Least-Privilege Calls to Amazon Web Services

Puneet Gill, Werner Dietl and Mahesh Tripunitara

Abstract—We address least-privilege in a particular context of public cloud computing: calls to Amazon Web Services (AWS) Application Programming Interfaces (APIs). AWS is, by far, the largest cloud provider, and therefore an important context in which to consider the fundamental security design principle of least-privilege, which states that a thread of execution should possess only those privileges it needs. There have been reports of over-privilege being a root cause of attacks against AWS cloud applications, and a least-privilege set for an API call is a necessary building-block in devising a least-privilege policy for a cloud application. We observe that accurate information on a least-privilege set for an invoker of a method to possess is simply not available for most such methods in AWS. We provide a meaningful characterization of least-privilege in this context. We then propose techniques to determine such sets, and discuss a black-box process we have devised and carried out to identify such sets for all 707 API methods we are able to invoke across five AWS services. We discuss a number of interesting discoveries we have made, some of which are surprising and some alarming, that we have reported to AWS. Our work has resulted in a database of least-privilege sets for API calls to AWS, which we make available publicly. Developers can consult our database when configuring security policies for their cloud applications, and we welcome contributors that augment our database. Also, we discuss example uses of our database via an assessment of two repositories and two full-fledged serverless applications that are available publicly and have policies published alongside. We observe that the vast majority of policies are over-privileged. Our work contributes constructively to securing cloud applications in the largest cloud provider.

Index Terms—Computer Security, Amazon Web Services, Least-Privilege.

1 Introduction

Infrastructure- and Platform-as-a-Service public cloud computing, with which a *developer*, who is a customer of the *cloud provider*, is able to deploy and run applications on the hardware and software computing resources of the cloud provider, has become a dominant paradigm over the past decade [1]. Amazon Web Services (AWS) [2] has been, by far, the largest provider of such services over the past few years [3]. AWS packages its offerings as services such as the EC2 compute service and the S3 storage service. Each of these services exposes an Application Programming Interface (API) that comprises a number of *methods*. Such a method can be *called* or *invoked*.

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. AWS is no exception. AWS perceives the security of a cloud application as a shared responsibility: "... 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..." [4].

As part of its responsibility for security, AWS provides constructs that the developer can use to specify their intended security policy. An example of such a construct is an *identity-based policy* [5]. An identity-based policy is attached to a user, role or group, and "controls what actions the identity can perform, on which resources, and under what conditions" [5]. Figure 1 is an example of such a policy. The policy authorizes a user, group or role to which the policy is attached the two actions `dynamodb:DeleteItem` and `dynamodb:DeleteTable` on the resource `arn:aws:dynamodb:region:accountId:table/myTable` and the action `s3:GetObject` on the resource `arn:aws:s3:::myBucket/*`. (An identity-based policy can also specify "Deny" for the **Effect** field; however, such syntactic details about policies are irrelevant to our work.)

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

Fig. 1. An AWS identity-based policy. It comprises a set of statements, in this example, of size two. The statement at the top authorizes a user, group or role to which the policy is attached the actions *dynamodb:DeleteItem* and *dynamodb:DeleteTable* to the *myTable* resource in the AWS DynamoDB service [6]. The other statement authorizes *s3:GetObject* to all objects within *myBucket* in the AWS S3 service [7]. The mnemonic "*" is a wildcard.

Motivation and Our Work: Misconfiguration of security policies in AWS that results in over-privilege is by now a well-known source of insecurity of cloud applications. For example, as CloudSploit [8] explains the so-called Capital One Hack [9] which lead to the breach of about 100 million pieces of private information, "at the root of the hack lies... misconfiguration [that] allowed an unauthorized user to elevate her privileges..." Another incident relates to a policy that is managed by AWS itself, which was over-

The authors are with the Department of Electrical and Computer Engineering, University of Waterloo, Canada.
E-mails: {p24gill,wdietl,tripunit}@uwaterloo.ca
Manuscript received April 19, 2005; revised August 26, 2015.

privileged [10]. We address the following more fundamental question that is necessary to preclude such misconfigurations: given a method $c()$ of an AWS service that a caller intends to invoke with arguments a, what is a least-privilege set for the caller to be able to successfully make the call $c(a)$? It is unsurprising to us that misconfigurations that result in over-privilege occur in practice when documentation from AWS is inadequate to answer this basic question. Worse still, there is documentation and examples from AWS and others of policies that are over-privileged (see Section 2 under "Over-privilege and insufficiency from documentation" and Section 6). A necessary building-block of a least-privilege policy for a cloud application is a least-privilege set for a call $c(a)$ that the application makes.

We have answered this fundamental question for almost every method in a recent version of AWS's Java API for five services: EC2, S3, Kinesis, DynamoDB and Elastic Transcoder, for a total of 707 methods. (We discuss in Section 5 why this is "almost every" and not "every.") We chose the five services based on how widely used we think they are, and so that we consider services with a large (e.g., EC2), medium (e.g., S3) and small (e.g., ElasticTranscoder) number of methods (see Table 1 in Section 5 for the exact numbers.) Our choice of the Java API is arbitrary: we could have adopted any of the APIs that AWS publishes; there is no difference, as it pertains to our work.

We have devised an approach to identifying such least-privilege sets. This has been a significant challenge (see Section 4). We have compiled our least-privilege sets into a publicly available database of such sets [11]. Our database is immediately useful to developers to devise least-privilege policies for their applications. And, it is our hope that others augment our database so the community as a whole benefits from improved security. Also, our methodology is effective, as demonstrated by our ability to identify such sets at scale, and can be used for more services and API methods, and even, possibly, for other clouds. However, as we identify in Section 4 and in our discussion of future work in Section 8, such an effort would benefit greatly from AWS publishing a test harness or other tools that enable more automation.

Notation: We adopt the following notation in the remainder of this paper. To denote a method or call, we write $c()$. We state in each case as to whether it is a method or call to which we refer. We typically use the mnemonic a for the arguments to a call, and denote such a call as $c(a)$. To refer to a method of a particular AWS service, we identify the AWS service, followed by a dot, followed by the name of the method, e.g., `s3.getObject()`. To refer to an action, we adopt the syntax AWS uses to refer to it, e.g., `dynamodb:DeleteItem`.

Remainder of the paper: In the next section, we discuss classes of a first set of particularly interesting observations we have made in the course of identifying least-privilege sets. We have reported these to AWS. The contents of that section complement those in Section 5 in which we make additional observations. In Sections 3 and 4, respectively, we propose definitions and approaches for least-privilege that we have adopted, and the methodology we have devised and carried out. As an illustration of the use of our database, in Section 6, we analyze two sets of publicly available repositories of cloud applications and identify gaps in their policies which our database helps rectify. We discuss related work in Section 7, and conclude with Section 8, wherein we discuss future work as well.

2 OBSERVATIONS 1

In the course of our work, we have made several interesting observations about specific least-privilege sets, classes of which we discuss in this section as our first set of observations. These complement the ones we make in Section 5.

Over-privilege and insufficiency from documentation: There are instances in which documentation from AWS recommends policies that are, simultaneously, over-privileged and insufficient. An example of this is with `ec2.createStoreImageTask()`. The documentation for the method points, in the context of privileges, to "Permissions for storing and restoring AMIs using S3 in the Amazon Elastic Compute Cloud User Guide" [12], which suggests a policy which authorizes 16 actions. However, we observe that a least-privilege set of size at most seven exists [11]. (We say "at most" because a least-privilege set of size six exists if a field in the argument is not set.) Furthermore, the recommended policy of 16 actions does not include actions that are necessary. For example, for `ec2.createStore-ImageTask()`, our work tells us that when a particular field, `S3ObjectTags`, in the argument is set, then the action `s3:PutObjectTagging` is necessary; however, that action is not part of the set of 16 actions in the recommended policy. Thus, not only is the policy over-privileged in that it authorizes redundant actions, it is insufficient as it does not authorize actions that are necessary.

Nonetheless, in some ways, the recommended policy in the documentation to which we refer above is helpful in that it tells a developer that actions from three different services are needed for a call to `ec2.createStoreImageTask()` to succeed. There are other instances in which actions from several different services are needed, and we are unable to find documentation that guides us. An example is `ec2.modifyVpnTunnelCertificate()`, which requires the following actions from across three different AWS services: `ec2:ModifyVpnTunnelCertificate`, `acm:RequestCertificate`, `acm-pca:GetCertificate` and `acm-pca:IssueCertificate`. Prior to our work, it is unclear how a developer is expected to arrive at this least-privilege set. We suggest that this can cause a developer to specify an over-privileged policy.

Over-privilege from requiring the wildcard: We have discovered instances in which either the resource or the action in an identity-based policy must be the wildcard "$\star$," for a call to succeed. This is somewhat alarming as from a security standpoint, use of the wildcard can be seen as dangerously permissive. An example of this is `ec2.exportImage()`. A least-privilege set of actions is the singleton set {`ec3:ExportImage`}. However, for the resource in an identity-based policy, it does not suffice that the policy mentions anything more specific than the wildcard "$\star$". A narrower specification of the resource, such as "`arn:aws:ec2:*::image/ami-013ad70ab73da1ff2`," does not suffice.

We made a similar observation, but with the actions rather than the resource, for the calls `kinesis.startStreamEncryption()` and `kinesis.stopStreamEncryption()`. Even if we enumerate all actions that start with `kinesis:` in an identity-based policy, that set of actions was not sufficient. However, if we specify the action as `kinesis:*`, i.e., with the wildcard, then the call succeeds. AWS has fixed this after our report of this observation.

Over-privilege from unrelated arguments: There exist instances in which changing seemingly unrelated arguments changes the least-privilege set in a manner that under-privileged calls can succeed. An example of this is with the call `s3.deleteObject()` which takes an argument of type `DeleteObjectRequest`. That argument, in turn, has two fields, `bypassGovernanceRetention` and `versionId`. The former is a boolean value which may be set to true or false, and the latter may either be set or not. We observe that when `bypassGovernanceRetention` is set to true and the `versionId` is set, the action `s3:BypassGovernanceRetention` is necessary for the call to succeed. This makes sense given that `bypassGovernanceRetention` is set to true. However, when `bypassGovernanceRetention` is set to true but `versionId` is not set, the action `s3:BypassGovernanceRetention` is no longer necessary. This suggests the potential that under-privileged callers, who do not possess `s3:BypassGovernanceRetention` can make a successful invocation even though the argument `bypassGovernanceRetention` is set to true.

Mismatch between method name and action name: It appears that a design philosophy of AWS is to give methods and actions the same name. For example, corresponding to the `s3.getObject()` method is an action `s3:GetObject`. And it is indeed the case, in several instances, that a least-privilege set for a call is the singleton set of the action that corresponds to the method. However, this is not universally true and in some cases, it is unclear what a least-privilege set is. For example, one of the `s3.getObject()` methods takes an argument of type `GetObjectRequest`, which has a field called `versionId`. This field is optional; it may either be set or not. We observe that when the field is not set, a least-privilege set for a successful call is {`s3:GetObject`}. When the `versionId` field is set, a least-privilege set is {`s3:GetObjectVersion`}. That is, providing `s3:GetObject` (as well) in the latter case would result in over-privilege. See Appendix A for details.

Over-privilege from coarse granularity of action: Related to the above issue of a mismatch of method and action names is that there are instances in which a caller must possess a particular action to successfully make a call; however the caller is then over-privileged in that they are able to make other calls that we may not want them to be able to make. An example of this is with `s3.createBucket()`. The argument of type `CreateBucketRequest` has a field called `acl` which may be set to one of the constants PRIVATE, PUBLIC_READ, PUBLIC_READ_WRITE or AUTHENTICATED_READ. If the caller seeks `acl` to be set to a value other than PRIVATE which is the default, the action `s3:PutBucketAcl` is necessary. However, this then is sufficient for the caller to arbitrarily change the value associated with an ACL via the call `s3.putBucketAcl()`. Thus, if we want to empower a user or role to create buckets whose `acl` is set to, say, AUTHENTICATED_READ, we are forced to empower that caller to also change the ACL setting for buckets in any way they want, e.g., to PUBLIC_READ.

Red herring empty least-privilege sets: For most methods, if a caller does not possess sufficient privileges, an exception is thrown when a call is made. However, there are methods for which this is not the case. Even if the caller is authorized to no actions whatsoever, the call appears to succeed in that no exception is thrown when the call is made. However, when one attempts to access the returned object locally in particular ways, a local exception is thrown. This is the case with every call that pertains to a so-called paginator across all the five services which we have investigated, e.g., `dynamodb.queryPaginator()`. Thus, if we adopt no exception being thrown as a successful call, the least-privilege set is the empty set for these calls. However, that is a red herring because the returned object then throws an exception for certain kinds of accesses to it. It is unclear in such cases how one is to characterize least-privilege, because some local accesses to the returned object succeed. Also, this is inconsistent with what appears to be AWS's design for most methods, which is to throw an exception if an under-privileged call is made.

Action name corresponding to method name exists but does not suffice: There are numerous instances in which there exists an action name that corresponds to the name of a method; however, not only is that action necessary, but also others for a call to succeed. A particularly egregious example of this is the method `dynamodb.restoreTableToPointInTime()`. Not only is `dynamodb:RestoreTableToPointInTime` required, but also 7 other actions such as `dynamodb:Scan` and `dynamodb:Query`. It is unclear how a developer is to know what a least-privilege set is. We suggest that this situation can lead to over-privilege by the developer, for example, use of wildcards to ensure that a legitimate caller can succeed.

Vendor Notification and Response

Over the course of the past year we communicated these observations we have made to AWS. On the observation with Kinesis we discuss under "Over-privilege from requiring the wildcard" above, AWS has, subsequent to our reporting the issue, fixed it.

3 Definitions and Approach

In this section, we characterize least-privilege in a way that is meaningful for our context, and associated approaches that we have devised and used to determine least-privilege sets for calls to AWS. In the next section, we discuss the methodology we have devised and followed in its entirety; the approaches we discuss in this section are part of the methodology.

Definition 1 (Privilege)**.** *A privilege is a pair, $\langle action, resource\rangle$.*

To us, our definition of a privilege is apparent from, for example, identity-based policies in AWS (see Figure 1 in Section 1 for an example). Moving on to least-privilege, the notion of least-privilege comprises *sufficiency* and *necessity*. That is, we deem a set of privileges to be least-privilege if no additional privilege is needed for access (sufficiency) and if any privilege from the set is removed, then access is not authorized (necessity). As "access" in our context is successful invocation of an API method, we need to ask what happens when we make an invocation and the caller is under-privileged. The answer, in most cases, is that an exception is thrown, and indeed, this appears to be a design intent of AWS. Consequently, we adopt the following definition.

Definition 2. *We say that a call $c(a)$ fails if an exception is thrown when we make it. Otherwise, we say that it succeeds.*

Next, we recognize that a call may fail owing to causes unrelated to authorization. For example, a resource to which a method $c()$ pertains may not have been allocated. Thus, in a characterization of least-privilege and our approach to determining a least-privilege set, we need to account for such failures that are unrelated to authorization. A related issue is whether we can associate a least-privilege set with a method $c()$ only, or whether we need to consider arguments a to $c()$ as well. The answer, from our observation for AWS is that least-privilege is characterized meaningfully on a pair $\langle c, a\rangle$, i.e., both the method $c()$ and arguments a to it, and not $c()$ only. Finally, we need to account for the possibility that a least-privilege set is not necessarily unique because we have no definitive evidence to that in AWS. The following definitions capture these discussions. Constraining our notion of success or failure of a call to authorization only is achieved by our definition for sufficiency, specifically, by phrasing it as an implication. The possible non-uniqueness of a least-privilege set is addressed by our definition for necessity.

Definition 3 (Sufficient set of privileges). *Given an API method $c()$ in an AWS service and arguments a for it, we say that a set of privileges P is sufficient for $\langle c, a\rangle$ if:*

$$\begin{gathered}(\textit{caller possesses all privileges} \implies c(a) \textit{ succeeds})\\ \implies \\ (\textit{caller possesses privileges } P \textit{ only} \implies c(a) \textit{ succeeds})\end{gathered}$$

The above definition addresses only those situations that the caller is able to successfully invoke $c(a)$ when the caller is in possession of all possible privileges. And in such a situation, we deem the set of privileges P to be sufficient if the possession of only P by the caller also would have resulted in a successful invocation $c(a)$. We intentionally do not address situations in which the caller is unable to invoke $c(a)$ successfully despite possessing all privileges; to us, no meaningful notion of "sufficient set of privileges" exists in such situations.

Definition 4 (Necessary set of privileges). *Given an API method $c()$ in an AWS service and arguments a for it, we say that a set of privileges P is necessary for $\langle c, a\rangle$ if: there exists a set Q of privileges such that $P \subseteq Q$, Q is sufficient for $c(a)$, and for all $p \in P$, $Q \setminus \{p\}$ is not sufficient.*

Definition 5 (Least-privilege set). *Given an API method $c()$ in an AWS service and arguments a for it, we say that a set of privileges P is least-privilege for $\langle c, a\rangle$ if P is sufficient and necessary for $\langle c, a\rangle$.*

In the remainder of this section, we discuss pieces of our methodology that pertain to our definitions. Our exposition is as a progression: (i) satisfying the precondition in Definition 3, (ii) identifying for which arguments to a method we determine least-privilege sets, and, (iii) given a method-arguments pair, how we determine a least-privilege set.

Precondition in Definition 3: valid arguments: The precondition in Definition 3 is, "an invocation $c(a)$ succeeds given that the caller possesses all privileges." Thus, a challenge for us is that a call $c(a)$ may fail for reasons that have nothing to do with whether the caller possesses sufficient privileges. That is, given $\langle c, a\rangle$ for which the call $c(a)$ can succeed, we need to be able to setup an environment in which the call indeed succeeds. Effecting a successful call is the only way for us to know whether a are indeed *valid* arguments for c, i.e., arguments for which the only way the call $c(a)$ fails is that the caller has insufficient privileges. For example, `s3.createBucket()` fails, i.e., throws an exception, in a manner unrelated to authorization if the bucket to which the arguments refer already exists in S3. The manner in which we address this issue of fulfilling the precondition in Definition 3 is to wrap a call $c(a)$ as follows.

```
$wrap_c(a)$
    $setupEnv_c()$
    $c(a)$
    $teardownEnv_c()$
```

where $setupEnv_c()$ is a setup procedure and $teardownEnv_c()$ is a teardown procedure. That is, the wrapper method $wrap_c()$ given arguments a first invokes the setup method that corresponds to the method $c()$, then invokes $c(a)$ and then invokes the teardown method. The property we seek from $setupEnv_c()$ is that if a call $c(a)$ can succeed, then $setupEnv_c()$ ensures that it does succeed. In Section 5 we present the total number of $setupEnv_c()$ and $teardownEnv_c()$ methods we need.

Our above procedure $wrap_c(a)$ is correct in that it is sufficient. Specifically, if an invocation $wrap_c(a)$ returns successfully, then we know: (i) that $setupEnv_c()$ correctly setup a sufficient environment for the subsequent $c(a)$ call to succeed, (ii) the call $c(a)$ succeeded, and (iii) the privileges we provided the caller are sufficient (see Definition 3).

We intentionally denote the method name "c" as a subscript for each of *wrap*, *setupEnv* and *teardownEnv* above, and not, for example, as an argument, because this reflects what we have actually done. That is, we have been able to associate each setup procedure with a method $c()$ independent of the arguments to an invocation of $c()$. This may seem like a limitation; however, the only possible negative impact is efficiency: we may do some redundant things within $setupEnv_c()$ from the standpoint of an invocation to $c()$ with particular arguments a. So long as each $setupEnv_c()$ guarantees that the subsequent invocation $c(a)$ succeeds, correctness is not impacted.

Space of all valid arguments: Given the notion of a valid argument from the discussions above, a next challenge for us is to identify all valid arguments a for a method $c()$; i.e., all arguments a to each method $c()$ for which there exists an environment in which the call $c(a)$ succeeds given that the caller has all privileges. Enumerating all candidate a does not scale. Therefore, we rephrase our challenge as follows. First denote as $L_{c(x)}$ and $L_{c(y)}$ respectively least-privilege sets for valid arguments x and y to $c()$. Now, suppose A_c is the set of all valid arguments a for $c()$. Consider partitions of A_c into equivalence classes, denote them $A_{c,1}, \ldots, A_{c,n}$ with the properties that (i) for every $i = 1, \ldots, n$, and every $x, y \in A_{c,i}$, it is the case that $L_{c(x)} = L_{c(y)}$, and, (ii) given any two distinct i, j, for every $x \in A_{c,i}$ and $y \in A_{c,j}$, it is the case that $L_{c(x)} \neq L_{c(y)}$. Our challenge, for each method $c()$, is to identify at least one member of every such equivalence class $A_{c,i}$ of its arguments. We would then have identified every possible least-privilege set for $c()$. For example, based on our methodology we have identified that $n = 1$ for `ec2.createImage()` and $n = 5$ for `s3.createBucket()` [11].

The manner in which we address this challenge is to perceive it as similar to the challenge encountered in software testing: with what inputs should a piece of software be run so the set of inputs comprehensively tests the software? In our case, what "comprehensive" means is different from what it customarily means in software testing. Also our success criterion, i.e., whether a test passes or not, is different from what is customary in software testing. Nonetheless, feedback directed random test generation [13], specifically as realized by Randoop [14], has been effective for us.

To be able to use feedback directed random test generation, we first need to identify a superset of possible argument values. To identify such a superset, we followed the category-partition method which was originally proposed for functional testing [15]. In the category-partition method, a human tester first identifies categories of inputs. Each category is then perceived as a partition of the input space, and a representative input from each partition is chosen for testing. The category-partition method is a good fit for us because we seek exactly to partition the space of all valid arguments. For us, each partition corresponds to a unique least-privilege set.

We first identify a set of all representatives for each argument in the arguments a to a method $c()$. For example, one of the `s3.createBucket()` methods takes as argument an object of type `CreateBucketRequest`. A `CreateBucketRequest` is composed of several fields. One of these fields is `aclAsString`. For an argument of type `CreateBucketRequest` to be valid for the method `s3.createBucket()`, the constituent field `aclAsString` must be one of five distinct valid values — the empty string, and four constant strings, and each of these values belongs to a partition of its own. Another field in `CreateBucketRequest` is `bucket`, which is the name of the bucket to be created, which is also of type `String`. There are constraints on this string for the call $c(a)$ to succeed: it must be at least three characters long and must not begin with a particular prefix. The remaining possible string values are all valid and belong to the same partition. There is also a field `objectLockEnabledForBucket` in `CreateBucketRequest`, which is a boolean, i.e., takes value either true or false, each of which is in a partition of its own. Thus, if `aclAsString`, `bucket` and `object-LockEnabledForBucket` were the only arguments to a call to `s3.createBucket()`, we would have $5 \times 1 \times 2 = 10$ possible combinations of arguments with which to invoke the method. It is this set of arguments which are provided as part of the configuration to Randoop, and Randoop then employs feedback directed random testing. In reality, `CreateBucketRequest` has several more fields and the total number of valid arguments we determine using the above approach is more than 8000. Randoop provides a way to choose from amongst those arguments, and an order in which the arguments are tried in invocations of the method.

A consequence of adopting the above approach is that our methodology is not necessarily complete; we do not necessarily identify every possible partition $A_{c,i}$ to which we refer above. That is, the database we have built which maps a pair $\langle c, a \rangle$ where $c()$ is a method and a is a valid argument for it does not necessarily contain an $a \in A_{c,i}$ for every partition $A_{c,i}$ for the method $c()$. This is the same situation that the software testing method that we have adopted encounters: we cannot guarantee that the set of inputs comprehensively tests the software.

Least-privilege set: Suppose we have chosen to identify a least-privilege set for $c(a)$, i.e., for an invocation of the method $c()$ with arguments a. How exactly do we go about identifying one? Our first-cut approach is expressed by the algorithm, `LP_linear` in Figure 2.

Apart from the method $c()$ and arguments a for it, `LP_Linear` takes as argument a set of privileges S. It invokes as sub-routine a method *failure*() that we construct. The method *failure*() returns true if the invocation $\mathit{wrap}_c(a)$ fails, and false if it succeeds, and we ensure that $\mathit{wrap}_c(a)$ fails if and only if the call $c(a)$ within it fails. (Recall, from the start of this section, that "fails" means "an exception is thrown.") That is, $\mathit{setupEnv}_c()$ and $\mathit{teardownEnv}_c()$ are guaranteed to not fail. If the set of privileges S is not sufficient for $c(a)$, then `LP_Linear` returns *error* in Line (2).

Otherwise, Corollary 1 below, which relies on Theorem 1, establishes that the returned `Result` is necessary. We have also checked for every returned `Result` for each pair $\langle c, a \rangle$ for which we have generated such a set, that that set is sufficient. We have done this by invoking $\mathit{wrap}_c(a)$ with the returned `Result` as the set of privileges the caller has and observing that it succeeds. Corollary 2 below establishes that this means that each sufficient set S with which we seeded our runs of `LP_Linear` has only one subset that is least-privilege, and this subset is the returned `Result`. Thus, we have not identified any method in AWS which has more than one least-privilege set. However, our work does not preclude this possibility. We point out also that we have not identified two actions whose names are different, but are the same from the standpoint of privilege. By Corollary 2, `LP_Linear` would identify such actions if it is provided as argument an S that includes both privileges that correspond to those actions by returning a `Result` that is not sufficient.

Theorem 1. *Suppose S is a sufficient set of privileges, and suppose $L_1, \ldots, L_n$ are all the distinct subsets of S each of which is least-privilege for $\langle c, a \rangle$. Then,* `Result` *that* `LP_Linear`(c, a, S) *returns is $L_1 \cap \ldots \cap L_n$.*

Proof. Suppose some privilege $p \in L_1 \cap \ldots \cap L_n$. Then $S \setminus \{p\}$ is not sufficient — if it is, there exists some least-privilege set, denote it $L_{n+1} \subseteq S \setminus \{p\}$, that is distinct from each of $L_1, \ldots, L_n$ because p is in all of $L_1, \ldots, L_n$. This is a contradiction to the assumption that $L_1, \ldots, L_n$ are all

```
LP_Linear($c, a, S$)
1  Assign the caller the privileges in $S$
2  **if** *failure*($wrap_c(a)$) **then return** *error*
3  Result ← $\emptyset$
4  **foreach** $p \in S$ **do**
5      Assign the caller the privileges in $S \setminus \{p\}$
6      **if** *failure*($wrap_c(a)$) **then**
7          Result ← Result $\cup \{p\}$
8  **return** Result
```

```
LP_Binary($c, a, S$)
1  **if** $|S| < 2$ **then return** LP_Linear($c, a, S$)
2  $S_1$ ← half the members of $S$
3  $S_2 \leftarrow S \setminus S_1$
4  Assign the caller the privileges in $S_1$
5  **if** *failure*($wrap_c(a)$) **then**
6      Assign the caller the privileges in $S_2$
7      **if** *failure*($wrap_c(a)$) **then**
8          **return** LP_Linear($c, a, S$)
9      **else return** LP_Binary($c, a, S_2$)
10 **else return** LP_Binary($c, a, S_1$)
```

Fig. 2. Our algorithms for determining least-privilege sets.

such distinct sets. Thus, $p \in$ `Result` that is returned by the algorithm because for that choice of p in the **foreach** loop of Line (4), the **if** condition of Line (6) must evaluate to true, and we add p to `Result` in that iteration of the **foreach** loop in Line (7). Now suppose some $q \notin L_1 \cap \ldots \cap L_n$. To prove that $q \notin$ `Result` that is returned, consider the iteration of the **foreach** loop of Line (4) in which the privilege q is chosen from S. We know that $S \setminus \{q\}$ is sufficient because there exists $i \in \{1, \ldots, n\}$ such that $L_i \subseteq S \setminus \{q\}$. Thus, the **if** condition of Line (6) evaluates to false for this iteration, and we do not add q to `Result` in this iteration of the **foreach** loop, and therefore, as that is the only moment we could possibly add q to `Result`, we never add q to `Result`. □

Corollary 1. *If S is sufficient for $\langle c, a\rangle$, then* `Result` *that* `LP_Linear`(c, a, S) *returns is necessary.*

Proof. The returned `Result` is $L_1 \cap \ldots \cap L_n$ by Theorem 1, $L_1 \cap \ldots \cap L_n \subseteq L_1$, L_1 is necessary because it is least-privilege, and every subset of a necessary set is necessary. □

Corollary 2. *Suppose S is sufficient for $\langle c, a\rangle$. Then, the returned* `Result` *from* `LP_Linear`(c, a, S) *is sufficient if and only if there is only one subset of S that is least-privilege.*

Proof. For the "if" direction, we assume that S has only one subset that is least-privilege. Then, in the statement of Theorem 1, $n = 1$, and the returned `Result` $= L_1$ which is sufficient because L_1 is least-privilege. For the "only if" direction, suppose the returned `Result`, denoted R, is sufficient. By Corollary 1, it is also necessary and therefore R is least-privilege. Suppose $T \subseteq S$ and $T \neq R$ such that T is also least-privilege. Now, we have two possibilities only. (i) $T \supset R$ — this cannot be true because T is least-privilege and therefore necessary, and no strict superset of a sufficient set, R in this case, is necessary. (ii) $T \not\supseteq R$ and $T \neq R$ — then, by Theorem 1, the returned `Result` $\subseteq R \cap T \subset R$. This contradicts the assumption that R is the returned `Result` because no strict subset of a set is the set itself. Therefore, no such T exists. □

We have further improved our efficiency of converging to a least-privilege set in practice by adopting the algorithm `LP_Binary` in Figure 2, which, as its name suggests, realizes a kind of binary search. We show an example of a run of `LP_Binary` for the method `s3.copyObject()` for a particular starting S in Figure 3. In the next section, we clarify other pieces of our methodology that are needed to generate the corresponding policy, e.g., the manner in which we identify the resources.

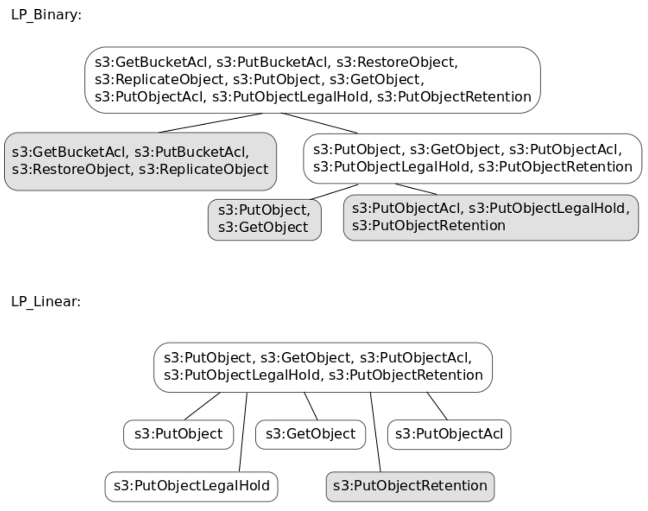
LP_Binary:
s3:GetBucketAcl, s3:PutBucketAcl, s3:RestoreObject, s3:ReplicateObject, s3:PutObject, s3:GetObject, s3:PutObjectAcl, s3:PutObjectLegalHold, s3:PutObjectRetention
s3:GetBucketAcl, s3:PutBucketAcl, s3:RestoreObject, s3:ReplicateObject
s3:PutObject, s3:GetObject, s3:PutObjectAcl, s3:PutObjectLegalHold, s3:PutObjectRetention
s3:PutObject, s3:GetObject
s3:PutObjectAcl, s3:PutObjectLegalHold, s3:PutObjectRetention
LP_Linear:
s3:PutObject, s3:GetObject, s3:PutObjectAcl, s3:PutObjectLegalHold, s3:PutObjectRetention
s3:PutObject
s3:GetObject
s3:PutObjectAcl
s3:PutObjectLegalHold
s3:PutObjectRetention

Fig. 3. An example run of `LP_Binary`, and a consequent run of `LP_Linear` for `s3.copyObject()`. Under `LP_Binary` at the top, an unshaded box is a sufficient set of actions; a shaded box is a set that is not sufficient. For `LP_Linear`, an unshaded box is a privilege we include in `Result`; a shaded box, we do not.

We point out that `LP_Binary` is not quite the same, for example, as binary search for an item in a sorted array. Because in an invocation of `LP_Binary`, both the checks in Lines (5) and (7) may fail, i.e., evaluate to true, in which case we fall back to `LP_Linear` in Line (8). Also, we point out that if `LP_Binary` is invoked with S a least-privilege set, then we are guaranteed to invoke `LP_Linear` either in Line (1) or Line (8). That is, `LP_Binary` relies on `LP_Linear` to confirm that it has indeed converged to a least-privilege set. In the worst case, `LP_Binary` is no more efficient than `LP_Linear`. However spot checks tell us that it has helped us converge faster in practice.

4 Methodology

In this section, we discuss the methdology we have devised and carried out; Figure 4 presents it. We have attempted to automate as many steps as we are able. However, some of our steps do require manual intervention; these are shown unshaded in Figure 4. A side-effect of our work is our observation that AWS really needs to make available a software harness so one can fully automate such a methodology. We now proceed to describing the steps in our methodology.

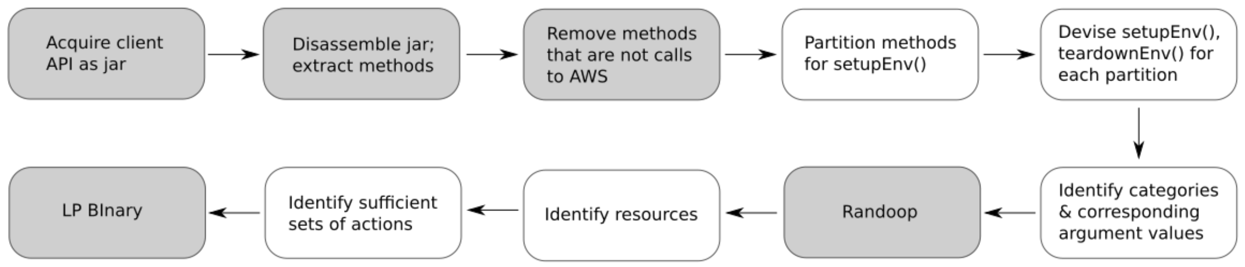
Acquire client API as jar
Disassemble jar; extract methods
Remove methods that are not calls to AWS
Partition methods for setupEnv()
Devise setupEnv(), teardownEnv() for each partition
Identify categories & corresponding argument values
Randoop
Identify resources
Identify sufficient sets of actions
LP BInary

Fig. 4. The steps in our methodology for identifying least-privilege sets. Steps that require manual intervention are unshaded.

Acquire client API as jar: Our first step is to acquire the client API for an AWS service. For example, for S3, this is AWS Java SDK :: Services :: Amazon S3 [16]. The version of each API we have adopted is 2.16.60, which was the latest version that was available at the time we were ready with our process. We downloaded the client API as a jar from the Maven Central Repository [17] using the Gradle build automation tool [18].

Disassemble jar; extract methods: We then disassembled the jar file in the IntelliJ integrated development environment [19] to access the method declarations.

Remove methods that are not calls to AWS: Not all methods in a client API are calls to AWS and consequently, the notion of a least-privilege set is not meaningful for those methods. An example is a constructor method for the client API class. We employed heuristics to first arrive at a superset of methods we would remove, and then inspected this superset for methods we should retain. For example, part of the heuristic is to identify static methods; for example for the S3 client, these are `create()`, `builder()` and `serviceMetadata()`. None is a call to AWS and therefore all were removed.

Partition methods for *setupEnv()*: As we discuss under "Precondition in Definition 3: valid arguments" in the previous section, for each method $c()$, we need to identify what arguments a are valid, i.e., for what a there exists an environment in which $c(a)$ succeeds if the caller has all privileges. As we say there, our approach was to first partition the methods based on whether each method in a partition can have the same *setupEnv()* procedure. There is a trade-off in placing two methods in the same partition. The larger a partition, the fewer the total number of partitions, but the more likely we have redundant setup being performed in $\mathit{setupEnv}_c()$ as it pertains to a method $c()$, which impacts the time it takes to converge to least-privilege sets for $c()$.

Devise *setupEnv()*, *teardownEnv()* for each partition: Once we partitioned the methods as we discuss above, we devised the $\mathit{setupEnv}()$ and $\mathit{teardownEnv}()$ methods that are common to every method in a partition. For example, we use the same $\mathit{setupEnv}()$ and $\mathit{teardownEnv}()$ methods that we devised for both `dynamodb.deleteBackup()` and `dynamodb.updateContinuousBackups()`. Similarly, for both `ec2.describeInstances()` and `ec2.describe-VpcEndpoints()`, we use the same $\mathit{setupEnv}()$ and $\mathit{teardownEnv}()$ methods, which happen to be empty.

Identify categories and corresponding argument values: Once we had the wrapper methods for each API method, we moved on to the arguments for each method. As we discuss under "Space of all valid arguments" in the previous section, for each method, we went through one argument at a time, and identified a set of values for that argument such that all categories of the argument are covered. From the method declarations and argument types to each method that we had extracted in the first two steps, we automatically drilled down to basic Java object types such as `String` and `Integer`. Our output from this part of the process was a hierarchical JSON file where a parent node is an opaque type, and a child node is either a basic Java type or one step lower in our drilling down towards a basic Java type. We had to pay special attention to composite data types such as `List`. Typically, an instance of type `List` is further parameterized by the type of object the list contains, which may be of a type that is defined by AWS. We would extract and drill down on that parameterized type.

Once we had such a JSON file with a hierarchy which bottoms-out in basic Java types, we then automatically created code-snippets to associate values with objects. For example, for an object of type `String`, we would associate the Java constant `null`, an empty string `" "` and a non-empty string. These values are a starting point in establishing categories for each argument and choosing a value for each category for that argument. As we mention as an example under "Space of all valid arguments" in the previous section, for the `aclAsString` argument to `s3.createBucket()`, we arrived at five different categories and therefore five different values. There are two properties we need to keep in mind: whether an argument value would result in an erroneous call, and whether we had addressed all categories. For example, for `aclAsString` in `s3.createBucket()`, a value of `null` results in a call that fails. Therefore, it is not a candidate argument value for the next step. To determine whether a candidate value would result in a failed call, we would invoke the method.

Randoop: Once we have a superset of values for arguments for each method, we then use Randoop [14] to employ feedback generated random testing to cull the set of arguments. For example, the set of more than 8000 arguments for `s3.createBucket()` was culled to a set of size 31 only. We configured Randoop, and wrote post-processed its output. The output of this step is a set of arguments A_c for each method $c()$. Our focus in the next step was then narrowed to pairs $\langle c, a\rangle$ for every $c()$ and every $a \in A_c$.

Identify resources: Our next step was to identify resources that would appear in an identity-based policy we would include in our database. That is, this step identifies all resources that must appear as the second component in an $\langle \text{action}, \text{resource} \rangle$ pair that is a privilege. Once we had the output pairs of methods $c()$ and arguments a, we identified a superset of resources that would be associated with any privileges. This requires an understanding of the semantics of $c(a)$, i.e., what the method $c()$ does when invoked with arguments a, and trial-and-error. While carrying out the trial-and-error, we would grant a caller all actions to every resource. In our database, a resource appears as the second component of a least-privilege set only if at least one action needs to be authorized to it.

Identify sufficient sets of actions: For each pair $\langle c, a \rangle$, we then identified a sufficient set of actions $S_{c,a}$. The intent was for this $S_{c,a}$ to be the third argument to a call to `LP_Binary` to then identify a least-privilege set for $c(a)$. As we discuss under "Least-privilege set" in the previous section, we sought two properties for $S_{c,a}$: it needs to be sufficient, and it should have only one subset that is least-privilege. This required checking whether there exists an action with the same or similar name as $c()$, consulting documentation, various user manuals and technical posts on the Internet. Engaging in this step of the process, in particular, demonstrated to us the severe lack of quality documentation with regards to even sufficient sets of actions for $c(a)$, let alone least-privilege sets. We could have simply choosen large sets of actions for $S_{c,a}$; for example, we could have chosen all possible actions. However, then, the cardinality of $S_{c,a}$ would have been more than 1000. Indeed, even if we had considered only actions that correspond to the five AWS services we have considered, the number of actions is more than 600 (see Appendix A for details). This would have resulted in each invocation of `LP_Binary` taking a prohibitively long time — a call to AWS can, in some cases, take minutes.

Another issue is that considering only the actions from the five services would not have identified necessary actions in all cases, as there exist methods for which an action from a service that is not one of the five we considered is necessary, e.g., an action with prefix "`iam:`". To mitigate this, we identified a sufficient set $S_{c,a}$ for each $\langle c, a \rangle$ and grouped pairs of $\langle c, a \rangle$ together based on the presumed semantics of $c()$ such that a single S set was sufficient for each of them.

`LP_Binary`: Our final step was to run `LP_Binary` from the previous section to identify least-privilege sets with arguments $\langle c, a, S_{c,a} \rangle$ as we discuss in the previous section. Part of this process is a step that is not part of `LP_Binary` as we present it in the previous section; this was to check that the returned `Result` from a run of `LP_Binary` was indeed sufficient. As we mention in the previous section, in every instance, the answer to this question was "yes." That is, we did not identify any method for which there is more than one least-privilege set. The outputs of this last step comprise our database [11].

5 Observations 2

In this section, we present observations in addition to the ones we discuss in Section 2. We remind the reader that in AWS parlance, which we adopt, an action is a privilege or permission. A caller needs to be authorized to certain actions to be able to successfully invoke a method. Each action and method belongs to an AWS service. The name of an action identifies the service to which it belongs, e.g., "`ec2:...`" and "`s3:...`."

	# setupEnv() and teardownEnv() methods	**# methods after third step of methodology**	**# methods ≥ 1 least-privilege set identified**
DynamoDB	18	56	56
EC2	275	547	507
Elastic Transcoder	17	20	20
Kinesis	17	28	28
S3	22	97	96
Total	349	748	707

TABLE 1
The number of methods in each service that remained after the third step, "Remove methods that are not calls to AWS" in our methodology, and the number for which we identified at least one least-privilege set.

	# distinct actions in least-privilege sets	**# distinct actions of service in least-privilege sets**	**# distinct actions of service in database**
DynamoDB	50	48	48
EC2	426	408	408
Elastic Transcoder	16	16	16
Kinesis	27	27	27
S3	70	69	69

TABLE 2
Actions in least-privilege sets. The column "**# distinct actions in least-privilege sets**" is the number of actions of any service that appear in a least-privilege set of a method of a particular service. The next column is the number of those actions that are of the service itself. The last column is the number of distinct actions of a service that appear anywhere in our entire database of least-privilege sets.

Table 1 reports the number of *setupEnv*(), *teardownEnv*() pairs we created for each service, the number of API methods for each service that we were left with after the third step, "Remove methods that are not calls to AWS," in our methodology from the previous section, and the number for which we have identified at least one least-privilege set. As the table shows, for one method in S3 and 40 methods in EC2, we were unable to identify any least-privilege set. The broad reason is that we were unable to effect a successful invocation to any of those methods. The specific reason differs by method. In some cases, the cost is too prohibitive for us, e.g., API methods that require AWS outpost [20]. In some other cases, the method pertains to the so-called EC2-classic platform, which is not available to us as our AWS accounts were created later than 2013 [21]. In yet other cases, notwithstanding our best efforts to setup an adequate environment to effect a successful call, we were unable to, and the error messages were unhelpful.

Actions in least-privilege sets: Table 2 reports information on actions that appear in the least-privilege sets we have identified [11]. The column "**# distinct actions in least-privilege sets**" is the total number of distinct actions that appear in least-privilege sets for methods in a particular service. For example, for methods in EC2, the total number of distinct actions across all least-privilege sets we have identified is 426. The next column, "**# distinct actions of**

	# least-privilege sets by cardinality								
	0	1	2	3	4	5	6	7	8
DynamoDB	1	44	2	0	0	0	0	0	2
EC2	1	386	36	7	2	1	2	1	0
Elastic Transcoder	1	16	0	0	0	0	0	0	0
Kinesis	1	27	0	0	0	0	0	0	0
S3	1	68	5	3	4	2	0	0	0

TABLE 3
The cardinalities of distinct least-privilege sets we identified for each service. For example, DynamoDB has two distinct least-privilege sets of cardinality 8 each across all its method-argument pairs that appear in our database.

	# methods by # distinct least-privilege sets			
	1	2	3	4
DynamoDB	52	0	1	3
EC2	479	28	0	0
Elastic Transcoder	19	1	0	0
Kinesis	28	0	0	0
S3	83	9	2	2

TABLE 4
The number of methods in each service with particular numbers of distinct least-privilege sets. So, for example, S3 has 9 methods each of which has two different least-privilege sets. This is the same as the number, n, of equivalence classes to which we refer in Section 3.

service in least-privilege sets," is the number of actions that are of the service itself; e.g., the number of actions with prefix "`ec2:`" that appear in any least-privilege set for methods in EC2 is 408. The last column, "**# distinct actions of service in database**," are actions of a service, e.g., actions with prefix "`ec2:`", that appear in any least-privilege set across all five services in the database. For example, if an action with prefix "`s3:`" appears in a least-privilege set of a method "`ec2...`", this would be counted in this last column, but not in the column immediately prior.

We observe, in Table 2, that for all five services we have examined, the values in the last column are identical to the immediately prior column. This means that an action appears in a least-privilege set in our database only if it appears in a least-privilege set of a method of the same service to which the action belongs. For example, if an action with prefix "`ec2:`" appears in a least-privilege set for a method in S3, then that action appears also in some least-privilege set for a method in EC2. This is unsurprising. We observe also that the overwhelming number of actions that appear in least-privilege sets we have identified for a service are that of a service itself. The biggest gap is with EC2 in which $426-408 = 18$ actions that do not have prefix "`ec2:`" appear in least privilege sets for `ec2....()` methods. Also, actions from seven AWS services other than the five whose methods we have examined appear in our database, e.g., `iam:`, `tiros:` and `acm:`. Yet another observation is that for none of the five services we have examined do all actions of that service appear in the least-privilege sets we have determined for methods of that service. (See Appendix A for details.) This suggests a possibility that there are actions that appear in no least-privilege set. We leave further investigation of this for future work.

Number and cardinalities of least-privilege sets: Table 3 shows, for each of the five services we consider, the number of distinct least-privilege sets with a particular cardinality. Table 4 shows the number of API methods in each service that have a particular number of distinct least-privilege sets. The latter number, i.e., the number that is the heading of each of the last four columns of Table 4, is the number of equivalence classes, n, to which we refer under "Space of all valid arguments" in Section 3.

We observe from Table 3 that every one of the five services has a method with a "red herring empty least-privilege set" as we call it in Section 2. Also, in overwhelmingly many cases, a least-privilege set is a set of size one. This appears to validate the mindset that indeed, a design intent of AWS is to have one action per method as its least-privilege set. However, as Table 3 shows, this is not always the case. We have methods with a somewhat large cardinality for the least-privilege set, e.g., two in DynamoDB each of size eight, and several methods for which a least-privilege set has cardinality > 1, e.g., in S3.

Table 4 suggests that in overwhelmingly many cases, a least-privilege set is a function of the method only, and not arguments to it. This is good news for a developer who seeks to devise a least-privilege policy for their cloud application. However, as the table shows, this is not always the case. Our database informs a developer as to which methods these are.

Repo	**# apps**	**# apps with policy**	**# least-privilege policies**
AWS	15	15	0
Serverless, Inc.	68	21	9

TABLE 5
Data on two publicly available repositories. The "**# apps**" is the number of applications in the repository that exercise at least one of the services we have considered: EC2, S3, DynamoDB, Kinesis and Elastic Transcoder. We analyze a policy against the least-privilege sets we have determined only.

6 Example Uses of Our Database

In this section, we discuss work we have carried out in demonstration of the utility of our database of least-privilege sets. In Section 6.1 we discuss two repositories, each of which contains several snippets of serverless applications. In Section 6.2, we discuss two full-fledged serverless applications. All of these have identity-based policies that have been published alongside, against which we are able to compare policies that would have been devised had our database of least-privilege sets been available.

6.1 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) Serverless, Inc. [23]. Table 5 presents our quantitative observations.

AWS repository: This is a repository from AWS that has been made available for developers to adopt 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. Customer-managed policies, which is our focus, are published as so-called policy templates, which allow for placeholder values for resources. AWS publishes such policy templates [24].

As Table 5 indicates, there are 15 applications in the repository, with policies alongside, that invoke a method from at least one of the five AWS services we have considered. We find that none of the policies adheres to least-privilege. An example of over-privilege from the repository is for the Microservice HTTP Endpoint application [25]. As the `template.yaml` file there indicates, it adopts the `DynamoDBCrudPolicy` that is part of the policy templates published by AWS [24]. For the application, that policy is over-privileged. The application needs four actions only. `DynamoDBCrudPolicy` grants a number of additional, redundant privileges.

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, all of which exercise one of the five services that we have considered. As Table 5 indicates, 21 of the 68 applications have a policy published alongside and 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. We find only nine of the 21 to be least-privilege.

Both least-privilege and over-privileged policies in this repository are less straightforward than the ones in the AWS repository. For example, policies with more than one action are amongst those that are least-privilege; for example, an application that needs two actions on the same resource [26]. Amongst the policies that are over-privileged, we have some whose cause is the use of wildcards [27]. We also have policies with redundant actions, for example, one that grants six actions, when only five are needed [28].

6.2 Two Applications

We have looked also at two full-fledged applications for which identity-based policies have been published alongside, which we discuss in this section. The two applications are for a Bookstore [29] and for a retail outlet [30].

The Bookstore application: The Bookstore application [29] 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 privileges 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.

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 action that begins with "`s3:Get`" and "`s3:List`," which is redundant. Indeed, none of the nine Lambda functions exercises S3; therefore any action from S3 that is awarded is redundant. Actions from DynamoDB are excessive as well.

The Hello, Retail! application: Hello, Retail! is an application for a retail store [30]. It has won a serverless architecture award [31]. We analyzed 13 of the Lambda functions of Hello, Retail! We found only a somewhat subtle problem with over-privilege that we discuss below. Every policy in Hello, Retail! that we examined appears to be "hand-crafted" carefully, and not, for example, blindly copied from other sources. We observe that 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. The issue is that the `kinesis:DescribeStream` and `kinesis:-ListStreams` actions are needed only when a Lambda function is registered to be the callback when a Kinesis event happens. Those actions are not needed subsequently.

Summary We have found that the vast majority of identity-based policies that we have examined and discussed in this section suffer from over-privilege. We caution developers from blindly adopting those policies in their applications.

7 Related Work

Our work deals with least-privilege, a security design principle that, to our knowledge, was first articulated by Saltzer and Schroeder [32]. That work discusses a number of other principles as well, that 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 section, 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. [33]. Our objectives are similar to theirs, but that work is for Android. 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 [34] so a piece of code in an app can clearly call out the permissions it requires.

Another piece of work that is related to ours and focuses on AWS is of Shimizu et al. [35]. The objective of their work is to find least-privilege policies for Infrastructure as Code (IaC) templates, specifically AWS CloudFormations, which are used to provision resources in a cloud environment. Our work differs from theirs in the sense that we are finding a minimum set of privileges for an API method, a building-block for a cloud application, and involves the role of arguments for an API method changing the least-privilege set for that API method. The work of Sanders et al. [36] focuses on minimizing privilege errors in creating Attribute Based Access Control (ABAC) policies for a system by mining through audit logs. They view least-privilege as a balance between minimizing under-privilege and over-privilege assignment errors. This is different from the manner in which we characterize least-privilege. Furthermore, our work does not deal with specific classes of policies such as ABAC policies, but rather considers privileges that are needed to invoke an API method in AWS.

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. [37] 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. As such, it is different from our work as we determine least-privilege sets for API calls to AWS.

Then, there is work from AWS itself. There is work on security best practices for AWS [38], the work of Cook [39] that discusses the use of formal methods for security within AWS, and the work of Backes et al. [40] that discusses an automated approach to reasoning about AWS access policies. These endeavours underlie AWS services that provide security assessments: AWS Config [41], Amazon Macie [42], AWS Trusted Advisor [43], Amazon GuardDuty [44] and AWS IAM Access Analyzer [5]. 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 [38], 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. [40] 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 is needed for least-privilege in an application, but as to whether and how, is a missing link. Furthermore, the mapping of an API method to privileges, the focus of our work, does not seem to be information that is available readily. 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 [5] seems related most closely to our work. It is based on the work of Backes et al. [40], 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 policies, particularly resource-based policies, 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 a policy is not least-privilege, but not solving the problem of determining least-privilege.

8 Conclusions

We have addressed least-privilege for calls to AWS. We have observed that identifying least-privilege sets is not easy and necessitates a clear characterization of what we mean by a least-privilege set, which we provide, and a somewhat painstaking black-box experimental process, which we have devised and carried out. We have identified least privilege sets for 707 methods across five different AWS services; 46 of these methods are associated with more than one least-privilege set because the least-privilege set for those methods changes with their arguments. We have reported 27 observations, which we have discovered through our work, to AWS via their vulnerability reporting program over the course of the past year, and discussed classes of those observations in this work. Some of the observations are alarming to us, e.g., documentation from AWS that suggests a policy that is simultaneously over-privileged and insufficient for particular methods, and methods which seem to require the specification of the wildcard for the resource or actions in a policy. Some others suggest inconsistent design that can impede security policy configuration, e.g., "red herring" empty least-privilege sets. We have studied also two repositories of Lambda functions, and two full-fledged applications, all publicly available, and found that over-privilege is pervasive in them. We have made our database of least-privilege policies available publicly [11]. Developers can immediately use our database to devise least-privilege policies for their cloud applications, and we hope that the broader community contributes to the database.

There is tremendous scope for future work. Our work involved steps that were heavily manual (see Figure 4 in Section 4). It would be meaningful to automate the manual steps as much as possible because AWS itself has several more than the five services we have considered, and we would want to repeat this work for those services, and for other cloud providers such as Google Cloud and Microsoft Azure. There are also a number of specific questions our work has thrown up which we have not answered; for example, (i) are there actions that are members of no least-privilege set?, and, (ii) do there exist actions that are distinct in that they have different names, but are identical from the standpoint of privilege? There is also the question of whether mechanisms can be conceived and created to automatically use a database such as the one we have created with least-privilege sets to secure an entire cloud application. For example, could we introduce new types and associated rules so we are able to automatically identify a least-privilege policy for code that resides in a cloud application?

References

[1] Amazon Web Services (AWS), "Types of cloud computing," https://aws.amazon.com/types-of-cloud-computing/, Apr. 2020.
[2] ——, "https://aws.amazon.com/," Apr. 2020.
[3] 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/, Aug. 2019.
[4] Amazon Web Services (AWS), "Security Overview of AWS Lambda," https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf, Apr. 2019.

[5] ——, "AWS Identity and Access Management – User Guide," https://docs.aws.amazon.com/IAM/latest/UserGuide/, Apr. 2020.

[6] ——, "Amazon DynamoDB – Fast and flexible NoSQL database service for any scale," https://aws.amazon.com/dynamodb/, Apr. 2020.

[7] ——, "Amazon S3 – Object storage built to store and retrieve any amount of data from anywhere," https://aws.amazon.com/s3/, Apr. 2020.

[8] CloudSploit, "A Technical Analysis of the Capital One Hack," https://blog.cloudsploit.com/a-technical-analysis-of-the-capital-one-hack-a9b43d7c8aea, May 2020.

[9] 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.

[10] 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.

[11] P. Gill, W. Dietl, and M. Tripunitara, "Database of least-privilege policies for AWS," https://github.com/puneetgill05/AWSDatabase, Apr. 2022.

[12] Amazon Web Services (AWS), "Store and restore an AMI using S3," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ami-store-restore.html#ami-s3-permissions, Aug. 2021.

[13] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 75–84.

[14] Randoop, "Randoop: Automatic unit test generation for Java," https://randoop.github.io/randoop/, Aug. 2021.

[15] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988. [Online]. Available: https://doi.org/10.1145/62959.62964

[16] Amazon Web Services (AWS), "Interface S3Client," https://sdk.amazonaws.com/java/api/latest/software/amazon/awssdk/services/s3/S3Client.html, 2021.

[17] ——, "AWS Java SDK :: Services :: Amazon S3 2.16.60," https://mvnrepository.com/artifact/software.amazon.awssdk/s3/2.16.60, 2021.

[18] Gradle Inc., "Gradle Build Tool," https://gradle.org/, 2021.

[19] JetBrains, "IntelliJ IDEA," https://www.jetbrains.com/idea/, 2021.

[20] Amazon Web Services (AWS), "Working with local gateways," https://docs.aws.amazon.com/outposts/latest/userguide/outposts-local-gateways.html, Aug. 2021.

[21] ——, "EC2-Classic," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-classic-platform.html, Aug. 2021.

[22] ——, "AWS Serverless Application Repository Examples," https://github.com/aws-samples/serverless-app-examples, Apr. 2020.

[23] Serverless, Inc., "Serverless Examples – A collection of ready-to-deploy Serverless Framework services," https://github.com/serverless/examples, Apr. 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] ——, "Microservice HTTP Endpoint," https://github.com/aws-samples/serverless-app-examples/tree/master/javascript/microservice-http-endpoint, Oct. 2019.

[26] Serverless, "FFmpeg app," https://github.com/serverless/examples/tree/master/aws-ffmpeg-layer, Feb. 2020.

[27] ——, "Receive an email, store in S3 bucket, trigger a lambda function," https://github.com/serverless/examples/tree/master/aws-node-ses-receive-email-body, Oct. 2018.

[28] ——, "Serverless REST API," https://github.com/serverless/examples/tree/master/aws-node-rest-api-with-dynamodb, Aug. 2019.

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

[30] Nordstrom, Inc., "Hello, Retail!" https://github.com/kalevalp/hello-retail-baseline, Sep. 2018.

[31] J. McKim, "Announcing the Winners of the Inaugural ServerlessConf Architecture Competition," https://read.acloud.guru/announcing-the-winners-of-the-inaugural-serverlessconf-architecture-competition-1dce2db6da3, May 2017.

[32] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Communications of the ACM*, vol. 17, no. 7, Jul. 1974.

[33] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 627–638. [Online]. Available: https://doi.org/10.1145/2046707.2046779

[34] Android developer guides, "RequiresPermission," https://developer.android.com/reference/androidx/annotation/RequiresPermission, Dec. 2019.

[35] R. Shimizu and H. Kanuka, "Test-based least privilege discovery on cloud infrastructure as code," in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2020, pp. 1–8.

[36] M. W. Sanders and C. Yue, "Mining least privilege attribute based access control policies," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 404–416. [Online]. Available: https://doi.org/10.1145/3359789.3359805

[37] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A Security Analysis of Amazon's Elastic Compute Cloud Service," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 1427–1434. [Online]. Available: https://doi-org.proxy.lib.uwaterloo.ca/10.1145/2245276.2232005

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

[39] B. Cook, "Formal Reasoning About the Security of Amazon Web Services," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 38–47.

[40] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Rungta, O. Tkachuk, and C. Varming, "Semantic-based automated reasoning for AWS access policies using SMT," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–9. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8602994

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

[42] ——, "Amazon Macie – A machine learning-powered security service to discover, classify, and protect sensitive data," https://aws.amazon.com/macie/, Apr. 2020.

[43] ——, "AWS Trusted Advisor – Reduce Costs, Increase Performance, and Improve Security," https://aws.amazon.com/premiumsupport/technology/trusted-advisor/, Apr. 2020.

[44] ——, "Amazon GuardDuty – Protect your AWS accounts and workloads with intelligent threat detection and continuous monitoring," https://aws.amazon.com/guardduty/, Apr. 2020.

Puneet Gill received his MASc and BASc in Computer Engineering from the University of Waterloo, in Canada, where he is currently working towards his PhD. He researches information security.

Werner Dietl is an Associate Professor in the Electrical and Computer Engineering (ECE) Department at the University of Waterloo, Canada. He researches safe and productive software development, towards which he combines theoretical results with practical tools so developers can create high-quality, trustworthy software.

Mahesh Tripunitara is a Professor in the Electrical and Computer Engineering (ECE) Department at the University of Waterloo, Canada. He researches various aspects of information security, with particular use of algorithm design, computational complexity and mathematical logic.