Go Serverless: Securing Cloud via Serverless Design Patterns

Sanghyun Hong*
University of Maryland

Abhinav Srivastava
Frame.io

William Shambrook
Frame.io

Tudor Dumitraș
University of Maryland

Abstract

Due to the shared responsibility model of clouds, tenants have to manage the security of their workloads and data. Developing security solutions using VMs or containers creates further problems as these resources also need to be secured. In this paper, we advocate for taking a serverless approach by proposing six serverless design patterns to build security services in the cloud. For each design pattern, we describe the key advantages and present applications and services utilizing the pattern. Using the proposed patterns as building blocks, we introduce a threat-intelligence platform that collects logs from various sources, alerts malicious activities, and takes actions against such behaviors. We also discuss the limitations of serverless design and how future implementations can overcome those limitations.

1 Introduction

Cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and IBM Cloud offer a shared responsibility model when it comes to security in the cloud. In this model, the cloud provider manages the security of the physical infrastructure and hypervisors; the tenants are responsible for the security of resources, workloads, and data. Given that security is one of the leading concerns in the broader adoption of cloud computing, cloud providers offer many security services to help tenants meet their security and compliance requirements. To this end, providers offer services such as vulnerability scanning, configuration change detection, and stateful firewalls to protect tenant resources and critical workloads.

While these cloud security services help tenants to some extent, tenants still have to go through the tedious process of developing security automation, misuse detection, intrusion detection, virus scanning, etc., before they execute their code securely in the cloud. Developing this security infrastructure using VMs or containers exacerbate the problem as now these resources require similar protection themselves. Serverless architecture helps solve this last mile problem.

Serverless architecture, aka Function-as-a-Service (FaaS), simplifies the code deployment and eliminates the need for system administration, allowing developers to focus on the core security logic without creating additional overhead by instantiating resources such as VMs or containers in the monitoring infrastructure. In this programming model, developers execute their logic in the form of functions and submit to the cloud provider to run the task in a shared runtime environment; cloud providers manage the scalability needs of the function by running multiple functions in parallel. Due to the simplicity and ease of deployment, many serverless architecture has been proposed [27, 29, 33, 35, 39, 41]. While the past works focus on the design, implementation, and security of serverless architecture itself [4], in this work, we focus on how serverless architecture can help cloud developers and security operation personnel to develop a variety of security services in a scalable manner by adhering to simple design patterns.

Based on our extensive experience in developing serverless applications, we have identified six design patterns for serverless architectures: periodic invocation, event-driven, data transformation, data streaming, state machine, and bundling multiple patterns (Sec. 2). These design patterns allow developers to create many security services such as virus scanning, compliance checking, and incident response. For each pattern, we discuss: 1) how the design pattern is composed, 2) what are the advantages compared to non-serverless designs, and 3) provide examples of a few services that the pattern can help build. Using the fundamental patterns as building blocks, we also propose a threat-intelligence platform for the cloud that analyzes many data sources, generates alerts on suspicious activities and automatically takes responsive actions to recover from attacks (Sec. 3).

*This work has been performed during the internship at Frame.io.

At the end, we discuss the limitations of current serverless architecture and how future research can overcome those limitations (Sec. 4). By sharing these design patterns with the wider research and development community, we hope to encourage others to develop more security applications using serverless architecture and explore similar serverless design patterns in other areas.

1.1 AWS Lambda

To focus our attention on one specific serverless architecture, we only consider AWS Lambda [16] in the rest of this paper. AWS Lambda supports various runtime environments, e.g., Python, Node.js, Java, Go, or C#, and is tightly integrated with the rest of the AWS ecosystem. Lambda functions have many advantages compared to the conventional server-oriented architectures such as:

- **Deployment**: The speed at which developers can go from code to executing it is much faster than using servers such as VMs or containers.
- **Scalability**: The Lambda allows thousands of concurrent executions of a function out of the box, without any operational overhead to the cloud developers.
- **Cost**: Unlike traditional architectures using servers, where we need to pay for the running time of an instance, we only pay for the time when the code is executed by a Lambda.
- **Integrations**: Lambda can subscribe to various event sources such as CloudWatch [7], S3 [19], API Gateway [5], SNS [10], and Kinesis [9].
- **Stateless**: Lambda functions are stateless, which provides the simplicity in implementing security systems such as data analytic services that process records individually. However, some security services, e.g., firewalls, are stateful, thus, to implement such service using Lambda will require an external database to maintain states.

1.2 AWS Lambda Security

To provide security services using serverless design patterns, Lambda functions that facilitate serverless architecture should be secure. As securing the services running in the cloud follows the shared responsibility model, cloud providers ensure the security of a shared runtime environment and provide primitives that back the Lambda functions, whereas customers are responsible for securing their functions by using those resources.

Cloud Providers. Service providers are likely to have motivations and resources to invest heavily in the security of their infrastructure. A motivated attacker, for example, can attempt to break into a host operating system (OS) or shared runtime environment to install malware that monitors Lambda functions. To thwart such attacks, providers deploy state-of-the-art intrusion detection and prevention systems [31]. Nevertheless, instead of compromising a host OS, an attacker can establish side-channels [44,45] to eavesdrop a user's sensitive data being processed by a Lambda. For such cases, cloud providers utilize hardware-based solutions, e.g., Intel SGX [26, 32], that enable safe executions of Lambda functions with the compromised host OS or runtime. In addition, providers limit the execution time of Lambda functions for few minutes to make it difficult for attackers to probe and establish such channels.

Customers. Securing the code running in a Lambda and the data coming in/out of the function is the responsibility of customers. Attackers have motivations to modify users' Lambda functions or eavesdrop the communication between a Lambda function and data storage to steal customers' sensitive information. Customers, for instance, can use access control policies such as AWS IAM [20] that provide temporary credentials for Lambda functions to communicate with other services and AWS key management service (KMS) to encrypt their secrets, making it harder for attackers to access the credentials. However, not all the attack scenarios are covered by these services, e.g., an attacker can utilize the vulnerabilities in a customer's code or perform man-in-the-middle attacks by inserting malicious contents to Lambda messages. In such cases, tenants can utilize other security products such as JSON web tokens (JWTs) to strengthen their Lambda functions. In Sec. 4, we further discuss improving the security of a Lambda.

2 A Taxonomy of Serverless Design Patterns

In this section, we introduce a taxonomy of serverless design patterns that we realize using Lambda and primitives provided by AWS. We categorize serverless design patterns into six groups: 1) periodic invocation, 2) event-driven, 3) data transformation, 4) data streaming, 5) state machine, and 6) bundled pattern. We also discuss how these patterns can be used to build various security services.

2.1 DP1: Periodic Invocation Pattern

A periodic invocation design pattern (see Figure 1) represents the kind of models that invokes Lambda functions periodically by using schedulers such as cron in Unix operating systems or CloudWatch monitoring service [7]. Each Lambda function carries out a simple task and reports the execution results to notification channels such as asynchronous message buses or emails. For instance,

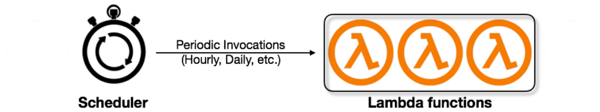

Figure 1: Periodic invocation pattern (DP1).

we can archive the data not accessed for an extended period of time into long-term backup storage, such as AWS cold storage service called Glacier, by using a Lambda function that scans the data using the LRU manner and copies them to the cold storage. The periodic invocation approach also allows cloud tenants to build applications that provide continuous compliance as required by system and organization controls (SOC2) [3] or cloud security alliance (CSA) [1]. Those applications periodically check the compliance status of resources, e.g., if any VM is subscribed to a security group that has SSH port open to all IPs (0.0.0.0/0). The compliance status is stored in a data store for later viewing and auditing purposes.

2.2 DP2: Event-Driven Pattern

An event-driven design pattern, as described in Figure 2, is where a set of Lambda functions subscribe to events from cloud resources, such as accessing files in the object store or updating a table in the database. These events trigger the execution of the subscribed Lambda function passing the necessary context. Unlike the periodic invocation pattern, the event-driven approach can reduce the latency between the occurrence of events and the action taken by the invoked Lambda. For example, cloud tenants can implement an anti-virus application for the S3 object store that performs the virus scanning of uploaded files to S3 [42]. Once the file-upload event occurs, S3 invokes corresponding Lambda function that removes the malicious files based on its virus scanning results. In another use case, we can implement layer-7 intrusion detection systems by attaching Lambda functions to application load balancers [11]. Given that all web connections are terminated at the load balancer, a Lambda function, ingesting load balancer logs, has complete visibility of incoming requests. We can perform two types of detections at the Lambda: 1) signature-based detection that identifies attacks such as SQL injection or cross-site scripting (XSS), and 2) anomaly-based detections that isolate malicious web requests, deviating from the normal behavior. The event-based design pattern has several advantages: 1) it minimizes the cost by invoking the Lambda function only when an event occurs, and 2) Lambda functions scale automatically based on the number of events, providing a scalable design.

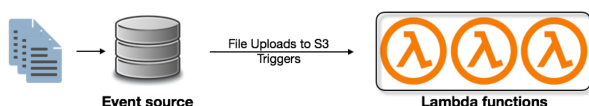

Figure 2: Event-driven pattern (DP2).

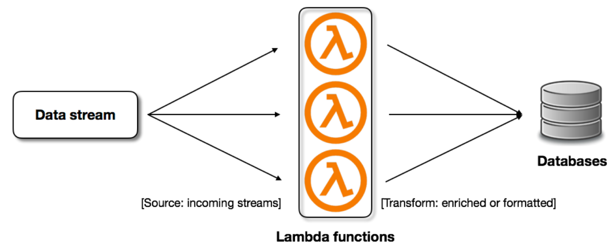

Figure 3: Data transformation pattern (DP3).

2.3 DP3: Data Transformation Pattern

The ETL (extract-transform-load) data processing pipelines usually require three steps: 1) **extract** data from a data source, 2) **transform** data by using frameworks such as Apache Spark [25] or Flink [23], and 3) **load** the transformed data into a database. Realizing these ETL pipelines into the cloud environment presents several problems: it requires persistent execution of VMs or containers to process incoming data, and the data transformation code is not easy to update once deployed because it requires pausing the input data streams.

Using Lambda-based architecture, as shown in Figure 3, solves these issues. The data processing tasks can be implemented as Lambda functions, and when the data is available, those Lambda functions perform transformations and store the results. Lambda functions are not required to run persistently when there is no data, and it is quite easy to update a processing pipeline by only modifying target Lambda functions and redeploying it on the fly. Data processing pipelines that utilize Lambda functions provide various advantages in security because many security applications require data enrichment or change in the data format for further analysis. For example: suppose that we want to append the geolocations of IP addresses in incoming network packets using MaxMind GeoIP Database [36]. In non-serverless data processing pipelines, we first store the original packets in a database, extract only IP fields from the stored data, and update the data in the database with the geolocation information. However, with the Lambda-based transformation patterns, we can enrich the incoming data on the fly as it is available using Lambda, which does not demand any other database or data processing framework. In another example, Lambda functions can transform the data into the Apache Parquet [24] on the fly, which is a columnar structure [22], reducing the cost and query processing time of Amazon Athena [6].

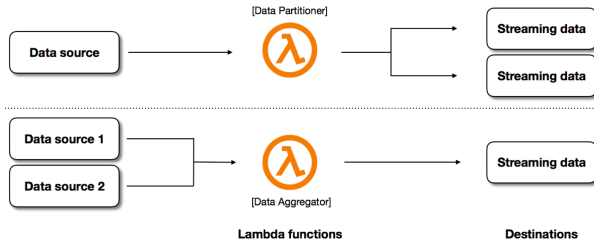

Figure 4: Data streaming pattern (DP4).

2.4 DP4: Data Streaming Pattern

In the data streaming design pattern (see Figure 4), a Lambda function sits in the path of data stream and functions either as an aggregator or data partitioner. For example, a lambda can separate an incoming data-stream into multiple small streams (partition) or merge several incoming streams into one large data-stream (aggregation). This partitioning functionality also helps Lambda act as a load balancer, which divides the data into many streams of the same size and transfer to multiple streaming services based on the size of incoming streams.

A data streaming pattern is useful to filter events from the data stream. For instance, we want to be notified immediately if an internal cloud API is invoked from malicious IP addresses. Unlike the traditional designs that require the deployment of another data processing pipeline, we deploy a Lambda function in the API processing path, filter the request using IP addresses, and generate alerts indicating whether there is suspicious traffic or not. Many ChatOps solutions such as Slack [40] provide seamless integrations with the programming languages used by Lambda, which makes it easier to receive security notifications.

2.5 DP5: State Machine Pattern

The state machine pattern in Figure 5 enables building a complex, stateful procedure by coordinating a collection of discrete Lambda functions using a tool such as AWS Step Functions. This pattern provides several advantages: 1) customers are not required to store states to cloud storage since Step Functions manage them seamlessly and 2) do not need to scale entire pattern as tasks

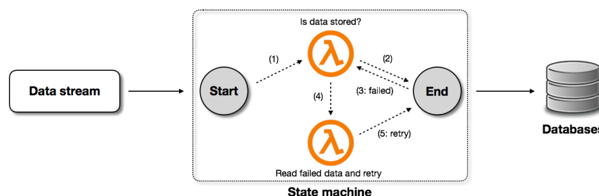

Figure 5: State machine pattern (DP5). Note that we extended data transformation pattern with a state machine.

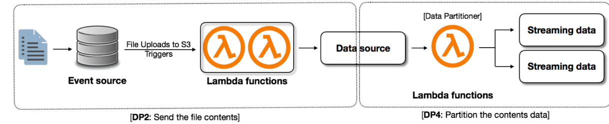

Figure 6: Bundled pattern (DP6).

defined for each state can be scaled-up/down individually.

As an example, in Figure 5, we make the data streaming pattern more stable by using a state machine. With a single Lambda function, a failure in delivering a batch of streaming data means the data will be lost, which could be the serious problem in security monitoring services. On the other hand, this state machine pattern can deal with this problem; the failure state from the data-processing Lambda function invokes another Lambda which tries the same request again until it succeeds. In addition, the state machine offers a try/catch mechanism so that we can invoke different functions depending on the failure reason.

2.6 DP6: Bundled Pattern

The bundled pattern combines two or more of the previously described patterns together by easily passing events sequentially between them. Conceptually, this is very much like UNIX pipelines, where each function is small, precise and does one thing, but the great power comes from chaining these together. As proposed in Figure 6, a collection of functions forms a data processing pipeline, which combines the data-driven pattern (DP2) with the data streaming pattern (DP4).

2.7 Cost and Scalability Analysis

Due to the time-bound execution (see Sec. 4), long duration tasks are difficult to run by a Lambda function. In addition, for tasks that require short latency (5-10ms), the non-serverless architectures consisting of VMs or containers are better suited because the time to start a Lambda function (50-100ms) is significantly higher than that of running a VM or container [29]. Thus, when we compare the cost and scalability of serverless and non-serverless architecture, we consider tasks with the running time between 100ms and 5min and expected latency between 50-100ms.

Cost Analysis. The task is to process load balancer logs, streaming 200 requests per minute, where each request has 5000 log entries, i.e., 1 million log entries per minute. If we use a Lambda function with 256Mb memory (sufficient to process this workload in memory) running 1 seconds for each request, it costs $37.74/month based on Lambda pricing [14]. However, once we serve requests

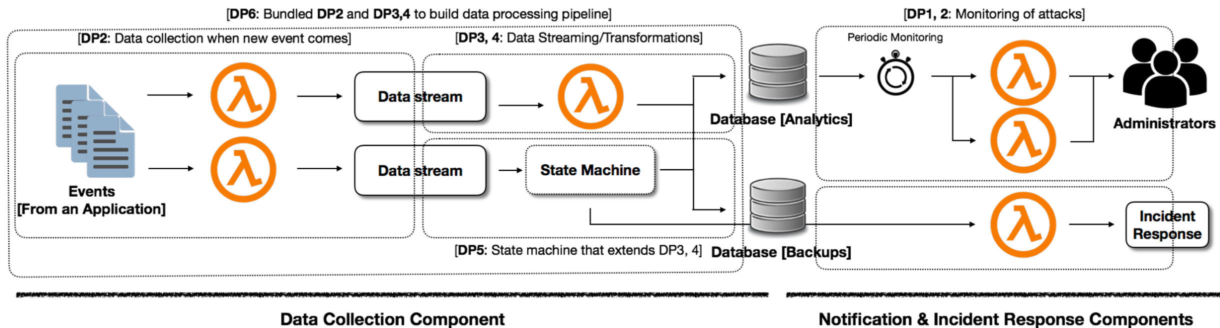

Figure 7: Proposed threat intelligence platform: the boxes with dashed-lines indicate what design pattern is used.

using 2 EC2 instances of the m5.large type equipped with 2CPUs and 8Gb memory [13], we need to pay $138.24/month[1] In this case, the serverless implementation is a lot cheaper than the non-serverless architecture. In addition, if the load is unpredictable and irregular, the cost of running instances can be a lot more than Lambda functions since the minimum number of containers or VMs are always required to run. However, if the Lambda function has to run one minute for each request, since the number of log entries per request is increased from 5,000 to 300,000 with the same configuration, the cost is $2,162.16/month, whereas the instances cost $138.24 (the same), which makes Lambda very expensive for the long-running tasks. The additional cost that we need to consider is the operational cost, which is not reflected in the above numbers. Lambda is a managed service that requires minimum administration and efforts to scale-up/down whereas containers and VMs are required to be configured with these scaling options. This saves time and effort of customers who operate large-scale infrastructures for security services.

Scalability Analysis. Both non-serverless and serverless architecture can be scaled-up/down well with the regular and predictable loads. However, with unpredictable loads, serverless patterns have better scalability as they can release the resources when there is no running task.

3 Serverless Threat-Intelligence Platform

To illustrate how these individual design patterns, as described in Sec. 2, can be combined to build security services, we propose a threat-intelligence platform that analyzes various data sources in the cloud, notifies suspicious events and takes responsive actions against them. Our proposed architecture, as illustrated in Figure 7, consists of three components: 1) data collection, 2) alert notifications, and 3) incident response workflows. In describing the architecture of the threat-intelligence platform, we emphasize on the relevant serverless design patterns, as described in Sec. 2, pertaining to the functionality of components.

[1]Note that the instance type is the cheapest EC2 General Purpose compute instance, and we run at least two instances in case of failures.

Data Collection Component: The AWS cloud has many data sources, usually one for each cloud resource, that is exposed to tenants for performance, debugging, and security purposes. For example: application firewall [17], load balancer [11], S3 access logs [19], DNS [21], and API calls [12] are some of the log types that tenants can utilize. The data collection module utilizes multiple serverless patterns to collect the data from varying sources and stores them in a centralized location. As soon as the new data becomes available at those data sources, the event-driven pattern (DP2) attached to sources reads the new data and streams the incoming data into transformation/streaming sub-modules. The data transformation pattern (DP3) enriches the input data with additional information, such as by identifying the geolocation corresponding to an IP address in the data, and then the data streaming pipeline (DP4) streams the enriched data into both S3 and Elasticsearch cluster. When a failure occurs in sending the data to either S3 or Elasticsearch, the state machine pattern (DP5) catches the failure and resends the failed entries by invoking another Lambda.

Notification Component: The notification component incorporates both the periodic invocation (DP1) and event-driven (DP2) patterns to alert system operators and developers of suspicious activities against their cloud resources. For example: when the data collection component stores the API call data (collected by AWS CloudTrail) in S3, the event-driven Lambda function verifies if the API calls are invoked from outside the USA (which may signify in certain cases that the API keys used to invoke APIs are misused). On the other hand, there are attacks, where we need evidence to be collected for a certain period of time before we can detect them. For instance, detecting login brute force attacks requires analyzing a number of failed login attempts over time. How-

ever, the Lambda functions attached to the incoming data streams can only see the evidence present in the current stream; it has no access to historical data. In this case, the periodic invocation design (DP1) solves the problem by periodically invoking a Lambda function, querying the data from our Elasticsearch cluster and extracting failed login attempts over time.

Incident-Response Component: This component responds to attacks identified by the notification component. For example, it helps to automate the forensic analysis of a compromised VM (or a container). When the notification component notifies on a compromised VM, the best incident response action is to quarantine the infected VM by blocking all incoming and outgoing traffic from it and to analyze the memory of the VM. The memory analysis includes the installation of the kernel module such as LiME [2] into the compromised VM to extract the memory dump and perform forensic actions by using the known tools such as Volatility [43]. This complete end-to-end incident response workflow requires a number of well-orchestrated Lambda functions, which can be achieved using the state machine pattern (DP5).

Cost and Scalability Analysis: To implement each component as a non-serverless architecture, few containers or VMs will be essentially running all the time because of the high spin-up time of a container or VM. Lambda functions can avoid such resource consumption by not being invoked when there is no data or incident. In terms of scalability, as each design pattern can be scaled-up/down individually, which is managed by cloud providers, the threat-intelligence platform in Figure 7 provides more flexibility when we want to add or remove data, notifications, or incident-response modules.

4 Discussion

Despite the versatile capabilities of the Lambda function, there are limitations that restrict our design choices. In this section, we describe the limits and discuss how to avoid them by using workarounds, followed by potential approaches that solve such limitations, systematically.

4.1 Resource Constraints

Time-Bound Execution. Lambda functions have a maximum execution time limit [15, 30, 37], which prohibits using Lambda for tasks that have an unknown duration as once the limit is reached, AWS will terminate the execution without waiting for the completion and any state will be lost. This limitation can be avoided by splitting the original task across multiple executions, which is not possible for all workloads. Thus, the proper solution is to either increase the execution time limit or to automatically pass state between executions so that the task can continue in another execution with the previous state.

Lack of Computing Power. From prior experience with applications such as video encoding, the amount of computing power available to a Lambda function is insufficient for CPU intensive workloads. CPU resources are also not directly configurable; instead, they are proportionally allocated depending on the amount of memory configured to a Lambda function. Thus, such workloads currently have to be executed inside VMs or containers. An ultimate solution for this problem is to make computing resources configurable or to support a Lambda that uses powerful computing resources such as GPUs.

Disk Space. AWS limits you to 512MB of disk space under the "/tmp" directory exposed to a Lambda function which again restricts using Lambda for workloads like video encoding. There is also no documentation about whether the Lambda disk is encrypted. We would like to see the ability to increase this as a simple configuration option or a way to mount disks like AWS EBS or AWS EFS which would also allow encryption.

4.2 Limited Functionalities

Event Tracing. There is a lack of tooling to trace an event through an intricate serverless system to help with troubleshooting issues and to understand where the bottlenecks exist in the system. This also impacts handling suspicious activity analysis as it is hard to identify where the event has originated, and what other components that the event may have affected. Tools like Zipkin [46] and AWS X-Ray [18] provide the required database and visualizations, but lack integrations with cloud services such as AWS Cloudwatch and AWS SNS to fully trace an event propagation. Since this limits the visibility of serverless systems, either the integrations with existing cloud services should be supported or cloud providers need to support such tools.

Security. AWS Lambda functions are offered as a managed service. However, there are no security services integrated with the Lambda functions, currently. To develop secure function code, developers resort to security tools, such as bandit [38], integrated into continuous integration/continuous deployment (CI/CD) pipeline that statically analyzes the function code to discover unsafe functions and security bugs. To prevent introducing security bugs from third-party libraries [34], many Lambda functions only include AWS provided packages and libraries. To allow seamless security to Lambda functions, AWS should integrate AWS Inspector [8] with the Lambda function that provides vulnerability scanning.

5 Conclusions

To ease the development of security services in the cloud, this paper describes six serverless design patterns that can be used to build serverless applications and services. In each design pattern, we highlighted the key advantages and presented several serverless applications. We also conceptually demonstrated that a large-scale security system for cloud can be composed of proposed design patterns by introducing a threat-intelligence system. In addition, we described the limits of Lambda functions and provided ways to overcome them. We envision that the proposed serverless design patterns will revolutionize security systems in the cloud and become dominant by being a standard of serverless application development.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Michael Swift, for their feedback.

References

[1] Cloud Controls Matrix - Cloud Security Alliance : Cloud Security Alliance. https://cloudsecurityalliance.org/group/cloud-controls-matrix/#_overview, 2018. [Accessed 03-10-2018].

[2] LiME - Linux Memory Extractor. https://github.com/504ensicsLabs/LiME, 2018. [Accessed 03-12-2018].

[3] System and Organization Controls: SOC Suite of Services. https://www.aicpa.org/interestareas/frc/assuranceadvisoryservices/sorhome.html, 2018. [Accessed 03-10-2018].

[4] ALPERNAS, K., FLANAGAN, C., FOULADI, S., RYZHYK, L., SAGIV, M., SCHMITZ, T., AND WINSTEIN, K. Secure serverless computing using dynamic information flow control. *arXiv preprint arXiv:1802.08984* (2018).

[5] AMAZON WEB SERVICES. Amazon API Gateway. https://aws.amazon.com/api-gateway/, 2018. [Accessed 03-14-2018].

[6] AMAZON WEB SERVICES. Amazon Athena - Serverless Interactive Query Service - AWS. https://aws.amazon.com/athena/, 2018. [Accessed 03-11-2018].

[7] AMAZON WEB SERVICES. Amazon CloudWatch - Cloud & Network Monitoring Services. https://aws.amazon.com/cloudwatch/, 2018. [Accessed 03-11-2018].

[8] AMAZON WEB SERVICES. Amazon Inspector. https://aws.amazon.com/inspector, 2018. [Accessed 03-15-2018].

[9] AMAZON WEB SERVICES. Amazon Kinesis. https://aws.amazon.com/kinesis/, 2018. [Accessed 03-12-2018].

[10] AMAZON WEB SERVICES. Amazon Simple Notification Services (SNS) — Event Notifications for Distributed Applications and Microservices — AWS. https://aws.amazon.com/sns/, 2018. [Accessed 03-12-2018].

[11] AMAZON WEB SERVICES. AWS — Elastic Load Balancing - Cloud Network Load Balancer. https://aws.amazon.com/elasticloadbalancing/, 2018. [Accessed 03-12-2018].

[12] AMAZON WEB SERVICES. AWS CloudTrail. https://aws.amazon.com/cloudtrail/, 2018. [Accessed 03-12-2018].

[13] AMAZON WEB SERVICES. AWS EC2 Pricing - AWS. https://aws.amazon.com/ec2/pricing, 2018. [Accessed 05-10-2018].

[14] AMAZON WEB SERVICES. AWS Lambda - Pricing. https://aws.amazon.com/lambda/pricing, 2018. [Accessed 05-10-2018].

[15] AMAZON WEB SERVICES. AWS Lambda Limits - AWS Lambda - AWS Documentation. https://docs.aws.amazon.com/lambda/latest/dg/limits.html, 2018. [Accessed 03-11-2018].

[16] AMAZON WEB SERVICES. AWS Lambda Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/, 2018. [Accessed 02-21-2018].

[17] AMAZON WEB SERVICES. AWS WAF - Web Application Firewall. https://aws.amazon.com/waf/, 2018. [Accessed 03-12-2018].

[18] AMAZON WEB SERVICES. AWS X-Ray Distributed Tracing System. https://aws.amazon.com/xray, 2018. [Accessed 03-14-2018].

[19] AMAZON WEB SERVICES. Cloud Object Storage — Store & Retrive Data Anywhere — Amazon Simple Storage Service. https://aws.amazon.com/s3/, 2018. [Accessed 03-12-2018].

[20] AMAZON WEB SERVICES. Identity and Access Management (IAM) - Amazon Web Services (AWS). https://aws.amazon.com/iam/, 2018. [Accessed 03-15-2018].

[21] AMAZON WEB SERVICES. Managed Cloud DNS - Domain Name System — AWS Route 53 — AWSl. https://aws.amazon.com/route53/, 2018. [Accessed 03-12-2018].

[22] AMAZON WEB SERVICES. Using Amazon Redshift Spectrum, Amazon Athena, and AWS Glue with Node.js in Production — AWS Big Data Blog. https://aws.amazon.com/blogs/big-data/using-amazon-redshift-spectrum-amazon-athena-and-aws-glue-with-node-js-in-production/, 2018. [Accessed 03-11-2018].

[23] APACHE SOFTWARE FOUNDATION. Apache Flink: Scalable Stream and Batch Data Processing. https://flink.apache.org/, 2018. [Accessed 03-11-2018].

[24] APACHE SOFTWARE FOUNDATION. Apache Parquet. https://parquet.apache.org/, 2018. [Accessed 03-11-2018].

[25] APACHE SOFTWARE FOUNDATION. Apache Spark - Lighting-Fast Cluster Computing. https://spark.apache.org/, 2018. [Accessed 03-11-2018].

[26] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M., ET AL. Scone: Secure linux containers with intel sgx. In *OSDI* (2016), vol. 16, pp. 689–703.

[27] BALDINI, I., CASTRO, P., CHANG, K., CHENG, P., FINK, S., ISHAKIAN, V., MITCHELL, N., MUTHUSAMY, V., RABBAH, R., SLOMINSKI, A., ET AL. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[28] ELASTICSEARCH. Open Source Search & Analytics Elasticsearch — Elastic. https://www.elastic.co/, 2018. [Accessed 03-12-2018].

[29] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. *Elastic 60* (2016), 80.

[30] IBM CLOUD. System Details and Limits - IBM Cloud Docs. https://console.bluemix.net/docs/openwhisk/openwhisk_reference.html, 2018. [Accessed 03-12-2018].

[31] JAIN, B., BAIG, M. B., ZHANG, D., PORTER, D. E., AND SION, R. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 605–620.

[32] JAIN, P., DESAI, S. J., SHIH, M.-W., KIM, T., KIM, S. M., LEE, J.-H., CHOI, C., SHIN, Y., KANG, B. B., AND HAN, D. Opensgx: An open platform for sgx research. In *NDSS* (2016).

[33] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 445–451.

[34] KRUG AND JONES. Hacking Serverless Runtimes - Profiling Lambda, Azure, and More. https://www.blackhat.com/docs/us-17/wednesday/us-17-Krug-Hacking-Severless-Runtimes.pdf, 2018. [Accessed 03-15-2018].

[35] MALAWSKI, M., GAJEK, A., ZIMA, A., BALIS, B., AND FIGIELA, K. Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems* (2017).

[36] MAXMIND, INC. MaxMind GeoIP2. https://www.maxmind.com/en/geoip2-services-and-databases, 2018. [Accessed 03-11-2018].

[37] MICROSOFT AZURE. Azure Functions scale and hosting — Microsoft Docs. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#consumption-plan, 2018. [Accessed 04-27-2018].

[38] OPENSTACK. openstack/bandit: Python AST-based static analyzer from OpenStack Security Group. https://github.com/openstack/bandit, 2018. [Accessed 03-15-2018].

[39] PÉREZ, A., MOLTÓ, G., CABALLER, M., AND CALATRAVA, A. Serverless computing for container-based architectures. *Future Generation Computer Systems 83* (2018), 50–59.

[40] SLACK. Slack: Where work happens. https://slack.com/, 2018. [Accessed 03-11-2018].

[41] SPILLNER, J. Snafu: Function-as-a-service (faas) runtime design and implementation. *arXiv preprint arXiv:1703.07562* (2017).

[42] UPSIDE™. S3 Anti-virus Scanning with Lambda and ClamAV. https://engineering.upside.com/s3-antivirus-scanning-with-lambda-and-clamav-7d33f9c5092e, 2018. [Accessed 03-11-2018].

[43] VOLATILITY FOUNDATION. Volatility - An Advanced Memory Forensics Framework. https://github.com/volatilityfoundation/volatility, 2018. [Accessed 03-12-2018].

[44] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 305–316.

[45] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 990–1003.

[46] ZIPKIN. OpenZipkin - A Distributed Tracing System. https://zipkin.io, 2018. [Accessed 03-12-2018].