VALVE: Securing Function Workflows on Serverless Computing Platforms

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

Prabuddha Kumar
Stony Brook University
prabkumar@cs.stonybrook.edu

Tristan Morris
Silicon Valley Bank
tmorrismain@gmail.com

Michael Grace
Samsung Electronics
m1.grace@samsung.com

Amir Rahmati
Stony Brook University
amir.rahmati@stonybrook.edu

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

ABSTRACT

Serverless Computing has quickly emerged as a dominant cloud computing paradigm, allowing developers to rapidly prototype event-driven applications using a composition of small functions that each perform a single logical task. However, many such application workflows are based in part on publicly-available functions developed by third-parties, creating the potential for functions to behave in unexpected, or even malicious, ways. At present, developers are not in total control of where and how their data is flowing, creating significant security and privacy risks in growth markets that have embraced serverless (e.g., IoT).

As a practical means of addressing this problem, we present VALVE, a serverless platform that enables developers to exert complete fine-grained control of information flows in their applications. VALVE enables workflow developers to reason about function behaviors, and specify restrictions, through auditing of network-layer information flows. By proxying network requests and propagating taint labels across network flows, VALVE is able to restrict function behavior without code modification. We demonstrate that VALVE is able defend against known serverless attack behaviors including container reuse-based persistence and data exfiltration over cloud platform APIs with less than 2.8% runtime overhead, 6.25% deployment overhead and 2.35% teardown overhead.

CCS CONCEPTS

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

KEYWORDS

Serverless Computing, Information Flow, Security

ACM Reference Format:
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.* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3366423.3380173

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380173

1 INTRODUCTION

Serverless Computing is a new paradigm in cloud computing that abstracts away infrastructure management tasks like load-balancing and scaling from tenants, enabling them to focus solely on application development. It has become popular among industry practitioners, with less time spent on managing servers and reduced cost being cited as its most significant advantages [**?**]; the serverless market's growth is expected to exceed $8 billion per year by 2021 [**?**]. In serverless computing, tenants implement an application's logic as an interdependent set of functions, each of which performs a specific task. To achieve the broader goals of the application, these functions coordinate through interaction with other components including event triggers, message queues and object stores. Furthermore, the serverless ecosystem enables rapid prototyping [**?**] of applications, allowing tenants to use purpose-built functions from public markets [**? ? ?**], closed-source license-based functions [**? ?**], or third-party dependencies in their custom function code [**? ?**],

The diverse and distributed nature of serverless applications has led to the resurgence of prevalent cloud security issues. In an evaluation of 1000 open-source serverless projects, 21% of them contained critical vulnerabilities or misconfigurations [**?**]. Issues such as cross-tenant side-channels [**? ?**] are still present in the serverless ecosystem, as are canonical web application vulnerabilities [**?**] like event injection [**? ? ? ? ?**], access control and security misconfigurations [**? ?**], and bugs in library and platform code [**? ? ? ?**]. Past experience tells us that these vulnerabilities can be leveraged to gather information and steal secrets from cloud customers, at times leading to even more severe attacks [**?**].

While the existence of canonical security issues is unsurprising, the ephemeral nature of serverless infrastructure might suggest that such vulnerabilities are more difficult to exploit in practice. A function's lifecycle, from creation to destruction, typically spans just milliseconds of time; is it possible to exploit a function and perform malicious acts so quickly? Unfortunately, growing evidence suggests that determined attackers are finding creative work-arounds. One important attack strategy is to rapidly exfiltrate stolen data [**?**]. Worse, attackers have also discovered that persistent function compromise is possible – after writing malware or toolkits to an in-memory `/tmp` partition, attackers can game cloud platforms' "warm

container" re-use policy to cache a compromised copy of the function that persists across invocations [**?**]. Furthermore, simply restricting functions' network access is ineffective as attackers have developed methods of laundering stolen data through legitimate platform APIs in order to reach the open Internet [**? ?**].

Given the popularity of serverless platforms, it is unsurprising that industry has rushed forward with a variety of security solutions. Existing solutions primarily focus on traditional vulnerabilities, statically analyzing function source code and explicitly defining fine-grained policies before deployment [**? ? ? ? ? ?**]. Dynamic solutions have also been proposed that may be useful in safeguarding serverless computings' novel attack surfaces. For example, products that model function behavior using machine learning to detect anomalous behaviors [**? ?**] or wrap function event handler wrappers to inspect specific activities [**?**]. Other tools could assist in post-mortem attack investigation by tracing function activities and collecting error reports [**? ? ? ?**].

While promising, we note several limitations of existing techniques that prevent them from being a complete solution to serverless security. Many of these techniques require either source code access or specific knowledge of internal function state; this approach is simply not practical when considering the use of pre-compiled third-party objects (e.g., [**? ?**]) or proprietary closed-source functions (e.g., [**? ?**]). Moreover, existing products rarely consider the *interactions* between functions, giving rise to emergent attack vectors such as API-based data exfiltration [**?**]. Similarly, logging and debugging support in serverless platforms [**?**] lacks the ability to monitor a serverless application as a whole and therefore struggle to trace sophisticated attacks. Assumptions of unrestricted function access, combined with a function-centric perspective on serverless security, undermine the efficacy of these approaches.

In this paper we introduce Valve a transparent and *workflow-centric* approach to authorizing serverless information flows. Rather than analyzing or modifying function code, Valve deploys agents residing within each function-instance (i.e., container)[1] to monitor function's file and network behaviors. As a function-agnostic mean of interpreting network activity, we leverage the ubiquity of REST-based APIs in event-driven serverless applications, which are traced by a network proxy in each Valve agent. Agents dynamically generate taint labels that describe each function's file accesses and network requests. These labels are reported to a centralized controller. The Valve controller then aggregates this information to discover the flow paths of the application. These learned flows provide insight into the information flow across the serverless application. They also constitute a default security policy for the serverless application which can optionally be further restricted by the workflow designer. When set to enforcement mode, Valve uses this policy to mediate the network activities of all serverless components. Finally, to address the problems of persistent function compromise and cross-invocation side channels, Valve agents contain a garbage collection mechanism that sanitizes containers between invocations.

We implement Valve on OpenFaaS [**?**], an open-source serverless computing platform and evaluate it across three representative FaaS applications: (1) An open-source E-commerce serverless application, (2) a selection of popular OpenFaaS functions from its function store [**?**], and (3) exemplar Trigger-Action workflows frequently used in IoT environments. We demonstrate that Valve can protect workflow developers against common classes of attacks on serverless platforms, while providing them with auditing tools to understand the information flow across their applications. Our experiments show that Valve can provide complete, fine-grained information flow control across FaaS platform, while incurring less than 2.8% runtime overhead.

In summary, this paper makes the following contributions:

- We present Valve a serverless platform that enables dynamic information flow tracking and control in distributed function workflows to prevent data exfiltration and offer better observability into the serverless ecosystem.
- We thoroughly explore several exemplar case studies to demonstrate how Valve can be used to secure and audit real-world serverless application.
- We implement Valve on an open-source serverless computing platform, OpenFaaS [**?**], with Kubernetes [**?**] orchestration support. We evaluate the effectiveness of our system by providing security guarantees in an open-source serverless retail framework [**?**].

2 BACKGROUND

While early cloud computing solutions like IaaS (Infrastructure-as-a-service) gave enterprises access to seemingly infinite backend computing infrastructure [**?**], the serverless cloud paradigm has freed tenants from the burden of infrastructure management, allowing them to concentrate on application development. Serverless platforms adopt a pay-per-use model where users are billed according to fine-grained resource usage (CPU, memory and network), significantly reducing the application deployment cost [**?**]. Serverless developers implement application logic as interdependent sets of functions each performing a specific task. Individual functions are executed in response to different event source triggers (e.g., http web request, message broker services, object storage server events, cron jobs) and can be chained together to form workflows. In turn, the cloud service provider takes care of spawning and managing function instances (in isolated sandboxes or containers) according to developer-defined resource usage caps [**? ? ?**] in addition to handling load-balancing, auto-scaling and operational monitoring. To further ease the task of building serverless applications, assisting frameworks (e.g., [**? ? ?**]) are also available for orchestrating different serverless components like message queues, functions, and object stores.

Functions are intended to be stateless, meaning that the output of the function (usually returned to the client as JSON) should exclusively be the result of its explicit inputs. Accordingly, each function invocation should ideally take place in a "sterile" environment, such as a fresh container that is immediately destroyed following execution. In practice, however, due to the cost of setting up an entire runtime environment for each function execution, "warm" containers are cached and reused for future invocations of the same function within a pre-configured timeout window [**? ? ?**]. Opaque platform policies [**? ? ?**] and scheduling algorithm details obscure this practice, making it difficult for customers to account for such issues during application development.

[1] We will use 'function-instance' & 'container' interchangeably in the rest of the paper.

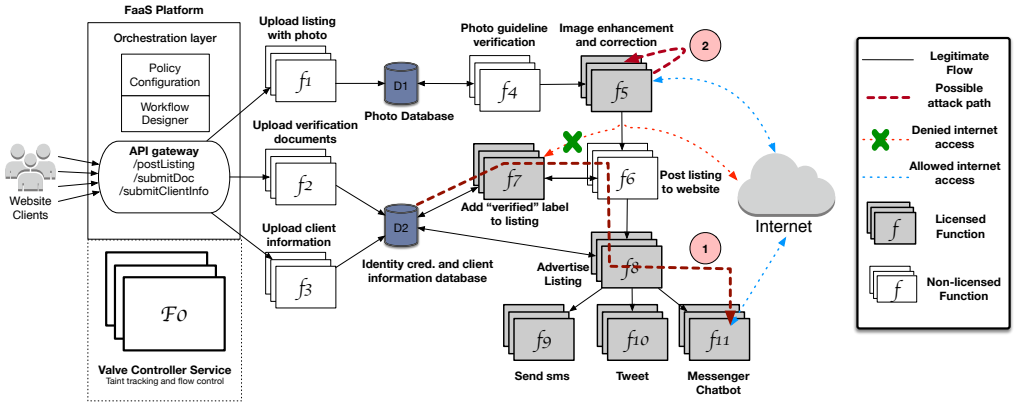


Figure 1: A reference architecture of serverless real estate listing website.

3 MOTIVATION

3.1 Serverless Application Scenario

Let us now consider a real-estate website that has transitioned to using a serverless cloud backend, as has already been done by popular companies including Realtor [?] and Zillow [?]. The website serves three stakeholders: the website owner, the customers of the website (real estate companies) and the clients of the customers (i.e. website users). An exemplar serverless architecture for such a website, based on these case studies made available by Amazon AWS [? ?], is shown in Figure ??. The website backend consists of several functions, data stores, and an API gateway. The API gateway exposes REST endpoints to customers for several website features, including: (1) posting home and apartment listings, (2) submitting identity proof documents to get a "verified" label on their listings and (3) uploading client information (e.g. phone number, email address) for targeted advertisements of new listings. The functions can be explicitly invoked by an orchestrated workflow or triggered by data store events. The orchestration layer of the FaaS platform provides different forms of inter-function communication (i.e. asynchronous callbacks, event sources, workflow designing tools)[2] to establish flow paths that connect the different functions. Finally, the developers also leveraged the FaaS platforms security settings to set a static network policy that restricts Internet access to all functions except where strictly necessary (e.g,. f_5, f_{11}).

In building the real estate website, the developers drew from a variety of heterogenous function sources, including custom logic, open source third-party functions, and proprietary services for which they pay licensing fees. Specifically, they make use of third-party software and packages for f_5 [? ?], f_7 [?], f_8 [? ?], f_9 [?], f_{10}, and f_{11} [? ? ? ?]. While these components dramatically simplify the process of building the website, they also introduce the application to code and dependencies that are outside of the direct control of the developer. One potential consequence of these dependencies is a data breach resulting from buggy or malicious functions; we introduce two plausible attacks in Table ??. The first attack demonstrates a method for bypassing the developer's static network policy by exfiltrating data through intermediate functions. Function f_7 is able to access customer credentials in database D_2, but cannot directly leak this data to the Internet.

[2] AWS step functions, for example. [?]

Attacks	Description	VALVE solution
Data leak through network (Attack ①)	Static network policies bypassed by passing data to downstream functions with network access	Network level taint tracking
Cross invocation side channel (Attack ②)	Residual data in warm containers leaked across invocations	File access taint tracking and function garbage collection

Table 1: Summary of attacks prevented by VALVE.

Instead, the attacker utilizes an indirect flow $f_7 \rightarrow f_6 \rightarrow f_8 \rightarrow f_{11}$ to leak credentials out via messenger chatbot (Fig ??, ①). The second attack demonstrates a method of surveilling website customers from the vantage point of a malicious co-tenant utilizing the same image enhancement service (Fig ??, ②) . Because f_5 is subject to container reuse, upon function compromise the attacker installs persistent malware that reads customer photographs during future invocations and transmits them to a remote location on the Internet.

3.2 Limitation of existing security tools.

A number of commercial security solutions may be useful in combating these threats, which we broadly categorize into *pre-deployment effort* and *runtime protection* tools. Pre-deployment efforts include language runtime libraries that secure a single function according to developer defined policies as part of the source code [? ? ?], static analysis of function source code and configuration files to detect violations of the principle of least privilege [? ? ?], and checking function dependencies against vulnerability databases [? ? ?]. These solutions are largely function-centric and their efficacy depends on the correctness of policies written by the function developers, complete access to source code and configuration files, and the compatibility of the tool with specific language runtimes, platforms, and event sources. Among other shortcomings, such pre-deployment security tools will be unable to reveal the implicit flows in Figure ?? due to lack of source code access (f_5,f_7,f_8).

Run time protections include machine learning based detection of anomalous function behaviors [? ?]. Other approaches attempt to prevent event-data injection prevention by inspecting incoming function invocation requests [?] using existing penetration testing techniques like sqlmap [?]. The robustness of the machine learning models, or the completeness of code injection analysis, determines the

Participant	Roles
Cloud Provider	Hosts Public FaaS Cloud Exposes security mechanisms to *Workflow Developer* Provides isolation between customers
Workflow Developer	Customer of *Cloud Provider* Designs serverless web application Specifies security policy
Function Writer	Publishes Functions used by *Workflow Developer* Functions include potentially-unwanted features

Table 2: Summary of participant roles in our system's design.

usefulness of such solutions. Another variation of runtime solutions focus on providing better observability into serverless component usage. Serverless platform providers offer execution tracing, error reporting, alerts, and monitored metrics of function executions [**? ? ? ?**]. However, existing monitoring techniques offer limited observability into the interactions between functions and most of these monitoring services are limited by strict usage limits [**?**]. Third party observability tools [**? ? ? ?**] offer more features (distributed tracing, cost analysis), but are limited to certain language runtimes and platforms. These solutions are mostly geared towards function-level protection and do not consider more complex information flow violations. For example, while individual transitions in the flowpath $f_7 \rightarrow f_6 \rightarrow f_8 \rightarrow f_{11}$ are all legitimate in the example real estate website, the entire flowpath should not occur and could indicate a serious data breach. Because the observability tools only track cross-component transitions in isolation, they would be unable to detect or prevent this attack. Moreover, we are aware of no approach that considers the matter of intra-container interactions (e.g., container reuse), which is necessary to prevent cross-invocation attacks (e.g., f_5).

3.3 Our approach.

To address emerging threats in the serverless ecosystem, we argue that a holistic approach is required, one that can mediate information flows across multiple functions as well as cross-invocation flows of the same function. To this end, we propose Valve, a security-enhanced serverless platform that performs runtime tracing and enforcement of network- and file-based information flow. Function-level operations are described as taint labels that can be propagated across workflows to capture inter-function security violations. Our solution is agnostic to function source (e.g., third party, closed-source), but can also be used in concert with more invasive security solutions when they are applicable.

Valve runs as a service within the FaaS platform, augmenting the orchestration layer to provide stronger security guarantees, described in Figure **??** as F_0. A Valve agent residing in every container performs function-level monitoring and taint tracking (not shown in Figure **??**). Valve enables network-level taint tracking to monitor network accesses of each function and generates restrictive dynamic flow policies to prevent attack ①. Valve prevents attack ② by file-access based taint tracking and garbage collection at the end of every function execution.

4 DESIGN

4.1 Threat Model & Assumptions

This work considers a typical public compute cloud, focusing specifically on 3 stakeholders that are summarized in Table **??**. A *Cloud Provider* hosts a public FaaS cloud that allows customers to design complex applications based on one or more functions, charging customers by the invocation. The Cloud Provider is responsible for assuring isolation between customers and making available security mechanisms (e.g., firewalls, virtual networks) that are directly configurable by the customer. We call these customers *Workflow Developers* (or simply *Developers*), as they leverage a number of functions to design complex heterogenous web services. Workflow Developers are also responsible for configuring the security mechanisms made available by the Cloud Provider. Third-Party functions are made available to the Workflow Developer by *Function Writers*. Function Writers may offer a licensed service to Developers (e.g.,[**? ? ?**]) or publish functions to public markets like the AWS Serverless Application Repository.[3]

The goal of the Developer is to design a web service that handles sensitive data such as personally-identifiable information, financial transactions, or user credentials. However, all functions used in workflows may contain vulnerabilities, as well as potentially unwanted features in the case of third-party functions. Thus, we assume that functions are untrusted and can behave arbitrarily, or even maliciously, with the goal of leaking sensitive data. The malicious functions may use any permissible system flow to exfiltrate data, including transmissions to the external network, writing to persistent storage somewhere in the cloud, or even writing to ephemeral storage inside the function container for later retrieval. The latter includes the possibility of cross-invocation side channels [**?**]. Multiple malicious functions may also collude, creating the possibility that legitimate flows through the system could be abused to exfiltrate data.

We make the following assumptions about this environment: First, we assume that the Cloud Provider is trusted and will not mishandle customer data or tamper with the Workflow Developer's security policies. We also assume that all serverless functions are invoked through use of REST API calls or other form of Remote Procedure Calls (event triggers, asynchronous callbacks). This assumption stands valid because web and API serving are the most popular use cases in the serverless paradigm [**?**]. Finally, our solution will introduce profiling and mediation mechanisms into parts of the FaaS platform, including function containers; we assume that mandatory access controls (e.g., SELinux [**?**]) are sufficient to prevent the attacker from being able to disable or subvert these components.

4.2 Design Goals

The limitations of existing tools discussed in section **??** inform the following high level design goals of our system:

G1 *Transparent*: FaaS benefits tremendously from the public sharing of functions under various licensing agreements, rendering unsuitable any solution that impedes function publishing. Thus, Valve should not require access to function source code, which may be unavailable, nor may it instrument functions, which may violate licensing agreements (e.g., [**? ?**]).

G2 *Widely Applicable*: Because the FaaS paradigm is language independent, Valve should make no assumptions regarding language runtimes, nor should it assume the presence of specific functions or components (e.g., Apache Kafka) which may not be present in all serverless applications.

[3] https://aws.amazon.com/serverless/serverlessrepo/

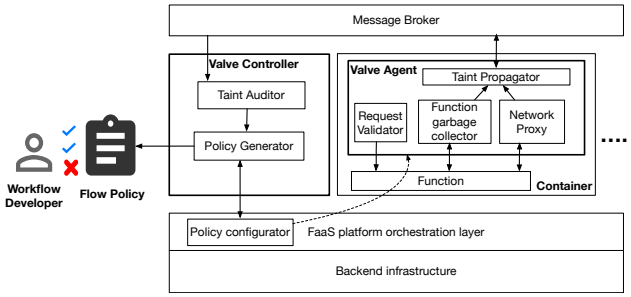


Figure 2: A reference architecture of VALVE.

G3 *Fine-Grained Observability*: The serverless concept belies the considerable complexity of individual functions; functions are often comprised of many control flow paths and may exhibit behaviors that are rarely seen during testing. To account for this, VALVE should support a fine-grained view of functions that permits it to differentiate between different control flow paths.

G4 *Dynamic Flow Control*: In light of the complexity of functions and their potential for exploitation, VALVE must be capable of dynamically determining the security level of a given application flow rather than employ a (potentially brittle) static policy that could lead to overprivilege (e.g., Attack 1 in Figure **??**).

4.3 Overview

This section provides an overview of the VALVE architecture. VALVE consists of two main components: the VALVE agent and the VALVE controller. Figure **??** provides an overview diagram of these components.

VALVE *Agent* The VALVE agent resides inside the function-instance container and monitors (and optionally mediates) activity on a per-invocation basis. This monitoring takes two forms, both of which are completely transparent to the operation of the function:

- **API Monitoring.** The Agent keeps track of every REST-based API call made by the function by proxying network requests issued from within the container. These calls are recorded as taints, where the taint is defined by the specific API call, and sent to the VALVE controller. When operating in enforcement mode, the VALVE Agent authorizes or denies each API request according to the active security policy.
- **Disk Access Monitoring.** The Agent records the disk activity of the function in order to track the possibility of information leakage through cross-invocation side-channels. When operating in enforcement mode, the VALVE Agent performs post-invocation garbage collection by deleting or truncating files that were modified by the function.

VALVE *Controller* The VALVE controller runs as a service alongside other functions in the cloud platform. It performs four tasks: 1) audits the information flows of the serverless application by accumulating the taints generated by VALVE agents, 2) generates a default application security policy based on the audit trace, 3) incorporates Workflow Developer-specified authorization rules and other configuration information into the security policy, and 4) orchestrates enforcement by pushing the security policy to the Request Validation component of the VALVE agents.

Listing 1: Function invocation request payload for function f_5 as shown in Figure ??.

```
{"input":[

    {"imageId":"image1","desc":"unique image id","type":"string"},
    {"customerId
":"Bob","desc":"website client owning the image","type":"string"},

{"imagePayload":"<imagedata>","desc":"input image","type":"base64"}
  ],
  "taints":[
    {"
label":"f_1","invocation-id":"1234","time":"2019-10-11T00:43:42Z"},
    {"label":"D1.datastore
.com","invocation-id":"1238","time":"2019-10-11T00:43:46Z"},
    {
"label":"f_4","invocation-id":"1239","time":"2019-10-11T00:43:50Z"}
  ],
    ...
  ]
 }
```

4.4 VALVE Agent

In standard FaaS design, 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. The VALVE agent works alongside the request handler to transparently monitor the execution and to enforce important security features of VALVE. The VALVE agent consists of several sub-units.

4.4.1 Network Profiler. A central challenge in our design is tracing function activities while avoiding dependence on source code, instrumentation, or assumptions of language (**??**). To address this, we leverage the ubiquity of REST-based APIs in the serverless ecosystem. The event-driven nature of serverless prompts function to lean heavily on Web and API design paradigms [**?**], such that most of the interactions take place in form of REST based API calls. These APIs are not only function-agnostic, but due to the nature of URLs define a resource hierarchy that is easy for human agents to interpret. For example, it is intuitive that the REST endpoint *https://api.github.com/users/Bob/repos* responds with a list of GitHub repositories of user Bob. The generic and intuitive properties of these APIs make them an ideal mediation point within our architecture.

To observe API usage, VALVE deploys a transparent forward proxy in each container that begins proxying network requests when the container starts running. This network proxy fields all incoming and outgoing network requests originating from the container. To address the matter of encrypted traffic, the VALVE Agent contains an HTTPS proxy (*mitmproxy* [**?**]), which is a common technique in enterprise environments for monitoring security-sensitive network flows [**?**]. Thus, the network proxy serves as a transparent mediation point that allows us to observe the dynamic behavior of functions, identify hidden flow paths, and ultimately derive flow control policies (**??**).

We extend the mitmproxy-based network proxy to perform *network level tainting*. The proxy inspects REST call headers and body fields to determine appropriate labels with which to taint the current flow. The generated taint is sent to the VALVE controller along with an invocation ID and the time of invocation. Additionally the Agent propagates the cumulative taint for the current workflow along with all outbound API requests from the current function; an example of

how we extend these requests is given in Listing **??**. Since each invocation request is a unit of work in FaaS, functions are short-lived and taint labels are assigned per request. As a result, the "taint explosion" problem that commonly affects taint analysis tools is not an issue in this domain. Moreover, the network taints can be summarized when a function makes multiple calls to the same domain, providing significant compression of taint labels.

4.4.2 API Request Validator. In protecting a serverless application, one advantage enjoyed by the system defender is that functions provide clear and well-defined interfaces. The request validator component leverages this advantage to create an enforcement point for function activities.

Upon creation of the function-instance container, the API Request Validator registers with the Valve Controller in order to receive the relevant policy module in the application's security policy. We will discuss the Valve policy language in greater detail in Section **??**, but examples of function modules are given in the policies depicted in Figure **??**. When a function invocation request is received in the container, the Request Validator extracts the taint label from the inbound request (see Listing **??**), and references the policy to determine if the request is authorized. The Request Validator can allow, deny or forward the requests according to the flow policies enforced by the Valve controller (**??**).

If the request is prohibited according to the security policy, the Validator will drop the request silently and return error code to the caller function module (**??**). In typical applications, silently terminating an access request would frequently result in cascading failures that could cause the application to crash. Fortunately, the ephemeral nature of functions means that we do not need to worry about the stability of the application following a denied access request – at worst, this denial will cause a single web session workflow to fail. The API Request Validator will only interfere with legitimate workflows if the security policy instructs it to do so; we describe the process through which the Workflow Developer can iteratively specify an effective security policy in Section **??**.

4.4.3 File Access Profiler. Cross-invocation interference is an important tool for serverless adversaries that enables persistence as well as surveillance of application customers [**?**], as shown in Attack ② of our motivating example. While Valve's network-layer profiling will allow us to trace and prevent data exfiltration attempts, it is also necessary to monitor file access behaviors to deny attackers these cross-invocation capabilities.

Each Valve Agent contains a system call tracing mechanism for monitoring function file I/O, allowing Valve to detect cross-invocation flows resulting from container reuse (**??**). The tracer intercepts and records all system calls issued by the function using the *strace* utility. For file operations (`open`, `read`, `write`, `lseek`, etc.), the tracer records the accessed file, offset, and bytes accessed. After the function finishes execution, all data on disk that was modified by the function is erased from the container. Because current attacks require an explicit data flow from one function execution to another, this procedure is sufficient to deny cross-invocation capabilities to the attacker.

We make the following observations about the practicality of this approach. First, we note that cross-invocation data flows violate the serverless abstract in which functions are intended to be stateless.

⟨policy⟩	::=	⟨function⟩ : { ⟨ruleset⟩ } ⟨policy⟩ | ϵ
⟨ruleset⟩	::=	⟨direction⟩ : { ⟨rules⟩ } |
		⟨direction⟩ : { ⟨rules⟩ } ⟨ruleset⟩
⟨direction⟩	::=	`Ingress` | `Egress`
⟨rules⟩	::=	⟨condition⟩ ⟨action⟩ ⟨subject⟩ |
		⟨condition⟩ ⟨action⟩ ⟨subject⟩ , ⟨rules⟩
⟨condition⟩	::=	`NULL` | `if` ⟨labels⟩ `in {` ⟨flowtaint⟩ `}`
⟨flowtaint⟩	::=	⟨labels⟩ | ⟨labels⟩ , ⟨flowtaint⟩
⟨labels⟩	::=	$T_1 ... T_n$
⟨action⟩	::=	`Allow` | `Deny` | `FwdTo` ⟨ip⟩
⟨subject⟩	::=	`NULL` | ⟨web-address⟩ | ⟨port⟩ |
		⟨function⟩ | ⟨restapi⟩
⟨port⟩	::=	⟨number⟩ : ⟨protocol⟩
⟨function⟩	::=	⟨url-string⟩
⟨restapi⟩	::=	⟨method⟩ ⟨web-address⟩
⟨method⟩	::=	`GET` | `PUT` | `POST` | `DELETE`
⟨web-address⟩	::=	⟨ip⟩ | ⟨url-string⟩
⟨protocol⟩	::=	$prot_1$ | $prot_2$ | ... | $prot_k$
⟨ip⟩	::=	*ip address*
⟨url-string⟩	::=	*web url*
⟨number⟩	::=	`[0-9]+`

Table 3: Policy Specification Grammar in Valve.

There is therefore no legitimate purpose for cross-invocation data flows, making our garbage collector compatible with existing functions. Second, because commercial platforms [**? ?**] provide only a small writable partition (e.g., just `/tmp`) using an in-memory filesystem, our approach is significantly more efficient than destroying and reprovisioning the entire container.

File Access tainting Garbage collection is sufficient to prevent explicit cross-invocation data flows by denying persistence to the attacker; however, this is not to say that file access behaviors (specifically reads) will not inform the security of the application. For example, the function-instance may be provisioned with sensitive information (e.g., authorization tokens) that itself may be the target of attacks. To account for this, Valve Agents also report taint labels on file accesses. From the full set of file I/O calls captured by `strace`, the Agent filters all files not accessed with read permission. This list is further filtered to remove special Linux virtual files (e.g. `dev`, `proc`, network sockets) since these files do not result in security-sensitive data flows [**?**]. The remaining list of accessed files are included in the taint label of the function invocation. Since garbage collection is performed after each function invocation, the set of modified files are not tainted by previous function executions and thus can be excluded from taint-label generation. This step permits workflow designers to express policies that constrain the flow of file-based data through the application.

4.5 Valve controller

The Valve controller runs as a service in the FaaS platform and coordinates among the different application components to manage the application workflow. It consists of the following components:

4.5.1 The Flow Control Policy Model. At the core of Valve is an expressive policy model for controlling information flows in serverless applications. Our policy grammar is given in Table **??**. Each *policy*

```
• • •

f7:
    ingress:
      - If taint-label in {f1,D1,f4,f5,f6} Allow f6
    egress:
      - If taint-label in {f1,D1,f4,f5,f6,f7} Allow
        https://D2.datastore.com #database D2 url
f6:
    ingress:
      - If taint-label in {f1,D1,f4,f5} Allow f5
    egress:
      - If taint-label in {f1,D1,f4,f5,f6} Allow f7
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7} Allow f8
f8:
    ingress:
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7} Allow f6
    egress:
      - If NULL Allow https://D2.datastore.com
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7,f8}
        - Allow f9
        - Allow f10
        - Allow f11
f11:
    ingress:
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7,f8} Allow f8
    egress:
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7,f8,f11}
        - Allow http://cerberhhyed5frqa.fkr84i.win
        - Allow https://bestboy.top
        - Allow POST https://www.googleapis.com/upload/gmail/v1/
          users/userId/messages/send
        - Allow POST https://graph.facebook.com/v4.0/me/messages?
          access_token=<PAGE_ACCESS_TOKEN>

• • •
```

(a) Default Policy generated by VALVE.

```
• • •
f7:
    ingress:
      - If taint-label in {f1,D1,f4,f5,f6} Allow f6
    egress:
      - If taint-label in {f1,D1,f4,f5,f6,f7} Allow
        https://D2.datastore.com #database D2 url
f6:
    ingress:
      - If taint-label in {f1,D1,f4,f5} Allow f5
    egress:
      - If taint-label in {f1,D1,f4,f5,f6} Allow f7
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7} Allow f8
f8:
    ingress:
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7} Allow f6
    egress:
      - ~~If NULL Allow https://D2.datastore.com~~
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7,f8}
        - Allow f9
        - Allow f10
        - Allow f11
f11:
    ingress:
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7,f8} Allow f8
    egress:
      - If taint-label contains {D2}
        - Deny https://www.googleapis.com/upload/gmail/v1/
          users/userId/messages/send
        - Deny https://graph.facebook.com/v4.0/me/messages?
          access_token=<PAGE_ACCESS_TOKEN>
      - If taint-label in {f1,D1,f4,f5,f6,D2,f7,f8,f11}
        - ~~Allow http://cerberhhyed5frqa.fkr84i.win~~
        - ~~Allow https://bestboy.top~~
        - Allow POST https://www.googleapis.com/upload/gmail/v1/
          users/userId/messages/send
        - Allow POST https://graph.facebook.com/v4.0/me/messages?
          access_token=<PAGE_ACCESS_TOKEN>
• • •
```

(b) Refined Policy updated by the Workflow Developer.

Figure 3: Security policies for the Real Estate website in Figure ??.

is defined by a set of *function* policy modules governing the permitted `Ingress` and `Egress` behaviors of the function. Each *rule* in the function policy can be expressed as a *condition* over the active taint *labels* of the flow to specify the permissibility of data exchanges with a given *subject* (e.g., web address, function, API call). Rules can either be restrictive or permissive (i.e., `Allow` or `Deny`). The restrictive rules are evaluated at the enforcement point prior to permissive. As we describe below, permissive rules are used by the VALVE Controller as it automatically produces a whitelist policy after profiling the application, while restrictive rules can be added by the developer to exert further control over information flow.

In designing this policy model, our objective was to adhere closely to existing security paradigms in cloud computing. We note that, by defining our policy in terms of ingress and egress rules over functions, our policy is actually an extension of the existing standard network policy specifications on commercial cloud platforms. Our adherence to this style of policy specification demonstrates a possible path forward for integrating information flow controls into production cloud systems. Below, we explain how VALVE assists the developer in specify a workflow-wide security policy through taint auditing and automated policy generation, then provide a guided example of how our policy can be used to secure the exemplar application presented in Section **??**.

4.5.2 Taint Auditor. This component serves as an orchestration mechanism for the network and file taints generated by the VALVE Agents. The Taint Auditor gathers taint labels sent from function-instances after each invocation and propagates the taints to downstream functions in the workflow. As a result, each flow in the application accumulates taints as it progresses through the workflow. The Taint Auditor also records application-side taint data in a centralized log. This log will be used to derive a default security policy, but also assist the developer in understanding and iteratively debugging function workflows.

4.5.3 Policy Generator. Specifying an effective security policy is a notoriously difficult problem (see, e.g., verification efforts of SELinux policies [**???**]) – failure to adequately restrict flows violates the principle of least privilege and leaves the system vulnerable (e.g., [**?**]), but defining overly-restrictive rules prevents the correct operation of the system (e.g., [**?**]). While a complete solution to policy specification is well beyond the scope of this work, VALVE simplifies the process of policy writing through the inclusion of an automated policy generator capable of profiling serverless applications in order to define a default whitelist security policy. Inspired by SELinux's `audit2allow` utility,[4] VALVE supports a permissive mode in which the taint audit log is used to produce an `Allow` rule for every flow observed by the Taint Auditor. Following this application profile phase, the default policy is presented to the Workflow Developer . This default policy not only serves as a starting point for iterative policy refinement, but

[4] https://linux.die.net/man/1/audit2allow

also summarizes the contents of the audit log into a definitive guide on the behavior of the constituent functions within an application.

After the default policy is derived, Valve can be switched into enforcement mode, at which point the API Request Validators within the Valve Agents will begin to deny accesses that do not adhere to the function ruleset. While our intent is for Workflow Developer to add and modify rules, it is worth noting that the default security policy already enforces least privilege on information flows within the application. We envision the primary reasons for further revision to the policy will be 1) the Workflow Developer wishes to deny unwanted functionality contained in third party functions, and 2) the Workflow Developer identifies coverage gaps from the application profiling phase that require rule modification. During enforcement mode, the Taint Auditor assists in these revisions by logging unauthorized accesses, enabling the Workflow Developer to iteratively refine the active security policy.

4.5.4 Policy Case Study. We now return to our motivating example (Figure **??**), to demonstrate how Valve can be used to prevent attacks on serverless applications. In the real estate application, it is necessary for the function f_6 to to communicate with f_7 in order to support verified listings. It is also necessary for the website to advertise new listings, which among other data flows includes f_6 transmitting data that reaches the open Internet via f_{11} (i.e., $f_6 \rightarrow f_8 \rightarrow f_{11}$). The Valve Policy Generator will therefore produce a ruleset that permits $D_2 \rightarrow f_7 \rightarrow f_6 \rightarrow f_8 \rightarrow f_{11}$; unfortunately, this is the precise authorization needed by the attacker to exfiltrate the contents of D_2.

This default security policy generated by Valve is given in Figure **??**. The taint auditor observes that f_7 receives incoming traffic from https://D2.datastore.com (database D_2) and then invokes the downstream function f_6. Similarly, information flows from $f_5 \rightarrow f_6$ and $f_6 \rightarrow f_8$ are authorized. The taint auditor also observes f_{11} contacts malicious domains http://cerberhhyed5frqa.fkr84i.win and https://bestboy.top [**?**]. Moreover, f_{11} invokes a non-malicious gmail API to send email, although it is only supposed to invoke messenger apis. The policy generator allows all such flows during training phase and generates the default policy to present to the Workflow Developer.

After profiling the application to produce this default policy, the Workflow Developer audits the system and is surprised to see that their licensed functions are contacting some less-than-reputable domains within the advertising network, and is also using an unadvertised gmail API for some unknown communication. This last flow may serve a legitimate purpose, but it is unclear to the Workflow Developer because the source code is proprietary. As a result, the Workflow Developer becomes concerned that there exists a flow from their credential database to the Internet. They are unable to use existing network controls provided by the cloud to address this problem because such mechanisms are not information flow-based.

To address these shortcomings, the Workflow Developer makes the following modifications to the policy, which is given in Listing **??**. Having written f_6, the Workflow Developer is certain that data from D_2 should not reach the advertising functions; in fact, only events triggered by f_5 prompt f_6 to post a new listing. To solve this problem, the Workflow Developer specifies a new restrictive rule that denies egress from f_{11} under any circumstances if taint label T_{D2} is active on the flow. This allows a middle ground between usability and security, because the Workflow Developer does not know if the gmail API is essential to f_{11}'s functionality. Now, the gmail API nor any other egress flow may carry data from D_2. Regardless of the possibility of data exfiltration, the Workflow Developer is also concerned about advertising functions contacting known-malicious domains. To address this, they strike the rules from f_{11}'s policy that permitted these connections.

Workflow	Function	Description
Product Catalog	product-catalog-builder	The owner can add products to the catalog
	product-catalog-api	The products can be fetched from the catalog
Product Purchase	product-purchase-authenticate	Authenticate the username and password
	product-purchase-get-price	Fetch the item price
	product-purchase-authorize-cc	Authorize the card for the transaction
	product-purchase-publish	Publish whether the transaction succeeded or failed
Product Photos	product-photos-assign	Assign the task to click a photo to a photographer
	product-photos-message	Message the photographer about the assignment
	product-photos-record	Update the db to reflect the assignment
	product-photos-receive	Receive the photograph from the photographer
	product-photos-success	Denote successful receiving of the photograph
	product-photos-report	Update the database to reflect the completion of the assignment

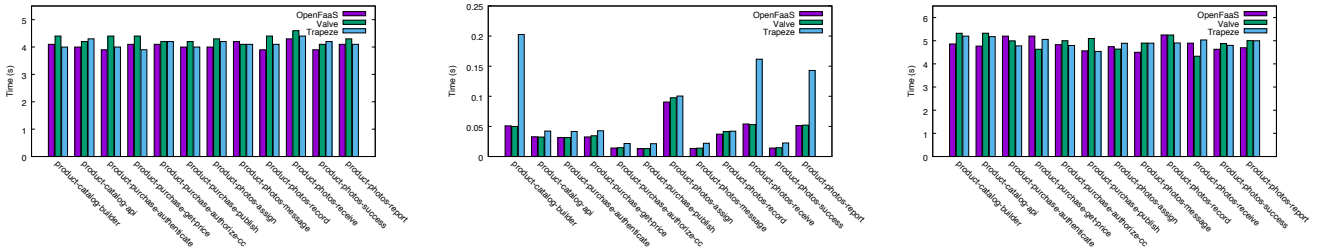
Table 4: Summary of Hello,Retail! functions.

5 IMPLEMENTATION

We implemented Valve using programming language Go 1.12 on top of OpenFaaS version 0.18.1 [**?**], an open source serverless computing platform. We have minimally modified (33 lines of Go code excluding build scripts, comments and blank lines) the OpenFaaS source code to instrument the request handler in each container to proxy network requests and track system calls as part of Valve agent. This demonstrates that the principles of Valve can be easily incorporated into existing systems making it widely applicable (**??**). We have used `mitmproxy` [**?**] python library to proxy HTTP and HTTPS requests generated by the functions. The system call tracing mechanism is implemented using the *strace* utility. The Valve controller is implemented as an independent service that runs on top of OpenFaaS without requiring any change in the underlying platform. Further add-ons and improvements can be easily integrated into Valve by adding the functionalities as vertical services on top of the underlying platform (e.g., OpenFaaS).

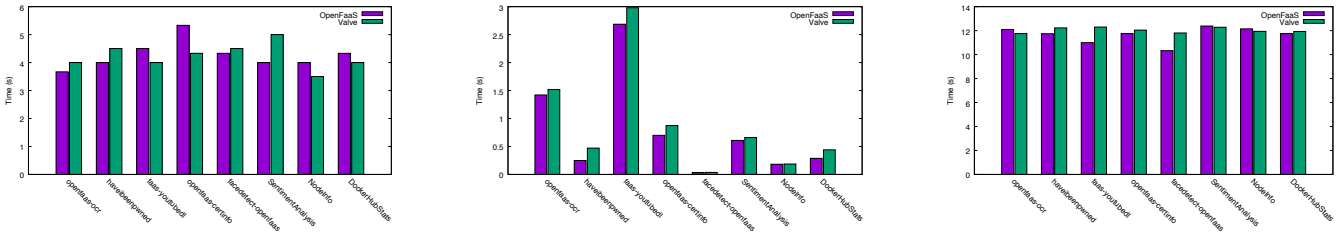
6 EVALUATION

In this section, we evaluate the performance of Valve. We deployed Valve on a server-class machine with 8-core Intel(R) Xeon(R) CPU E5620 @ 2.40GHz and 12 GB memory running Ubuntu 16.04.6 LTS 64 bit. Using this server, we created a Kubernetes v1.16.0 cluster with Docker runtime version 18.09.0 to conduct all our experiments. We compare the performance of Valve against two baselines – a standard OpenFaaS deployment (Vanilla) and a language-based information flow control mechanism (Trapeze) [**?**]. To ensure the generality of our results, we run each experiment against several different corpora of serverless applications:



(a) Function Deployment (b) Function Invocation (c) Function Teardown

Figure 4: Performance characteristics of VALVE for the *Hello, Retail!* functions as compared to vanilla OpenFaaS baseline and Trapeze.



(a) Function Deployment (b) Function Invocation (c) Function Teardown

Figure 5: Performance characteristics of VALVE for the *OpenFaaS Store* functions as compared to vanilla OpenFaaS baseline.

Corpus	Function	Vanilla OpenFaaS		OpenFaaS with VALVE	
		Size (MB)	Time (s)	Size (MB)	Time (s)
Hello, Retail!	catalog-builder	101.0	59.19	169.0 (67.33%)	81.71 (28.21%)
	catalog-api	105.0	62.91	173.0 (64.76%)	88.49 (40.66%)
	purchase-authenticate	101.0	57.81	169.0 (67.33%)	81.61 (41.17%)
	purchase-get-price	101.0	58.19	169.0 (67.33%)	96.68 (66.15%)
	purchase-authorize-cc	101.0	58.00	169.0 (67.33%)	86.22 (48.66%)
	purchase-publish	98.5	38.66	162.0 (64.47%)	55.64 (43.92%)
	photos-assign	103.0	56.49	169.0 (64.08%)	82.20 (45.51%)
	photos-message	90.3	37.16	161.0 (78.29%)	66.36 (78.58%)
	photos-record	103.0	57.16	169.0 (64.08%)	88.19 (55.55%)
	photos-receive	108.0	65.08	174.0 (61.11%)	121.93 (87.35%)
	photos-success	90.3	37.28	161.0 (78.29%)	60.83 (63.17%)
	photos-report	105.0	64.37	173 (64.76%)	91.48 (42.12%)
OpenFaaS store	openfaas-ocr	511.0	105.31	530.0 (3.72%)	118.19 (12.23%)
	haveibeenpwned	31.3	53.93	98.5 (214.70%)	60.30 (11.81%)
	faas-youtubedl	74.4	23.69	143.0 (92.20%)	48.97 (106.71%)
	openfaas-certinfo	29.4	30.22	97.0 (229.93%)	41.60 (37.66%)
	facedetect-openfaas	80.8	70.13	142.0 (75.74%)	84.16 (20.00%)
	SentimentAnalysis	370.0	51.22	443.0 (19.73%)	86.47 (68.82%)
	NodeInfo	64.5	19.82	134.0 (107.75%)	32.99 (66.45%)
	DockerHubStats	26.0	40.33	87.6 (236.92%)	55.60 (37.86%)
TA Workflows	gmail-trigger	603.0	639.35	686.0 (13.76%)	666.66 (4.27%)
	slack-notification	112.0	37.28	191.0 (70.54%)	91.22 (144.66%)
	test-flow	14.2	29.48	87.2 (514.08%)	80.10 (171.67%)
	geolocation-service	114.0	38.71	193.0 (69.30%)	59.21 (52.95%)
	update-sheet	131.0	66.45	204.0 (55.73%)	94.43 (42.11%)
	location-flow	14.2	30.06	87.2 (514.08%)	77.19 (156.82%)

Table 5: Build Time (MB) and Build Size (s) for VALVE as compared to Vanilla OpenFaaS. Percent overhead is given in parenthesis.

- *Hello, Retail! [?]* This is an event-driven AWS serverless application that leverages AWS services like Lambda, Kinesis, Dynamo and S3. We base our experiments on Alpernas et al.'s fork of *Hello, Retail!* which replaces calls to the AWS S3 and Dynamo databases with their own data score [?]. This allows us to compare VALVE to their Trapeze system, which is a language-based information flow control solution [?] system. We present the description of constituent functions of this application in Table ??.
- *OpenFaaS Store Functions [?].* OpenFaaS provides a function store [?] containing community developed serverless functions for public use which closely resembles our motivating scenario. We selected 8 functions from this store to include in our evaluation suite as listed in Table ??.
- *Exemplar Trigger-Action Workflows.* With the growth of IoT, we envision that trigger-action platforms will move to serverless platforms due to their support for event triggers and actions [? ?]. We implemented two proof-of-concept workflows based on trigger-action rules: 1) an *Email-Slack* workflow in which the gmail service (f_1) notifies an orchestration function (f_2), causing a slackbot f_3 to post an email notification to a slack channel; and 2) a *Location Sheets* workflow in which a location service (f_1) sends updates of a user's location to an orchestration function (f_2), which are then recorded in a Google docs spreadsheet (f_3).

6.1 Build Costs

We first measure the pre-deployment cost of building and storing a function image with VALVE. These costs are *one-time* and primarily incurred by the cloud provider. The build size and built time for each function corpus are given in Table ??. On average, the increase in image size imposed by VALVE is 97%, while the average increase in build time is 51%. The primary source of these overheads is caused by adding `mitmproxy` to the container image. This is because `mitmproxy` is a mature and trusted software artifact that is considerably more feature-rich than what is strictly needed by our VALVE implementation. VALVE's design is ultimately independent of the proxy used and `mitmproxy` can be replaced with lightweight alternatives.

While considerable, we note that these overheads are a one-time cost and that large image sizes are a long standing problem in the container ecosystem [**?**]. One possibility for bringing down these overheads is the use of container debloating techniques, e.g., [**?**].

6.2 Runtime Costs

We next measure the set of runtime costs imposed by Valve. Unlike the build overhead, these costs are largely passed on to the serverless customer (or Workflow Developer) and are therefore of vital importance. We consider three factors affecting runtime performance: *Function Deployment*, *Function Invocation*, and *Function Teardown*. Function Deployment measures the time it takes Kubernetes to transition the container from the "Container Creating" state to the "Ready" state. Function Invocation measures the RTT of a request to the function issued by another host on the local network. Function Teardown is a measure of the time required to transition the container from the "Running" state to the "Terminated" state. We present these results for the *Hello, Retail!* corpus in Figure **??** and the *OpenFaaS Store* corpus in Figure **??**, each experiment averaged over 50 repetitions, omitting the similar *TA Workflows* results for brevity.

Deployment time is only minimally impacted by the presence of Valve. With Valve enabled, we measured an average deployment overhead of just 6.25% . The most critical performance cost, function invocation time, enjoys similarly modest overheads with Valve. The average runtime overhead of Valve is just 2.8% per invocation. Valve exhibits fairly constant runtime overhead compared to Trapeze where overhead varies depending on function internals as shown in Figure **??**. To achieve transparency, Valve's flow control model is intentionally coarse-grained. Compared to Trapeze which offers flow control at memory-level while being more intrusive, expensive and onerous for developers, Valve offers lightweight handles to developers to control functions at the file-access and network-access level. Valve invocation overhead is dependent on the number of network requests proxied by the Valve Agent; the worst-case scenario for our performance would be a function that issues many serialized network requests. This access pattern could potentially appear, e.g., in an orchestration function that communicates with many other functions, but we did not observe it in our tests.

The teardown cost reflects the time required for terminating a running function. As a result, we observe performance characteristics similar to deployment costs for both function corpora, with Valve seldom outperforming vanilla OpenFaaS due to scheduling artifacts. The average function deletion cost is 2.35% in Valve.

7 RELATED WORK

Serverless computing attacks. Arbitrary code execution, imperfect tenant isolation (i.e. VM and function co-location vulnerabilities) and the ability to gather knowledge about runtime and infrastructure are common issues in serverless platforms [**? ?**]. Since functions communicate to different event sources, object stores and integrated third-party services, it leads to broader and newer attack surfaces susceptible to canonical vulnerabilities [**?**]. For Example, event injection attacks [**? ? ?**] may target the function source code, other secrets stored in the container [**? ?**], or databases associated with the function. Access control and security misconfigurations (e.g., read/write access to object stores accessed by the function, long timeout, buggy configuration rollout [**?**]) enables attackers to retrieve sensitive informative as part of reconnaissance [**?**] or launch denial-of-service (or *denial-of-wallet* [**? ?**]) attacks by exhausting allocated resource limits and expanding usage bill. Common vulnerabilities and bugs in SDKs, third-party libraries and platform code [**? ? ? ? ?**] plague serverless functions. Existing security solutions [**? ? ? ? ? ? ? ? ? ?**] each solve some part of these problems. Valve adds to the growing set of artilleries against serverless attacks.

Serverless Security Research. Alpernas et al propose Trapeze [**?**], a language-based approach to dynamic information flow control. Trapeze wrap each serverless function in a security shim that intercepts data access from shared data stores, external communication channels (i.e. Internet), and messages exchanged with other functions. Similar to Valve, the shim tracks information flow and enforces a global security policy based on a combination of static and dynamic security labelling (i.e., **??**, **??**). However, Trapeze places a greater burden on developers to correctly define information flow policies and implement declassification functions, whereas Valve assists workflow developers in policy specification and employs a transparent coarser-grained (i.e., function-level) information flow model that does not require declassification (i.e., **??**). Further, Trapeze makes assumptions about the programming language of the serverless function, violating goal **??**. Trapeze completely forgoes serverless warm-start performance optimizations and faces higher overheads than Valve; the fork-optimized Trapeze does not work for some API calls and requires effort at the external API implementation level to fix it. We believe that, where applicable, Trapeze is complementary to Valve in providing robust defense against data breach attacks in serverless cloud.

Several studies have surveyed the security of different serverless computing platforms. Baldini et. al. examined several popular platforms and concluded the lack of proper function isolation is a major problem [**?**]. Solving this problem is a major driving force of Valve. Wang et. al. have measured scalability, cold-start latency, instance lifetime and several other metrics in Google Cloud functions, Microsoft Azure Functions and AWS lambda [**?**] . They discovered that Azure Functions have exploitable placement vulnerabilities and that the ability to run arbitrary binary code in containers makes them vulnerable to many kinds of side-channel attacks. However, they did not suggest defenses to these attacks, which is the central concern of our work. Some researchers have attempted to formally model serverless platforms [**? ?**], and perform semi-automated troubleshooting based on log data [**?**], to ease reasoning about function behavior. This may aid in the security analysis of functions, but is orthogonal to the goals of Valve.

Serverless design. Another line of research in the serverless computing domain attempts to improve serverless architectural designs. Hendrickson et al. address the high memory overhead of storing paused containers in memory and improved load balancing to exploit session, code and data locality [**?**]. Sock extends OpenLambda to decrease the cold start latency of the containers to reduce response time [**?**]. Sock leverages lightweight linux primitives to replace costly container initialization mechanisms and process caching, i.e. usage of common system libraries for all containers to reduce the overall initialization cost. However, they admit that their process-caching

methodology has security implications. SAND achieves resource efficiency by application-level sandboxing by orchestrating function executions of the same application locally [**?**]. McGrath et al. propose metrics to evaluate the execution performance of serverless platforms [**?**] and identify hosting arbitrary code in containers on multi-tenant systems as a potential security problem which calls for attention. Combining SDN and serverless computing [**? ?**] and serverless execution parallelization [**?**] to achieve performance improvement, better serverless programming models [**?**], serverless pricing models [**?**], serverless analytics optimizations [**? ? ?**] are active research areas in the serverless paradigm.

8 CONCLUSION

As our understanding of serverless computing (in)security evolves, it has become clear that many of the threats to FaaS are fundamentally based on violations of information flow. In this work, we have presented Valve, a generic and transparent solution for dynamic information flow control. We have demonstrated that Valve can mitigate common classes of attacks against serverless platforms, and also assist workflow developers in auditing the information flows of their web applications, while imposing as little as 2.8% runtime overhead on function invocation. Valve thus represents a viable path forward to the integration of information flow enforcement in serverless platforms.

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 16-57534. The views expressed are those of the authors only.