Least-Privilege Identity-Based Policies for Lambda Functions in Amazon Web Services (AWS)

by

Puneet Gill

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public

Abstract

We address least-privilege in a particular context of public cloud computing: identity-based policies for callback functions, called Lambda functions, in serverless applications of the Amazon Web Services (AWS) cloud provider. We argue that this is an important context in which to consider the fundamental security design principle of least-privilege, which states that every thread of execution should possess only those privileges it needs. We observe that poor documentation from AWS makes the task of devising least-privilege policies difficult for developers of such applications. We then describe our experimental approach to discovering least-privilege for a method call, and our observations, some of which are alarming, from running it against 171 methods across five different AWS services. We discuss also our assessment of two repositories, and two full-fledged serverless applications, all of which are publicly available, for least-privilege, and find that the vast majority of policies are over-privileged. We conclude with a few recommendations for developers of Lambda functions in AWS. Our work suggests that much work is needed, both from developers and providers, in securing cloud applications from the standpoint of least-privilege.

Acknowledgements

I would like to express my sincere gratitude to my advisors, Professor Mahesh Tripunitara and Professor Werner Dietl, for guiding me through my research. I would also like to thank readers of my thesis, Professor Arie Gurfinkel and Professor Patrick Lam, for taking their time to review my work and provide feedback.

I am thankful to my parents and family for all their love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Infrastructure- and Platform-as-a-Service public cloud computing, with which a *customer* is able to deploy and run applications on the hardware and software computing resources of a *cloud provider*, has become a dominant paradigm over the past decade [26]. Amazon Web Services (AWS) [14] has been the largest provider of such services over the past few years [58]. Along with several others, a particular cloud service that AWS provides is *serverless computing* [25]; Lambda [19] is AWS's serverless offering. A customer packages their[1] business logic as callback functions, each of which is associated with an event that triggers its execution. A callback function in AWS Lambda is called a Lambda function. An intent is that the customer no longer needs to explicitly specify the hardware and software resources that should be provisioned; the cloud provider does such provisioning "under the covers." AWS Lambda is a dominant provider of serverless computing, and is also the largest amongst all cloud services provided by AWS [57].

Security, and in particular, the protection of resources from unauthorized principals, is an essential requirement in any system in which such resources are stored and manipulated. Lambda functions in AWS are no exception. AWS perceives the security of an application in AWS Lambda as a shared responsibility, between itself and the customer, i.e., the owner of the application [6]: "Security . . . is a shared responsibility between AWS and the customer. . . For AWS Lambda, AWS manages the underlying infrastructure and foundation services, the operating system, and the application platform. [The customer is] responsible for the security of [the customer's] code, the storage and accessibility of sensitive data, and identity and access management. . . to the Lambda service and within [the customer's Lambda] function."

[1] We use the gender-neutral pronoun "they" and its variants as a singular.

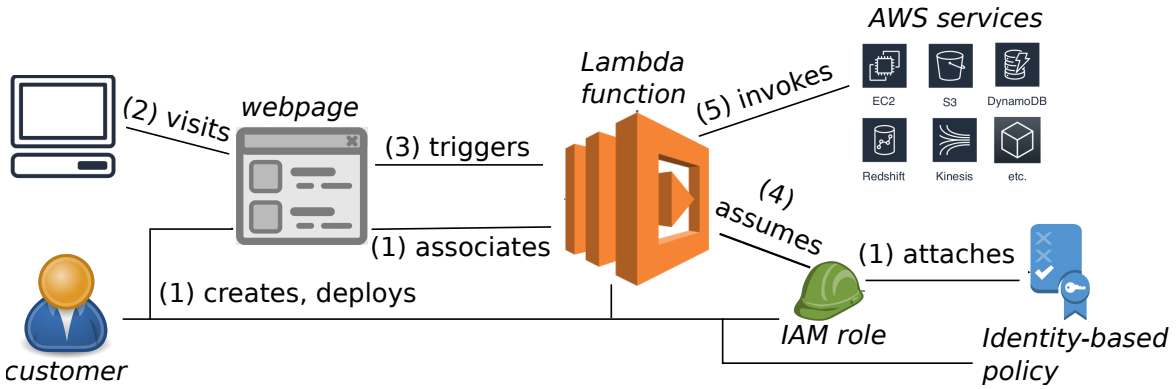

Figure 1.1: Overview of the components and actions that are relevant to our work. Numbers indicate chronology. A customer of a cloud provider (1) creates and deploys a Lambda function, an Identity and Access Management (IAM) role and an identity-based policy. They also associate an event, a visit to a web page in this example, with invocation of the Lambda function, and attach the identity-based policy to the role. When a user (2) visits the web page, this (3) triggers the invocation of the Lambda function. The Lambda function executes by first (4) assuming the role, and then perhaps (5) invoking other AWS services.

As part of its responsibility towards security, AWS provides constructs that the customer can use to specify their intended security policy. One such construct is an *identity-based policy* [18]. An identity-based policy is attached to an identity, a role in the example in Figure 1.1, and "controls what actions the identity can perform, on which resources, and under what conditions" [18]. The manner in which an identity is bound to the entity is beyond the scope of an identity-based policy. Figure 1.2 is an example of such a policy (see Section 2.3 of Chapter 2 for more details). The policy comprises two statements, one of which denies privileges, and the other grants privileges, as indicated by "Deny" and "Allow," respectively, for the **Effect** field. The statement at the top denies the `DeleteItem` and `DeleteTable` privileges in the `DynamoDB` AWS service [7] to a resource that is listed under **Resource**. The second grants `GetObject` in the `S3` AWS service [13] to a set of resources; "*" is a wildcard.

How it all works In Figure 1.1 we illustrate how the components we discuss above work together. In the figure, in the first chronological step, Step (1), the customer creates and deploys the Lambda function, an Identity and Access Management (IAM) role which the Lambda function assumes when it runs, and an identity-based policy which the customer attaches to the role. Thus the "identity" here in "identity-based policy" is the role. Also,

```
{ "Statement": [{
        "Effect": "Deny",
        "Action": [ "dynamodb:DeleteItem",
                    "dynamodb:DeleteTable" ],
        "Resource": "arn:aws:dynamodb:region:accountId:table/myTable" },
    {   "Effect": "Allow",
        "Action": "s3:GetObject",
        "Resource": "arn:aws:s3:::myBucket/*" }] }
```

Figure 1.2: An AWS identity-based policy. It comprises a set of statements, in this example, of size two. One statement denies privileges as indicated by *Deny* for "Effect" and the other grants privileges as indicated by *Allow*. The statement at the top denies *dynamodb:DeleteItem* and *dynamodb:DeleteTable* to the *myTable* resource in the AWS DynamoDB service [7]. The other statement grants *s3:GetObject* to all objects within *myBucket* in the AWS S3 service [13]. The mnemonic "*" is a wildcard.

the customer associates the Lambda function with the events that trigger it—in the figure, it is a visit to a webpage that is also created and deployed by the customer. Step (1) is the deployment step. The subsequent steps (2)–(5) in the figure show what happens at run time. When a user visits the webpage in Step (2), they cause the event to occur that, in Step (3), triggers the Lambda function to run. The Lambda function, in Step (4), assumes the IAM role that was specified for it at deployment, and, in Step (5), invokes other AWS services as specified in its code.

Least-privilege The intent of the identity-based policy in Figure 1.1 is to ensure that when the Lambda function runs, it has sufficiently many privileges, and no more. That is, we seek identity-based policies that are neither *under-* nor *over-privileged*. The motivation for preventing over-privilege is security: once an attacker compromises, for example, a user's account, over-privilege allows them to effect more damage than they otherwise would have been able to [46]. The motivation for precluding under-privilege is that the reason we create and deploy a Lambda function, in the first place, is for it to perform certain tasks. The Lambda function cannot perform these unless it possesses sufficient privileges.

If we seek to preclude one of over- or under-privilege only, and not both, it is easy to devise an identity-based policy. If we require preclusion of over-privilege only, we would simply deny access to everything. If we require preclusion of under-privilege only, we would simply allow access to everything. The balance between over- and under-privilege is the problem of devising policies that adhere to *least-privilege*. As articulated by Saltzer and Schroeder [51], least-privilege is the principle that "every program and every user of the

system should operate using the least set of privileges necessary to complete the job." The reason is to "limit the damage that can result from an accident or error." Least-privilege is customarily adopted as fundamental to good design if security is of concern.

Our motivation Apart from it being acknowledged as an important design principle, we are motivated to investigate least-privilege in identity-based policies in AWS for two reasons. One is the wide adoption of AWS as we mention above in this chapter, and the fact that identity-based policies are the most widely used mechanism for managing access in AWS applications — identity-based policies are listed first amongst all policy-types "in order of frequency" [17]. Our other reason is two recent security incidents that have occurred related to identity-based policies in AWS.

One is the Capital One Hack [38, 31] which led to the breach of about 100 million pieces of private information. As CloudSploit [38] explains, "At the root of the hack lies a common refrain: the misconfiguration of cloud infrastructure resources allowed an unauthorized user to elevate her privileges and compromise sensitive documents." And, "While it may be easy to blame... developers for the loss of data, the truth is that IAM role misconfigurations are likely present in nearly every single AWS account. Very few developers take the time to carefully list each permission required; oftentimes a wildcard is used in many more places than it should be, resulting in unintended access." The other incident relates to an identity-based policy that is managed by AWS, rather than the customer (we discuss the two policy types in Section 2.3). The problem is that within the policy, a particular action called `iam:PutRolePolicy` is granted, which in turn allows someone who is able to assume the role to which the policy is attached to grant administrative access to another role [56]. This incident suggests that it is not only customers of AWS who have difficulty devising correct least-privilege identity-based policies, but AWS themselves.

Our focus and the breadth of its applicability Our focus, as Figure 1.1 suggests, is least-privilege in identity-based policies for Lambda functions in AWS. We have already motivated why we consider identity-based policies in AWS above. We now first motivate our focus on Lambda functions, and then discuss, given these foci, how broadly our work applies.

We have two reasons that we focus on Lambda functions. One is the wide use of AWS Lambda as we discuss earlier in this chapter — it is the most widely used service within AWS, which, in turn, is the most widely used cloud provider. Our other reason is more technical, and pertains to our interpretation of least-privilege. Specifically, at what granularity should least-privilege be enforced? Should it be that, for example, at any moment in time, a thread of execution possesses a smallest set of privileges only? This somewhat strong interpretation may be appropriate, for example, for a process in a computer operat-

ing system. However, in cloud computing, and AWS, in particular, a more coarse-grained interpretation seems appropriate; our focus on Lambda functions concretizes this. We make the notion of least-privilege for a Lambda function precise in Chapter 3, and overview it as follows.

The specification that a particular identity-based policy be bound to a Lambda function via a role customarily happens at the time the customer deploys the Lambda function (see Figure 1.1). Furthermore, while a Lambda function runs, it is not customary for its privileges to change. Therefore, least-privilege for a Lambda function in AWS is characterized at deployment-time, based on the method calls it makes to AWS services.

We now discuss how broadly our work applies given our focus on (i) AWS, (ii) identity-based policies, and (iii) Lambda functions. Our focus (i), on a single cloud provider, AWS, is, admittedly, a limitation of our work. Sufficiently detailed technical work at the level we carry out requires customization to each cloud-provider, as each has their own approach to a need. We observe that Google Cloud appears to have similar constructs as AWS in this regard [41], and therefore conjecture that our work can be repeated there. Our focus (ii) on identity-based policies can also be seen as a limitation. However, AWS has other constructs such as resource-based policies [17] to which our observations certainly apply. The reason is that the difference between identity- and resource-based policies is only that the latter is bound to a resource and not an identity, but otherwise, they are almost identical. Furthermore, as we discuss above, AWS touts identity-based policies as the most frequently used mechanism.

Our focus (iii) on Lambda functions, as we mention above, helps us concretize the granularity at which we address least-privilege. In any context within AWS that this, or a coarser, granularity applies, our work is relevant. This is not Lambda functions only, but also those applications that use Elastic Compute Cloud (EC2) [8], for example, for which this granularity is appropriate. Furthermore, identity-based policies are not used for Lambda functions only; they can be, and are, used for security by AWS applications that use services other than AWS Lambda.

Our intent and work Underlying the work that we have done is an intent which is expressed by the following question: if I am a developer of AWS Lambda functions and I plan to use identity-based policies to secure them, in what manner should I configure those policies so they are least-privilege?

Towards this intent, we have done two complementary pieces of work. In Chapter 3, we discuss our work that, across the Java API of five different services in AWS, identifies a least-privilege identity-based policy for several methods. We then make several observations in that chapter, including the poorness of documentation that necessitates a

somewhat painstaking experimental process. Then, in Chapter 4, we leverage our work from Chapter 3 to assess Lambda functions in two repositories of sample applications [22, 55], and two serverless applications [15, 45] that have been written to run on AWS. We ask: is an identity-based, or another similar, policy configuration made available with the code, and if yes, is it least-privilege? The two repositories we consider host what we can call snippets of serverless code, with the intent that someone who builds a realistic serverless application can adopt those snippets as part of their code. The two applications are standalone, and intended to be realistic. Thus, these are two qualitatively different assessments. Based on our observations from Chapters 3 and 4, and towards the intent we articulate above, in Chapter 6, we make recommendations for developers of Lambda functions.

Terminology We use the terms "permission" and "privilege" interchangeably. Our choice in a particular case is based on customary usage, for example, "least-privilege." We sometimes refer to a mnemonic that may appear as an action in an identity-based policy (see Figure 1.2) as a permission. For example, we may say, "if the role possesses the permission `s3:GetObject`, then..." In this case, we mean that the effect of all identity-based policies which are attached to the role is to grant that permission to the resources in question.

Pseudonymous repository We have created a pseudonymous repository [35], to which we refer as appropriate. The repository contains working code to illustrate three of the several issues we discuss in this paper. A README file in the repository gives detailed instructions on how to install and run the code within, configure identity-based policies we provide, and what outputs to expect.

Layout Apart from Chapters 3, 4 and 6 that we discuss above, the remainder of our paper is organized as follows. In Chapter 2, we summarize the background on access control, related work and the AWS security architecture. We discuss the foundations of access control in Section 2.1. In Section 2.2, we discuss related work. In Section 2.3, we overview identity-based policies and Lambda functions, and make broad observations about the overall design of such policies and their syntax. In Chapter 5, we discuss the steps we have taken in automating the least-privilege Identity-based policy generation process. We conclude with Chapter 7, where we discuss also future work.

Chapter 2

Background

In this chapter, we discuss the foundations of access control in Section 2.1, followed by a discussion of related work and lastly, a description of the how identity-based policies fit in the security architecture of AWS.

2.1 Access Control

In this section, we discuss the foundations of access control and security policies and mechanisms. Content of this section is a summary of Chapters 1, 2 from the work of Bishop [34].

Computer Security of a system is often described as maintaining *confidentiality, integrity* and *availability* of the system. The confidentiality aspect of security deals with the concealment of information or resources of a system. This means that either the unauthorized parties are unable to use the information or resources, or they are oblivious to the existence of them. In addition to confidentiality, we are interested in answering whether the information or a resource of a system has been improperly altered. This corresponds to the integrity of a system. Integrity pertains to the the trustworthiness of information or resources. Lastly, the availability of a system is its ability to use the information or resources desired by authorized users.

Definition 2.1 (Access Control)**.** Access Control is a process by which the use of system resources is regulated according to a *security policy* and is permitted only by authorized entities (users, programs, processes, or other systems). On a high level, a security policy is a statement comprising of what is, and what is not, allowed [50].

	`alice`	`bob`	`file 1`	`file 2`
`alice`	*own*		*read, write*	
`bob`		*own*	*read*	*read, write*

Table 2.1: An access control matrix with two subjects `alice, bob` and two other objects `file 1, file 2`

Access control mechanisms support confidentiality, it may conceal information to unauthorized parties via use of cryptography or hide the existence of information or resources altogether. Moreover, following the principle of least privilege in access control can be viewed as a prevention mechanism for ensuring the integrity of a system. Hence, the goal for a security administrator, configuring access control, is to limit the access to resources by authorized entities.

In a computer system, a *state* is the collection of the current values of all memory locations, secondary storage, and all registers and other components. A subset of these states that deal with protection is called the *protection state* of the system. One of the tools that describes a protection state of the system at a given point in time is the *access control matrix*. It encodes the protection state using the following entities:

- A set of all protected entities called *objects*, O
- A set of active objects called *subjects*, S
- A set of *rights*, R

The access control matrix A captures the relationship between the subjects and objects using the rights that each subject has on all objects. Each entry in A is a right $A[s, o] \subseteq R$ that a subject $s \in S$ possess over an object $o \in O$. Table 2.1 shows an example of an access control matrix. The rows of the matrix are indexed by subjects and the columns by objects. In the example, the entries in the matrix specify that `alice` can both *read* and *write* to `file 1`, but has no right over `bob` and `file 2`. Similarly, `bob` can *read* `file 1`, can both *read* and *write* to `file 2` and has no rights over `alice`. Both `alice` and `bob` has *own* right to themselves. The *own* right allows the owner of the object to add or delete rights for themselves or grant rights to others.

In the context of AWS, a Lambda application that makes a request for an action on an AWS resource can be thought of as a subject in the access control matrix, where the AWS

resources are the passive objects. The IAM permissions that a user/role associated with a Lambda application may possess over an AWS resource are the rights available for that particular type of resource. Although, it is difficult to use an access control matrix in practice due to the space requirements, it is often used for theoretical analysis of security problems.

2.2 Related Work

Our work deals with least-privilege in the context of AWS Lambda applications. The security design principle of least-privilege, to our knowledge, was first articulated by Saltzer and Schroeder [51]. That work discusses a number of other principles as well, which are considered important to the design of secure systems. Apart from least-privilege, our work is at the intersection of several topics that have been addressed in research in information security: authorization and access control languages and systems, checking for security properties in real-world systems, and security of cloud computing. It is well beyond our scope to discuss each of these comprehensively. Rather, in the remainder of this chapter, we focus on work that we see as most closely related to ours.

A piece of work that is related closely to ours is that of Felt et al. [40]. Our objectives are similar to their work. That work exercises the Android API to determine the permissions needed for each API method, and based on that determination, assesses several apps as to whether they adhere to least-privilege. Thus, we are strongly similar from the standpoint of objectives; however, the different settings, Android vs. AWS, result in different technical details and findings. We observe that since that work, and perhaps as a consequence, Android now provides a `RequiresPermission` annotation [29] so a piece of code in an app can clearly call out the permissions it requires. One of our recommendations in this work is exactly such a mechanism (see Chapter 6).

The other pieces of work that are related to ours are ones that assess aspects of security in AWS. The work of Balduzzi et al. [33] identifies security risks from the use of virtual server images from public catalogs of AWS. It identifies that some of these risks can allow unauthorized, remote access to an AWS application. That work considers virtual server images and AWS EC2, and not identity-based policies and Lambda functions, as ours does.

Then, there is work from AWS itself. There is work on security best practices for AWS [2], the work of Cook [39] that discusses the use of formal methods for security within AWS, and the work of Backes et al. [32] that discuss an automated approach to reasoning about AWS access policies. These endeavours underlie AWS services that provide security

assessments: AWS Config [16], Amazon Macie [12], AWS Trusted Advisor [23], Amazon GuardDuty [10] and AWS IAM Access Analyzer [17]. None of these calls out least-privilege as a property of interest, nor do we see mention of properties that seem to lie at the same level of abstraction.

The discussions in the first of these, security best practices [2], is at a higher level than least-privilege. A representative example of a recommendation there is, "Use bucket-level or object-level permissions alongside IAM policies to protect resources from unauthorized access and to prevent information disclosure, data integrity compromise, or deletion." The work of Backes et al. [32] mentions a number of properties; for example: "... [checks for] AWS Lambda Functions granting unrestricted access, ... S3 buckets granting unrestricted read access, ... S3 buckets granting unrestricted write access, deny putObject requests that do not have server side encryption, and deny actions that do not allow https traffic." These are at a level of abstraction lower than least-privilege, in that it is possible that one or more of these properties are needed for least-privilege in an application, but as to whether and how, is a missing link. Furthermore, given that the mapping of an API method to privileges does not seem to be information that is available readily, we conjecture that that contribution of ours alone, which is in Chapter 3, may be useful to enhance these endeavours and tools. It would be interesting to investigate also whether the expressive power of the approaches and tools that are the current focus of such techniques within AWS suffices for a property such as least-privilege.

Of all the AWS services we list above, the AWS IAM Access Analyser [17] seems related most closely to our work. It is based on the work of Backes et al. [32], and it helps identify what it calls unintended access to resources. It does so by first identifying what it calls a zone of trust, and then checking for accesses by entities from outside this zone. While the AWS IAM Access Analyzer can tell us whether some of our policies, particularly resource-based policies (see Section 2.3) are least-privilege, it is unclear how one should go from a warning issued by the analyzer to a least-privilege policy. Thus, we see the Access Analyzer as alerting us to the possibility that our policy is not least-privilege, but not solving the problem of devising a least-privilege policy.

2.3 Identity-based Policies, Lambda functions, and overall design

In this section, we first describe identity-based policies in AWS and their place in the broader security architecture of AWS. Then, we overview Lambda functions. Finally, we make some observations about the design of identity-based policies in AWS.

Identity-based policy Our description of identity-based policies is from the user guide [17] and the work of Backes et al. [32]. An identity-based policy in AWS is a set of statements, each of which is a tuple. There are three mandatory components in each statement, and three optional components. The mandatory components are: (i) `Effect`: either "allow" or "deny," (ii) `Action`: a verb, and (iii) `Resource`: an identifier. The optional components are a `Version` of the policy language, an `Sid`, an identifier for the statement, and a `Condition`, which qualifies when this statement applies. Our work is meaningful with the mandatory components only, and carries over easily when optional components are included.

An identity-based policy is bound to a Lambda function via either a user or a role. The user or role is the identity the Lambda function assumes when it runs. Then, the semantics of a policy is the following. A policy allows an action a on a resource r by the Lambda function if and only if there is a statement in the policy that matches a and r with the `Effect` "allow," and no statement in the policy matches a and r with the `Effect` "deny." That is, we have the customary default-deny, and deny-overrides semantics. We use the word "matches" and not "is" because the resource r may be specified as, for example, a wildcard, as the example in Figure 1.2 indicates for the resource '. . . `myBucket/*`'.

If more than one identity-based policy is bound to a Lambda function, then the union of the statements in those policies applies. Also, apart from identity-based policies, AWS provides a number of other constructs for authorization. These are the following [17]. (i) Resource-based policies: these are identical to identity-based policies except that they are bound to a resource rather than a principal. Correspondingly, the `Resource` mandatory component in a statement is supplanted by one or more `Principal` components, which specify to which principals the statement applies. (ii) Permission boundaries, Organizational service control policies, and Session policies: with these, one can delimit the permissions that are granted. And, (iii) Access Control Lists (ACLs): these are identical to resource-based policies, but with a different syntax.

We focus on identity-based policies only, from the standpoint of technical details. But our work certainly can apply almost directly to the other mechanisms we mention above. AWS

specifies a "policy evaluation logic" [17] that expresses the manner in which these policies co-exist from the standpoint of enforcement.

Identity-based policies are further classified into inline policies and managed policies. An inline policy is embedded with a user or role and cannot be attached to any other user or role. A managed policy may be attached to more than one user or role. Our work can be seen as applying to both kinds of policies. Managed policies are further classified into two: AWS managed policies and customer managed policies. AWS provides the former for use by customers, but only AWS is allowed to specify and change those policies. Also, AWS may change them at any time. A customer managed policy is specified and maintained by the customer. A practice that AWS suggests is that a customer could first copy-and-paste the contents of one or more AWS managed policies, and then edit it as the customer deems appropriate, to yield a customer managed policy. One of the recent security incidents we discuss in Chapter 1 pertains to over-privilege in an AWS managed policy [56].

Lambda functions We now discuss Lambda functions, broadly, to an extent that is necessary and meaningful for our work. We narrow our characterization of a Lambda function in Definition 3.1 in Chapter 3 in a manner that is appropriate for the technical aspects of our work.

A Lambda function is a function written in a programming language such as Python or Java. It is to be written as a callback for when an event occurs. A Lambda function is deployed to be run by AWS Lambda. To deploy a function, one needs to create a deployment package and an execution role. The deployment package contains the code for the function and any dependencies such as libraries. As for the execution role, "An AWS Lambda function's execution role grants it permission to access AWS services and resources. You provide this role when you [deploy] a function, and Lambda assumes the role when your function is invoked." [19]

Our focus is exactly the permissions that are granted to such an execution role of a Lambda function. Identity-based policies are attached to such a role, which specify the permissions that are granted. The policies may be either AWS or customer managed, or inline policies (see above under "Identity-based policy").

Figure 2.1 shows an example of a Lambda function, `ListBooks` for the AWS Bookstore application that we analyze in Chapter 4. As shown in the figure, the Lambda is triggered by an API Gateway endpoint and queries a DynamoDb table. The `ListBooks` lambda function takes on the execution role of `newbookstore-DynamoDbLambda` which grants the permissions to perform operations on DynamoDb, S3 and CloudWatch Logs as shown in Figure 2.2. There are three identity-based policies attached to the `newbookstore-DynamoDbLambda` execution role. The first one is an inline policy with the name "`PutRidePolicy`". The

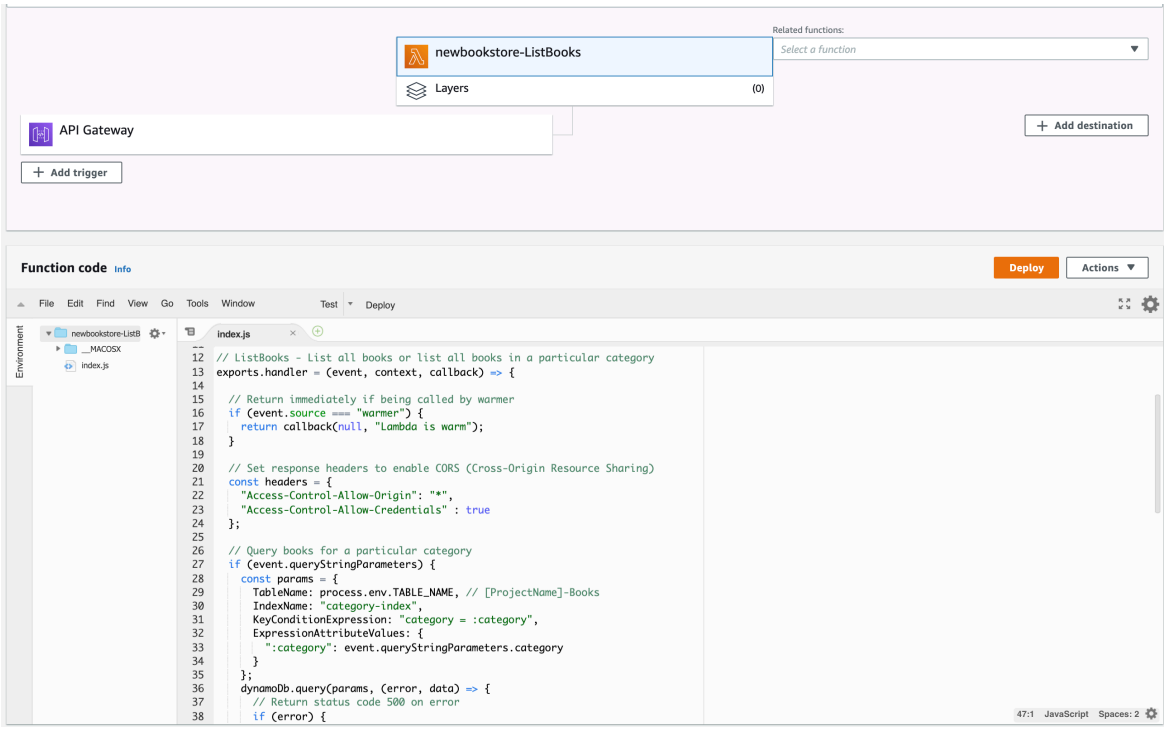

Figure 2.1: An example of a Lambda function from the AWS Bookstore application.

other two are AWS managed. We analyze these identity-based policies in detail in Chapter 4.

On the design We now make two observations on the design of identity-based policies for Lambda functions in AWS that impacts least-privilege. One regards actions that appear in such policies. A design choice AWS has made is for each action to be associated with an AWS service such as S3 or EC2. For example, in the policy in Figure 1.2, the actions begin with "`dynamodb:`" and "`s3:`". Then, within each service, there are a number of such actions. This design choice has led to a proliferation in the number of possible actions that can appear in an identity-based policy. For example, in only the five services we have examined, there are almost 600 different actions (see Table 3.1 in section 3.3).

We contrast this design choice to, for example, file permissions in UNIX. In that system, there are three actions only: read, write, and execute. Those actions may be specified on different filesystem objects, and the semantics of an action is a function also of the kind of object to which it applies. For example, a read on a data file means something different from a read on a directory. The design in AWS appears to be more like Android. At the

```
{   "roleName": "newbookstore-DynamoDbLambda",
    "trustedEntities": "lambda.amazonaws.com",
    "policies": [
    { "document": {
        "Statement": [{
            "Effect": "Allow",
            "Action": [ "dynamodb:PutItem",
                        "dynamodb:Query",
                        "dynamodb:UpdateTable",
                        "dynamodb:UpdateItem",
                        "dynamodb:BatchWriteItem",
                        "dynamodb:GetItem",
                        "dynamodb:Scan",
                        "dynamodb:DeleteItem" ],
            "Resource": [ "arn:aws:dynamodb:region:accountId:table/Books",
                          "arn:aws:dynamodb:region:accountId:table/Orders",
                          "arn:aws:dynamodb:region:accountId:table/Cart",
                          "arn:aws:dynamodb:region:accountId:table/Books/*" ]}]},
        "name": "PutRidePolicy",
        "type": "inline" },
    { "document": {
        "Statement": [{  "Effect": "Allow",
                        "Action": [ "s3:Get*",
                                    "s3:List*" ],
                        "Resource": "*" }]},
    "name": "AmazonS3ReadOnlyAccess",
    "type": "managed",
    "arn": "arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess" },
    { "document": {
        "Statement": [{ "Effect": "Allow",
                        "Action": [ "logs:CreateLogGroup",
                                    "logs:CreateLogStream",
                                    "logs:PutLogEvents" ],
                        "Resource": "*" }]},
    "name": "AWSLambdaBasicExecutionRole",
    "type": "managed",
    "arn": "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole" }]}
```

Figure 2.2: Policy attached to the `newbookstore-DynamoDbLambda` execution role

time of writing, Android has 170 system default permissions [30]. These permissions are custom to resources, e.g., ACCESS_NETWORK_STATE, and in that respect, are similar to AWS actions which are custom to a resource that is part of an AWS service, e.g., `s3:getObject`.

Apart from our observations above on the design of actions, a second observation we make regards the binding of permissions to a thread of execution, i.e., a Lambda function. As in Android, the binding of permissions to a Lambda function is somewhat static in AWS. One can think of it as a deployment-time occurrence, though an identity-based policy can be changed at later times. However, the permissions to which a Lambda function is authorized customarily do not change as it runs. We contrast this to processes in the UNIX operating system, which can dynamically change their privileges as they run, for example, via a call to the setuid suite [37]. This in turn impacts our characterization of least-privilege (see Chapter 3).

These design choices by AWS suggest that issues with least-privilege in AWS will be similar to issues Android has suffered from the standpoint of least-privilege [40]. There certainly are similarities as we discuss in Section 2.2 on related work. There are also some differences, for example, in the ecosystem of an application in AWS vs. Android, that can impact least-privilege. For example, an Android app must specify the permissions it needs in a manifest file that is part of its "app bundle," which is a distribution format for an app. In AWS by contrast, there is an additional level of indirection: to deploy a Lambda function, one needs to specify a deployment package and execution role only. Any identity-based policies that are attached to the role are specified separately.

Chapter 3

Least-Privilege Identity-Based Policies

To be able to identify whether a Lambda function is least-privileged, we need to first identify the privileges it needs. This pertains to the resources the Lambda function accesses, which in turn, it does via AWS services. For example, if the Lambda function writes to a storage object, it would do so via a call to a service such as S3 [13]. For our work then, we identify whether a Lambda function is least-privileged as follows. We look at the code for the Lambda function, and consider all the methods it invokes in AWS services and their arguments $m_1(a_1), \ldots, m_n(a_n)$. We then perceive these methods and the sequence of arguments of each as a set of pairs, and ask what a least-privilege set of permissions is for that set of calls to AWS services.

Consider, for example, the snippet of JavaScript code in Figure 3.1. We omit some details, shown as "...", that do not pertain to us. We would associate that snippet of code with the set which comprises four pairs, each of which is a method in DynamoDB and the sequence of its arguments. We can think of the object on which a method is invoked as an argument. If a method takes no arguments, then we can think of its sequence of arguments as being represented by a special symbol, say ϵ. Of course, the arguments may be specified at run time only, and can change each time a Lambda function runs. As we restrict ourselves to configuring identity-based policies at deployment-time, with no changes to them at run time, we associate a set of argument-sequences, A, with each method m that is invoked within a Lambda function. The set A comprises every possible argument-sequence that may be supplied to m when it is called in the Lambda function. This suggests the following definition for a Lambda function in the context of our work. We adopt this definition for the remainder of our work.

```
switch (...)
{ case 'DELETE':
    dynamo.deleteItem(JSON.parse(event.body), done); break;
  case 'GET':
    dynamo.scan(
      {TableName: event.queryStringParameters.TableName},
      done); break;
  case 'POST':
    dynamo.putItem(JSON.parse(event.body), done); break;
  case 'PUT':
    dynamo.updateItem(JSON.parse(event.body), done); break;
  default: ...  }
```

Figure 3.1: Example JavaScript code in a Lambda function from the AWS repository [5].

Definition 3.1 (Lambda function)**.** A Lambda function is a set of pairs, each of the form $\langle m, A\rangle$, where m is a method in an AWS service, and A is the set of all possible sequences of arguments that may be supplied in a call to m when this Lambda function runs.

Least-privilege set for a Lambda function Given the above characterization of a Lambda function, we develop a characterization of a least-privilege set of permissions for it as follows. We define, separately, a sufficient and a necessary set. Then, we define that least-privilege means both sufficient and necessary. For sufficiency, our approach is to first characterize it for a pair of method and argument-sequence, $\langle m, a\rangle$, then for a pair of method and set of argument-sequences, $\langle m, A\rangle$, and then for a Lambda function, i.e., a set of pairs, each of the form $\langle m, A\rangle$.

Definition 3.2 (Sufficient set of permissions)**.** A set of permissions P is said to be sufficient for a pair $\langle m, a\rangle$, where m is a method in an AWS service and a a sequence of arguments for m, if: a call $m(a)$ succeeds given all possible permissions implies that that call $m(a)$ would have succeeded given the permissions P only. A set of permissions P is said to be sufficient for a pair $\langle m, A\rangle$, where m is a method in an AWS service and A a set of sequences of arguments for m if, for every $a \in A$, P is sufficient for $\langle m, a\rangle$. A set of permissions P is sufficient for a Lambda function L, i.e., a set of pairs $\langle m, A\rangle$, if P is sufficient for every $\langle m, A\rangle \in L$.

The reason we have an implication with the premise "a call succeeds given all possible permissions" in the above definition is in recognition of the fact that an instance of a call

may fail for any number of reasons, for example, if we seek to read an object, that is specified as an argument, that does not exist. Thus, with regards to a set of permissions P for which we attempt to assess sufficiency, we care only about calls that succeed given all permissions — the call fails on account of lack of privilege only.

Definition 3.3 (Necessary set of permissions)**.** A set of permissions Q is necessary for a Lambda function L if: there is some superset $P \supseteq Q$ such that (i) P is sufficient for L, and, (ii) for every $q \in Q$, $P \setminus \{q\}$ is not sufficient for L.

We define a necessary set for a Lambda function L only. Of course L may comprise one pair $\langle m, A \rangle$ only, and A may have one member only. Thus, we do not need, as we did for sufficiency, to start with a characterization for a method-argument pair, $\langle m, a \rangle$.

Definition 3.4 (Least-privilege set of permissions)**.** A least-privilege set of permissions for a Lambda function L is a set that is necessary and sufficient for L.

We make the assumption that there are no negative permissions in AWS. This means that for a permission p_1 there is no set $\{p_2, \ldots, p_k\}$ that nullifies p_1. This is different from the *allow* and *deny* effects on a permission.

A nuance is that we do not know whether a least-privilege set in AWS is unique. Definition 3.3 for necessity above accounts for this. In the definition, we do not demand that every $q \in Q$ is a member of every sufficient set. Rather, we demand only that there exists a sufficient set in which every $q \in Q$ is indispensable. From our Definition 3.4, a set of permissions $P = \{p_1, p_2, p_3\}$ are both necessary and sufficient if any strict subset of the permissions in P will be insufficient. From the standpoint of non-uniqueness, we are unsure about whether it is possible that there exists another least-privilege set of permissions $P' = \{p_4\}$. We should also clarify that the word *Least* in least-privilege does not mean the minimum sized set but instead reflects the idea of minimal privilege in computer security. Moreover, as we discuss in Section 3.3, most API methods only require a single permission. Therefore, the difference between the minimum set and a minimal set of permissions diminishes.

Constraints and choices We now identify choices we have made in our identification of least-privilege sets, and constraints we have had to reconcile. We have implemented our approach against the API that is part of the Java SDK for AWS. Also, we have chosen five services whose APIs we have examined: S3 [13], DynamoDB [7], Kinesis [11], Elastic Transcoder [9] and EC2 [8]. We have examined both version 1 [1] and version 2 [3] of the client interface or class for each of those five services in the Java SDK for AWS. The reason we have looked at both versions is that version 1 continues to be supported by AWS, and

there are features supported by version 1 that are not supported by version 2 [44]. The specific interfaces and classes we have examined are:

version 1 [1]: `com.amazonaws.services.`

- `s3.AmazonS3Client`
- `dynamodbv2.AmazonDynamoDBClient`
- `kinesis.AmazonKinesisClient`
- `elastictranscoder.AmazonElasticTranscoderClient`
- `ec2.AmazonEC2Client`

version 2 [3]: `software.amazon.awssdk.services.`

- `s3.S3Client`
- `dynamodb.DynamoDbClient`
- `kinesis.KinesisClient`
- `elastictranscoder.ElasticTranscoderClient`
- `ec2.Ec2Client`

In the case of version 2, we instantiate a class that implements an interface using a corresponding builder; for example, for S3, this is `s3.S3ClientBuilder`, which comes with a static `builder`() method. Within each of the above client classes, we have excluded constructors, static methods, deprecated methods and any method that does not exercise a capability of the service; for example, the `shutdown`() method in Elastic Transcoder that shuts the client instance down.

We have attempted to cover seemingly different kinds of services with our five choices. Table 3.1 shows the services, and the number of API methods in each service for which we have identified a least-privilege set. As the table indicates, we have examined a total of 171 API methods across the five services. A constraint we have had to circumvent is that AWS code is closed-source and we have no access to it. Therefore, we have had to determine a least-privilege set in a black box manner.

3.1 The poorness of documentation

If documentation is adequate, we would not have to identify least-privilege sets experimentally. Indeed, good documentation should reasonably be expected in this regard. Unfortunately, we have found AWS's documentation for necessary and sufficient permissions for an API method to be poor. In this context, it is meaningful to compare documentation for versions 1 and 2 of the API: between the two versions, the idiom "pick your poison" is appropriate.

With version 1, the documentation on Java methods is largely non-existent. When there is documentation on permissions, it regards ACLs, and not identity-based policies. There is no clear mapping from one to the other. For example, `Permission.Write` may be specified in an ACL for `putObject`() in S3. It is not easy to determine what the corresponding action for an identity-based policy is. The Developer Guide for version 1 [21], under "Security → Identity and Access Management" redirects a reader to IAM. The IAM User Guide [17] has an overview, and presents several examples of policies. It is unclear whether those policies are least-privilege, and even how comprehensive they are. The examples address specific scenarios, for example, "EC2: Start or Stop Instances Based on Tags" and "Amazon S3: Allows Amazon Cognito Users to Access Objects in Their Bucket." Thus, it can be reasonably said that clear, comprehensive guidance in version 1, on permissions for one's Lambda function, is absent.

With version 2, the Developer Guide [20] is identical to version 1 under "Security → Identity and Access Management." The documentation for the Java API does mention permissions for the S3 client [4]; but for the other four services we have examined, the situation is the same as with version 1. However, the documentation for the S3 client [4] in this regard is almost always incomplete, and in some cases, simply incorrect. Consider, for example, the `getObject`() call. The documentation says, "You need the `s3:GetObject` permission for this operation." Unfortunately, this unqualified assertion is wrong. The assertion should really be qualified based on a particular field in the `getObjectRequest` argument. If a particular field in that argument, `versionId`, is set, then it turns out that a least-privilege set for `getObject`() is {`s3:GetObjectVersion`}. That is, in this case, `s3:GetObject` is not needed. If, on the other hand, the `versionId` field is not set, then indeed, `s3:GetObject` is necessary.

We point out also incorrect documentation immediately above where it says that `s3:GetObject` is needed. The line from the documentation is, "Assuming you have permission to read object tags (permission for the `s3:GetObjectVersionTagging` action), the response also returns the x-amz-tagging-count header…" Unfortunately this is also not correct. The

Assuming you have permission to read object tags (permission for the `s3:GetObjectVersionTagging` action), the response also returns the `x-amz-tagging-count` header that provides the count of number of tags associated with the object. You can use GetObjectTagging to retrieve the tag set associated with an object.

Permissions

You need the `s3:GetObject` permission for this operation. For more information, see Specifying Permissions in a Policy. If the object you request does not exist, the error Amazon S3 returns depends on whether you also have the `s3:ListBucket` permission.

- If you have the `s3:ListBucket` permission on the bucket, Amazon S3 will return an HTTP status code 404 ("no such key") error.
- If you don't have the `s3:ListBucket` permission, Amazon S3 will return an HTTP status code 403 ("access denied") error.

Versioning

By default, the GET operation returns the current version of an object. To return a different version, use the `versionId` subresource.

If the current version of the object is a delete marker, Amazon S3 behaves as if the object was deleted and includes `x-amz-delete-marker: true` in the response.

Figure 3.2: Documentation excerpt for the `s3:GetObject` method from the `S3Client` class [4]

permission `s3:GetObjectTagging` is needed when the `versionId` field in the argument is not set; `s3:GetObjectVersionTagging` is needed when it is set. We shows the excerpts from the documentation for `s3:GetObject` in Figure 3.2.

It is interesting to compare the documentation for `getObject`(), and similar methods such as `getObjectAsBytes`() which suffer from the same issues that we point out above, with the documentation for `deleteObject`(), in version 2 of the `S3Client` API [4]. The documentation for `deleteObject`() includes the following regarding permissions. "If you want to block users or accounts from removing or deleting objects from your bucket, you must deny them the `s3:DeleteObject`, `s3:DeleteObjectVersion`, and `s3:PutLifeCycleConfiguration` actions." The excerpt is shown in Figure 3.3. This passage must be parsed somewhat carefully—it is a negation of a conjunction. It turns out that the passage is indeed correct—if one wants to ensure that an object cannot be deleted, all of those permissions must be denied. Thus, the documentation for `deleteObject`() appears to observe, correctly, that `s3:DeleteObjectVersion` may cause an invocation of `deleteObject`() to succeed. The documentation for `getObject`(), on the other hand, does not observe that `s3:GetObjectVersion` may enable a successful invocation of `getObject`().

There is a risk also, as we mention above, that example policies that have been published by AWS are not least-privilege. We discuss an example in our observations on EC2 in the next section: the policy in the user guide for VM import/export [27] is over-privileged. From the standpoint of incompleteness, we observe that the documentation in version 2 of S3's API for `getObject`(), and related methods such as `getObjectAsBytes`(), says only

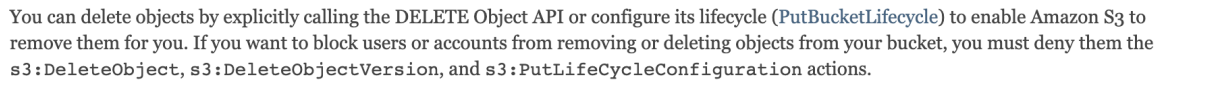

Figure 3.3: Documentation excerpt for the `s3:DeleteObject` method from the `S3Client` class [4]

that `s3:GetObject` is "needed," i.e., necessary. It does not say what is sufficient. Thus, in version 2 we have, in most cases for API methods, completely missing information on permissions. In cases that information is present, it is often incorrect and incomplete.

Our pseudonymous repository [35] includes code for the two issues above: S3's `getObjectAsBytes`() and EC2's VM export.

3.2 Our approach to identifying permissions for an API call

In this section, we discuss the manner in which we have identified a least-privilege set of permissions for a single call, i.e., method-arguments pair, for a service in AWS. In Chapter 4, we then leverage Definition 3.4 to identify a least-privilege set for a Lambda function and apply it to our study of serverless applications.

Our approach works as follows. For each method m in an AWS service, we have devised a method, called `SetupEnvAndArgs`(), which takes as argument the method m, and sets up an environment in AWS, and arranges a sequence of arguments a such that a call $m(a)$ is guaranteed to succeed if the caller is given all possible permissions. This is not easily automated because a particular environment is typically appropriate for one service only, and within that service, only for a small set of API methods. Furthermore, a least-privilege set may change based on the arguments to the method call. Thus, we have a collection of such environments in our set up. We abstract all of them into the single call `SetupEnvAndArgs`() for clarity of exposition. It takes as input the method m and returns a sequence of arguments for m.

Also, we have devised a method, call it `Failure`(), which takes two arguments: an AWS method m, argument-sequence a for it, and a set of permissions P, and returns true if and only if the call $m(a)$ fails given that the caller possesses the permissions P. Otherwise, i.e., if the call succeeds, it returns false. Thus, a property that expresses the manner in which the above two methods are composed is the following. If we invoke `SetupEnvAndArgs`(m)

and it returns a, and then we invoke $\texttt{Failure}(m, a, P)$, the latter call returns false if and only if the call $m(a)$ is successful given the environment set up by the former call, and the caller has the permissions P.

We now discuss two algorithms we have devised and run against the AWS API to identify a least-privilege set. Both algorithms take two inputs: a method m to an AWS service, and a set of permissions S. Each algorithm premises that the input S is sufficient (see Definition 3.2). It is easy to identify such a sufficient set; the set of all possible permissions in AWS is sufficient. Each algorithm then identifies a necessary set that is a subset of that sufficient set S. The result is a necessary and sufficient, i.e., least-privilege, set. Our first algorithm, `LPLinear` below, performs a linear-search through the members of S and identifies whether each is necessary.

```
LP_Linear(m, S)
    a ← SetupEnvAndArgs(m)
    if Failure(m, a, S) then return error
    Result ← ∅
    foreach p ∈ S do
        a ← SetupEnvAndArgs(m)
        if Failure(m, a, S \ {p}) then
            Result ← Result ∪ {p}
    return Result
```

`LP_Linear` is correct: it is guaranteed to terminate, and if the input set of permissions S is sufficient for $\langle m, a \rangle$, then the `Result` that `LP_Linear` returns is least-privilege for $\langle m, a \rangle$. The time-efficiency of `LP_Linear` is a function of two things: (i) the size of S, and, (ii) how long $\texttt{SetupEnvAndArgs}(m)$ and $\texttt{Failure}(m, a, S)$ take to run. Assuming the duration (ii) is not a function of S, there is nothing we can do to mitigate that time. We can mitigate (i) in two ways. One is that we can start with as small a set S as possible. This involves making a good guess for a sufficient set that is small; for example, much smaller than the set of all permissions. The other way is encoded in the following improved algorithm,

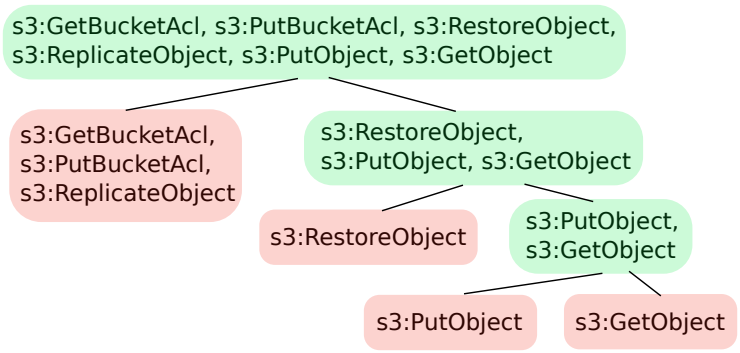

Figure 3.4: An example run of `LP_Binary` on `copyObject()` in S3. A green box is a sufficient set of permissions. A red box is a set that is not sufficient. The set with only the two permissions, `s3:GetObject` and `s3:PutObject` is least-privilege.

`LP_Binary`, which attempts a binary search.

`LP_Binary`(m, S)
  $S_1 \leftarrow$ half the members of S
  $S_2 \leftarrow S \setminus S_1$
  $a \leftarrow$ `SetupEnvAndArgs`(m)
  if *Failure*(m, a, S_1) **then**
    $a \leftarrow$ `SetupEnvAndArgs`(m)
    if *Failure*(m, a, S_2) **then**
      return *LP_Linear*(m, S)
    else return *LP_Binary*(m, S_2)
  else return *LP_Binary*(m, S_1)

`LP_Binary` invokes not only `SetupEnvAndArgs` and `Failure` as subroutines, but also `LP_Linear`. The reason is that `LP_Binary` partitions S into two sets, call them S_1, S_2 and invokes `LP_Binary` recursively in the hope that at least one of them is a sufficient set. Now assume, as an example, $S = \{p_1, p_2, p_3, p_4\}$, $S_1 = \{p_1, p_2\}$, $S_2 = \{p_3, p_4\}$, and the only least-privilege set for $\langle m, a \rangle$ that S contains is $\{p_1, p_3\}$. Thus, the call `Failure`(m, a, S) returns false, but both `Failure`(m, a, S_1) and `Failure`(m, a, S_2) return true. In `LP_Binary`, we fall back to `LP_Linear` starting at S if this happens. In the worst-case, `LP_Binary` is no more time-efficient than `LP_Linear`; in practice, it improved our time-efficiency considerably. Figure 3.4 shows an example run of `LP_Binary`.

The process is painstaking and time-consuming for two reasons. The subroutine `SetupEnvAndArgs()` is difficult to fully automate. Also, the process is different for different sets of

API calls. One of the drawbacks of a manual approach for generating arguments for invoking a method is incompleteness. Here, incompleteness refers to the fact that it is infeasible to test a method with all the possible valid inputs, which can be potentially infinite, using our manual approach. We mean that the argument space for an argument of type T is all the values that T can have. For example, testing all possible `int` values in Java will require a tester to test 2^{32} possibilities. We discuss an approach to address this issue in Chapter 5.

The subroutine `Failure`() can take several seconds, or even minutes, to return, as it is made remotely to AWS. As an example, exploring all the nodes we show in Figure 3.4 took around 10 minutes using our set up. Our experience further emphasizes the need for good documentation so developers do not have to use our methodology.

As a possible future exploration, one option is to invesigate the QuickXPlain algorithm proposed by Ulrich Junker, which provides a divide-and-conquer computation strategy to find within a given universe an irreducible subset with a particular (monotone) property. Our problem of finding least-privilege set of permissions can also be seen as finding an irreducible set. In fact, `LP_Binary` utilizes the divide-and-conquer strategy [49].

3.3 Observations

Before we assess whether policies that have been published alongside AWS serverless applications are least-privilege in Chapter 4, in this section, we make observations about per-API method least-privilege sets we have identified by running the algorithms from the previous section. We first present some quantitative data and inferences we make from it. Then, for each of the five services, we make qualitative observations. For each method m, only in some cases is a least-privilege set also a function of arguments to m. In this section, when we report quantitative observations, we report per-method only. We reserve discussions related to different least-privilege sets for different arguments to a method for qualitative discussions, as we do in Section 3.1 on documentation on S3's `getObject`() method.

Table 3.1 lists, in order, (i) the five AWS services we have examined, (ii) the number of methods of each service's Java API for which we have identified a least-privilege set, and the total number of methods in that service, (iii) the number of API methods for each service whose least-privilege set size is more than one, (iv) the total number of permissions that are associated with each service, (v) the number of API methods from the total number for each service which have no permission of the same name as the method, and, (vi) the

AWS Service	API methods		Least-privilege set size> 1	Total # permissions	# API methods with no permissions of the same name	# permissions with no API method of the same name
	Examined	Total				
S3	56	176	1	92	143	74
DynamoDb	31	63	0	48	3	10
Kinesis	41	41	2	26	2	0
Elastic Transcoder	18	18	3	17	0	1
EC2	25	458	2	393	10	0

Table 3.1: Quantitative observations from our identification of least-privilege sets of permissions per API method. These numbers are for version 1 of the Java API to AWS. The numbers for version 2 differ only slightly.

number of permissions associated with each service which have no API method of the same name. The reason we report (v) and (vi) is our conjecture that if the names are the same, a developer will be able to easily guess which permission is needed for the methods their Lambda function invokes, notwithstanding poor documentation.

Towards rationalizing why we have chosen to call out the data in Table 3.1, we recall our intent from Chapter 1: we seek to make it easy for a developer to identify a least-privilege set of permissions so they can then write out a least-privilege identity-based policy for their Lambda function. From the data in Table 3.1 and qualitative observations we have made which are discussed after we discuss the quantitative data, we make the following two conjectures on design intent in AWS. One, given the rather large number of permissions—576 across the five services in Table 3.1, and the large number of API methods whose least-privilege set is exactly one permission—is that a design intent is for an API method to be associated with its own unique permission. A second conjecture, based on the large number of permission names being exactly that of an API method, is that an exact match between a permission name and an API method name is a design intent.

With these conjectured design intents in mind, we take another look at the data in the table. We argue that the fact that these design intents are not adhered to strictly is a problem; that is, the third column, "Least-privilege set size > 1" is non-zero for four of the five services, for the API methods we have examined. Also problematic is the lack of

universality on whether an API method is associated with a permission of the same name. On this aspect, we observe in Table 3.1, that S3 is the worst-offender, with 143 of the 176 methods not being associated with a permission of the same name. For the other services, most methods are associated with a permission of the same name, but not universally so, except for Elastic Transcoder.

When we say "is a problem" in the last paragraph, we mean from the standpoint of a developer who attempts to devise a least-privilege identity-based policy. If indeed both design intents we articulate above are adhered to strictly, then it would be easy for a developer to write down a least-privilege set not only for a single method, but their entire Lambda function.

A reason why we have shown both the last two columns in Table 3.1, is that it is unclear, given an API method, whether it is easier to identify a least-privilege set starting with that API method, or alternately, starting with the set of all permissions, and then attempting to identify which of those may be associated with our API method of interest. For Elastic Transcoder, it appears that it is easier to identify a least-privilege set starting with the API method; for Kinesis and EC2, starting with the set of all permissions may be easier. We now make per-service, qualitative observations, while still consulting the data in Table 3.1.

S3: As Table 3.1 shows, S3 is the "worst-offender" from the standpoint of mismatch between API method names and permission names. Notwithstanding the fact that all but one of the 56 API methods for which we identified least-privilege sets is of size one, only 16 of the 56 have a permission that is the same as the method name. An example of a method that requires a single permission only, in version 1 of the API, is `setBucketAcl`(). The only permission in its least-privilege set is `s3:PutBucketAcl`. We call this example out particularly because we notice that in version 2 of the API, the name of the method has been changed to `putBucketAcl`(), i.e., to match that of the permission. This supports our conjecture above regarding a design intent for APIs in AWS: an API method name should match that of a permission.

The only API method in S3 amongst the ones for which we identified least-privilege sets that requires more than one permission is `copyObject`(), which copies one object, call it the source object, to another, call it the destination. We observe that the documentation for `copyObject`() in version 2 includes some discussions about permissions which are entirely missing in version 1. However, these discussions for version 2 are for setting up permissions in an ACL for the destination object; they have nothing to do whatsoever with permissions to invoke `copyObject`().

DynamoDB DynamoDB performs very well from the standpoint that each API method we examined has a least-privilege set of size one only, and only three of all the API methods in DynamoDB are associated with no permission of the same name. However, given that we have 48 total permissions only, and given the number of permissions that have no API method of the same name—10 in the last column of Table 3.1, there are a few amongst the $63 - 31 = 32$ methods that require more than one permission.

Kinesis As Table 3.1 indicates, we identified least-privilege sets for the entirety of the Kinesis API. We found only two methods with least-privilege sets of size greater than one. For these two, we made an additional interesting, and perhaps alarming, observation. The two methods are `startStreamEncryption`() and `stopStreamEncryption`(). For each of these, the policy in Figure 3.5 is sufficient. We observe that that policy grants every permission whose prefix is `kinesis:` to every resource. However, when we replace this policy with the policy shown in Figure 3.6, the method call fails, thereby suggesting that the policy in Figure 3.6 is under-privileged. The two policies should be equivalent—in the policy in Figure 3.5, we have used a wildcard, and in the policy in Figure 3.6, we have enumerated all the permissions in Kinesis. This suggests that there is some hidden permission in Kinesis which is granted in the former case only. Consequently, we are unable to devise a policy that is guaranteed to be necessary.

In Figure 3.7, we show a more restrictive policy than the one in Figure 3.5, but still sufficient for the `startStreamEncryption()` and `stopStreamEncryption()` calls in Kinesis. This policy is more complex, and difficult to devise. There is still no guarantee that it is least-privilege. We include code in our pseudonymous repository [35] that illustrates this issue.

```
{ "Statement": {
        "Effect": "Allow",
        "Action": "kinesis:*",
        "Resource": "*" } }
```

Figure 3.5: A sufficient policy for `startStreamEncryption()` and `stopStreamEncryption()` APIs for AWS Kinesis.

```
{ "Statement": {
        "Effect": "Allow",
        "Action": [ "kinesis:AddTagsToStream",
                    ...
                    "kinesis:UpdateShardCount" ],
        "Resource": "*" } }
```

Figure 3.6: A policy for `startStreamEncryption()` and `stopStreamEncryption()` in Kinesis that is under-privileged, but should not be, given that the policy in Figure 3.5 is sufficient. We use "..." to indicate that we enumerate every possible permission with prefix "`kinesis:`".

```
{ "Statement": [{   "Effect": "Allow",
                    "Action": [ "kinesis:ListStreams",
                                "kinesis:EnableEnhancedMonitoring",
                                "kinesis:ListShards",
                                "kinesis:UpdateShardCount",
                                "kinesis:DescribeLimits",
                                "kinesis:ListStreamConsumers",
                                "kinesis:DisableEnhancedMonitoring" ],
                    "Resource": "*" },
                {   "Effect": "Allow",
                    "Action": "kinesis:*",
                    "Resource": [ "arn:aws:kinesis:*:*:stream/*",
                                  "arn:aws:kinesis:*:*:*/*/consumer/*:*" ]},
                {   "Effect": "Deny",
                    "Action": [ "kinesis:DeregisterStreamConsumer",
                                "kinesis:SubscribeToShard",
                                "kinesis:DecreaseStreamRetentionPeriod",
                                "kinesis:PutRecords",
```

```
                    "kinesis:DescribeStreamConsumer",
                    "kinesis:CreateStream",
                    "kinesis:GetShardIterator",
                    "kinesis:DescribeStream",
                    "kinesis:RegisterStreamConsumer",
                    "kinesis:ListTagsForStream",
                    "kinesis:PutRecord",
                    "kinesis:RemoveTagsFromStream",
                    "kinesis:DeleteStream",
                    "kinesis:DescribeStreamSummary",
                    "kinesis:SplitShard",
                    "kinesis:MergeShards",
                    "kinesis:AddTagsToStream",
                    "kinesis:IncreaseStreamRetentionPeriod",
                    "kinesis:GetRecords" ],
        "Resource": [ "arn:aws:kinesis:*:*:stream/*",
                      "arn:aws:kinesis:*:*:*/*/consumer/*:*" ]},
    {   "Effect": "Deny",
        "Action": [ "kinesis:ListStreams",
                    "kinesis:EnableEnhancedMonitoring",
                    "kinesis:ListShards",
                    "kinesis:UpdateShardCount",
                    "kinesis:DescribeLimits",
                    "kinesis:ListStreamConsumers",
                    "kinesis:DisableEnhancedMonitoring" ],
        "Resource": "*" }]}
```

Figure 3.7: A more restrictive policy for `startStreamEncryption`() and `stopStreamEncryption`() in Kinesis than Figure 3.5, that is sufficient, but not necessarily with least-privilege.

Elastic Transcoder As Table 3.1 shows, we have identified a least-privilege set for every method in Elastic Transcoder's version 1 Java API. Indeed, one of the reasons we chose that service is its relatively small sized API, which gave us the ability to exhaustively examine it. It may seem counter-intuitive that every one of the 18 methods has a least-privilege set of size one, but that we have 17 permissions only. The reason for this is that two of the methods are overloaded, and one permission is associated with a deprecated method. Thus, we have 16 unique method names, and 16 unique permission names that match exactly.

Another observation about Elastic Transcoder's API relates to the three methods that

require more than one permission. These are: `createPipeline`(), `updatePipeline`() and `createJob`(). Of these, we discuss the first two in particular. For these calls to succeed, the instance of the AWS Elastic Transcoder service that the cloud application utilizes, itself needs to be authorized in particular ways to effect actions that underlie these methods. This, in turn, can be done using an identity-based policy that is attached to whatever role is assumed by the instance of AWS Elastic Transcoder[1]. For this, `createPipeline`() and `updatePipeline`() require `iam:passRole` which "... allows the service to later assume the role and perform actions on [the application's] behalf" [17]. We omit more details here, and emphasize only that it is not necessarily straightforward to identify least-privilege policies for these methods, because, in addition to a policy for the role the invoker of the method assumes, we need to devise a least-privilege policy for the role the instance of the AWS service, in this instance Elastic Trancoder, assumes.

EC2 23 of the 25 API methods for which we identified least-privilege sets in EC2 have size one only, and each such permission has the same name as the API method. For the remaining two, however, things are more complex. The two are `importImage`() and `exportImage`(), and the situation with them is similar to the methods `createPipeline`() and `updatePipeline`() for Elastic Transcoder that we discuss above. That is, each needs to authorize the EC2 service with a number of permissions. We show the least-privilege policies for the role that the EC2 service assumes in Figures 3.9 and 3.10. In this context, we caution that the policy in the user guide [27] is over-privileged on account of use of wildcards, and inclusion of permissions that are not necessary. The policy presented in the EC2 user guide is shown in Figure 3.8. We also include the code that demonstrates this issue in our pseudonymous repository [35].

[1] AWS calls such a role a *service role*. A separate *trust policy* must be specified to authorize the instance of the AWS service to assume a service role. Separately, identity-based policies may be attached to service roles.

```
{   "Statement": [{
        "Effect": "Allow",
        "Action": "s3:ListAllMyBuckets",
        "Resource": "*" },
{       "Effect": "Allow",
        "Action": [ "s3:CreateBucket","s3:DeleteBucket",
                    "s3:DeleteObject","s3:GetBucketLocation",
                    "s3:GetObject","s3:ListBucket","s3:PutObject" ],
        "Resource": ["arn:aws:s3:::mys3bucket","arn:aws:s3:::mys3bucket/*"]},
{       "Effect": "Allow",
        "Action": [
                    "ec2:CreateImage","ec2:CreateTags",
                    "ec2:DeleteTags","ec2:DescribeImages",
                    "ec2:DescribeTags","ec2:DescribeInstances",
                    "ec2:DescribeSnapshots","ec2:ImportVolume",
                    "ec2:ExportImage","ec2:ImportImage",
                    "ec2:ImportInstance","ec2:ImportSnapshot",
                    "ec2:StartInstances","ec2:StopInstances",
                    "ec2:CreateInstanceExportTask",
                    "ec2:DescribeConversionTasks",
                    "ec2:DescribeExportTasks",
                    "ec2:DescribeExportImageTasks",
                    "ec2:DescribeInstanceAttribute",
                    "ec2:DescribeInstanceStatus",
                    "ec2:CancelConversionTask",
                    "ec2:CancelExportTask",
                    "ec2:TerminateInstances",
                    "ec2:DescribeImportImageTasks",
                    "ec2:DescribeImportSnapshotTasks",
                    "ec2:CancelImportTask"],
        "Resource": "*" }]
}
```

Figure 3.8: Policy given in the EC2 User Guide [27] for the role assumed by the instance of EC2 for `importImage, exportImage` API methods.

```
{ "Statement": [{   "Effect": "Allow",
                    "Action": [ "s3:GetObject",
                                "s3:GetBucketLocation" ],
                    "Resource": [ "arn:aws:s3:::test-bucket",
                                  "arn:aws:s3:::test-bucket/
                                   image-to-import" ]},
                {   "Effect": "Allow",
                    "Action": [ "ec2:CopySnapshot",
                                "ec2:RegisterImage",
                                "ec2:ImportImage" ],
                    "Resource": "*" }]}
```

Figure 3.9: A least-privilege identity-based policy for the role assumed by the instance of EC2 for `importImage`. Note that this is different from the least-privilege policy for the role assumed by the Lambda function that invokes `importImage`().

```
{ "Statement":  [{  "Effect": "Allow",
                    "Action": [ "s3:PutObject",
                              "s3:GetBucketAcl" ],
                    "Resource": [ "arn:aws:s3:::test-bucket",
                                "arn:aws:s3:::test-bucket/
                                 image-exports/export-file" ]},
                {   "Effect": "Allow",
                     "Action": "ec2:DescribeImages",
                     "Resource": "*" }]}
```

Figure 3.10: A least-privilege identity-based policy for the role assumed by the instance of EC2 for `exportImage`(). Note that this is different from the least-privilege policy for the role assumed by the Lambda function that invokes `exportImage`().

3.4 Summary

Poor documentation is a major problem. Ideally, documentation would be adequate so one does not have to identify what permissions are associated with an API method experimentally as we have had to do. We have observed that AWS documentation is largely missing, and in cases that it exists, it is often incorrect and incomplete. Our experimental assessment has led to some interesting observations on the per-API method least-privilege sets. Some of our observations are alarming, for example, our observations in Kinesis that leads us to conjecture that there is a permission that is not public. In the next chapter, we leverage our identification of least-privilege sets of permissions to assess whether policies that have been made available publicly with Lambda functions are indeed least-privilege.

Chapter 4

Least-Privilege in Applications

We have analyzed two publicly available sets of policies for AWS serverless applications as to whether they adhere to least-privilege. Our identification of least-privilege sets against which we compare the published policies are from the approach we discuss in the previous chapter. In Section 4.1 we discuss our study of two repositories, each of which contains several serverless applications. These applications appear to be snippets of code, presumably with the intent that a developer of a large application can adopt or adapt these snippets as part of their application. In Section 4.2, we look at two complete demo serverless applications, each of which contains several Lambda functions. Both these applications have identity-based policies that have been published alongside; we analyze those policies for least-privilege.

4.1 Two Repositories

In this section, we discuss least-privilege in the context of serverless applications that have been made available publicly by (i) AWS [22], and, (ii) a company called Serverless, Inc. [55]. Table 4.1 presents our quantitative observations for the two repositories. We now discuss each, in turn.

AWS repository This is a repository from AWS that has been made available for developers to adopt and adapt in their own applications. We have analyzed the Lambda functions that have been written in Javascript in that repository, and the policies that have been published for each. In the AWS repository, customer managed policies are published as so-called policy templates, which allow for placeholder values for resources. The

Repo	# apps	# with policy	# policies AWS managed	# policies least-privilege
AWS	56	54	3	6
Serverless	68	21	0	8

Table 4.1: Data on two publicly available repositories.

intent is that those placeholder values are to appear in any adaptation of a developer of the code for the policy to have the intended effect. The only difference between a customer managed policy and a policy template is that AWS publishes the available policy templates [24]. This detail does not impact whether the policy is least-privilege or not.

As Table 4.1 indicates, there are 56 demo applications in the repository which are written in Javascript, which we have looked at. Of these, 54 have policies published alongside. Of these 54, three are AWS managed and the remainder are customer managed, i.e., policy templates. We find that only six of the 54 adhere to least-privilege. None of these is amongst the ones that are AWS managed. None of the six that are least-privilege has more than one permission; that is, they are amongst the easiest to devise. An example of over-privilege from the repository is for the Microservice HTTP Endpoint application [5]. As the `template.yaml` file there indicates, it adopts the `DynamoDBCrudPolicy` that is part of the policy templates published by AWS [24]. Unfortunately, for the application, that policy is over-privileged. The application needs the `dynamodb:DeleteItem`, `dynamodb:Scan`, `dynamodb:UpdateItem` and `dynamoDB:PutItem` permissions only (see Figure 3.1 in Chapter 3). `DynamoDBCrudPolicy` grants a number of additional, redundant permissions.

Thus, if a developer adopts not only the code, but also the policies from the AWS repository, they are likely to end up with over-privileged applications. If they adopt the code only and not the policies, then they have the additional task of devising policies.

Serverless, Inc. repository The intent of this repository seems to be the same as the one from AWS; for developers to be able to adopt and adapt in their own applications. The repository contains a total of 68 applications for AWS. We looked at all 68 applications; Table 4.1 presents our data. As the table indicates, only 21 of the 68 applications have a policy published alongside. This is reasonable: only those 21 applications make calls to AWS services. The others are written so they can be deployed in AWS Lambda, but either have code-logic only, or invoke non-AWS services only, such as an http request to a non-AWS service to find the current time. It is interesting to observe, as the table reports, that none of those 21 policies is AWS managed. Notwithstanding, we find only eight of the 21 to be least-privilege.

Overall, both least-privilege and over-privileged instances in this repository are less straightforward than the ones in the AWS repository. For example, policies with more than one permission are amongst those that are least-privilege; for example, an application that needs two permissions on the same resource [54]; thus, the policy, correctly, places both actions in the same statement of the policy. Amongst the policies that are over-privileged, we have some whose cause is the use of wildcards [52]. We also have policies with a few redundant permissions, for example, one that grants six permissions, when only five are needed [53]; the redundant permission in that case is `dynamodb:Query`.

4.2 Two Applications

Apart from the two repositories we discuss in the previous section that contain several applications, we have looked also at two applications for which identity-based policies have been published alongside, which we discuss in this section. Unlike the applications in the repositories, the two applications we discuss in this section are more full-fledged. They contain several Lambda functions each. The applications in the repositories in the previous section can be seen more as snippets. The two applications we have looked at are one for a Bookstore [15] and the other for a retail outlet [45]. We now discuss each, in turn.

The Bookstore application The Bookstore application [15] emulates Amazon's customer website, with functionality such as those to search books, add to carts, and view orders. The application has a total of nine Lambda functions. The policy that has been published with it, unfortunately, is grossly over-privileged. The root cause for this appears to be that the same role is assumed by all Lambda functions when they run. The policy that is attached grants all permissions across all those Lambda functions to the role, thereby resulting in over-privilege. A more careful design would be for each Lambda function whose least-privilege set is distinct to be associated with a role of its own. This design choice can be seen as separation of concerns in software engineering and the advantage is modularity of the application. With separate roles for every lambda, in an event of a security breach the affected lambdas in an application are limited. In large applications with many lambda functions, the execution role sharing many permissions across multiple lambda functions can be hard to manage. The downside is that every time a change is made to a lambda function, the identity-based policy attached to the execution role of the lambda may need to be changed as well. This is an additional burden on the developer.

Worse, still, even if we consider the union of method calls made across all the Lambda functions in the application, the policy is over-privileged. For example, it grants `s3:Get*` and `s3:List*` actions, i.e., every permission that begins with "`s3:Get`" and "`s3:List`,"

```
{ "Statement": [{   "Effect": "Allow",
                  "Action": [ "s3:Get*",
                              "s3:List*" ],
                  "Resource": "*" },
                {   "Effect": "Allow",
                    "Action": [ "dynamodb:PutItem",
                              "dynamodb:Query",
                              "dynamodb:UpdateTable",
                              "dynamodb:UpdateItem",
                              "dynamodb:BatchWriteItem",
                              "dynamodb:GetItem",
                              "dynamodb:Scan",
                              "dynamodb:DeleteItem" ],
                    "Resource": [ "arn:aws:dynamodb:region:accountId:table/Books",
                                  "arn:aws:dynamodb:region:accountId:table/Orders",
                                  "arn:aws:dynamodb:region:accountId:table/Cart",
                                  "arn:aws:dynamodb:region:accountId:table/Books/*" ]},
            {   "Effect": "Allow",
                  "Action": [ "logs:CreateLogGroup",
                              "logs:CreateLogStream",
                              "logs:PutLogEvents" ],
                  "Resource": "*" }]}
```

Figure 4.1: The over-privileged policy that has been published with the Bookstore application [15]. The same role is assumed by all the Lambda functions in the application, and therefore, this same policy is attached to all of them when they run.

```
{ "Statement": [{ "Effect": "Allow",
                  "Action": [ "dynamodb:Scan",
                              "dynamodb:Query" ],
                  "Resource": [ "arn:aws:dynamodb:region:accountId:table/
                                 Books/index/*",
                                 "arn:aws:dynamodb:region:accountId:table/Books" ]},
                { "Effect": "Allow",
                  "Action": [ "logs:CreateLogStream",
                              "logs:CreateLogGroup",
                              "logs:PutLogEvents" ],
                  "Resource": "*" }]}
```

Figure 4.2: A least-privilege policy for the `listBooks` lambda function in the bookstore application.

which is redundant. Indeed, none of the nine Lambda functions exercises S3; therefore any permission in S3 that is awarded is redundant. Permissions on DynamoDB are excessive as well. In Appendix 4.1, we reproduce the policy that has been published for the application. We then constructed least-privilege policies using our approach in Chapter 3 for two of the Lambda functions that are part of the application, `listBooks` and `addToCart`. Figure 4.2 shows the least-privilege policy for `listBooks` lambda function and Figure 4.3 shows the least-privilege policy for `addToCart` lambda function. Included in these least-privilege policies are permissions for calls for logging that are made implicitly, and are not explicit in the code for those Lambda functions. The logging permissions are implicit because AWS Lambda automatically reports metrics to AWS Cloudwatch. To discover the implicit permissions, the only way that we are aware of is to read the AWS Lambda documentation on setting up logging. Without those permissions, the Lambda functions will run successfully; however, logging will not be performed.

```
{ "Statement": [{ "Effect": "Allow",
                  "Action": "dynamodb:PutItem",
                  "Resource": "arn:aws:dynamodb:region:accountId:table/Cart" },
                { "Effect": "Allow",
                  "Action": [ "logs:CreateLogStream",
                              "logs:CreateLogGroup",
                              "logs:PutLogEvents" ],
                  "Resource": "*" }]}
```

Figure 4.3: A least-privilege policy for the `addToCart` lambda function in the bookstore application.

```
{ "Statement": [{ "Effect": "Allow",
                  "Action": [ "kinesis:GetShardIterator",
                              "kinesis:GetRecords",
                              "kinesis:DescribeStream",
                              "kinesis:ListStreams" ],
                  "Resource": "arn:aws:kinesis:region:accountId:stream/
                              stream-name"}]}
```

Figure 4.4: Policy in Hello, Retail! for Lambda functions that are triggered by AWS Kinesis events. The Visual Editor [36] reports a warning for this policy, as we discuss in the prose.

The Hello, Retail! application Hello, Retail! is an application for a retail store [45]. It has won a serverless architecture award [43]. The application is built for the scenario that a merchant adds a product to their store. Once a product is added to the store, a photographer takes a photograph of the product, followed by prospective customers of the store being able to see the product and its photograph. Hello, Retail! realizes its functionality by producing and consuming events anytime a photographer is registered, a new product is added by the merchant, and a photo for the product is taken by a photographer.

We analyzed 13 of the Lambda functions of Hello, Retail! We found only a somewhat subtle problem with over-privilege as we discuss below. We found also what appear to be under-privileged policies, which we discuss further below as well. Every policy in Hello, Retail! that we examined appears to be "hand-crafted" carefully, and not, for example, blindly copied from other sources. This may explain at least partly why the policies appear to be of the highest quality amongst all the ones we encountered in the course of carrying out our work for this thesis. It would certainly be interesting to analyze the entirety of Hello, Retail!; to do so, we would need to greatly expand the scope of AWS API methods for which we have identified least-privilege sets.

Before we discuss the instances of over- and under-privilege, we observe a good design choice in Hello, Retail!, unlike in the Bookstore application we discuss above: each Lambda function assumes a distinct role when it runs. The only instance of over-privilege we found regards a grant of `kinesis:DescribeStream` and `kinesis:ListStreams` to a particular Lambda function. On further investigation, the problem seems to be one of design in AWS, rather than a misconfiguration on the part of Hello, Retail!.

There are two issues here; we present the problematic policy in Figure 4.4. One issue is that the `kinesis:DescribeStream` and `kinesis:ListStreams` permissions are needed only when a Lambda function is registered to be the callback when a Kinesis event happens. Those permissions are not needed subsequently; in particular, they are not needed when the Lambda function is invoked and runs in response to an event. We have confirmed this by removing those permissions once the Lambda function is registered as the callback; we observe that the Lambda function is indeed invoked successfully. However, such a change to a policy is not a customary operation: as we say in Chapter 3, customarily, a policy is specified and attached to a role at deployment-time.

The other issue is that `kinesis:ListStreams` is not a permission that is to be bound to any particular resource, i.e., instance of a stream. Rather, it is a permission that applies to any stream. When the policy in Figure 4.4 is entered into the Visual Editor of AWS [36], it gives a warning, "The actions in your policy do not support resource-level permissions

```
{ "Statement": [{ "Effect": "Allow",
                  "Action": [ "kinesis:GetShardIterator",
                              "kinesis:GetRecords",
                              "kinesis:DescribeStream" ],
                  "Resource": "arn:aws:kinesis:region:accountId:stream/
                              stream-name"},
                { "Effect": "Allow",
                  "Action": "kinesis:ListStreams",
                  "Resource": "*" }] }
```

Figure 4.5: Modified version of the policy from Figure 4.4.

and require you to choose all resources." In Figure 4.5 we present a modified policy for which there is no warning; it grants `kinesis:ListStreams` to `*`. With either policy, the Lambda function works as expected. This suggests that the VisualEditor quietly grants `kinesis:ListStreams` to all resources, which would be undesirable.

We mention the two instances of under-privilege that we have discovered in Hello, Retail!. One regards logging: it appears that Hello, Retail! does not configure the required `logs:CreateLogStream` permission, and therefore, logging fails. The other instance of under-privilege we found regards the `decrypt`() call to the AWS Key Mananagement Service. This requires the `kms:Decrypt` permission; however, the role the Lambda function that invokes `decrypt`() assumes is not configured with a policy that grants that permission. Nevertheless, the application fails gracefully.

4.3 Summary

We have found that the vast majority of identity-based policies that have been published for AWS serverless applications suffer from over-privilege. Therefore, we caution AWS serverless application developers from blindly adopting those policies in their applications. Doing so can expose their applications to security compromises. One possibility is an insider attack. Here, an employee uses an application, with over-privileges assigned to it, to perform his daily activities. If one day he decides that he is soon going to leave the company, he can use the over-privilege application to cause havoc to company resources. Sometimes employees may cause damages to company resources unknowingly. In this case, the employee is a user of the application. In fact, any user using an over-privilege application is a possible threat to security of resources.

Chapter 5

Steps towards automated policy generation

5.1 Argument Partitioning

As we discussed earlier in Chapter 3, the least privileged identity based policy for an API is dependent on the API method and arguments for the method. We also encountered cases where different arguments to an AWS API method required different permissions. For the same method, there exist more arguments which may require a new set of permissions.

We mentioned the problem of invoking a method with all possible valid inputs using our manual approach in Chapter 3. The problem can be addressed if we can partition the argument space and only consider a representative argument from each partition in the argument space. Moreover, we are only interested in arguments that do not make the program go into an error state. This is assuming that the execution role of the API has full access, making sure that error is not due to lack of a privilege.

Definition 5.1. A *partition* of a set S is a collection of nonempty disjoint subsets of S whose union is S.

Let S be the set of all the possible arguments that can be passed into an API method. The partition of S provides a way to decompose the S into disjoint subsets based on some equivalence relation or characteristic. The advantage is that there is no need to test all possible arguments, one from each partition is sufficient.

This problem often comes up in testing of software as input space partitioning and the way to partition the input space is called input domain modelling. There are two main approaches to input domain modelling: Interface-based input domain modelling and Functionality-based input domain modelling.

In the interface-based approach each parameter is considered in isolation and ignores the dependencies of a parameter on other parameter. The strength of this approach is that it is easy to identify characteristics which are needed for partitioning. Since each characteristic limits itself to a single parameter, it is easy to come up with test cases as well. One of the weakness of this approach is that not all the information available to the test engineer will be reflected in the interface domain model. This raises the possibility of the model being incomplete. Another weakness is that due to the lack of considering dependencies between parameters, important sub-combinations may get missed [28].

In functional-based approach, the goal is to identify characteristics based on the functionality of the system rather than the interface. This allows the tester to incorporate some semantics or domain knowledge into the input domain modelling. Unlike the interface-based approach the dependencies among parameters are considered. The strength of this approach is that it is more goal-oriented but the down side is that it requires more design and thought [28].

In functional-based approach, characteristics can be derived from any of the following: preconditions, postconditions, the program specification, design information or the program. Ostrand and Balcer et al. [47] talk about a systematic specification based method, called *category-partition*, for generating arguments necessary for functional tests. Other approaches include boundary-value analysis and classification trees method, which is a refinement over the category-partition method. These are possible choices that we may explore in the future. Grindal et al. [42] did a thorough survey of such techniques.

Category-Partition method helps a tester devise what inputs and environment conditions to consider for testing that may affect the behaviour of a method. A tester begins with identifying *categories* of information that characterize each input and environment condition. A category is a major property of an input or environment condition. Each category can be further partitioned into distinct *choices* that include all possible values for a particular category. The choices are the partition classes from which a representative element is taken for testing a method.

We use the Category-Partition approach to come up with various categories and choices of parameters and environment conditions to invoke an AWS API method. This helps us tackle the problem of incompleteness regarding arguments. Figure 5.1 shows an example of using category-partitioning to come up with possible `CreateBucketRequest` for

s3:createBucket(CreateBucketRequest) method.

```
Arguments:
    bucket name size:
        is empty                              [error]
        is less than 3 characters long        [error]
        is more than 63 characters long       [error]
        is between 3 and 63 characters long [property length_valid]

    bucket name:
        is null                               [error]
        is not a unique string                [error]
        contains an uppercase character       [error]
        begins with 'xn-'                     [error]
        in IP address format                  [error]
        is unique and 'valid'                 [if length_valid]
                                              [property bucket_name_valid]

    acl:
        is empty                              [if bucket_name_valid]
        is an arbitrary string                [error]
        is UNKNOWN_TO_SDK_VERSION             [error]
        is PRIVATE                            [if bucket_name_valid]
        is AUTHENTICATED_READ                 [if bucket_name_valid]
        is PUBLIC_READ                        [if bucket_name_valid]
        is PUBLIC_READ_WRITE                  [if bucket_name_valid]

    is object lock enabled:
        is False                              [if bucket_name_valid]
        is True                               [if bucket_name_valid]

    grant full control:
        is empty                              [if bucket_name_valid]
        is an arbitrary string                [error]
        is a valid user id, email or uri
        (format: id="", emailAddress="",
        uri="")                               [if bucket_name_valid]
```

```
grant read:
    is empty                              [if bucket_name_valid]
    is an arbitrary string                [error]
    is a valid user id, email or uri
    (format: id="", emailAddress="",
    uri="")                               [if bucket_name_valid]

grant read ACP:
    is empty                              [if bucket_name_valid]
    is an arbitrary string                [error]
    is a valid user id, email or uri
    (format: id="", emailAddress="",
    uri="")                               [if bucket_name_valid]

grant write:
    is empty                              [if bucket_name_valid]
    is an arbitrary string                [error]
    is a valid user id, email or uri
    (format: id="", emailAddress="",
    uri="")                               [if bucket_name_valid]

grant write ACP:
    is empty                              [if bucket_name_valid]
    is an arbitrary string                [error]
    is a valid user id, email or uri
    (format: id="", emailAddress="",
    uri="")                               [if bucket_name_valid]

bucket location constraint:
    is AF_SOUTH_1                         [if bucket_name_valid] [single]
    is AP_EAST_1                          [if bucket_name_valid] [single]
    is AP_NORTHEAST_1                     [if bucket_name_valid] [single]
    is AP_NORTHEAST_2                     [if bucket_name_valid] [single]
    is AP_NORTHEAST_3                     [if bucket_name_valid] [single]
    is AP_SOUTH_1                         [if bucket_name_valid] [single]
    is AP_SOUTHEAST_1                     [if bucket_name_valid] [single]
    is AP_SOUTHEAST_2                     [if bucket_name_valid] [single]
    is CA_CENTRAL_1                       [if bucket_name_valid] [single]
```

```
        is CN_NORTH_1                          [if bucket_name_valid] [single]
        is CN_NORTHWEST_1                      [if bucket_name_valid] [single]
        is EU                                  [if bucket_name_valid] [single]
        is EU_CENTRAL_1                        [if bucket_name_valid] [single]
        is EU_NORTH_1                          [if bucket_name_valid] [single]
        is EU_SOUTH_1                          [if bucket_name_valid] [single]
        is EU_WEST_1                           [if bucket_name_valid] [single]
        is EU_WEST_2                           [if bucket_name_valid] [single]
        is EU_WEST_3                           [if bucket_name_valid] [single]
        is ME_SOUTH_1                          [if bucket_name_valid] [single]
        is SA_EAST_1                           [if bucket_name_valid] [single]
        is UNKNOWN_TO_SDK_VERSION              [error]
        is US_EAST_2                           [if bucket_name_valid] [single]
        is US_GOV_EAST_1                       [if bucket_name_valid] [single]
        is US_GOV_WEST_1                       [if bucket_name_valid] [single]
        is US_WEST_1                           [if bucket_name_valid] [single]
        is US_WEST_2                           [if bucket_name_valid] [single]

Environment:
    bucket with the specified name exists           [error]
    bucket with the specified name does not exist   [if bucket_name_valid]
```

Figure 5.1: Shows the categories and choices when the Category-Partition method applied to `s3.createBucket(CreateBucketRequest)`. A valid user id, email address or uri is one that has an associated AWS account with it. A random email address of the form *emailAddress="myrandomemail@gmail.com"* is invalid and results in error.

In the unrestricted version of the method, one without `[error]` and `[single]` annotations, there would be 4245696 possibilities of test cases we will need to generate for testing one value from each choice of a category. Luckily, by reading the specification we can identify choices that will result in errors and we can label them as such. Using the `[error]` annotation, we can reduce the number of tests needed to 8017. We got 8017 by not counting the combination of `[error]` choices with any choice from other categories.

We were interested in knowing whether there are choices that do not contribute to a change in permissions needed for an invocation of `s3.createBucket(CreateBucketRequest)`. In our testing, we started with a `CreateBucketRequest` object with a valid bucket name and tested values from the choices in the bucket location constraint category. The *bucket location constraint* describes the region the S3 bucket will be created in. We noticed that

the location constraint did not require additional permissions. Therefore, one test case for each choice in this category is sufficient. We use the `[single]` annotation, as introduced in the original paper, to describe the special and redundant condition that do not need to combined with all possible choices from other categories. With the introduction of the `[single]` annotation, we can further reduce the number of tests to 320. This is a significant decrease from the the number of tests in the unrestricted case. The number of tests needed are still too large to be generated manually. For scaling this process to many more APIs across various AWS services, there is a need for a way to generate these tests programmatically.

In the next section, we describe the use of a unit test generator to generate tests programmatically.

5.2 Test Case Generation

In the previous section, we discussed a way to partition the arguments and pick a representative argument from each class of arguments. We used the Category-Partition method to help us devise a strategy to pick the representative arguments. Due to the combinatorial nature of the method, the test cases are large to be constructed manually. We are interested in knowing whether the test cases can be constructed programmatically instead.

We consider Randoop as a plausible choice for test-case generation. Pacheco et al. [48] introduced a Feedback-directed random test generation techique. Randoop is an implementation of this technique to randomly generate tests by incorporating feedback obtained from executing test inputs as they are created. It takes in the `JAR` of the program that we wish to generate the tests for, in our case AWS SDK. The technique builds a sequence of method calls starting from an empty set. Each method call in the sequence contains the method name and the arguments it takes, which can be Strings, primitive values or reference values returned by previous method calls. The technique randomly selects a method call to add to a sequence and finds arguments from among previously-constructed methods in the sequence. Once the method is added to the sequence, it is executed and checked against a set of contracts and filters. Randoop stops generating the tests after a time limit has passed. The time limit can be configured by the `time-limit` option.

When the `testclass` option is set, Randoop considers user-specified argument values during the test-case generation. The user can further annotate primitive fields within this Java test class using `@TestValue`. An example of a test class is shown in Figure 5.2.

```
public class TestValues {
    private static final String EXECUTE_ROLE_ARN =
    "arn:aws:iam::accountId:role/execute-role";

    @TestValue
    public static String bucketName = "test-bucket";

    public static S3Client createS3Client() {
        AwsSessionCredentials awsSessionCredentials =
            getAwsSessionCreds(EXECUTE_ROLE_ARN);
        return S3Client.builder()
                .credentialsProvider(
                    StaticCredentialsProvider.create(awsSessionCredentials))
                .region(Region.US_EAST_1)
                .build();
    }

    public static CreateBucketRequest createBucketRequest(String bucket) {
        return CreateBucketRequest.builder().bucket(bucket).build();
    }

    private static AwsSessionCredentials getAwsSessionCreds(String roleArn) {
        StsClient sts = StsClient.builder().region(Region.US_EAST_1).build();
        AssumeRoleResponse assumeRoleResponse = sts.assumeRole(
            AssumeRoleRequest.builder()
                .roleArn(roleArn)
                .roleSessionName("session1")
                .build());
        Credentials sessionCreds = assumeRoleResponse.credentials();
        return AwsSessionCredentials.create(sessionCreds.accessKeyId(),
            sessionCreds.secretAccessKey(), sessionCreds.sessionToken()
        );
    }
}
```

Figure 5.2: A test class that can be set in the `testclass` option. The bucket name is annotated with the `@TestValue` which will now be considered by Randoop as an input to a method call during the test generation process. The `public` methods in the class are considered as methods that return reference values for an AWS API method.

In addition to the methods in the test class, a list of methods from the AWS SDK can be given as an input to Randoop using the `methodlist` option. This list describes the methods that Randoop should consider when generating the tests. For the AWS API method, `s3.createBucket(CreateBucketRequest)`, we discovered that `s3:CreateBucket` is not the only permission needed for all valid arguments. There are arguments that require 2 additional permissions. The findings are shown in Figures 5.3, 5.4

```
Arguments:
    acl:
        is AUTHENTICATED_READ
        is PUBLIC_READ
        is PUBLIC_READ_WRITE

    grant read:
        is a valid user id, email or uri
        (format: id="", emailAddress="", uri="")

    grant read ACP:
        is a valid user id, email or uri
        (format: id="", emailAddress="", uri="")

    grant write:
        is a valid user id, email or uri
        (format: id="", emailAddress="", uri="")

    grant write ACP:
        is a valid user id, email or uri
        (format: id="", emailAddress="", uri="")
```

Figure 5.3: `s3:createBucket, s3:PutBucketAcl` permissions are needed for the choices of arguments shown.

```
Arguments:
    object lock enabled:
        is True
```

Figure 5.4: `s3:createBucket, s3:PutBucketObjectLockConfiguration` permissions are needed for the choices of arguments shown.

5.3 Summary

We recognize the limitations of generating least-privilege policies manually. The main drawback is incompleteness of arguments in our manual approach of testing privileges. We use the Category-Partition method, introduced by Ostrand and Balceret al. [47] to partition the argument space. We claim that one representative argument from each partition is sufficient for coverage of the argument space. We use a random test generation tool, Randoop, based on feedback-directed technique and developed by Pacheco et al. [48]. With Randoop, we are able to generate tests for `s3.createBucket(CreateBucketRequest)` API method and discover additional permissions required for some of the arguments.

Chapter 6

Recommendations

We recall from Chapter 1 that our goal is to help developers with devising least-privilege policies for their serverless applications. Often security is perceived as an afterthought in software development process in many teams. The approach mentioned in this thesis is an effort to incorporate in the development process at a minimal time investment. In an ideal world, coming up with the permissions needed for a lambda function would be easy but in our experience it is not always the case. An iterative process can be used instead. This can be seen in the same light as why software testing is an important step in the software development life cycle. We encourage the developers to make an active effort to look for various AWS APIs invoked in every lambda function in an application and using our approach to come up with the least-privilege policies during the development process. This incorporation of security in the development process improves the security stature of the application early on, makes developers aware of security of their application and saves them time that would have been instead spent on ensuring security of the application later in the development process. Often times, improving on the over-privileged policies can be taxing and require many iterations.

We now make a few recommendations based on our work and consequent observations.

Recommendations for developers using AWS:

- **Do not blindly adopt published policies** Published policies appear overwhelmingly to be over-privileged. This includes policies in AWS user guides. A developer should carefully examine a policy, and edit it, before adopting it for their application.

- **Identify least-privilege sets for API methods** It is easy to cull any AWS API method calls one makes in one's Lambda functions. To us, a first step any developer should take before configuring an identity-based policy is to identify, for each such method and their anticipated arguments, what a least-privilege set of permissions is. This process can be painstaking as documentation may be non-existent or faulty. However, it is worthwhile if one cares about security. Then, using our definition in Chapter 3, the developer of an application can identify least-privilege sets of permissions.

- **Provision distinct roles** It is necessary to provision a distinct role for each set of API methods. That is, suppose an application has several Lambda functions. Now suppose we perceive a Lambda function as defined in Chapter 3, i.e., as a set of pairs of API methods and arguments that are anticipated for those methods. We first partition our Lambda functions based on whether their sets are identical. This determines the number of different roles we need—one per partition.

Recommendations for both AWS and developers using AWS :

- **Be wary when using AWS managed policies** AWS managed policies "are designed to provide permissions for many common use cases." [17] However, as one may expect, AWS cannot possibly anticipate every possible use case. Hence, it behooves the adopter of an AWS managed policy to carefully look through it, and ensure that the policy indeed meets the adopter's needs. Furthermore, the User Guide warns, "AWS occasionally updates the permissions defined in an AWS managed policy. When AWS does this, the update affects all principal entities (users, groups, and roles) that the policy is attached to." Thus, such a change could be effected unbeknownst to the adopter. The safest option, in our view, is to not adopt AWS managed policies at all if one cares about security. They can be used, for example, as a starting point for a customer managed policy. But that is the extent of their utility, in our view.

- **Anticipate bugs in code and documentation** Based on our observation on documentation (see Section 3.1), and on Kinesis, in which we conjecture that there is a hidden permission (see under "Kinesis" in Section 3.3), we caution developers that, like other complex pieces of software, the authorization components of AWS, and their documentation, may have bugs. We recommend that a developer be aware of this possibility.

Chapter 7

Conclusions

We have addressed least-privilege identity-based policies for Lambda functions in AWS. We have motivated our focus on each of identity-based policies, Lambda functions and AWS, and pointed out that devising such policies is not easy. A problem is poor documentation, which necessitates a somewhat painstaking experimental process for determining what a least-privilege set of permissions is for a method and arguments for that method. We have identified least privilege sets for 171 methods across five different AWS services and made observations about them from the standpoint of how easy it is for a developer to devise least-privilege policies. Some of our observations are alarming: for example, on being unable to guarantee a least-privilege policy for two methods in AWS Kinesis. We have studied also two repositories of Lambda functions, and two sample applications, all publicly available, and found that over-privilege is pervasive in them. We have concluded with a few recommendations for developers of serverless applications in AWS.

There is tremendous scope for future work. It would be invaluable for developers of serverless applications to have a tool to automatically generate least-privilege policy for their applications. Another key insight with the progress made in Chapter 5 is to observe what different set of permissions are needed for a method and arguments based on the argument partitioning. In addition, an interesting piece of work would be to examine other cloud providers in the same manner as we do in this work. Even within AWS itself, it is interesting to examine AWS managed policies, and ask how well designed they are. This would require a semantic understanding of the intent of each of those policies, which in itself can be challenging. Finally, there is the question of whether an alternative design for the policy language would be better, for example, one without a proliferation of actions as in AWS.

References

[1] Amazon Web Services (AWS). AWS SDK for Java API Reference – 1.11.772. https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html, 2013.

[2] Amazon Web Services (AWS). AWS Security Best Practices. https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Best_Practices.pdf, August 2016.

[3] Amazon Web Services (AWS). AWS SDK for Java API Reference – 2.13.6. https://sdk.amazonaws.com/java/api/latest/, 2019.

[4] Amazon Web Services (AWS). Interface S3Client. https://sdk.amazonaws.com/java/api/latest/software/amazon/awssdk/services/s3/S3Client.html, 2019.

[5] Amazon Web Services (AWS). Microservice HTTP Endpoint. https://github.com/aws-samples/serverless-app-examples/tree/master/javascript/microservice-http-endpoint, October 2019.

[6] Amazon Web Services (AWS). Security Overview of AWS Lambda. https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf, April 2019.

[7] Amazon Web Services (AWS). Amazon DynamoDB – Fast and flexible NoSQL database service for any scale. https://aws.amazon.com/dynamodb/, April 2020.

[8] Amazon Web Services (AWS). Amazon EC2 – Secure and resizable compute capacity in the cloud. https://aws.amazon.com/ec2/, April 2020.

[9] Amazon Web Services (AWS). Amazon Elastic Transcoder. https://aws.amazon.com/elastictranscoder/, April 2020.

[10] Amazon Web Services (AWS). Amazon GuardDuty – Protect your AWS accounts and workloads with intelligent threat detection and continuous monitoring. https://aws.amazon.com/guardduty/, April 2020.

[11] Amazon Web Services (AWS). Amazon Kinesis – Easily collect, process, and analyze video and data streams in real time. https://aws.amazon.com/kinesis/, April 2020.

[12] Amazon Web Services (AWS). Amazon Macie – A machine learning-powered security service to discover, classify, and protect sensitive data. https://aws.amazon.com/macie/, April 2020.

[13] Amazon Web Services (AWS). Amazon S3 – Object storage built to store and retrieve any amount of data from anywhere. https://aws.amazon.com/s3/, April 2020.

[14] Amazon Web Services (AWS). Amazon web services. https://aws.amazon.com, April 2020.

[15] Amazon Web Services (AWS). AWS Bookstore Demo App. https://github.com/aws-samples/aws-bookstore-demo-app, January 2020.

[16] Amazon Web Services (AWS). AWS Config – Record and evaluate configurations of your AWS resources. https://aws.amazon.com/config/, April 2020.

[17] Amazon Web Services (AWS). AWS Identity and Access Management – User Guide. https://docs.aws.amazon.com/IAM/latest/UserGuide/, April 2020.

[18] Amazon Web Services (AWS). AWS Identity and Access Management: User Guide. https://docs.aws.amazon.com/IAM/latest/UserGuide/iam-ug.pdf, May 2020.

[19] Amazon Web Services (AWS). AWS Lambda – Run code without thinking about servers; pay only for the compute time you consume. https://aws.amazon.com/lambda/, April 2020.

[20] Amazon Web Services (AWS). AWS SDK for Java 2.0 Developer Guide. https://docs.aws.amazon.com/sdk-for-java/v2/developer-guide/, 2020.

[21] Amazon Web Services (AWS). AWS SDK for Java Developer Guide. https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/, 2020.

[22] Amazon Web Services (AWS). AWS Serverless Application Repository Examples. https://github.com/aws-samples/serverless-app-examples, April 2020.

[23] Amazon Web Services (AWS). AWS Trusted Advisor – Reduce Costs, Increase Performance, and Improve Security. https://aws.amazon.com/premiumsupport/technology/trusted-advisor/, April 2020.

[24] Amazon Web Services (AWS). Policy Template List. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-policy-template-list.html, 2020.

[25] Amazon Web Services (AWS). Serverless – Build and run applications without thinking about servers. https://aws.amazon.com/serverless/, April 2020.

[26] Amazon Web Services (AWS). Types of Cloud Computing. https://aws.amazon.com/types-of-cloud-computing/, April 2020.

[27] Amazon Web Services (AWS). Vm import/export user guide. https://docs.aws.amazon.com/vm-import/latest/userguide/vm-import-ug.pdf, May 2020.

[28] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 2nd edition, 2016.

[29] Android developer guides. RequiresPermission. https://developer.android.com/reference/androidx/annotation/RequiresPermission, December 2019.

[30] Android developer guides. Manifest.permission. https://developer.android.com/reference/android/Manifest.permission, April 2020.

[31] Appsecco. An ssrf, privileged aws keys and the capital one breach. https://blog.appsecco.com/an-ssrf-privileged-aws-keys-and-the-capital-one-breach-4c3c2cded3af, May 2020.

[32] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[33] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A Security Analysis of Amazon's Elastic Compute Cloud Service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1427–1434, New York, NY, USA, 2012. Association for Computing Machinery.

[34] Matt Bishop. *Computer security : art and science*. Addison-Wesley, Boston, 2003.

[35] ccssubeodee2ieTe. Three examples; AWS identity-based policies. https://github.com/ccssubeodee2ieTe/ccssubeodee2ieTe/, April 2020.

[36] Joy Chatterjee. Use the New Visual Editor to Create and Modify Your AWS IAM Policies. *AWS Security Blog*, November 2017.

[37] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, page 171–190, USA, 2002. USENIX Association.

[38] CloudSploit. A technical analysis of the capital one hack. `https://blog.cloudsploit.com/a-technical-analysis-of-the-capital-one-hack-a9b43d7c8aea`, May 2020.

[39] Byron Cook. Formal Reasoning About the Security of Amazon Web Services. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 38–47, Cham, 2018. Springer International Publishing.

[40] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. CCS '11, pages 627–638, New York, NY, USA, 2011. Association for Computing Machinery.

[41] Google Cloud. Granting, changing, and revoking access to resources. `https://cloud.google.com/iam/docs/granting-changing-revoking-access`, April 2020.

[42] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[43] John McKim. Announcing the Winners of the Inaugural ServerlessConf Architecture Competition. *A Cloud Guru*, May 2017.

[44] Matthew Miller. AWS SDK for Java 2.x released. *AWS Developer Blog*, November 2018.

[45] Nordstrom, Inc. Hello, Retail! `https://github.com/Nordstrom/hello-retail`, September 2018.

[46] Charlie Osborne. The top 10 security challenges of serverless architectures. *Zero Day*, January 2018.

[47] T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Commun. ACM*, 31(6):676–686, June 1988.

[48] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, 2007.

[49] P. Rodler. Understanding the QuickXPlain Algorithm: Simple Explanation and Formal Proof. *ArXiv*, abs/2001.01835, 2020.

[50] R.Shirey, Network Working Group. Internet Security Glossary, Version 2. `https://tools.ietf.org/html/rfc4949`, August 2007.

[51] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Communications of the ACM*, 17(7), July 1974.

[52] Serverless. Receive an email, store in S3 bucket, trigger a lambda function. `https://github.com/serverless/examples/tree/master/aws-node-ses-receive-email-body`, October 2018.

[53] Serverless. Serverless REST API. `https://github.com/serverless/examples/tree/master/aws-node-rest-api-with-dynamodb`, August 2019.

[54] Serverless. FFmpeg app. `https://github.com/serverless/examples/tree/master/aws-ffmpeg-layer`, February 2020.

[55] Serverless, Inc. Serverless Examples – A collection of ready-to-deploy Serverless Framework services. `https://github.com/serverless/examples`, April 2020.

[56] Sharath AV. AWS Security Flaw which can grant admin access! `https://medium.com/ymedialabs-innovation/an-aws-managed-policy-that-allowed-granting-root-admin-access-to-any-role-51b409ea7ff0`, May 2020.

[57] The Register. AWS won serverless – now all your software are kinda belong to them. `https://www.theregister.co.uk/2018/05/11/lambda_means_game_over_for_serverless/`, May 2018.

[58] ZDNet. Top cloud providers 2019. `https://www.zdnet.com/article/top-cloud-providers-2019-aws-microsoft-azure-google-cloud-ibm-makes-hybrid-move-salesforce-dominates-saas/`, August 2019.