Granularity and Usability in Authorization Policies

Boyun Zhang[1], Puneet Gill[1], Nelu Mihai[2], and Mahesh Tripunitara[1](✉)

[1] University of Waterloo, Waterloo, Canada
{boyun.zhang,p24gill,tripunit}@uwaterloo.ca
[2] Cloud of Clouds Project, San Francisco, USA
nelumihai@icloud.com

Abstract. Emerging security systems need to carefully reconcile usability considerations in their design. In this context, we address authorization policies, which are used to limit the actions a principal may exercise on a resource. We compare two designs from the standpoint of the ease with which such policies can be devised and expressed. The two designs we consider are read-write-execute policies in UNIX, which was designed many decades ago, and identity-based policies in Amazon Web Services (AWS), which is a modern system. These can be seen, in the evolution of such designs, as two extremes—in the former, only the three actions read, write and execute are allowed in an authorization policy; in the latter, more than a thousand actions are allowed. While a richer set of actions lends to finer-grained authorization policies, the question we pose is: are such policies easier to formulate? Our question is important because a trend in the design of such policy languages in real systems over the years has been to enrich the set of actions. For a meaningful comparison between the two extremes, we design an overlay authorization policy syntax for AWS that allows the three actions read, write and execute only. We then describe our design of an ethics-approved, human participants study to assess whether a richer set of actions indeed results in better usability, and our results from carrying out the study. Using carefully chosen statistical methods that are appropriate for our study, we find that there is indeed evidence that allowing for a richer set of actions lends to better usability. Our work has significant implications to design in emerging security systems that seek to reconcile usability.

Keywords: Usability · Authorization · Policy

1 Introduction

Access control is widely, and rightfully, seen as an essential component of the security of a system. It deals with whether a principal, e.g., a user or role, is allowed to exercise an action, e.g., read or write, over a resource, e.g., a file in

Portions of this work were supported via grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mitacs, Canada.

the system. At the time a principal seeks to exercise an action over a resource, a trusted entity called a reference monitor checks whether that principal is indeed authorized to do so [11]. We address the setting in which the reference monitor bases its decision on an authorization policy that can change with the system or even over time within the same system, rather than, for example, a universally established set of rules. An authorization policy can be seen as a collection of triples, ⟨Principal, Action, Resource⟩, where such a triple authorizes the Principal to exercise the Action over the Resource.

```
$ stat --format=%A /etc/passwd
-rw-r--r--
```

Fig. 1. UNIX example.

Each system adopts a syntax for a language, and associated semantics, in which an authorization policy is specified. Two examples from designs for such a language which are the foci of this work are shown in Figs. 1 and 2.

In Fig. 1 we show the output of a command called `stat` in a version of the UNIX system [31]. The command shows us the so-called file permission bits for a file whose name we specify as the argument to the command, in this example, `/etc/passwd`. The file permission bits express an authorization policy over the particular file. In this example, a principal called the `owner` is allowed to both read and write the file, and the principals `group` and `others` are allowed to read the file only. Inherent to the design of the authorization policy syntax in the UNIX system and its variants is that only the three actions, `read`, `write` and `execute` are allowed to be specified in a policy. The semantics of the possession of one of those actions by a principal on a resource depends on the kind of resource it is. For example, what it means to possess the `execute` action over a directory is different from what it means to possess the `execute` action over a file.

In Fig. 2, we show an example of an authorization policy from Amazon Web Services (AWS) [7]. AWS allows for a few different kinds of authorization policies; we focus on so-called identity based policies, which are the most frequently used in AWS [8]. The figure shows an example of what AWS calls a statement, which specifies that a principal, which is a user or role, to whom this statement is bound is authorized the two actions `dynamodb:DeleteItem` and `dynamodb:DeleteTable` to the resource.

We recognize that AWS identity based policies may be seen more as capabilities each of which is bound to a principal, whereas UNIX file permission bits may be seen as Access Control Lists (ACLs) each of which is bound to a resource [11]. This distinction is relevant to our work only in that we really would be comparing across two rather different kinds of systems if we were to directly compare UNIX file permission bits with AWS identity based policies. Apart from that, our focus is only on the number of different actions each design permits to be specified in an authorization policy.

In comparing the two different designs for the possible actions in a policy, we ask: does one design choice lend to more usable authorization policies than the other? Usability in our context refers to the ease with which a policy can be specified in a particular syntax given goals for that policy. Prior work recognizes that it can be difficult for a human to specify authorization policies correctly, and with sufficient expediency (see, for example, [10,17,20,30]). And indeed, in such systems, it is a human, for example the owner of a file or a systems administrator, who specifies such policies.

```
{ "Statement": [{
        "Effect": "Allow",
        "Action": [ "dynamodb:DeleteItem",
                    "dynamodb:DeleteTable" ],
        "Resource": "arn:aws:dynamodb:::table/myTable"
}] }
```

Fig. 2. An example of an identity-based policy in AWS.

In this context, we observe that since the invention of the UNIX system and its file permission bits syntax for authorization, over time, the trend seems to be to increase the size of the set of actions in a design. For example, the Windows NT system allows for six actions [22]; in addition to `read`, `write` and `execute`, we have `delete`, `change permission` and `take ownership`. (These actions are, in turn, grouped into so-called "permissions" such as `full control`, which grants all the six actions, and `change`, which grants `read`, `write` and `delete` only.) More recently, in the Android system, we have about 250 app permissions, for example, ADD_VOICEMAIL and WRITE_CALL_LOG [14]. In AWS identity-based policies, the possible set of actions is of size more than a thousand.

It is then reasonable to ask: why has such an evolution occurred from a set of three actions only to a set of more than a thousand? We do not have a definitive answer to this question, as we are unable to find any prior work that reasons about these design choices in the corresponding systems. However, we observe that a richer set of actions lends to finer-grained authorization policies. In the example from AWS in Fig. 2, we observe that actions are bound to a particular service, `dynamodb` in the example. Thus, they seem more custom to the particular action that is being authorized, rather than the more generic `read`, for example. Thus, our work can be seen as relating the ability for such finer-grained specification of authorizations, to the ease with which authorization policies can be specified. Does one correlate with the other?

Our Work. Our hypothesis is that a system with a richer set of actions is more usable than one with `read`, `write` and `execute` only. Our work tests this hypothesis. We have done two things, which we discuss in sequence. First, for a meaningful comparison, we have devised a syntax for identity-based policies in AWS that allows the actions `read`, `write` and `execute` only. We discuss our design in

Sect. 2 for this. Our design leverages our observations regarding the similarity between the manner in which resources are referred to in the UNIX filesystem, and those in AWS. Then, we discuss our design of a human participants study entirely on AWS. Our study adopts a split participants design [21]—of all the human participants, a randomly chosen half are assigned to one of the two syntaxes, and the other half to the other syntax (see Sect. 3). We then discuss the manner in which we carried out the study, and our observations and inferences from the data we collected (see Sect. 4). We conclude by observing that there is indeed evidence to the better usability of an authorization syntax with a richer set of actions. We discuss related work in Sect. 5 and conclude with Sect. 6, with suggestions for future work.

2 A Read-Write-Execute Syntax for AWS Policies

As we discuss in Sect. 1, for us to be able to carry out an "apples to apples" comparison with a focus on the set of permitted actions only, we designed a syntax for AWS identity-based policies whose set of actions is `read`, `write` and `execute` only. In this section, we discuss our design. We begin with an example.

```
{ "Statement": [
    {   "Effect": "Allow",
        "Actions": [
            "elastictranscoder:ReadPreset" ],
        "Resources":[
            "arn:aws:elastictranscoder:::
            preset/apple-mp4" ]
    } ] }
```

```
{ "Statement": [
    {   "Effect": "Allow",
        "Actions": [ "read" ],
        "Resources":[
            "arn:aws:elastictranscoder:::
            preset/apple-mp4" ] },
    {   "Effect": "Allow",
        "Actions": [ "execute" ],
        "Resources":[
            "arn:aws:elastictranscoder:::
            preset"
    ] } ] }
```

Fig. 3. A before and after example that helps explain our design for identity-based policies in AWS with `read`, `write` and `execute` actions only. To the left is an example identity-based policy in AWS, and to the right is the policy to the left under our design with those three actions only. As we explain in the prose, we need not only `read` to the resource `present/apple-mp4` that is mentioned in the policy on the left, but also `execute` on its "parent" resource, `preset`.

To the left of Fig. 3 is a policy in AWS's Elastic Transcoder service. It grants a user or role who is bound with that policy an action called `elastictranscoder:ReadPreset` on the resource `preset/apple-mp4`. The entire string that constitutes the name of the resource, `arn:aws:...`, is called an Amazon Resource Name (ARN), which is AWS's way of uniquely identifying a resource. We omit portions of the ARN such as partition and region as those are not relevant to our work. We can correctly refer to an ARN as the canonical identity of a resource and do not belabour AWS's choices in naming its resources as that is not a focus of our work.

Resource type	ARN
job	`arn:...:job/${JobId}`
pipeline	`arn:...:pipeline/${PipelineId}`
preset	`arn:...:preset/${PresetId}`

Fig. 4. The three types of resources in AWS's Elastic Transcoder service, and their corresponding ARNs, i.e., manner in which they are identified. We omit portions that are not relevant to our work using ellipses "..."

Our design of AWS identity-based policies with `read`, `write` and `execute` actions only is based on our observation that there is a natural and straightforward similarity between the naming scheme in the UNIX filesystem and resources in AWS. Before discussing this similarity, we overview the naming scheme of the UNIX filesystem, which has subsequently been adopted by its more modern variants, such as the `ext4` filesystem [12]. Figure 5 shows the naming scheme in UNIX and the naming scheme in AWS side-by-side to emphasize the similarity. They are both shown as trees because that is a natural way to represent the structure inherent to both naming schemes.

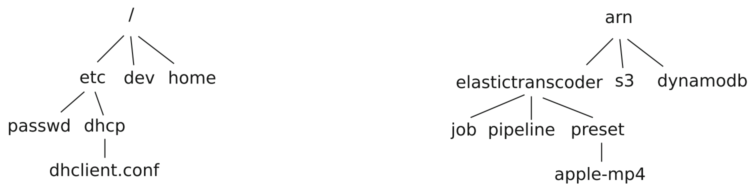

Fig. 5. The similarity between the naming scheme in the UNIX filesystem and in AWS. To the left is a portion of the directory and file structure in UNIX. To the right is a portion of AWS's services and resource types. We omit some portions of the names (ARNs) in AWS that are not relevant to our work.

In UNIX, there are two basic kinds of filesystem objects: directories and files. A directory contains other directories and files. A file contains data, or is a reference to some other information such as a device. The fully qualified name of a file starts with the symbol "`/`," which refers to the root directory, and is followed by one or more directory names separated by "`/`," with the file name at the end. For example, `/etc/dhcp/dhclient.conf` is the fully qualified name of a particular file. There are directories that are immediate sub-directories of the root level directory that have specific names and are associated with specific semantics. Examples of these are `/etc` and `/dev`.

We now discuss the manner in which resources in AWS are named. AWS classifies what it provides into services such as Elastic Transcoder, DynamoDB and S3. Each such service consolidates some common functionality that AWS provides. Elastic Transcoder, for example, is intended to provide functionality for converting media files into formats that can be played back on end-user devices [4]. Associated with every AWS service is a set of resource types [5]. Every resource which is mentioned in any identity-based policy is one of these types. The resource's name, i.e., ARN, includes the name of the service and the resource type within that service that the resource is. In the Elastic Transcoder service of AWS, for example, there are three resource types: job, pipeline and preset. We reproduce in Fig. 4 the table from the documentation for the Elastic Transcoder service which mentions these types and their corresponding ARNs [2].

Now that we have discussed the manner in which UNIX names its files and AWS names its resource, we now discuss the similarity we see between the two. Using the Elastic Transcoder service as an example again, we perceive the prefix `arn:...` up to one of the mnemonics `job`, `pipeline` or `preset` as the root-level directory in the UNIX filesystem "`/`." Each of the resource types `job`, `pipeline` and `preset` in the Elastic Transcoder service can be seen as a sub-directory of the root-level directory in UNIX, e.g., `/etc` and `/dev`, which must be named exactly as such and has specific semantics associated with what it contains.

Necessary and Sufficient Permissions in UNIX. Having observed the similarity between the naming schemes in UNIX and AWS, which in turn is a consequence of the manner in which resources are structured in the respective systems, we consider what necessary and sufficient permissions in the UNIX filesystem are for a user to be able to exercise an action on a resource. For example, consider that a user wants to read the file `/etc/dhcp/dhclient.conf`. Does it suffice that that user possess `read` over that file? The answer is no. In addition to possessing `read` over the file, the user needs to possess `execute` permission to each of the directories `/etc` and `/etc/dhcp`. This is the reason that in the example in Fig. 3, in translating the permission to exercise `elastictranscoder:ReadPreset`, we give the user both `read` to the resource `preset/apple-mp4` and `execute` to its "parent directory," `preset`.

More broadly, Fig. 6 shows a table that expresses the permissions that are necessary and sufficient to perform some actions in a UNIX directory and file structure that is `mydir/myfile`. That is, we have a directory named `mydir`, within which we have a file named `myfile`. For example, to be able to read `myfile`, the necessary permissions are `execute` on `mydir` and `read` on `myfile`.

In Fig. 6, we present a table of three actions in the Elastic Transcoder service in AWS, and our design of permissions from amongst `read`, `write` and `execute` only that are necessary and sufficient. The action ReadPipeline, given a pipeline ID, is similar to reading a "file" within the `pipeline` "directory." Therefore, as the table in Fig. 6 suggests for the UNIX filesystem, we need `execute` on the `pipeline` "directory," and `read` on the "file" that corresponds to pipeline ID that is part of the argument to the action. DeletePreset is similar to deleting a

Action	Permissions	
	`mydir`	`myfile`
read `myfile`	{x}	{r}
delete `myfile`	{w, x}	∅
create file in `mydir`	{w, x}	∅
write `myfile`	{x}	{w}

Action	Resource	Permissions
ReadPipeline	`pipeline/` `${pipeline ID}`	{x} to `pipeline/` {r} to `pipeline/${pipeline ID}`
DeletePreset	`preset/` `${preset ID}`	{w, x} to `preset/`
CreateJob	`job/` `${job ID}`	{w, x} to `job/` {w, x} to `job/${job ID}`

Fig. 6. To the left are necessary and sufficient permissions for some actions in a UNIX filesystem fragment that is `mydir/myfile`. We adopt `r` for `read`, `w` for `write` and `x` for `execute`. To the right are permissions that are necessary and sufficient for some actions in the Elastic Transcoder service of AWS in our design that allows `read (r)`, `write (w)` and `execute (x)` only.

"file" within the `preset` "directory," and therefore, as per the table in Fig. 6, requires `write` and `execute` on the `preset` "directory."

And as a final example, CreateJob given a job ID corresponds to creating a "file" within the `job` directory and also writing to that file, because the action of creating a job in Elastic Transcoder in AWS causes some data to the written to the "file" that corresponds to that job. Thus, this corresponds to both creating a file and write to that file from the table in Fig. 6 on UNIX permissions. That is, we require `write` and `execute` to both the "file" that corresponds to the job ID, and the "directory" `job` within which that "file" is contained.

Summary. In summary, we first observe the correspondence between the manner in which resources in AWS are organized and the manner in which files and directories in the UNIX filesystem are organized. We then adopt exactly the same permissions that would be necessary and sufficient in the UNIX filesystem, in our design of AWS policies that allow `read`, `write` and `execute` only. Thus, we anticipate that anyone who is familiar with UNIX file permissions and is informed of the correspondence between the two schemes, will be able to formulate `read`, `write` and `execute` permissions for resources in AWS. As we discuss in our next section, we exploit this in the design of our human participants study.

3 The Design of a Human Participants Study

The intent of our human participants study is to test the hypothesis that policies are easier to write with a richer set of actions than read-write-execute only. Towards this, each human participant was asked to formulate and write an identity-based policy in AWS in one of two syntaxes: the original syntax that AWS has designed for such policies that allows for a rich set of actions, or the syntax we designed for AWS that allows the actions `read`, `write` and `execute` only that we discuss in Sect. 2. In this section, we discuss details of the design of our study.

Nature of Human Participants. Our study follows a split participants design [21], wherein a human participant is randomly assigned to one of the two syntaxes,

```
response =
dynamodb.GetItem(
    Item = "cart-item",
    TableName="mycart")
```

```
dynamodb.CreateTable(
    TableName="mynewcart")
for item in items_in_cart:
    dynamodb.DeleteItem(
        Item = "cart-item",
        TableName="myoldcart")
```

```
dynamodb.UpdateTable(
    ReadCapacityUnits=15,
    TableName="mycart")
sqs.UpdateQueue(
    QueueName="update-success")
dynamodb.DeleteItem(
    Item="cart-item",
    TableName="mycart)
```

Fig. 7. The three snippets of code that we use in the three tasks in our human participants study. The snippet to the far left corresponds to Task 1: Get cart item. The snippet in the middle corresponds to Task 2: Change cart. And the snippet to the far right corresponds to Task 3: Edit cart.

such that each syntax has an equal number of participants. Each participant was scheduled for one hour. Of the hour, the first 15 min were dedicated to training, and the remainder to fulfill three tasks that each participant was assigned. For each syntax, every participant was assigned the same three tasks which we discuss below. On account of the pandemic, all interactions were via teleconferencing. The participants were provided with a consent letter at the time their appointment was made, which detailed various aspects of the study and assured their anonymity. Each participant was required to verbally consent to the study at the outset. Each participant was provided a feedback letter at the end. With the consent of each participant, we required the participant to share their screen so we could record the manner in which they worked. Part of the training was in thinking aloud. We recorded each participant's audio as well. All our materials were approved as part of our institution's ethics approval process. We remunerated each participant a small amount of money, again with the approval of our ethics board, for their participation.

As our study involves fairly sophisticated technical aspects, we wanted participants who most closely had the qualifications of a security or systems administrator. Consequently, all our participants were final- (fourth-) year undergraduate students, or first or second year graduate students in computer engineering and computer science. We required that all of our participants be (i) familiar with UNIX file permissions, and, (ii) not be familiar with identity-based policies in AWS, nor serverless applications in AWS. Our intent with (i) was that we wanted to "factor out" difficulties with what we wanted to consider baseline knowledge in our study, and, with (ii) was that we did not want such prior knowledge to give an advantage to a participant for which we could not compensate with our statistical methods.

Number of Human Participants. We recruited 20 participants in total, 10 each for each of the two syntaxes. Separately, we had three participants earlier who piloted our study, from whose observations we made improvements and fixes. A common concern in such studies is the number of participants: is 20 participants sufficient? There is no magic answer to this question. We observe that in studies of a technical nature as ours, around 10 participants appears to suffice. For instance, recent work that assesses the usability of static analysis tools for security [32] recruited 12 participants. Maxion and Reeder [19], whose work is related closely to ours, recruited 24 participants. Non-parametric tests, such as the Mann-Whitney U test on which we rely, have shown strong performance with small samples [25]. What is more important than a large number of participants is the nature of the participants, the design of and the manner in which we carry out our study, and appropriate choice and use of statistics on whose basis we make inferences. We have taken great care in these aspects.

An Application. We chose to focus on tasks that involve a single serverless cloud application that AWS provides as an example of such applications—a shopping cart [1]. Within that application, we specify tasks that involve two services: DynamoDB [3] and SQS [6]. The former is a database service and the latter, a queue service. DynamoDB supports a set of 81 actions across the entire service which can appear in an identity-based policy. SQS supports 20 actions. Of these, the ideal policies for our tasks require the specification of four actions from DynamoDB, and one from SQS. The only resource type from the six types DynamoDB specifies to which our tasks pertain is `table`, which is a database table. SQS has one resource type only, `queue`.

Tasks. We devised three tasks that a participant is to fulfill. Each involves formulating and writing an identity-based policy in the shopping cart application that we mention above [1]. For each task, we provide the human participant a snippet of the code in the python programming language, in which the backend of the application is written. Our code snippets are not fully faithful to the original application. Specifically: (i) We reproduce in our snippets only those pieces that pertain to our tasks so as not to burden a human participant with extraneous information. For example, in a call to get an item from a database table, we mention as arguments the table name and an identifier for the item only. (ii) We have changed the names of function calls to match exactly the names of actions that would appear in a policy. For example, rather than `get_item()`, which is the name of the corresponding method in AWS's python bindings, we use `GetItem()`, which exactly matches the action `dynamodb:GetItem` that would appear in an identity-based policy. (iii) We use keyword arguments that are allowed in python. That is, in python, if we have a subroutine that is defined as `def myfunc(name, id)`, then it can be invoked as `myfunc(id = 5, name = 'alice')`, which explicitly specifies which argument-value matches each parameter to the subroutine. (iv) We removed the "effect" field (see Figs. 2 and 3), with the assumption that we have a default deny, and any permissions that are specified are "allow." All participants were trained as such.

All of our adaptations (i)–(iv) are intended to not distract from the tasks on which we seek to focus. A human participant was scheduled for one hour. We could have scheduled each participant for longer; however, we would then have to account for fatigue.

The three snippets of code that correspond to the tasks are shown in Fig. 7. We call Task 1 "Get cart item." As the code snippet to the far left of Fig. 7 suggests, we retrieve a data item from a table using the `GetItem()` method. We call Task 2 “Change cart,” and it corresponds to the code snippet to the middle of Fig. 7. In it, we create a new table that appears to correspond to a new cart, and delete items one-by-one from an existing table, which is another cart. We do not show more actions to keep the policy that is needed somewhat compact. We call Task 3 “Edit cart.” It corresponds to the code snippet to the far right of Fig. 7. In it, we modify an attribute of the table that corresponds to the cart via a call to `UpdateTable()`. We then invoke the SQS service to report on the update, and then delete an item from the table that is the cart.

For both syntaxes, for each snippet of code, we had an ‘ideal’ solution against which we compared the results. For example, in Fig. 8, we show an ideal solution for the task to the far left of Fig. 7.

```
{ "Statement": [
    { "Actions": [ "dynamodb:GetItem" ],
      "Resources":[
          "dynamodb/mycart/cart-item" ]
    }] }
```

```
{ "Statement": [
    { "Actions": [ "execute" ],
      "Resources":[
          "dynamodb",
          "dynamodb/mycart" ] },
    { "Actions": [ "read" ],
      "Resources":[
          "dynamodb/mycart/cart-item" ]
    }] }
```

Fig. 8. An ideal policy for the task to the far left of Fig. 7. To the left in this figure is an ideal policy for the AWS syntax with a rich set of actions, and to the right is an ideal policy in the syntax with read, write and execute actions only.

Training. Both sets of participants were trained for 15 min at the start of their session. For a participant who was assigned to the read-write-execute syntax, there were two components to the training, both of which were done via pre-recorded videos. The first component was a review of UNIX file permissions. The second was AWS identity-based policies, but using read, write and execute actions only. This component was driven by an example of a code snippet and a policy for it. For a participant who was assigned the AWS syntax with a rich set of actions, we provided training on identity-based policies in AWS, where to find information on the various actions that can appear in policies, and an example of a snippet of code, and a policy for it. During our training for both sets of users, we incorporated training on thinking aloud, so we could capture the participants’ thought processes as they worked through their tasks.

Limitations. While we have been careful with our design, it certainly has a few limitations. One is an assumption that the participants we have chosen are representative of the population of human users who would author such policies. Another limitation is that the design our study addresses authoring policies from scratch only. It does not addresses changes to policies, which may be an event that occurs with more frequency than authoring a policy from scratch. We hypothesize that ease of authoring policies from scratch corresponds to ease of editing policies; however, this hypothesis itself needs to be validated. Yet another limitation lies in the choice of our tasks. We have chosen to focus on an application, which in turn is limited in the AWS services it exercises. It is possible that with a different set of services and/or a different set of possible actions, authoring policies in AWS's current syntax with a rich set of actions is harder than our study would suggest.

Notwithstanding these limitations, we argue that our study is novel, and has value, in that rejection of an appropriately posed null hypothesis with a sufficiently low p value based on our study is a valid indicator that the null hypothesis is indeed false in the broader population.

4 Results

Our study ran smoothly, and we were able to collect data as planned from all of our participants. None of the participants chose to withdraw upon commencement of their participation; this was an option they were allowed to exercise.

We consider the accuracy with which a participant fulfilled their task and their time-to-completion to be meaningful quantitative measures towards establishing or rejecting our hypothesis. That is, if a participant is able to achieve a task correctly, we credit the corresponding syntax as being usable, i.e., easy with regards to formulating a policy. Similarly, if the time to complete a task is short, this suggests that the syntax lends itself to usability. Of course with time, we need to consider whether a participant completed a task correctly as well. In addition to these quantitative results, an identification of the nature of errors our participants made is informative in that it tells us possible weak points from the standpoint of usability with a particular syntax.

Our raw observations for accuracy and time-to-completion are shown in Figs. 9 and 10. Not surprisingly, Task 1 was the easiest from the standpoint of accuracy for the participants—it is similar to example on which the participants were trained. Tasks 2 and 3 are more challenging to both sets of participants, however, it appears that the tasks were easier for the participants with the richer set of actions which is AWS's current design. It appears that across the board, participants who worked with the richer set of actions took less time. Also, there are cases that those who completed the tasks correctly took less time than those who did not, for example, Task 1 for read-write-execute, and Task 3 for the richer set. This is not necessarily surprising—it suggests to us that our participants were committed to the study.

Statistical Significance. We adopt a p value of 0.05 for every test at the outset. Thus, if the computed p value is < 0.05, then we deem the evidence strong enough to reject the null hypothesis. As a reminder, our null hypothesis is that authoring identity-based policies in AWS with the richer set of actions is no easier than authoring them when the set of actions is read, write and execute only.

As our statistical method, we adopt empirical bootstrap [26,33] with the median as the central tendency. Empirical bootstrap works as follows. Given a set of observations $\{X_1, \ldots, X_n\}$ which has sample median M_n, suppose we

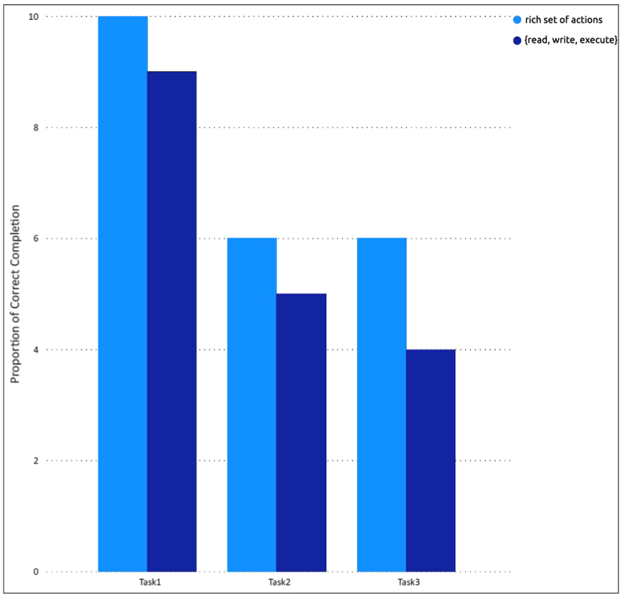

Fig. 9. The proportion of accurate responses per task.

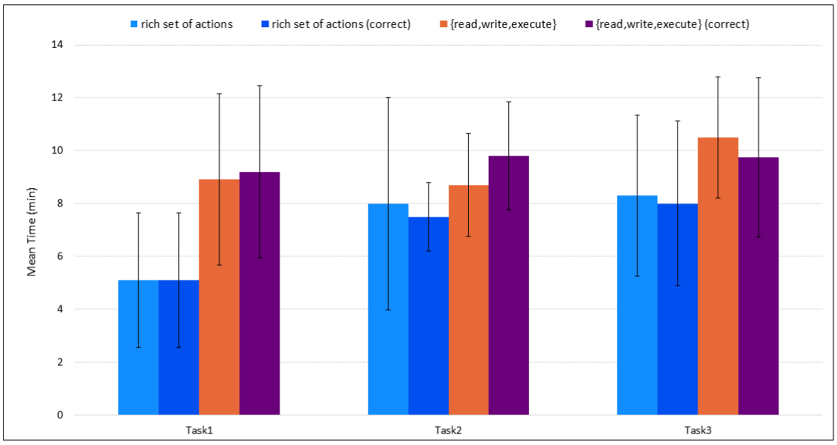

Fig. 10. The mean time-to-completion per task for both correct and all participants. The standard deviation is shown using error bars.

sample with replacement from the observations to get a "new" set of observations $X_1^{*(1)}, \ldots, X_n^{*(1)}$ with median $M_n^{*(1)}$. We repeat this, i.e., sample with replacement again with our original set of observations denote it B times. Thus, we have B sets of observations, each with a median $M_n^{*(i)}$ of its own. The key result is: the Cumulative Distribution Function (CDF) of a bootstrap median approximates the CDF of the true median. Indeed, both the estimate of variance and of the mean-squared error are functions of the number B [33]. In contrast to the median, there is no advantage to using bootstrap to establish statistical significance for a mean.

The median time for participants who were assigned our syntax with read-write-execute actions only was 9 minutes and 8 seconds. The median time for the participants who were assigned AWS's richer set of actions was 7 min and 10 s. For our bootstrap, we adopted $B = 200$. Study-wide, i.e., across all participants and tasks, a Mann Whitney U test's results are: $N = 400, U = 2674, p < 0.001$. Thus, as the p value is < 0.05, there is evidence to strongly reject the null hypothesis.

Task-by-task, considering both the participants who completed each task correctly and incorrectly, we have the following results.

- For Task 1, the Mann Whitney U test's results are: $N = 400, U = 3244.5, p < 0.001$. Thus, there is evidence to strongly reject the null hypothesis.
- For Task 2, the Mann Whitney U test's results are: $N = 400, U = 19804, p = 0.433 > 0.05$. Thus, we are unable to reject the null hypothesis.
- For Task 3, the Mann Whitney U test's results are: $N = 400, U = 7118, p < 0.001$. Thus, there is evidence to strongly reject the null hypothesis.

If we consider those participants who completed each task correctly only, we have the following task-by-task results.

- For Task 1, the Mann Whitney U test's results are: $N = 400, U = 3604, p < 0.001$. Thus, there is evidence to strongly reject the null hypothesis.
- For Task 2, the Mann Whitney U test's results are: $N = 400, U = 2802, p < 0.001$. Thus, there is evidence to strongly reject the null hypothesis.
- For Task 3, the Mann Whitney U test's results are: $N = 400, U = 10988.5, p = 0.0158 < 0.05$. Thus, there is evidence to reject the null hypothesis.

Overall Inferences. Based on our study, there is evidence to the better usability with a richer set of actions than read-write-execute only. Thus, our work suggests that an authorization policy syntax that allows for finer-grained authorizations is more usable from the standpoint of ease of formulating policies.

4.1 Nature of Errors

In addition to our observations we report in the previous section, we have analyzed the errors that were made by those participants who were unable to complete a task correctly, across both sets of participants. In the following, for each set of participants, we first identify all the kinds of errors we observed, and then present the number of participants who made one of those errors. We discuss our overall observations from this part of our analysis at the end of this section.

For the set of participants who were assigned the set of actions read-write-execute only, we observed the following kinds of errors (Fig. 11).

(A) *Too few actions*: the policy does not award sufficient authorizations on account that some actions were not granted. In particular, an action to the "parent directory" was not granted.
(B) *Incorrect action granted*: the policy grants an incorrect action, for example, read instead of execute.
(C) *Redundant grants*: the policy grants unnecessary actions.
(D) *Incorrect naming of a resource*: a resource was not named correctly, likely on account of incorrect mapping of AWS resources to the UNIX filesystem scheme as we discuss under our design in Sect. 2.

Error	**# instances**	**Tasks in which error occurred**
(A)	5	1, 2, 3
(B)	3	2, 3
(C)	3	2, 3
(D)	2	3

Error	**# instances**	**Tasks in which error occurred**
(E)	4	2, 3
(F)	2	2, 3
(G)	2	3

Fig. 11. To the left are the kinds of errors made by participants whose set of actions was read-write-execute only, the number of instances of each error, and the tasks in which the error occurred. To the right are the kinds of errors made by participants who were assigned the rich set of actions, the number of instances of each error, and the tasks in which the error occurred.

For the set of participants who were assigned the rich set of actions, we observed the following kinds of errors.

(E) *Incorrect naming of a resource.* Note that even though we associate the same phrase as error (D) for the other set of participants, the kind of error is different, as for this set of participants, no mapping to the UNIX filesystem is needed.
(F) *Incorrect action granted*: the policy grants an incorrect action, perhaps on account of the inability of the participant to navigate the information we provided on the list of all actions in an AWS service. Even though we use the same phrase as error (B) from the other set of participants, the cause of the error is different, and so we associate it with a different letter, i.e., (F).
(G) *Redundant actions granted*: this error appears to have been caused by lack of recognition that two actions needed to be associated with the same resource. Again, even though we use the same phrase as error (C) for the other set of participants, we deem this error to be of a different nature than (C).

The most prevalent error for the participants who dealt with the read-write-execute set of actions was (A), too few actions. Based on our study of the instances of these errors, it appears that the participants were not entirely familiar with UNIX filesystem permissions, e.g., that a permission to a parent directory is necessary for access to a file. This is the case notwithstanding the fact that familiarity with UNIX file system permissions was a required qualification for all participants, and we included a review of those permissions for those participants who were assigned the read-write-execute set of actions. Thus, it appears that lack of adequate knowledge of UNIX permissions may have been coupled with over-confidence in one's knowledge.

Incorrect naming of resources, errors (D) and (E), come as a surprise to us. We take this as part of errors that occur with such textual specification of policies. We point out that AWS does provide a visual editor for specification of policies, which may mitigate both this kind of error, and error (F), incorrect action. We do not anticipate that a visual editor will mitigate the counterpart error (B) for the participants with the read-write-execute actions only, as the likelihood of making the same kind of error (F) is small, given the highly limited set of actions.

Finally, errors (C) and (G) suggest a dangerous consequence: over-privilege. Over-privilege is known to be a source of security issues in cloud applications [23]. Our study did not focus on this problem; but our discovery of such errors suggests that work in over-privilege in this context may be of interest and value.

5 Related Work

The usability issues that plague security systems, authorization systems in particular, from the standpoint of formulating and expressing policies is well documented in the literature. The work of Maxion and Reeder [19] is pioneering in this regard. It considers the file-permissions interface of the Windows XP operating system, and compares it against an alternative design that it calls Salmon. It adopts a split-participants design for its human participant study, and based on the accuracy, time to completion and elimination of a class of errors, it concludes that Salmon is a better interface. Reeder and Maxion have since followed up that work with work that focuses on a particular class of errors called goal errors [27] and prevention of defects in an interface through hesitation analysis [28], and Reeder et al. [30] have devised a novel interface for Windows XP file permissions called expandable grids and analysed it. Similarly, Brostoff et al. [13] have devised and analysed an interface for Role-Based Access Control (RBAC) policies. Krishnan et al. [17] have addressed the usability of file Access Control Lists (ACLs) in Linux systems, and proposed an alternative interface to the existing one and shown its benefits via a human participants study. Our work adopts a number of elements from those pieces of work with regards to the manner in which a human participants study is conducted, and the results are analyzed. However, unlike such prior work, we do not seek to tout a new design for an interface, but rather compare a design that has been of historical interest and influence, which is the UNIX file permissions, to a modern design, which is AWS identity-based policies, from the standpoint of the number of actions that are allowed in a policy.

Mazurek et al. [20] have addressed the aspect of the manner in which users think about access control in a home setting, and have interviewed 33 users across 15 households in this regard. One of their key findings is that the mental models that humans have for access control do not match well with the interface and mechanisms that access control systems provide to articulate policies. That work clearly establishes some of the challenges in formulating access control policies, even in the seemingly simpler setting, compared to, for example, an enterprise setting, of households. Our work does not directly address the challenges that work raises. Rather, our focus is on whether a richer, more fine-grained set of actions renders policy formulation easier.

Beznosov et al. [10] have posed and answered a number of interesting questions from the standpoint of usability challenges in access control. Some of the questions that they address are novel; for example, whether the articulation of policies should be "transparent," and the manner in which federated policies required by technologies such as grid and distributed computing can be articulated and managed. Our work does not address questions of the nature that that panel discussion addressed, but rather, focuses on the rather more specific question in the context of real-world systems that have seen heavy use over time: UNIX file permissions and AWS identity-based policies.

Inglesant et al. [16] have devised a natural language interface for the specification of RBAC policies in grid computing systems. They discuss the process by which they arrived at their design, which includes interviews with 45 practitioners. They observe that while their design does lend to better usability, iterative-refinement is still needed for users to converge to correct policies. In contrast to their work, we consider a much more controlled setting, and compare two real-world designs for articulating access control policies. While our policies are textual and not visual, they are not based on a natural language.

Reeder et al. [29] recognize that access control policies can be in conflict with one another, and observe that the manner in which conflicts are resolved can have a significant impact on usability from the standpoint of policy authoring. Our work is different from theirs in that while we do consider usability from the standpoint of policy authoring, we do not consider conflicts nor the impact of the approach to conflict-resolution on usability.

Bauer et al. [9] address access control in a physical setting, and in particular, compare physical access to office rooms using a smartphone instead of a physical key. A particular focus of that work is that it is with real users, in the sense that those are the users to whom the office rooms have been assigned. In contrast to that work, our work chooses human participants that we hypothesize are representative of the kinds of humans who will author policies in our setting. Also, we consider textually specified policies to protect digital assets rather than physical access control with keys and smartphones.

Gusmeroli et al. [15] have incorporated considerations of usability in their work on devising a capability-based access control model for an Internet of Things (IoT). In the context of their work, usability is a design consideration. However, unlike our work, they do not carry out a human participants study to validate their design. Also, our work is in the entirely different context of identity-based policies in AWS. Lipford et al. [18] and Paul et al. [24] have studied the usability problems in the Facebook social network with regards to specification of permissions, and have found a number of deficiencies. Our work is entirely different from theirs in that we consider AWS identity-based policies, with a focus on the richness of the set of actions.

6 Conclusion and Future Work

We have compared two designs from the standpoint of the richness of the set of actions for authoring access control policies. Both designs are for identity-based policies in AWS. One of the designs allows three actions only: read, write and execute, and is based on file system permissions in the UNIX operating system. The other is the design from AWS which, in total, supports a set of more than 1000 actions. We see these two designs as extremes, and also at two ends of the continuum in the design of such syntaxes for policy authoring. We designed a syntax for identity-based policies in AWS, which we have discussed in this work, to allow us to carry out an "apples-to-apples" comparison between the two designs. We have carried out an ethics-approved human participants study to test the hypothesis as to whether a richer set of actions lends to better usability from the standpoint of policy formulation and authoring. Our study establishes that there is indeed evidence that a richer set of actions lends to better usability.

There is considerable scope for future work. One is a more elaborate study that incorporates users who author such policies in AWS. It may be difficult to assemble such a set of users, however. Another is a study with more realistic serverless applications, at scale, and the challenge involved in formulating and authoring policies for such applications. Yet another topic for future work is a study of over-privilege. We have identified in this work that over-privilege can be related to how usable a policy authoring syntax is. And over-privilege is dangerous from the standpoint of security. Consequently, a study with a focus on the manner in which usability as characterized by ease of policy formulation and authoring, and the risk of over-privilege, would be interesting future work for emerging systems.

References

1. Amazon Web Services (AWS): Serverless shopping cart microservice, January 2021.https://github.com/aws-samples/aws-serverless-shopping-cart
2. Amazon Web Services (AWS): Actions, resources, and condition keys for amazon elastic transcoder. https://docs.aws.amazon.com/service-authorization/latest/reference/list_amazonelastictranscoder.html. Accessed 31 Jan 2021
3. Amazon Web Services (AWS): Amazon dynamodb, https://aws.amazon.com/dynamodb/. Accessed 31 Jan 2021
4. Amazon Web Services (AWS): Amazon elastic transcoder. https://aws.amazon.com/elastictranscoder/. Accessed 31 Jan 2021
5. Amazon Web Services (AWS): Amazon resource names (ARMS). https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html. Accessed 31 Jan 2021

6. Amazon Web Services (AWS): Amazon simple queue service. https://aws.amazon.com/sqs/. Accessed 31 Jan 2021
7. Amazon Web Services (AWS): Amazon web services (AWS) - cloud computing services. https://aws.amazon.com. Accessed 31 Jan 2021
8. Amazon Web Services (AWS): Aws identity and access management – user guide – access management – policies and permissions in IAM. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html. Accessed 31 Jan 2021
9. Bauer, L., Cranor, L.F., Reeder, R.W., Reiter, M.K., Vaniea, K.: A user study of policy creation in a flexible access-control system. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2008, pp. 543–552. ACM, New York (2008)
10. Beznosov, K., Inglesant, P., Lobo, J., Reeder, R., Zurko, M.E.: Usability meets access control: challenges and research opportunities. In: Proceedings of the Symposium on Access Control Models and Technologies, SACMAT 2009, pp. 73–74. ACM, New York (2009)
11. Bishop, M.: Introduction to Computer Security, 1st edn. Addison-Wesley, Boston (2004)
12. Both, D.: An introduction to linux's ext4 filesystem. opensource.com, May 2017. https://opensource.com/article/17/5/introduction-ext4-filesystem
13. Brostoff, S., Sasse, M.A., Chadwick, D., Cunningham, J., Mbanaso, U., Otenko, S.: 'R-what?' development of a role-based access control policy-writing tool for e-scientists. Softw. Pract. Exp. **35**(9), 835–856 (2005)
14. Google Developers: Android API reference – android platform – manifest.permission, https://developer.android.com/reference/android/Manifest.permission. Accessed 31 Jan 2021
15. Gusmeroli, S., Piccione, S., Rotondi, D.: A capability-based security approach to manage access control in the internet of things. Math. Comput. Model. **58**(5), 1189–1205 (2013)
16. Inglesant, P., Sasse, A.M., Chadwick, D., Shi, L.L.: Expressions of expertness: the virtuous circle of natural language for access control policy specification. In: Proceedings of the Symposium on Usable Privacy and Security, SOUPS 2008, ACM, New York (2008)
17. Krishnan, V., Tripunitara, M.V., Chik, K., Bergstrom, T.: Relating declarative semantics and usability in access control. In: Proceedings of the Eighth Symposium on Usable Privacy and Security. SOUPS 2012, ACM, New York (2012)
18. Lipford, H.R., Besmer, A., Watson, J.: Understanding privacy settings in facebook with an audience view. In: Proceedings of the 1st Conference on Usability, Psychology, and Security. UPSEC2008, USENIX Association (2008)
19. Maxion, R.A., Reeder, R.W.: Improving user-interface dependability through mitigation of human error. Int. J. Hum.-Comput. Stud. **63**(1), 25–50 (2005)
20. Mazurek, M.L., et al.: Access control for home data sharing: attitudes, needs and practices. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 645–654. ACM, New York (2010)
21. McLeod, S.A.: Experimental design. Simply Psychology, January 2017. https://www.simplypsychology.org/experimental-designs.html
22. Network Encyclopedia: NTFS permissions (windows NT). https://networkencyclopedia.com/ntfs-permissions-windows-nt/. Accessed 31 Jan 2021
23. Osborne, C.: The top 10 security challenges of serverless architectures. Zero Day, January 2017. https://www.zdnet.com/article/the-top-10-risks-for-apps-on-serverless-architectures/

24. Paul, T., Puscher, D., Strufe, T.: Improving the usability of privacy settings in Facebook. arXiv e-prints arXiv:1109.6046, September 2011
25. Pero-Cebollero, M., Guardia-Olmos, J.: The adequacy of different robust statistical tests in comparing two independent groups. Psicologica **34**, 407–424 (2013)
26. Ramesh Johari: MS & E 226: "Small" Data, Lecture 13: The bootstrap (v3). September 2020.http://web.stanford.edu/~rjohari/teaching/notes/226_lecture13_inference.pdf
27. Reeder, R.W., Maxion, R.A.: User interface dependability through goal-error prevention. In: 2005 International Conference on Dependable Systems and Networks (DSN 2005), pp. 60–69 (2005)
28. Reeder, R.W., Maxion, R.A.: User interface defect detection by hesitation analysis. In: International Conference on Dependable Systems and Networks (DSN 2006), pp. 61–72 (2006)
29. Reeder, R.W., Bauer, L., Cranor, L.F., Reiter, M.K., Vaniea, K.: More than skin deep: measuring effects of the underlying model on access-control system usability. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2011, pp. 2065–2074. ACM, New York (2011)
30. Reeder, R.W., et al.:Expandable grids for visualizing and authoring computer security policies. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2008, pp. 1473–1482. ACM, New York (2008)
31. Ritchie, D.M., Thompson, K.: The Unix time sharing system. Commun. ACM **17**, 365–375 (1974)
32. Smith, J., Nguyen Quang Do, L., Murphy-Hill, E.: Why can't johnny fix vulnerabilities: a usability evaluation of static analysis tools for security. In: Proceedings of the Symposium on Usable Privacy and Security. SOUPS2020, Usenix, Aug 2020
33. Chen,Y.-C.: STAT/Q SCI 403: introduction to resampling methods, Lecture 5: Bootstrap, April 2017. http://faculty.washington.edu/yenchic/17Sp_403/Lec5-bootstrap.pdf,