Workflow Integration Alleviates Identity and Access Management in Serverless Computing

Arnav Sankaran
University
of Illinois at Urbana-Champaign
arnavs3@illinois.edu

Pubali Datta
University
of Illinois at Urbana-Champaign
pdatta2@illinois.edu

Adam Bates
University
of Illinois at Urbana-Champaign
batesa@illinois.edu

ABSTRACT

As serverless computing continues to revolutionize the design and deployment of web services, it has become an increasingly attractive target to attackers. These adversaries are developing novel tactics for circumventing the ephemeral nature of serverless functions, exploiting container reuse optimizations and achieving lateral movement by "living off the land" provided by legitimate serverless workflows. Unfortunately, the traditional security controls currently offered by cloud providers are inadequate to counter these new threats.

In this work, we propose WILL.IAM,[1] a workflow-aware access control model and reference monitor that satisfies the functional requirements of the serverless computing paradigm. WILL.IAM encodes the protection state of a serverless application as a permissions graph that describes the permissible transitions of its workflows, associating web requests with a permissions set at the point of ingress according to a graph-based labeling state. By proactively enforcing the permissions requirements of downstream workflow components, WILL.IAM is able to avoid the costs of partially processing unauthorized requests and reduce the attack surface of the application. We implement the WILL.IAM framework in Go and evaluate its performance as compared to recent related work against the well-established Nordstrom "Hello, Retail!" application. We demonstrate that WILL.IAM imposes minimal burden to requests, averaging 0.51% overhead across representative workflows, but dramatically improves performance when handling unauthorized requests (e.g., DDoS attacks) as compared to past solutions. WILL.IAM thus demonstrates an effective and practical alternative for authorization in the serverless paradigm.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Information flow control**; **Access control**.

KEYWORDS

Serverless Computing; Access Control; Information Flow Control

[1] WILL.IAM is short for Workflow Integration aLLeviates Identity and Access Management. IAM is the role-based access control system offered by Amazon Web Services.

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427665

ACM Reference Format:
Arnav Sankaran, Pubali Datta, and Adam Bates. 2020. Workflow Integration Alleviates Identity and Access Management in Serverless Computing. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427665

1 INTRODUCTION

Projected to exceed $8 billion per year by 2021 [7], serverless computing has experienced rapid growth and is expected to become the dominant pattern for cloud computing [63]. Also known by Function-as-a-Service (FaaS), serverless computing abstracts away the need to manage not only physical hardware but also the need to manage the life cycle of virtual machines; in serverless, the customer is no longer responsible for launching or tearing down virtual machines, provisioning virtual computer clusters, or management of software below the application level. This is achieved through the decomposition of applications into small discrete functions, stateless microservices that can be orchestrated into high-level workflows. As a result, developers can focus on the core logic of their application, eliminating the burdens of infrastructure management [73] while enabling rapid prototyping of services [12].

While serverless is often credited as being intrinsically more secure than prior cloud paradigms [83], in actuality most common cloud and web insecurities continue to fester [15]. A large-scale analysis of open-source serverless projects revealed that upwards of 20% contained critical vulnerabilities or misconfigurations [1]. Numerous event injection techniques have been demonstrated [2, 6, 10, 71], and challenges related to cross-tenant side-channels remain in the ecosystem [76, 92]. Even though these vulnerabilities were still present, for a time it was thought that it would be far more difficult to exploit them due to the stateless and ephemeral nature of serverless functions. Unfortunately, attackers have proven more than capable of surmounting these obstacles. For example, they have exploited the ubiquitously employed "warm container" reuse optimization, the practice of caching the containers of recently invoked functions in server memory, to transport toolkits into the application and establish quasi-persistence [64].

The primary mechanisms made available by cloud providers for mitigating these threats are role-based access controls (RBAC), known as Identity and Access Management (IAM) roles in the popular Amazon Lambda service [38]. Using IAM, cloud customers can statically assign each function to a role that is associated with a set of permissions for accessing other functions, datastores, or the open Internet. Accepting the reality that exploitable vulnerabilities will continue to exist in the serverless landscape, strict IAM roles can be configured such that functions are restricted to communicating only

with those components necessary to fulfill their task, thus reducing overprivilege. Unfortunately, there is already ample evidence that static RBAC alone is insufficient; not only are IAM roles often misconfigured [4, 19], but even when correctly defined, attackers are able to leverage legitimate function transitions to move laterally through the application in advancement of their goals [10, 64, 71]. Fundamentally, this attack is the result of a mismatch of abstractions – application developers express program logic in the form of inter-function workflows, yet authorization is performed only within the context of individual function transitions. As a result, attack opportunities are not bound by the end-to-end workflows specified by the developer.

In this work, we reconceptualize IAM roles as dynamic, efficient, and workflow-sensitive. We present WILL.IAM, an access control framework that, rather than (or in addition to) assigning static permissions to functions, performs authentication and role assignment to web requests at their point of ingress. Carrying this role assignment forward from function to function allows WILL.IAM to bound attackers to the permissions associated with a legitimate workflow, dramatically reducing the attack surface of the serverless application. WILL.IAM security policies are defined as a directed graph representation of the application's workflows, with the terminal nodes of the graph encoding permission requirements for traversing the workflow. As end-to-end determination of permissions for an entire workflow can frequently be pre-computed at the point of ingress, we extend WILL.IAM with a *proactive authorization* mechanism that rejects requests that cannot satisfy downstream permission requirements. Thus, WILL.IAM provides an intuitive extension to IAM-style RBAC that satisfies the unique functional requirements of serverless application security.

In this work, we make the following contributions:

- *Workflow-sensitive Access Control.* We present the design of a novel access control model for serverless computing that mediates inter-function information flow as requests are processed. Our approach follows the same design principles as serverless applications, while simultaneously integrating with the notion of IAM-style role-based access controls, thus avoiding the need for a dramatic reconceptualization of security on cloud platforms.
- *The* WILL.IAM *framework.* We implement our access control model for the popular OpenFaaS serverless platform.[2] As a case study on the efficacy of our approach, we define and analyze a complete security policy for the canonical "Hello, Retail!" reference application [88]. Our code, policies, and datasets will be made publicly available upon publication.
- *Performance Evaluation.* We rigorously evaluate the performance of WILL.IAM as compared to vanilla OpenFaaS as well as two baseline access control systems from related work (Trapeze [36], Valve [47]). We demonstrate that WILL.IAM has much less overhead compared to Trapeze and Valve, with an average workload overhead of just 0.51% compared to vanilla OpenFaaS; further, our performance optimization for proactive authorization of requests reduces wasted computation by 22% when considering a traffic profile comprised of 30% unauthorized requests.

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

2 BACKGROUND

Commercialized cloud computing began in 2006 with Amazon first releasing Elastic Compute Cloud (EC2) [37] and provided enterprises access to unlimited backend infrastructure without the burden of managing it. More recently, serverless computing platforms have emerged to further abstract away the need to manage the software stack (virtual machine provisioning, operating system and networking layer patches and upgrades) running below the application layer. Thus, serverless computing has enabled cloud tenants to focus on developing the business logic of their applications while the cloud provider handles load-balancing, auto-scaling of resources and other management tasks. The tenants are billed according to the resource usage (CPU, memory and network) when their functions are executed.

Serverless application developers implement the business logic as a set of functions that can be chained together to form task-specific workflows. These functions often read or write private data stored in the cloud infrastructure in order to achieve their operational goal (e.g. serverless data analytics applications [86]). This necessitates a robust access control mechanism in the serverless platforms to determine if a function invocation request is properly authenticated and has the required permissions to access a piece of data. Cloud platforms offer access control techniques [44, 85] as part of cloud services (e.g., AWS Identity and Access Management (IAM) [25]) provided by them. Cloud providers generally implement traditional role-based access control (RBAC) [49] and attribute-based access control (ABAC) [58] methods in their IAM services [31].[3] In serverless platforms, roles are attached to the functions allowing them to access other cloud resources (e.g., invoke other functions, access datastores) according to the access policy and permissions attached to the role. The roles and permissions should be defined to grant the least privilege to a function required for its operations.

For a traditional monolithic application deployed on the cloud, a single IAM authentication per request suffices to verify the entire application's adherence to the access policy. However, to enforce the same level of security in a serverless setting, each component function of the serverless application will need to perform authentication and authorization for every request. This quickly becomes infeasible in a serverless scenario, where a high throughput production-level application consists of numerous atomic functions each of them requiring various permissions to perform specific data operations. This leads to increased complexity of IAM policies, additional network latency for authentication, and high billing cost [33] for the cloud tenants. The resulting complex policies are often festered with misconfigurations [4, 19] and create greater opportunity for attackers. Moreover, attackers can even leverage legitimate function transitions to move laterally through the application [10, 64, 71] since each function activation is authorized in isolation, without considering the historic context (i.e. the series of function activations and permissions acquired in the workflow starting at the point of entry) of the current request. Currently, the static function-level IAM policies in serverless platforms do not offer an ideal security-performance trade-off for the tenants.

[3]For the remainder of this paper, we will use IAM as a generic term to describe role-based access controls assigned to functions in serverless clouds.

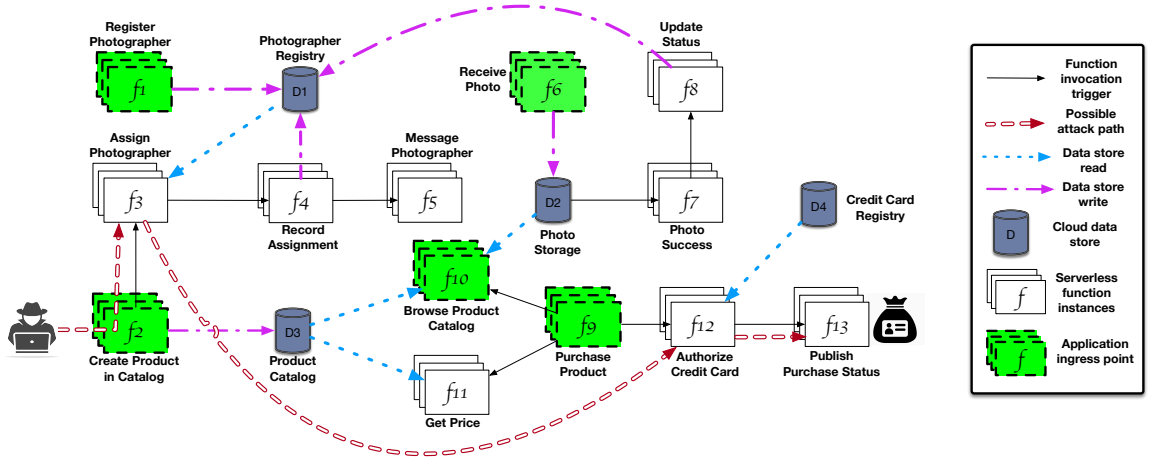


Figure 1: A reference architecture of serverless application *Hello, Retail!*.

Workflow	Description
$f_1(W_{D_1})$	Registration of a photographer in photographer registry
$f_2(W_{D_3}) \rightarrow f_3(R_{D_1}) \rightarrow f_4(W_{D_1}) \rightarrow f_5$	Creation of a product in the catalog and assign a photographer to submit a photo of the product
$f_6(W_{D_2}) \rightarrow f_7 \rightarrow f_8(W_{D_1})$	Receive photo from photographer and update assignment status
$f_9 \rightarrow f_{10}(R_{D_2};R_{D_3}) \rightarrow f_{11}(R_{D_3}) \rightarrow f_{12}(R_{D_4}) \rightarrow f_{13}$	Purchasing a product listed in the catalog
$f_{10}(R_{D_2};R_{D_3})$	Catalog browsing

Table 1: Summary of primary *Hello, Retail!* **workflows, as presented in Figure 1.**

3 MOTIVATION

In this section we describe the limitations of existing IAM policies in the context of serverless computing with an example application, *Hello, Retail!* [88], an open-source event-driven retail application from Nordstrom Technology. A simplified conceptual architectural diagram of *Hello, Retail!* is shown in Figure 1.

The application consists of two types of resources: serverless functions and datastores. The functions can be triggered by explicit HTTP requests to an API gateway, other functions, or data store events (e.g., creation of a new object in a datastore). The functions which expose public REST endpoints through the API gateway are designated *Application Ingress Points* and can be invoked through HTTP requests originating from the open Internet. The other functions should only be invoked from another function or a data store event. For example, f_1, f_2, f_6, f_9 and f_{10} are the application ingress points in this scenario. The internal function f_3 can only be executed with necessary permissions, which are associated with the IAM role assigned to f_2, and thus cannot be invoked directly through HTTP. Function f_7 is invoked by new object creation in datastore D_2 and is the only function in Figure 1 that has a datastore event trigger. The functions communicate with datastores D_1, D_2, D_3 and D_4 with appropriate permissions in order to complete their tasks.

The ingress points mark the beginning of different workflows in the application. The five primary workflows are described in Table 1. The $\rightarrow$ denotes a function invocation. Required datastore communications are encoded in the parenthesis next to the function, e.g., $f_1(W_{D_1})$ denotes function f_1 writes to datastore D_1, whereas R_{D_1} would denote a read from the datastore.

Limitations in existing serverless access control techniques. Serverless cloud platforms offer access control policy enforcement at the granularity of functions to ensure proper security of the application. Each function needs to be configured with the least privilege IAM roles and policies. For example, function f_3 only requires read access to the photographer registry D_1 and function f_4 should only be able to add new assignments to D_4, but is barred from accessing existing records. While these are the obvious permissions to be granted, there are several implicit permissions (e.g., access to the function source files stored in a cloud [26]) that if misconfigured can allow attackers access to the function code [19, 64]. The complex policy evaluation logic (e.g., [29]) employed by the cloud providers at every function to authorize a request makes the task of defining IAM policies even more unintuitive and cumbersome. Even with correctly configured IAM policies in place, attackers can still leverage leaked cloud access keys [27, 30, 74] for nefarious purposes, such as a cloud data store breach [28].

Returning to the example in Figure 1, consider a common scenario [19, 64] in which the IAM role for f_3 has been misconfigured, granting it the ability to invoke any function in the application. The attacker invokes f_2, passing in malcrafted data that exploits a vulnerability in f_3. After gaining control of f_3, they relay commands to invoke f_{12}, passing a bad argument that causes the entirety of D_4 to be returned to the function and subsequently transmitted to site visitors by f_{13}. Since traditional IAM authorizes each function in isolation, it is not clear to the platform that an information flow has been violated as f_{13} executes, in spite of the fact that the attacker reaches the credit card registry from a completely disjointed application workflow. This is what has motivated us to create our system, WILL.IAM, which associates requests with workflow-level permissions to reduce serverless applications' attack surface.

4 THREAT MODEL & ASSUMPTIONS

In this work, we consider an attacker whose primary goal is to exploit some security vulnerability in a serverless function or misconfigured IAM roles to use the function for malicious motives. The abundance of accidental access key leakage [27, 28, 74] makes it easier for such attackers to leverage leaked or stolen keys to launder sensitive data stored in cloud data stores. We assume that the cloud provider employs an IAM service to define cloud resource access policies to prevent data breaches. The cloud provider is trusted and will not mishandle or tamper with the security policies defined by the tenants. We also assume the presence of an API gateway in the cloud platform to handle external requests originating from the public internet and a trusted authentication service which properly authorizes ingress requests. Components like IAM and API gateway are part of the standard cloud design paradigm confirming the validity of the assumptions. We further make the assumption that all serverless functions are invoked through the use of REST API calls or other forms of Remote Procedure Calls (event triggers, asynchronous callbacks). This assumption is valid because web and API serving are the most popular use cases in the serverless paradigm [62].

5 POLICY DESIGN

In this section, we present the design of the access control model and policy representation. In Section 6, we present additional details of the WILL.IAM architecture, including how it is integrated into the cloud platform.

In WILL.IAM, access control roles are assigned to workflows, not individual functions. In practice, this means that a web request is assigned a role at its point of ingress and is bound to the role's associated permissions throughout its lifecycle in the serverless application. The security policy is comprised of two components, a *Labeling State* and a *Protection State*. An example policy is given in Figure 2. The Labeling State specifies the permissions associated with a given role, while the Protection State specifies the required permissions to complete a logical routine within the application. Critically, the Protection State does not describe function-by-function permission requirements, but instead the end-to-end permission requirements of the workflow. As we will later show, this allows authorization to be performed proactively at the earlier stages of a workflow to reduce the unnecessary use of compute resources.

5.1 Labeling State

We represent the Labeling State as a directed acyclic graph of the form $G=<V,E>$. Each $v \in V$ is of the form $v=<label,type>$, where *label* is an arbitrary string and $type \in \{$*"token", "role", "data"*$\}$. Token vertices correspond to authentication tokens, roles vertices to RBAC roles, and data vertices to explicit data permissions in the application. The permissible edges in E are constrained by vertex types: $\{token \rightarrow role, role \rightarrow role, role \rightarrow data\}$. Each token is associated with at most one role, each role is associated with zero to many data permissions, and for space efficiency roles can be hierarchical such that parent roles encompass all permissions of their children.

The design of our security policy is intentionally vague on the authentication method that should be employed. This is because authentication is ultimately an orthogonal problem that is best resolved by the application developers. For example, the developers may wish to use a password gateway or an OAuth-based approach for role assignment. In our proof-of-concept implementation of WILL.IAM, we make use of a token-based authentication scheme where clients transmit their token in the "Authorization" HTTP header. The WILL.IAM framework then hands the token off to an authentication service to determine the role to be associated with the request. The problem of web service authentication is well researched (e.g., [43]) so we will not explore it in greater detail in the remainder of this paper.

5.2 Protection State

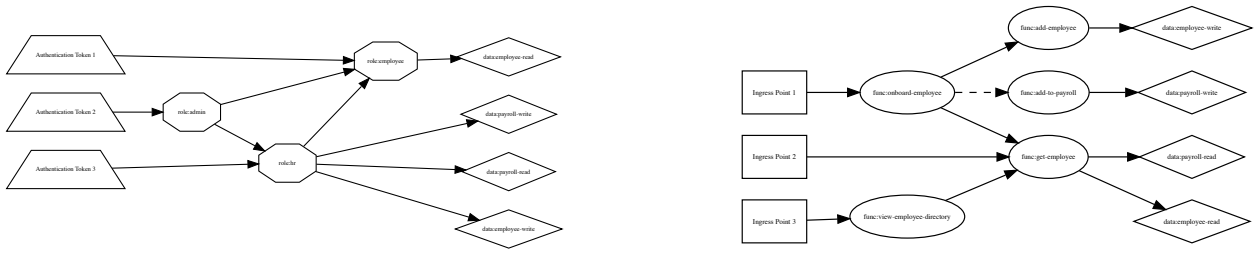
The Protection State is also represented as a directed acyclic graph of the form $G=<V,E>$. Each $v \in V$ is of the form $v=<label,type>$, where *label* is an arbitrary string and $type \in \{$*"ingress", "function", "data"*$\}$. Each edge $e \in E$ is of the form $<v_{src},v_{dst},type>$. Ingress vertices correspond to ingress points of the serverless applications, functions correspond to individual computation components of the application, and data vertices correspond to required explicit data permissions. The permissible edges in E are constrained by vertex types: $\{ingress \rightarrow function,\ function \rightarrow function,\ function \rightarrow data\}$. Each ingress vertex is linked to at most one function, each function can link from zero to many intermediate functions, and each last-level function is linked with zero to many data permissions.

The protection state graph can be used to perform authorization as follows. A function v_{f_1} may only invoke the API of a function v_{f_2} if there exists an edge $v_{f_1} \rightarrow v_{f_2}$; this is comparable to a traditional IAM role on Amazon Lambda, where only "single hop" transitions can be specified. Each path from an ingress vertex to a last-level function vertex encodes a programmed workflow in the application. The end-to-end workflow is authorized if the request is associated with a role that carries all of the required permissions encoded by the children of the last-level function. Thus, in this policy it is possible to proactively deny a request at the point of ingress if it lacks a necessary permission, even if that permission is not required until deep into the function workflow.

Despite their simplicity, functions often complex internal workflows, to the point that some functions may only conditionally invoke downstream functions depending on the context of the request. As a result, it may be that the permission set of a workflow is undecidable at the point of ingress. Allowing the request to proceed only if it contained *all possibly necessary* permissions would be overly restrictive; instead, to account for this we introduce a type attribute to each edge in the Protection State where $e.type \in \{$"$Mandatory$", "$Conditional$"$\}$. If a conditional edge exists in a workflow, we perform *conditional authorization* on the request at the point of ingress in which only the data permissions required by mandatory paths are checked. At each intermediary function in the workflow, we then check to see if any conditional requirements have been resolved, potentially re-authorizing the request if they have. The request is only fully authorized to continue once all conditional requirements have been resolved.

5.3 Example Policy Walkthrough

Figure 2 depicts an example security policy for a simple and imaginary human resources application. The application contains five functions (`onboard-employee`, `add-employee`, `add-to-payroll`,



(a) Labeling State Graph **(b) Protection State Graph**

Figure 2: Example WILL.IAM Security Policy. In Figure 2a, trapezoids are authentication tokens, octagons are the roles associated with those tokens, and diamonds are the permissions associated with those roles. Traversing the graph from a token to its terminal children specifies the token's permission set. In Figure 2b, rectangles are application ingress points, ovals are functions, and diamonds are permissions. Each path between an ingress point and a terminal child represents an application workflow, with the terminal children of the last function specifying the required permissions of the entire workflow.

`get-employee`, and `view-employee-directory`) and two datastores (`employee` and `payroll`), each with `read` and `write` permissions. Requests can be assigned to one of three roles (`employee`, `hr`, or `admin`). The `employee` role only possesses permission to read the `employee` datastore, while the `admin` role is permitted to write to the `employee` datastore and to read and write to `payroll`. The `admin` role possesses all permissions, which is expressed in Figure 2a by positioning `admin` as the parent of both the `employee` and `hr` roles.

Consider the workflow associated with Ingress Point 3. If a request is issued that authenticates and is assigned the `admin` role, it is possible to make an authorization decision for the entire workflow at the point of ingress. This is because the `admin` role possesses the `payroll-read` and `employee-read` permissions. On the other hand, if the request is assigned the `employee` role, it is possible to proactively deny this request at the point of ingress. This role technically has the necessary permissions to execute `view-employee-directory`, but not the permissions required to execute `get-employee`. Allowing the request to execute a portion of the workflow both wastes computation and expands the attack surface of the application, and should thus be avoided.

Let us now consider the workflow associated with Ingress Point 1. This workflow relates to an employee-onboarding routine performed by the Human Resources department. The `onboard-employee` function calls the `add-employee` function to register the new employee, as well as the `get-employee` function to return the new record for confirmation. If the employee has already completed and uploaded their direct deposit paperwork, the employee is added to the payroll system as well, but if not they are permitted to do so at a later date. As a result, the `onboard-employee` function contains a conditional dependency to `add-to-payroll`, which impacts the permission set of the entire workflow. Therefore, at the ingress point we can only *conditionally* authorize the workflow, comparing the role's permission set to the mandatory permissions `employee-write` and `payroll-read`. If `add-to-payroll` is invoked, the mandatory permission set changes to include `payroll-write`, which must then be verified.[4]

6 WILL.IAM ARCHITECTURE

In this section, we present the design of the WILL.IAM architecture, which manages and enforces the security policies described above.

6.1 Overview

A diagram of WILL.IAM is presented in Figure 3. WILL.IAM is comprised of three main components: the API gateway, the policy evaluation service and the request handler. The API gateway is built into the FaaS platforms to provide an external route to the deployed functions. In WILL.IAM, the API gateway is augmented to forward externally generated requests to the policy evaluation service and to forward internally generated invocation requests to the proper function instances. The policy evaluation service is the enforcement point of the access control policies defined for various serverless workflows deployed on the cloud. There is a request handler running in each function-instance (i.e. container) that transparently adjusts WILL.IAM-specific headers in the invocation request before passing it to the function, thus making WILL.IAM function-agnostic. These three components collaborate to enforce access control in serverless cloud platforms as described in Figure 3.

When an external function execution request arrives at the API gateway (①) in a FaaS platform with WILL.IAM enabled, the request is forwarded to an authentication server (②) to assign the designated role to the request as per the authorization header information in the request (③). Next, the request is passed to the policy evaluation service (④) to verify whether the assigned role possesses necessary data permissions to successfully execute the workflow activated by this request. The policy evaluation service fetches the protection state associated with the request and the labeling state associated

[4]Note that, in our simplified example, only the `hr` role can execute this workflow regardless of whether the conditional branch is taken.

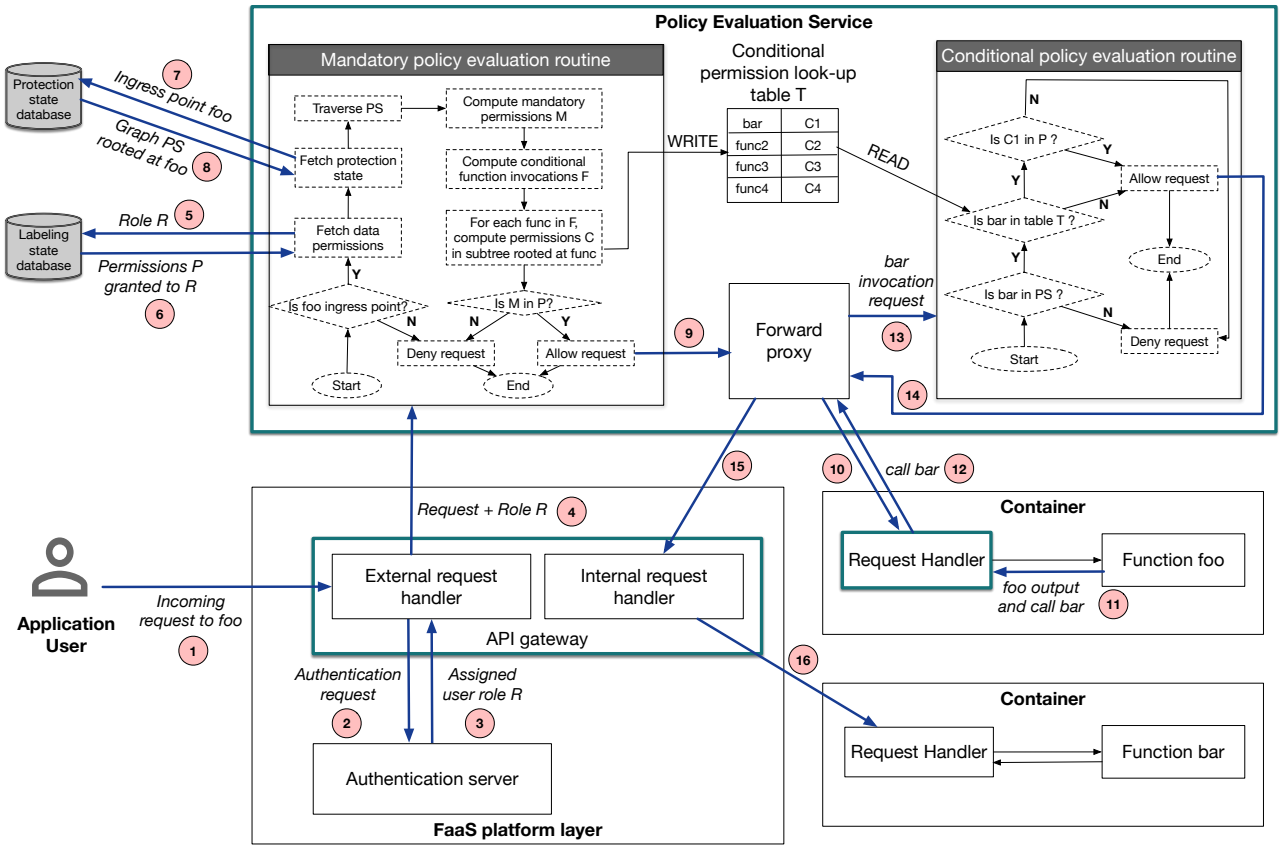


Figure 3: An overview of the WILL.IAM architecture and its authorization workflow.

with the role and invokes the *mandatory policy evaluation routine* (⑤-⑧). This routine accepts the request if it has the data permissions for executing absolute function invocations in its trajectory through the workflow and forwards the request to the appropriate function container (⑨-⑩). Otherwise the request is denied right at the ingress point even before any function executes. For each of the subsequent function invocations in the workflow, the policy evaluation service fires the *conditional policy evaluation routine* (⑪-⑬). If the function invocation is absolute, then no further processing is necessary and the request is forwarded to the function container, whereas for a conditional invocation data permissions required for that particular invocation are verified before forwarding (⑭-⑯).

6.2 API Gateway

In FaaS platforms, the API Gateway handles all incoming requests from the public internet. WILL.IAM extends the API gateway to introduce a centralized workflow-aware access control enforcement point in the lifecycle of an incoming request. Upon receiving an external request, the WILL.IAM API gateway uses an authentication service to exchange the request's authorization token for an IAM role which can be passed along with the request to the policy evaluation service. The API gateway is also responsible for forwarding pre-approved internal requests to their destination function containers. Requests rejected by WILL.IAM generate an unauthorized error, and the API gateway propagates the error to the user with necessary information for debugging purposes following the standard design practice of FaaS. The API gateway introduces no additional overhead in executing its tasks and transmits all access control information in-band with the requests that flow between the cloud components, thus avoiding TCP and HTTP overheads.

6.3 Policy Evaluation Service

The policy evaluation service is the centralized access control enforcement point in WILL.IAM. This component is designed as a plug-in with defined interfaces to interact with existing serverless platforms. In WILL.IAM, the API gateway forwards all externally originated requests to the policy evaluation service that proactively denies requests with insufficient permissions from executing the entire workflow associated with the ingress request. It also enforces access control for conditional policy violations in internal function requests. The policy evaluation service employs the following methods to achieve its goal.

6.3.1 Mandatory Policy Evaluation. The mandatory policy evaluation routine takes care of denying ingress requests with insufficient permissions. It reads the Protection State graph to obtain the set of permissions M required to execute the workflow, and the Labeling State graph associated with the corresponding IAM role to obtain the set of permissions P granted to the request. If $M \nsubseteq P$, then the request

is immediately rejected because eventually the workflow will lack necessary permissions to execute some downstream function. To account for the conditional function invocations in the protection state graph, this routine computes the conditional permission look-up table T which stores data permissions that will be required by these invocations for later processing, but does not immediately verify the presence of the conditional permissions. The accepted requests are forwarded to the designated function instances.

6.3.2 Conditional Policy Evaluation. All internal function requests are directed to the conditional policy evaluation routine. It trivially accepts all absolute function requests, as access permissions for them had already been verified in the mandatory policy evaluation. However, for functions present in table T, this routine determines whether the workflow should continue to execute or not. It checks if the required permissions C by the target function f are satisfied in P. Requests with unsatisfied permissions are aborted, whereas accepted requests are forwarded to the gateway for routing to the correct function instance.

6.4 Request Handler

In serverless platforms, a tiny webserver (i.e., a request handler) runs inside the function container that accepts function invocation requests. This request handler parses the incoming request object and starts execution of the function. WILL.IAM extends the design of this request handler to remove the in-band headers used by WILL.IAM before handing the request off to the function routine. This allows WILL.IAM to be completely transparent to the function implementation and easily deployable to existing platforms without any function modification. Additionally, the request handler runs a reverse proxy for communicating with other functions in the workflow. This reverse proxy will readjust the in-band WILL.IAM-specific headers in the outgoing request and then redirect it to the policy evaluation service to verify conditional violations.

7 IMPLEMENTATION

We implemented WILL.IAM into the OpenFaaS serverless framework. OpenFaaS can be deployed on multiple container orchestration platforms, however in this paper we deploy OpenFaaS over Kubernetes. We primarily modified two components of OpenFaaS, the "gateway" and "of-watchdog", representing an addition of approximately 400 lines of Go code.

Gateway. The gateway is exposed to the public internet and accepts incoming requests to functions. We modify the gateway by adding an extra HTTP middleware which modifies the body of the incoming request as it exits the gateway. This modification handles exchanging the token specified in the "Authorization" header of the incoming HTTP request for a policy. The gateway then uses the policy name and policy graph to build a list of data permissions granted to the request. The gateway also uses the target function name, which is specified in the URI path, and the Protection State graph in order to build the list of data permissions that are absolutely and conditionally required. If there is an element in the set of absolute permissions that is not in the request's permission set, the gateway rejects the incoming request with an unauthorized error. If there are conditional violations, the gateway encodes the allowed data permissions for the request into a serialized in-band header and transmits them to the target function. The target function runs behind a modified version of the of-watchdog component described below.

```
{
  "functions": {
    "product-catalog-api": {
      "permissions": [
        {
          "dataType": "productCategory",
          "operation": "read"
        }
        ... additional permissions omitted for space ...
      ],
      "absoluteDependencies": [],
      "conditionalDependencies": []
    },
    ... additional functions omitted for space ...
  },
  "policies": {
    "customer": {
      "dependencies": [
        "public"
      ],
      "permissions": [
        {
          "dataType": "creditCardsName",
          "operation": "read"
        }
      ]
    },
    ... additional policies omitted for space ...
  }
}
```

Figure 4: Snippet from an example JSON configuration file used for defining WILL.IAM security policies.

The gateway also handles routing of calls between functions in the Protection State graph. For these intra-function requests, the in-band header is already encoded into and the gateway checks for conditional violations before forwarding the request. If a conditional violation becomes absolute, the gateway rejects the request and the unauthorized error is sent upstream to the user.

Of-Watchdog. The of-watchdog server is only reachable from within the OpenFaaS cluster. It handles both receiving incoming requests from the gateway and passing them onto the function. The OpenFaaS framework allows functions to receive requests from the of-watchdog server over standard input or over HTTP. We do not modify this functionality and our changes are transparent to the function.

We add extra HTTP middleware that removes the in-band header containing the encoded access control logic from the gateway. We also modify the watchdog to launch a reverse proxy server that is bound to a port within the cloud function's container. When the HTTP middleware removes the in-band header from the request it stores it in a map so that it can be looked up for the specific request. The reverse proxy uses this map to retrieve that in-band header associated with a request and add it back to the request before it is sent to the gateway as an intra-function request.

Configuration. We require the access control policy writer to provide a JSON configuration file which provides the required information about each function and policy in the access control model. Figure 4 provides an example configuration file.

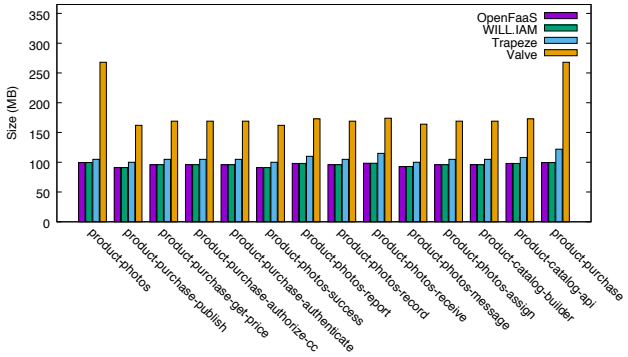


Figure 5: Container image build sizes for *Hello, Retail!*.

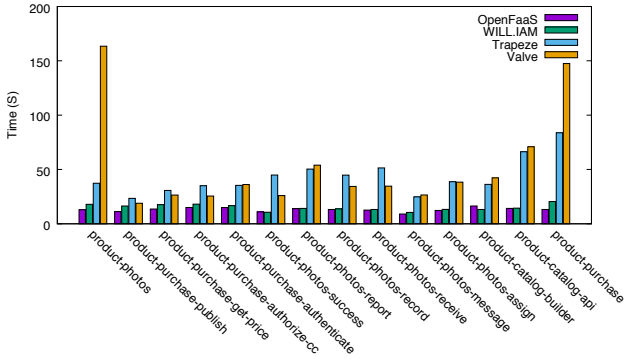


Figure 6: Container image build times for *Hello, Retail!*.

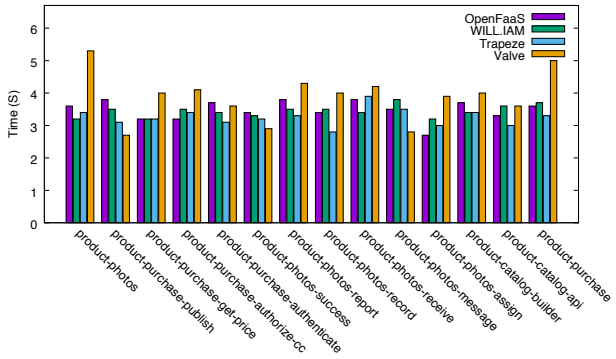


Figure 7: Container deployment times for *Hello, Retail!*,

8 EVALUATION

We evaluate our access control system using *Hello, Retail!* [88], a serverless application developed by Nordstrom Technology as a proof-of-concept approach to an event driven computing model in the retail industry. As one of the most mature open-source applications, *Hello, Retail!* has been used in many past studies of serverless computing [80], including access control research [36, 47, 59].

We compare the performance of WILL.IAM against two state-of-the-art serverless information flow control systems, Trapeze [5] [36] and Valve [6] [47], both of which have been open-sourced by their respective authors. Trapeze is a language-based approach that traces information flow at the language level, while Valve is a system-based approach that, like WILL.IAM, mediates events at the function level. To compare against these systems, we use the modified version of the application from Alpernas et al. [36], which replaces AWS-specific components with open-source components that can be deployed on top of Kubernetes and OpenFaaS. This is the same modified version used in prior work to evaluate these systems providing us with an apples-to-apples baseline upon which to compare performance. We also write a complete WILL.IAM security policy for *Hello, Retail!*, which is provided and discussed in Appendix A.

All experiments were performed on a server-class machine with an Intel(R) Xeon(R) CPU E5-2683v4 running at 2.10GHz and 135 GB of RAM. The containerization and orchestration software used was Docker 19.03.11 and Kubernetes 1.18.3. For the purposes of testing, the Kubernetes cluster was configured as a single node cluster with both control plane and user deployed pods being run on the single master node. All Docker images required for all of the following tests were pre-pulled in order to minimize the effects of external networking variations.

8.1 Build Time Performance

The build time and build size overheads, averaged across 30 invocations of each function, are given in Figure 5 and Figure 6. These figures indicate that WILL.IAM imposes very little overhead at build time when compared to Vanilla OpenFaaS. Additionally it substantially outperforms both Trapeze and Valve. For build size, the slightly increased container size over Vanilla is due to the additional Go code that is compiled into the of-watchdog binary for every function container. However, this addition pales in comparison to the increased build size of Trapeze and Valve, where many additional files are copied into the container image. Trapeze needs to copy in Javascript files that help with securing its key-value store, while Valve needs to copy in its HTTPS proxy binary. When comparing build times, there was no meaningful difference between WILL.IAM and Vanilla OpenFaaS. The slight differences observed here were due to variances in retrieving `npm` packages over the Internet during the build process. In contrast, Trapeze and Valve both took significantly longer to build due to the introduction of extensive additional dependencies to the container. Although these costs are one-time and can be largely ignored by the customer, they compound when considering millions of customers if these approaches were to be adopted at the platform layer, suggesting a significant advantage for the adoptability of WILL.IAM.

8.2 Orchestration Performance

Orchestration refers to platform management tasks, specifically the deployment and teardown of containers as functions are requested to be invoked. We report overheads for deployment and teardown, averaged across 30 invocations of each function, in Figures 7 and 8,

[5] https://github.com/kalevalp/trapeze
[6] https://github.com/Ethos-lab/Valve

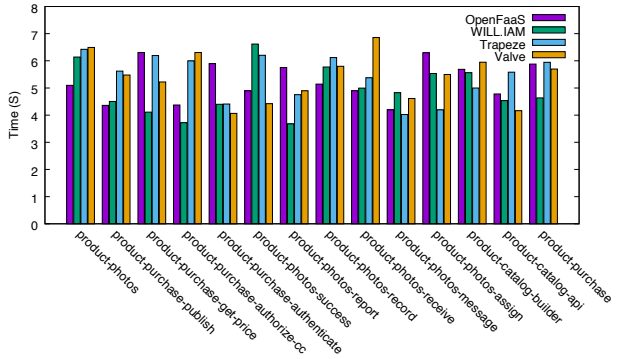


Figure 8: Container teardown times for *Hello, Retail!*.

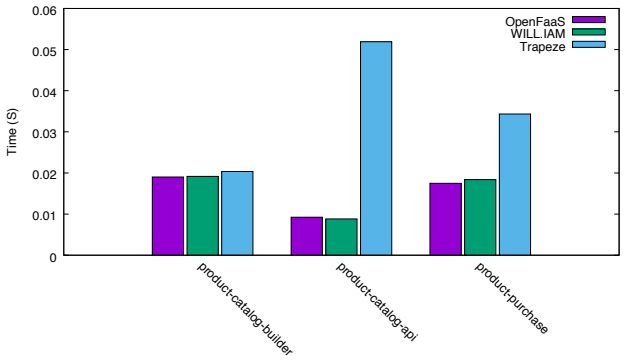


Figure 9: End-to-end workflow latency for web requests.

respectively, we did not observe any meaningful pattern of differences between any of the benchmarked systems. These results match up with our expectations, since the overhead associated with these operations is dominated by the performance of the orchestration system rather than the specific containers being handled. In spite of our results being averaged across many trials, no differences are observable that are not better explained by noise due to orchestration scheduling decisions made by `kube-scheduler` and `kubelet`. However, this is sufficient to demonstrate that WILL.IAM does not impose undue burden on orchestration routines.

8.3 Runtime Workflow Performance

As WILL.IAM mediates activity at the workflow level, we now compare performance of WILL.IAM to Vanilla OpenFaaS and Trapeze using 3 end-to-end *Hello, Retail!* workflows. We were unable to include Valve in these experiments because the authors did not make available their *Hello, Retail!* policy along with their source code. We measure the end-to-end latency of a request for 3 representative workflows in the reference application: Catalog Builder, Catalog API, and Product Purchase. These correspond to the ingress points f_2, f_{10}, and f_9 in Figure 1. They also represent 3 different kinds of workflows: Catalog Builder makes additional function calls and performs a database write operation, Catalog API makes no additional function calls but

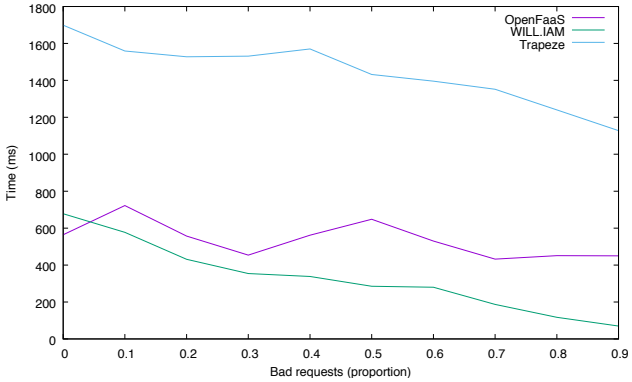


Figure 10: Concurrent request latency for variable proportions of bad (unauthorized) traffic.

performs a database read operation, and Product Purchase makes additional function calls and performs a database read operation. For each system, all requests made in this experiment had all required permissions to be fully processed.

Results, shown in Figure 9, are averaged over 1000 repetitions of each workflow. Across all three workflows, we observe almost negligible overheads imposed by WILL.IAM, averaging 0.51% and in thr worst case 5.2%. This overhead is due to the searches required to determine if the request has the required permissions and the encoding and decoding of the in-band header. In contrast, Trapeze imposes significant overheads on 2 of the 3 flows, specifically in conditions in which many datastore read operations occur. This is because the language-based Trapeze system transformation to programs incurs latency on datastore reads for each entry accessed in its key-value store.

Proactive Authorization. The above experiments assumed that all operations in the workflow were authorized, which does not capture the performance benefits of WILL.IAM's proactive authorization. To demonstrate the potential performance savings of this mechanism, we measured the latency of 100 concurrent requests with increasing proportions of unauthorized ("bad") traffic being issued to the ingress point. For this experiment, we make use of the Product Purchase workflow. Each bad request is correctly formatted, but lacks the required permissions to execute downstream functions after the ingress point, potentially resulting in wasted computation. If WILL.IAM's proactive authorization mechanism is effective, we expect to see decreased latencies for WILL.IAM as compared to other systems.

Results are shown in Figure 10, We observe that WILL.IAM greatly outperforms all of the other systems as the proportion of bad requests increase. This is due to the ability of WILL.IAM to preemptively reject requests with absolute violations at the gateway rather than allowing the request to be partially processed before being rejected. The Vanilla system does not have this ability and as a result it's average latency decreases with a much smaller slope. Even through Trapeze does allow for early rejection of requests with incorrect permissions, the overhead of the SQL operations is very high, leading to slow performance even in situations where a large proportion of the requests can be rejected early. When considering that Cloudflare estimates the percentage of Internet bot traffic to be 40% [46], much

of which emanates from malicious bots, this suggests potential for significant savings through use of WILL.IAM. While the workloads used in this experiment were synthetic, we discuss how this result may inform the use of WILL.IAM as a DDoS defense in Section 9.

9 DISCUSSION

Portability. The access control mechanism proposed in this paper and the implementation of the framework in OpenFaaS provides a way to enforce access control with low overhead in a serverless computing environment. Building the framework into OpenFaaS allows for this access control system to be deployed on top of Kubernetes and thus on every major cloud provider's infrastructure. Additionally this framework can easily be deployed on an in-house Kubernetes or Docker Swarm cluster. This portability is due to the fact that we do not depend directly on any functionality which is specific to a cloud providers serverless implementation. The entire framework runs inside of containers and utilizes common technologies for it's required functionality. For example, request and responses between services are serialized to and from JSON and performed over HTTP and routing between functions and the gateway leverages DNS.

Denial of Service Attacks. WILL.IAM's ability to determine if a request will fail ahead of time and reject it at the gateway has the potential to mitigate targeted denial of service attacks. This is because the wasted computation we identified above can be seen as a traffic amplifier; permitting these requests to travel partially through a complex workflow greatly increases the cost of the attack for the application owner. For example, in *Hello, Retail!* the function "purchase" depends on "get-price", "authorize-cc", and "publish" in that order. The first function in that ordering which requires a non-public data permission is "authorize-cc", therefore it is possible for a request without the proper permissions to invoke multiple functions before failing. However, when WILL.IAM is used, the upcoming permissions failure would be detected immediately and computational cost would be incurred only at the gateway. From the DDoS attack on GitHub [54] in 2018, we can see that bad requests accounted for upwards of 90% of incoming traffic. From the evaluation section above we can see that WILL.IAM access control model greatly outperforms the Vanilla model when proportion of bad requests is at 90%. Using Amazon Lambda's pricing model we determined the cost-saving of WILL.IAM under a DoS attack to be 64% less than that of the standard serverless platform. While we do not argue that WILL.IAM is a complete solution for DoS defense, it does prevent an application's own permissioned workflows from being weaponized against it.

10 RELATED WORK

Attacks on serverless platforms. Remote code execution, poor resource isolation and covert channels (i.e. VM, container and function co-location vulnerabilities) [52, 53], reconnaissance attacks and canonical cloud vulnerabilities [15] are rampant in serverless platforms [76, 92]. Researchers have demonstrated event injection attacks [2, 6, 10] and data exfiltration [64, 71] in serverless. Access control misconfigurations are shown to enable attackers to steal sensitive informative [4, 19] or launch denial-of-service (or *denial-of-wallet* [13, 14]) attacks by exhausting allocated resource limits and expanding usage bill. Common vulnerabilities and bugs in SDKs, third-party libraries and platform code [5, 9, 18, 20] plague serverless functions. Existing security solutions [3, 8, 11, 16, 17, 21–24] each solve some part of these problems. WILL.IAM adds to the growing set of defenses against described attacks.

Serverless Security Research. Alpernas et al. proposed Trapeze [36], a language-based approach to dynamic information flow control. Trapeze wraps each serverless function in a security shim that intercepts data accesses from shared data stores, external communication channels (i.e. Internet), and messages exchanged with other functions. Trapeze's implementation depends on the programming language of the function and usage of predefined key-value store functions from the Trapeze library. Trapeze's secure key-value store suffers high overhead induced by expensive SQL operations. Moreover, Trapeze completely forgoes serverless warm-start performance optimizations and worsens the overhead. The fork-optimized Trapeze does not work for some API calls and requires effort at the external API implementation level to enable the cloud function to work within Trapeze. In contrast, WILL.IAM takes a transparent approach to access control and is agnostic to function and platform implementation. WILL.IAM proactively evaluates access control policies at the ingress point of a serverless application and makes a decision on acceptance or rejection of the request even before any function executes leading to negligible overhead. Another flow-based framework, Valve [47], assists workflow developers in policy specification and employs a transparent coarser-grained (i.e. function-level) information flow control model that restricts unwanted function behavior through network proxying and taint propagation. We believe that Valve is complementary to WILL.IAM and will allow developers to better understand the data flows in their applications to write proper policy configuration for WILL.IAM. [61] optimizes authentication queries via caching in order to reduce the overhead of authenticating every function request. Instead, WILL.IAM leverages the idea of encoding absolute and conditional information flows within an application into a graph. This allows WILL.IAM to detect and disallow access policy violations at the point of ingress.

Baldini et. al. examined several popular platforms and concluded the lack of proper function isolation is a major problem [41]. Wang et. al. have measured several metrics like scalability, cold-start latency, instance lifetime in Google Cloud functions, Microsoft Azure Functions and AWS lambda [92]. They found placement vulnerabilities and arbitrary code execution bugs in Azure Functions that make the platform vulnerable to side-channel attacks. Robust access control can alleviate the damaging effect of such attacks as shown in this paper. Utilizing Intel's SGX to build secure containers [40] and cloud functions [35, 45] to provide better isolation, formal modeling of serverless platforms [51, 61], and semi-automated troubleshooting based on log data [75] are some related topics in serverless security research.

Access Control Models. Cloud platforms typically use federated identity management [65, 89] for access control which is insufficient to define expressive policies required in serverless application workflows. Graph based access control models [68, 81, 91] have used graphs to express hierarchical nature of user roles. Graph based frameworks have also been used to augment role based access control (RBAC) [69, 70] and relation based access control (ReBAC) [60] systems and have application in operating systems. WILL.IAM leverages the well-researched concepts of graph based access control to

propose flexible and dynamic access control model for serverless platforms.

Serverless computing application and design. Researchers have predicted the future of serverless computing [39, 62] and have evaluated its efficacy across several domains, including scientific computing [87], real-time systems [78], publisher-subscriber systems [55], internet-of-things [82], edge computing [56], and big data processing [42, 72, 93]. While this thread of research confirms the potential of serverless infrastructure, another thread focusses on improving the state-of-the-art serverless design. Improving serverless performance [34, 50, 57, 76, 79], combining SDN and serverless computing [32, 90], better serverless programming models [77], serverless pricing models [48], and serverless analytics optimizations [66, 67, 84] are some active research areas. Our paper adds to this literature by proposing an improved access control framework design to balance security-performance trade-off in serverless platforms.

11 CONCLUSIONS

In this paper we propose an access control mechanism which allows for requests to be preemptively rejected before known access control violations will occur, leading to time, compute, and cost saving. We implemented the proposed access control mechanism described for the OpenFaaS ecosystem and compared the implementation against prior work, Trapeze and Valve. In our evaluation we determined that there was no meaningful overheads at build time or during orchestration. We also determined that the runtime overhead was minimal when compared against prior access control system. When compared against the Vanilla implementation the average overhead was 0.51%. Furthermore we demonstrated that when load-testing the system at bad request proportion of 30%, WILL.IAM outperforms the Vanilla implementation by 22%.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is supported in part by NSF 17-50024 and NSF CNS 19-55228. The views expressed are those of the authors only.

REFERENCES

[1] 2019. 21% of Open Source Serverless Apps Have Critical Vulnerabilities. https://www.puresec.io/blog/puresec-reveals-that-21-of-open-source-serverless-applications-have-critical-vulnerabilities.
[2] 2019. A Deep Dive into Serverless Attacks, SLS-1: Event Injection. https://www.protego.io/a-deep-dive-into-serverless-attacks-sls-1-event-injection/.
[3] 2019. Aqua Cloud Native Security Platform. https://www.aquasec.com/products/aqua-container-security-platform/.
[4] 2019. AWS Lambda Container Lifetime and Config Refresh. https://www.linkedin.com/pulse/aws-lambda-container-lifetime-config-refresh-frederik-willaert/.
[5] 2019. CVE-2019-5736: runc container breakout. https://www.openwall.com/lists/oss-security/2019/02/11/2.
[6] 2019. Event Injection: Protecting your Serverless Applications. https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/.
[7] 2019. Function-as-a-Service Market by User Type (Developer-Centric and Operator-Centric), Application (Web & Mobile Based, Research & Academic), Service Type, Deployment Model, Organization Size, Industry Vertical, and Region - Global Forecast to 2021. https://www.marketsandmarkets.com/Market-Reports/function-as-a-service-market-127202409.html.
[8] 2019. FunctionShield. https://www.puresec.io/function-shield.
[9] 2019. Gathering weak npm credentials. https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md.
[10] 2019. Hacking a Serverless Application: Demo. https://www.youtube.com/watch?v=TcN7wHuroVw.
[11] 2019. Intrinsic: Software security, re-invented. https://intrinsic.com/.
[12] 2019. Lambda functions for rapid prototyping. https://developer.ibm.com/articles/cl-lambda-functions-rapid-prototyping/.
[13] 2019. Many-faced threats to Serverless security. https://hackernoon.com/many-faced-threats-to-serverless-security-519e94d19dba.
[14] 2019. New Attack Vector - Serverless Crypto Mining. https://www.puresec.io/blog/new-attack-vector-serverless-crypto-mining.
[15] 2019. OWASP Serverless Top 10. https://www.owasp.org/index.php/OWASP$_{S}$erverless$_{T}$op$_{1}$0$_{P}$roject.
[16] 2019. Protego Serverless Runtime Security. https://www.protego.io/platform/elastic-defense/.
[17] 2019. Puresec Serverless Security Platform. https://www.puresec.io/.
[18] 2019. ReDoS Vulnerability in "AWS-Lambda-Multipart-Parser" Node Package. https://www.puresec.io/blog/redos-vulnerability-in-aws-lambda-multipart-parser-node-package.
[19] 2019. Securing Serverless: Attacking an AWS Account via a Lambda Function. https://www.darkreading.com/cloud/securing-serverless-attacking-an-aws-account-via-a-lambda-function/a/d-id/1333047.
[20] 2019. Securing Serverless – by Breaking in. https://www.infoq.com/presentations/serverless-security-2018.
[21] 2019. Serverless Security for AWS Lambda, Azure Functions, and Google Cloud Functions. https://www.twistlock.com/solutions/serverless-security-aws-lambda-azure-google-cloud/.
[22] 2019. Snyk. https://snyk.io/.
[23] 2019. Sysdig Secure. https://sysdig.com/products/secure/.
[24] 2019. Vandium-node. https://github.com/vandium-io/vandium-node.
[25] 2020. AWS Identity and Access Management (IAM). https://aws.amazon.com/iam/
[26] 2020. AWS::Lambda::Function. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-lambda-function.html
[27] 2020. Cloud Breach: Compromising AWS IAM Credentials. https://rhinosecuritylabs.com/aws/aws-iam-credentials-get-compromised/
[28] 2020. List of AWS S3 Leaks. https://github.com/nagwww/s3-leaks
[29] 2020. Policy Evaluation Logic. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference$_{p}$olicies$_{e}$valuation-logic.html
[30] 2020. This Is What Happened When I Leaked My AWS Secret Key. https://alexanderpaterson.com/posts/this-is-what-happened-when-i-leaked-my-aws-secret-key
[31] 2020. What Is ABAC for AWS? https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction$_{a}$ttribute-based-access-control.html
[32] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein. 2019. Will Serverless Computing Revolutionize NFV? Proc. IEEE 107, 4 (April 2019), 667–678. https://doi.org/10.1109/JPROC.2019.2898101
[33] Gojko Adzic and Robert Chatley. 2017. Serverless Computing: Economic and Architectural Impact. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (Paderborn, Germany) (ESEC/FSE 2017). Association for Computing Machinery, New York, NY, USA, 884–889. https://doi.org/10.1145/3106237.3117767
[34] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus
[35] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-FaaS: Trustworthy and Accountable Function-as-a-Service Using Intel SGX. In Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop (London, United Kingdom) (CCSW'19). Association for Computing Machinery, New York, NY, USA, 185–199. https://doi.org/10.1145/3338466.3358916
[36] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. Proc. ACM Program. Lang. 2, OOPSLA, Article 118 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276488
[37] Amazon. 2006. EC2 Beta Announcement. https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/
[38] Amazon Web Services. 2020. Identity and access management for AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/security-iam.html.
[39] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. 2009. Above the Clouds: A Berkeley View of Cloud Computing. (2009).
[40] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI

16). USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[41] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. Springer Singapore, Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8$_1$

[42] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. 2019. FaaS Orchestration of Parallel Workloads. In Proceedings of the 5th International Workshop on Serverless Computing (Davis, CA, USA) (WOSC '19). Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/3366623.3368137

[43] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 2004. A Semantics for Web Services Authentication. In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Venice, Italy) (POPL '04). Association for Computing Machinery, New York, NY, USA, 198–209. https://doi.org/10.1145/964001.964018

[44] Eric Jason Brandwine. 2017. Permissions decisions in a service provider environment. US Patent 9,712,542.

[45] Stefan Brenner and Rüdiger Kapitza. 2019. Trust More, Serverless. In Proceedings of the 12th ACM International Conference on Systems and Storage (Haifa, Israel) (SYSTOR '19). Association for Computing Machinery, New York, NY, USA, 33–43. https://doi.org/10.1145/3319647.3325825

[46] Cloudflare. 2020. What Is Bot Traffic? https://www.cloudflare.com/learning/bots/what-is-bot-traffic/

[47] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, , and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan. Association for Computing Machinery, New York, NY, USA. https://adambates.org/documents/Datta$_W$ww20.pdf

[48] Tarek Elgamal. 2018. Costless: Optimizing cost of serverless computing through function fusion and placement. In 2018 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 300–312.

[49] David Ferriaolo and Richard Kuhn. 1992. Role-based access controls. In Proceedings of 15th NIST-NCSC National Computer Security Conference. 554–563.

[50] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In 2019 USENIX Annual Technical Conference (USENIX ATC 19). USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[51] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. 2019. No More, No Less. In Coordination Models and Languages, Hanne Riis Nielson and Emilio Tuosto (Eds.). Springer International Publishing, Cham, 148–157.

[52] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (London, United Kingdom) (CCS '19). Association for Computing Machinery, New York, NY, USA, 1073–1086. https://doi.org/10.1145/3319535.3354227

[53] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang. 2018. A Study on the Security Implications of Information Leakages in Container Clouds. IEEE Transactions on Dependable and Secure Computing (2018), 1–1. https://doi.org/10.1109/TDSC.2018.2879605

[54] GitHub. 2018. GitHub DDOS Incident Report. https://github.blog/2018-03-01-ddos-incident-report/

[55] Faisal Hafeez, Pezhman Nasirifard, and Hans-Arno Jacobsen. 2018. A Serverless Approach to Publish/Subscribe Systems. In Proceedings of the 19th International Middleware Conference (Posters) (Rennes, France) (Middleware '18). Association for Computing Machinery, New York, NY, USA, 9–10. https://doi.org/10.1145/3284014.3284019

[56] Adam Hall and Umakishore Ramachandran. 2019. An Execution Model for Serverless Functions at the Edge. In Proceedings of the International Conference on Internet of Things Design and Implementation (Montreal, Quebec, Canada) (IoTDI '19). Association for Computing Machinery, New York, NY, USA, 225–236. https://doi.org/10.1145/3302505.3310084

[57] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[58] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas. 2015. Attribute-Based Access Control. Computer 48, 2 (2015), 85–88.

[59] Huang, Xiaowei. 2019. Forensic Analysis in Access Control: a Case-Study of a Cloud Application. http://hdl.handle.net/10012/15265

[60] Padmavathi Iyer and Amirreza Masoumzadeh. 2019. Generalized Mining of Relationship-Based Access Control Policies in Evolving Systems. In Proceedings of the 24th ACM Symposium on Access Control Models and Technologies (Toronto ON, Canada) (SACMAT '19). Association for Computing Machinery, New York, NY, USA, 135–140. https://doi.org/10.1145/3322431.3325419

[61] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. Proc. ACM Program. Lang. 3, OOPSLA, Article 149 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360575

[62] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. CoRR abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[63] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv preprint arXiv:1902.03383 (2019).

[64] Rich Jones. 2019. Gone in 60 Milliseconds: Intrusion and Exfiltration in Server-less Architectures. https://media.ccc.de/v/33c3-7865-gone$_i$n$_6$0$_m$illiseconds.

[65] Bendiab Keltoum and Boucherkha Samia. 2017. A Dynamic Federated Identity Management Approach for Cloud-Based Environments. In Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing (Cambridge, United Kingdom) (ICC '17). Association for Computing Machinery, New York, NY, USA, Article 104, 5 pages. https://doi.org/10.1145/3018896.3025152

[66] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[67] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[68] M. Koch, L. V. Mancini, and F. Parisi-Presicce. 2001. On the Specification and Evolution of Access Control Policies. In Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies (Chantilly, Virginia, USA) (SACMAT '01). Association for Computing Machinery, New York, NY, USA, 121–130. https://doi.org/10.1145/373256.373280

[69] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. 2002. A Graph-Based Formalism for RBAC. ACM Trans. Inf. Syst. Secur. 5, 3 (Aug. 2002), 332–365. https://doi.org/10.1145/545186.545191

[70] M. Koch, L. V. Mancini, and F. Parisi-Presicce. 2004. Administrative Scope in the Graph-Based Framework. In Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies (Yorktown Heights, New York, USA) (SACMAT '04). Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/990036.990051

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

[72] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Karim El Tal, and Stefan Tai. 2019. An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing. In Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (Auckland, New Zealand) (UCC'19). Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3344341.3368796

[73] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2019. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. Journal of Systems and Software 149 (2019), 340 – 359. http://www.sciencedirect.com/science/article/pii/S0164121218302735

[74] B. Reaves M. Meli, M. McNiece. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In Proceedings of the Networked and Distributed Systems Security Symposium (NDSS).

[75] Johannes Manner, Stefan Kolb, and Guido Wirtz. 2019. Troubleshooting Serverless functions: a combined monitoring and debugging approach. SICS Software-Intensive Cyber-Physical Systems 34, 2 (01 Jun 2019), 99–104. https://doi.org/10.1007/s00450-019-00398-6

[76] G. McGrath and P. R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). 405–410. https://doi.org/10.1109/ICDCSW.2017.36

[77] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. 2018. Retro-λ: An Event-sourced Platform for Serverless Applications with Retroactive Computing Support. In Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (Hamilton, New Zealand) (DEBS '18). ACM, New York, NY, USA, 76–87. https://doi.org/10.1145/3210284.3210285

[78] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-Time Serverless: Enabling Application Performance Guarantees. In Proceedings of the 5th International Workshop on Serverless Computing (Davis, CA, USA) (WOSC '19). Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3366623.3368133

[79] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[80] Matthew Obetz, Anirban Das, Timothy Castiglia, Stacy Patterson, and Ana Milanova. 2020. Formalizing Event-Driven Behavior of Serverless Applications. In Service-Oriented and Cloud Computing, Antonio Brogi, Wolf Zimmermann, and Kyriakos Kritikos (Eds.). Springer International Publishing, Cham, 19–29.

[81] Sylvia Osborn and Yuxia Guo. 2000. Modeling Users in Role-Based Access Control. In Proceedings of the Fifth ACM Workshop on Role-Based Access Control (Berlin, Germany) (RBAC '00). Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/344287.344299

[82] Per Persson and Ola Angelsmark. 2017. Kappa: Serverless IoT Deployment. In Proceedings of the 2nd International Workshop on Serverless Computing (Las Vegas, Nevada) (WoSC '17). Association for Computing Machinery, New York, NY, USA, 16–21. https://doi.org/10.1145/3154847.3154853

[83] Protego. 2020. Is AWS Lambda the Most Secure Application Platform? Probably. https://www.protego.io/is-aws-lambda-secure/.

[84] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[85] Mark Ryland. 2016. Identity and access management-based access control in virtual networks. US Patent 9,438,506.

[86] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In Proceedings of the 19th International Middleware Conference Industry (Rennes, France) (Middleware '18). Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3284028.3284029

[87] Tyler J. Skluzacek, Ryan Chard, Ryan Wong, Zhuozhao Li, Yadu N. Babuji, Logan Ward, Ben Blaiszik, Kyle Chard, and Ian Foster. 2019. Serverless Workflows for Indexing Large Scientific Data. In Proceedings of the 5th International Workshop on Serverless Computing (Davis, CA, USA) (WOSC '19). Association for Computing Machinery, New York, NY, USA, 43–48. https://doi.org/10.1145/3366623.3368140

[88] Nordstrom Technology. 2019. Hello, Retail! https://github.com/Nordstrom/hello-retail

[89] Ivonne Thomas and Christoph Meinel. 2010. An Identity Provider to Manage Reliable Digital Identities for SOA and the Web. In Proceedings of the 9th Symposium on Identity and Trust on the Internet (Gaithersburg, Maryland, USA) (IDTRUST '10). Association for Computing Machinery, New York, NY, USA, 26–36. https://doi.org/10.1145/1750389.1750393

[90] Kailas Vodrahalli and Eric Zhou. [n.d.]. Using Software-defined Caching to Enable Efficient Communication in a Serverless Environment. ([n. d.]).

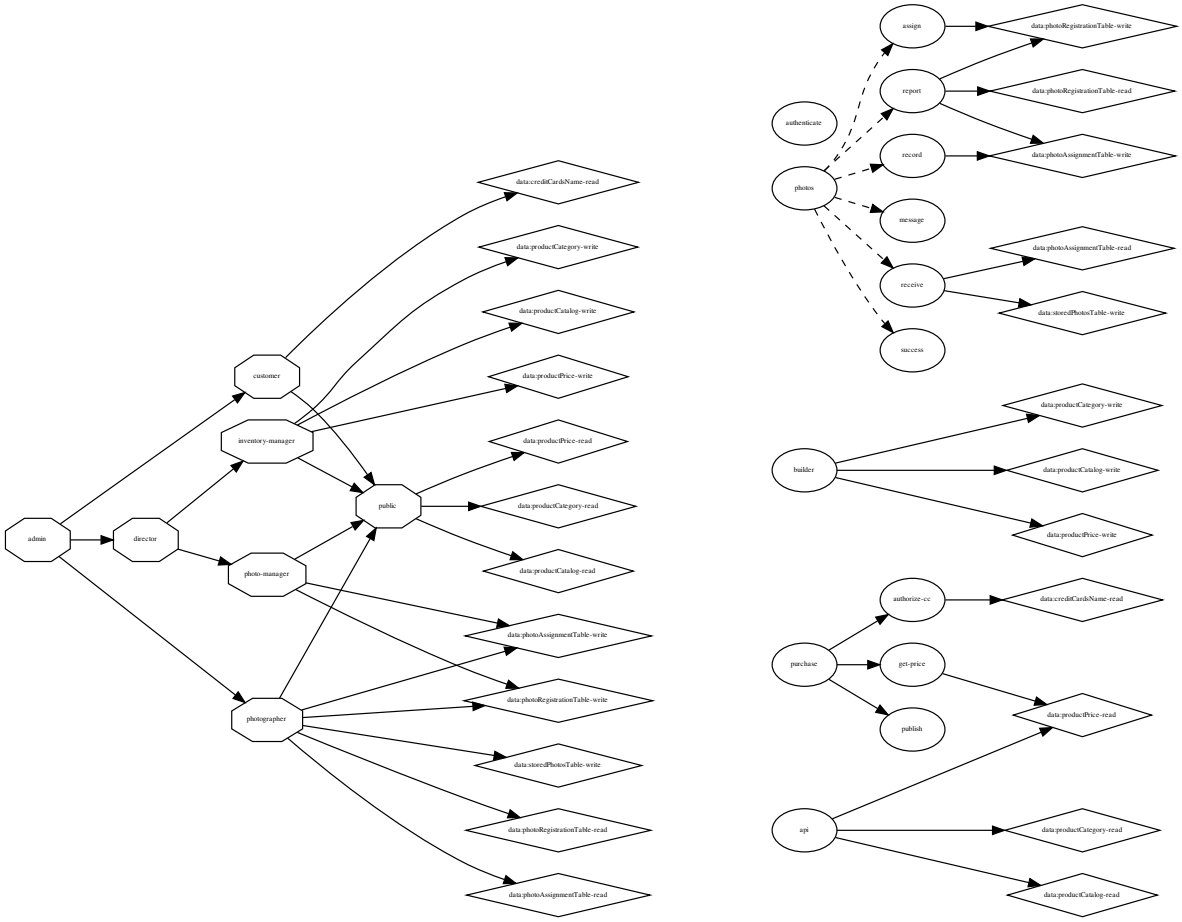
[91] He Wang and Sylvia L. Osborn. 2007. Discretionary Access Control with the Administrative Role Graph Model. In Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (Sophia Antipolis, France) (SACMAT '07). Association for Computing Machinery, New York, NY, USA, 151–156. https://doi.org/10.1145/1266840.1266865

[92] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[93] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. 2019. Video Processing with Serverless Computing: A Measurement Study. In Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (Amherst, Massachusetts) (NOSSDAV '19). Association for Computing Machinery, New York, NY, USA, 61–66. https://doi.org/10.1145/3304112.3325608

A HELLO, RETAIL! SECURITY POLICY

In order to configure our access control system for the Hello Retail application, we define the graphs for the Labeling State and Protection State graphs as shown in Figures 11a and 11b, respectively. These graphs follow the policy definitions laid out in Section 5. As can be seen in the Protection State graph, *Hello, Retail!* includes conditional dependencies that would make it difficult to enforce least privilege using traditional role-based access controls.



(a) Hello Retail Label State Graph **(b) Hello Retail Protection State Graph**

Figure 11: Hello Retail State Graphs