       functionReqs(F, SWReqs, HWReqs, FServicesReqs),
6       swReqsOK(SWReqs, SWCaps),
7       hwReqsOK(HWReqs, HWCaps, N, OldAllocHW, AllocHW),
8       compatibleNodeType(FType,N),
9       bindServices(AppOp, N, FServices, FType, FServicesReqs, FBinding),
10      mapping(AppOp, TFList, LatReqs, N, AllocHW, NewAllocHW, P).
```

Figure 7: The `mapping/7` predicate.

modelling of [20] to aggregate opinions from different stakeholders. Trust propagation is conditioned to a maximum radius of 3 hops in the trust graph, assuming that each stakeholder trusts herself with a $(1,1)$ opinion. Following the model of [20], we propagate trust from A to B as follows: opinions along paths from A to B are multiplied, while opinions across paths from A to B are summed. We employ the multiplication ($\otimes$) and sum ($\oplus$) operations of Fig. 8, implemented as in [17] via α-Problog.

Trust relations are checked between the application and the node operator where functions are placed. Analogously, trust relations are checked between the application operator and external service providers. Relying on α-Problog, FaaS2Fog annotates each output eligible placement with its overall trust assessment computed as described above.

$$\langle T,C \rangle \otimes \langle T',C' \rangle = \langle TT',CC' \rangle$$

$$\langle T,C \rangle \oplus \langle T',C' \rangle = \begin{cases} \langle T,C \rangle & \text{if } C > C' \\ \langle T',C' \rangle & \text{if } C' > C \\ \langle \max\{T,T'\},C \rangle, & \text{if } C = C' \end{cases}$$

Figure 8: Semiring $\otimes$ and $\oplus$ operations.

4. Motivating Example Revisited

In this section, we solve the motivating example[4] of Sect. 2 by answering the questions raised therein. Security policies to label services specify that all the services provided by `appOp` are labelled `top`, the `maps` services provided by `cloudProvider` are labelled `medium`, and all the other services are labelled `low`. We assume that input parameters generated by `userDevices` triggering the chain are labelled `[top,low,low,low]`, only considering user's data as sensitive information.

Given these ingredients, we simply query the main predicate of FaaS2Fog, `faas2fog(chainGath, Placement)`, to determine eligible placements for the FaaS chain. FaaS2Fog matches trigger types with the input labels of function f_{Login} (`UserInfo`, `Screenshot`, `Coordinates`, `SensorsInfo`) and, by propagation, it determines the labelling of functions[5] as: $(f_{Login},$ `top`$)$, $(f_{Shops},$ `low`$)$, $(f_{Geo},$ `low`$)$, $(f_{Gather},$ `medium`$)$, and $(f_{AR},$ `medium`$)$. Afterwards, eligible placements and service bindings meeting all software, hardware, latency, and security requirements of the chain of Fig. 1 are looked for by the placement phase. Table 1 summarises all obtained results. They are 9 eligible placements

---

[4]Full example code at: https://github.com/di-unipi-socc/FaaS2Fog/tree/main/examples/ITASEC21

[5]Note that, labelling `Coordinates` as `medium`, we would obtain a different labelling, i.e. $(f_{Login},$ `top`$)$, $(f_{Shops},$ `medium`$)$, $(f_{Geo},$ `medium`$)$, $(f_{Gather},$ `medium`$)$, and $(f_{AR},$ `medium`$)$.

and bindings – *P1*-*P9*, one per row – that satisfy all set requirements. Each column in the table corresponds to a chain function and lists the placement node and the service bindings. It is worth mentioning that while service bindings are already specified for f_{Login} (with the service myUserDB) and for f_{Gather} (with the service myGatherS), the binding of f_{Geo} was left unbound and is determined by FaaS2Fog so to meet the required low security context. As a consequence, depending also on function-service latency considerations, such requirement is bound either to the openM (e.g. *P1*) or to the cMaps instance (e.g. *P2*).

	f_{Login}		f_{Shop}		f_{Geo}		f_{Gather}		f_{AR}	
	Node	Bindings	Node	Bindings	Node	Bindings	Node	Bindings	Node	Bindings
P1	ispRouter	myUserDB	labServer	-	ispRouter	openM	switch	myGathS	southAntenna	-
P2	ispRouter	myUserDB	labServer	-	ispRouter	cMaps	switch	myGathS	southAntenna	-
P3	ispRouter	myUserDB	labServer	-	switch	openM	ispRouter	myGathS	southAntenna	-
P4	ispRouter	myUserDB	southAntenna	-	ispRouter	cMaps	switch	myGathS	labServer	-
P5	ispRouter	myUserDB	southAntenna	-	ispRouter	openM	switch	myGathS	labServer	-
P6	ispRouter	myUserDB	southAntenna	-	northAntenna	openM	ispRouter	myGathS	labServer	-
P7	ispRouter	myUserDB	southAntenna	-	private1	openM	ispRouter	myGathS	labServer	-
P8	ispRouter	myUserDB	southAntenna	-	switch	openM	ispRouter	myGathS	labServer	-
P9	labServer	myUserDB	ispRouter	-	labServer	openM	switch	myGathS	southAntenna	-

Table 1
Eligible placements for the example.

We now also consider the trust network of Fig. 9, where links are annotated with (Trust, Confidence) values. For instance, appOp has a trust level of 0.99 with confidence 0.9 toward the telco provider, and a trust level of 0.9 with confidence 0.9 toward the cloudProvider provider. Table 2 lists the trust and confidence values associated to the found placements *P1*–*P9*, listed in Table 1. Trust assessment allows application operators to select one (or more) best candidate placement(s), as well as to set a minimum trust level to meet. For instance, blindly choosing the first result *P1* of Table 1 actually leads to selecting one of the placements which can be trusted less (viz. (0.27, 0.23)) among the eligible ones. By considering trust assessment, the application operator will instead likely choose *P4*, with the best trust level (viz. (0.77, 0.48)).
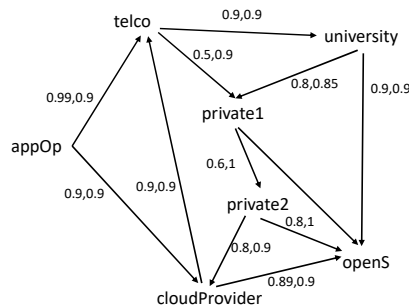
Figure 9
Example trust network.
.

	Trust
P1	(0.27, 0.23)
P2	(0.61, 0.31)
P3	(0.27, 0.23)
P4	(0.77, 0.48)
P5	(0.34, 0.35)
P6	(0.34, 0.31)
P7	(0.17, 0.28)
P8	(0.30, 0.28)
P9	(0.24, 0.21)

Table 2
Ranking of eligible placements.

5. Related Work

In this section, we survey some closely related work proposing declarative solutions to resource management in Cloud or Fog scenarios and techniques to assist the placement of orchestrated serverless functions in the Fog.

Speaking of declarative approaches to resource management, some have been proposed to manage Cloud resources (e.g. [28]) and to improve network usage (e.g. [29]). We employed α-Problog prototypes to assess the security and trust levels of different multiservice application placements [17], and to securely place VNF chains and steer traffic across them [30]. The taxonomy of Fig. 4 to express security policies was introduced in [17], where trust relations modelled via semirings were introduced. Differently from FaaS2Fog, it determines placements of multiservice applications (i.e. not FaaS-based) without considering information flow security, external service interactions, nor hardware and software requirements of services to be placed. Similarly, focussing on Software-defined Networking (SDN) domains, [30] considers neither trust relations nor information flow security, but only security requirements as AND-OR combinations of the taxonomy elements. Closely related to these, [31] devises a (non-declarative) solution to the problem of placing multiservice applications over multiple nodes, configuring hardware/software security controls without considering information-flow nor trust.

Targeting FaaS architectures, Pinto et al. [32] dynamically decide whether to run a serverless function within the local Edge network or in the Cloud, based on monitored data. An Edge proxy is in charge of making such a decision, also considering network failures. On the same line, Das et al. [33] present an edge-based framework to dynamically decide whether to execute a function in the Cloud or locally, by estimating operational costs and improving end-to-end latencies. Similarly, Aske and Zhao [34] present a serverless monitoring and scheduling system to select FaaS providers, based on average execution time, affinity constraints, and costs – possibly considering user-defined scheduling policies. Cho et al. [35] discuss a solution to distribute FaaS tasks over hierarchical Fog infrastructures, by employing a *token bucket* algorithm and einforcement learning to optimise workload distribution and response times.

Cicconetti et al. [36] propose an architecture to realise serverless computing SDN scenarios where network routers assign function execution to edge devices based on arbitrary costs (e.g. latency, bandwidth, energy). The infrastructure is monitored and updated by SDN controllers, and different strategies try to optimise operational costs and load-balancing. More recently, based on their model to annotate function requirements [37], Rausch et al. [38] discuss a Kubernetes-based scheduler to optimise the placement of FaaS in the Fog based on a linear combination of proximity to image registries and to data producers, available node resources and Edge or Cloud locality. Bermbach et al. [39] take an infrastructure provider's perspective to FaaS placement in the Fog, using distributed auctions. Programmers submit functions to target nodes along with a resource bid, based on which nodes decide whether to run functions.

Only [40] and [41] exploit information flow security to check that no leak is present in FaaS orchestrations, at runtime. Differently from FaaS2Fog, security types are not exploited to determine eligible FaaS placements. To the best of our knowledge, none of the previously proposed approaches considers information-flow security, or infrastructure security countermeasures, or trust relations when performing latency-aware placement of FaaS orchestrations in the Fog.

6. Concluding Remarks

This article introduced a life-like example concerning the problem of determining eligible placements of FaaS chains to Fog infrastructures, that is addressed with a declarative methodology and its prototype, FaaS2Fog. Our approach considers hardware, software and latency

requirements, and exploits information-flow security policies as well as infrastructure security capabilities. Eligible placements are ranked by employing semiring-based trust relations among the involved stakeholders. From the architectural viewpoint, FaaS2Fog employs information on the application, on its security types and on infrastructure labelling policies (declared by application operators), data on infrastructure capabilities (collected via distributed monitoring tools, and declared by infrastructure operators), and trust relations (declared by all involved providers).

Our approach shows some qualifying strenghts to address FaaS placement in Fog scenarios. Its declarative nature makes it easier to define FaaS chain requirements and security policies, infrastructure capabilities, and trust relations. Moreover, it naturally supports a flexible approach since it allows the definition of new requirements (e.g. bandwidth, availability of specialised hardware) or security considerations (e.g. provider whitelisting). Finally, placement explanation can be obtained by taking advantage of proof paths determined via the α-Problog engine. Thus, employing explainable placements combined with information-flow and trust assessment can reduce the possibility of leaking critical data to untrusted or insecure parties during the deployment of FaaS chains. On the other hand, FaaS2Fog does not yet handle structured FaaS orchestrations, obtained by combining functions with traditional control mechanisms (e.g. branches, loops). This needs to be addressed to model more complex use cases, also extending information flow analyses to the general case. Additionally, human-readable explanations on *why* a certain placement was (or was not) output would further improve interactions with the users and their trust in systems based on the proposed methodology. Finally, FaaS2Fog incurs in worst-case exp-time to determine valid placements. As serverless functions are usually on-demand and short-lived, it is crucial to assess execution times over larger examples, and to improve on those via heuristics or continuous reasoning [42].

In addition to tackling all aforementioned points, in our future work we plan to extend the methodology underlying FaaS2Fog by formally defining type judgement rules for a feature-complete orchestration language, showing how they can be naturally expressed in α-Problog. Last, we intend to implement a service based on FaaS2Fog and on the Logic-Programming-as-a-Service paradigm [43], and to experiment with actual FaaS deployments in Fog settings by relying on open-source orchestration engines such as OpenWhisk with IBM Composer [21].

Acknowledgments

References

[1] AWS Lambda, https://aws.amazon.com/it/lambda/, Accessed: Apr. 2021.

[2] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, F. Leymann, FaaSten your decisions: A classification framework and technology review of Function-as-a-Service platforms, Journal of Systems and Software 175 (2021) 110906.

[3] F. Bonomi, R. Milito, P. Natarajan, J. Zhu, Fog computing: A platform for internet of things and analytics, in: Big data and internet of things, volume 546, 2014, pp. 169–186.

[4] P. Habibi, M. Farhoudi, S. Kazemian, S. Khorsandi, A. Leon-Garcia, Fog Computing: A Comprehensive Architectural Survey, IEEE Access 8 (2020) 69105–69133.

[5] R. Mahmud, S. N. Srirama, K. Ramamohanarao, R. Buyya, Quality of Experience (QoE)-aware placement of applications in Fog computing environments, J. Parallel Distributed Comput. 132 (2019) 190–203.

[6] C. Guerrero, I. Lera, C. Juiz, Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures, Future Gener. Comput. Syst. 97 (2019) 131–144.

[7] A. Brogi, S. Forti, A. Ibrahim, Optimising QoS-assurance, Resource Usage and Cost of Fog Application Deployments, in: CLOSER (Selected Papers), CCIS, volume 1073, 2018, pp. 168–189.

[8] I. B. et al., Serverless computing: Current trends and open problems, in: Res. Adv. in Cloud Comp., 2017, pp. 1–20.

[9] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, D. A. Patterson, Cloud Programming Simplified: A Berkeley View on Serverless Computing, CoRR abs/1902.03383 (2019).

[10] M. S. Raghavendra, P. Chawla, A review on container-based lightweight virtualization for fog computing, in: ICRITO, 2018, pp. 378–384.

[11] D. von Leon, L. Miori, J. Sanin, N. El Ioini, S. Helmer, C. Pahl, A lightweight container middleware for edge cloud architectures, Fog and edge computing (2019) 145–170.

[12] T. Pfandzelter, D. Bermbach, tinyfaas: A lightweight faas platform for edge environments, in: ICFC, 2020, pp. 17–24.

[13] A. Brogi, S. Forti, QoS-Aware Deployment of IoT Applications Through the Fog, IEEE Internet Things J. 4 (2017) 1185–1192.

[14] M. Großmann, C. Ioannidis, D. T. Le, Applicability of serverless computing in fog computing environments for iot scenarios, in: UCC Companion, 2019, pp. 29–34.

[15] J. Ni, K. Zhang, X. Lin, X. Shen, Securing fog computing for internet of things applications: Challenges and solutions, IEEE Comm. Surveys & Tutorials 20 (2017) 601–628.

[16] L. M. Vaquero, F. Cuadrado, Y. Elkhatib, J. Bernal-Bernabe, S. N. Srirama, M. F. Zhani, Research challenges in nextgen service orchestration, Future Gener. Comput. Syst. 90 (2019) 20–38.

[17] S. Forti, G. L. Ferrari, A. Brogi, Secure cloud-edge deployments, with trust, Future Gener. Comput. Syst. 102 (2020) 775–788.

[18] A. Bocci, S. Forti, G.-L. Ferrari, A. Brogi, Secure FaaS orchestration in the fog: how far are we?, Computing (2021) 1–32.

[19] A. Sabelfeld, A. C. Myers, Language-based information-flow security, IEEE J. Sel. Areas Commun. 21 (2003) 5–19.

[20] S. Bistarelli, S. N. Foley, B. O'Sullivan, F. Santini, Semiring-based frameworks for trust propagation in small-world networks and coalition formation criteria, SCN 3 (2010) 595–610.

[21] IBM OpenWhisk, https://cloud.ibm.com/docs/openwhisk, Accessed: Apr. 2021.

[22] H. Hagras, Toward human-understandable, explainable AI, Computer 51 (2018) 28–36.

[23] F. Pottier, C. Skalka, S. F. Smith, A systematic approach to static access control, ACM Trans. Program. Lang. Syst. 27 (2005) 344–382.

[24] M. Bartoletti, P. Degano, G. L. Ferrari, R. Zunino, Semantics-based design for secure web services, IEEE Trans. Software Eng. 34 (2008) 33–49.

[25] S. Forti, M. Gaglianese, A. Brogi, Lightweight self-organising distributed monitoring of Fog infrastructures, Future Gener. Comput. Syst. 114 (2021) 605–618.

[26] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, V. Stankovski, Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review, J. Syst. Softw. 136 (2018) 19–38.

[27] A. Kimmig, G. Van den Broeck, L. De Raedt, An algebraic Prolog for reasoning about possible worlds, in: AAAI, 2011.

[28] S. Kadioglu, M. Colena, S. Sebbah, Heterogeneous resource allocation in Cloud Management, in: NCA 2016, 2016, pp. 35–38.

[29] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, S. Shenker, Practical declarative network management, in: WREN, 2009, pp. 1–10.

[30] S. Forti, F. Paganelli, A. Brogi, Probabilistic QoS-aware Placement of VNF chains at the Edge, Theory Pract. Log. Program. (2021). In press.

[31] Z. Á. Mann, Secure software placement and configuration, Future Gener. Comput. Syst. 110 (2020) 243–253.

[32] D. Pinto, J. P. Dias, H. S. Ferreira, Dynamic allocation of serverless functions in iot environments, in: EUC, 2018, pp. 1–8.

[33] A. Das, S. Imai, S. Patterson, M. P. Wittie, Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement, in: CCGRID, 2020, pp. 41–50.

[34] A. Aske, X. Zhao, Supporting multi-provider serverless computing on the edge, in: ICPP, 2018, pp. 20:1–20:6.

[35] C. Cho, S. Shin, H. Jeon, S. Yoon, Qos-aware workload distribution in hierarchical edge clouds: A reinforcement learning approach, IEEE Access 8 (2020) 193297–193313.

[36] C. Cicconetti, M. Conti, A. Passarella, A decentralized framework for serverless edge computing in the internet of things, IEEE Trans. Netw. Serv. Manag. (2020) 1–14.

[37] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, S. Dustdar, Towards a serverless platform for edge AI, in: HotEdge, 2019.

[38] T. Rausch, A. Rashed, S. Dustdar, Optimized container scheduling for data-intensive serverless edge computing, FGCS 114 (2021) 259–271.

[39] D. Bermbach, S. Maghsudi, J. Hasenburg, T. Pfandzelter, Towards auction-based function placement in serverless fog platforms, in: ICFC, 2020, pp. 25–31.

[40] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, K. Winstein, Secure serverless computing using dynamic information flow control, OOPSLA 2 (2018) 1–26.

[41] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, A. Bates, Valve: Securing function workflows on serverless computing platforms, in: WWW, 2020, pp. 939–950.

[42] S. Forti, A. Brogi, Continuous Reasoning for Managing Next-Gen Distributed Applications, in: ICLP (Technical Communications), volume 325 of *EPTCS*, 2020, pp. 164–177.

[43] R. Calegari, E. Denti, S. Mariani, A. Omicini, Logic programming as a service, Theory Pract. Log. Program. 18 (2018) 846–873.