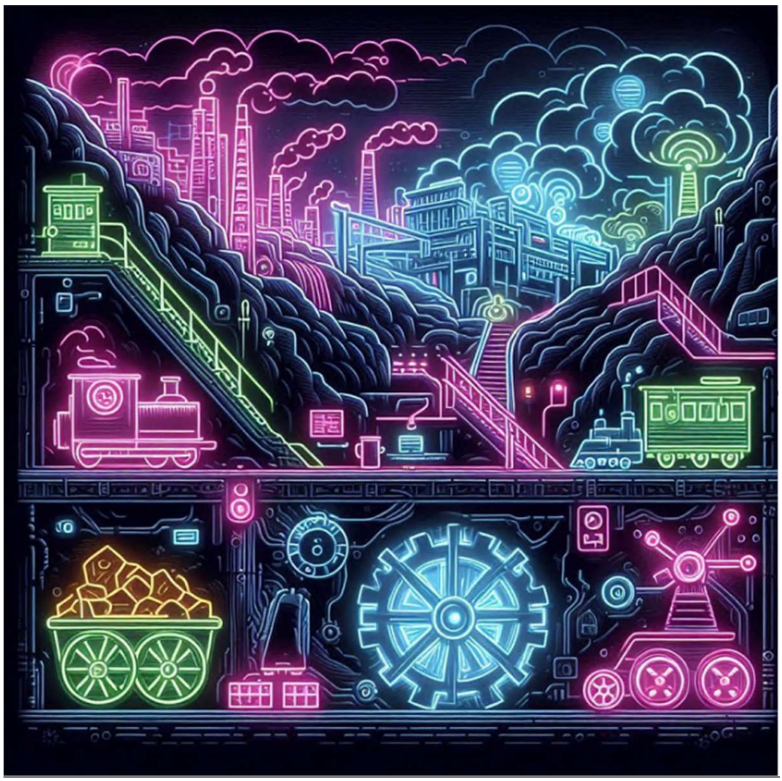
Automated Role mining

Author: Christophe PARISEL



Introduction

In many fields of computation, a fundamental principle known as the verifier/generator asymmetry plays a pivotal role. This principle highlights the inherent disparity between the ease of verifying a solution and the difficulty of generating it. In this chapter, we use this paradigm to define a semi-supervised SPNs role mining process.

1. Mining phase: a SMT solver mines roles according to the desired silhouette of SPN clusters calculated with the WAR norm defined in the previous chapter. This is the computationally intensive part.
2. Verification phase: roles are manually verified using a new causal interpretation framework producing auditable "proofs of equivalence".

Problem statement

By engaging in semi-supervised role mining, the roles we intend to mine:

1. always relate to a cluster of similar SPNs (the concept of cluster is explained in the previous chapter);
2. must enforce the desired silhouette. Hence permission scopes along Write, Action and Read axis must not exceed the upper bounds set by the WAR norm of the SPNs cluster;
3. attempt to follow least privileges principles. “attempt” means that least privileges is not always guaranteed. This is the reason why we call this semi-supervision (we need a verification phase).

The mining process

Overview

The idea is to build Azure role assignments in a bottom-up fashion (i.e from the “ground truth” in Azure Activity Logs to the RBAC “golden source” in AAD) , by running through all logs of the cluster’s SPNs over a long observation period (say 3 months) and coalescing them into Azure roles by using an equivalence relation called SP-equivalence.

Here is the takeaway:

Two Azure activity logs of a cluster of SPNs belong to the same Azure role if and only if they are SP-equivalent.

Roughly, SP-equivalence states that two activity logs are “equal” if they belong to the same scope (the same subscription, for instance) <u>or</u> they permit the same action / data action (say, loadbalancers/write).

Elucidating SP-equivalence would almost end there if the underlined “or” statement in the previous sentence was replaced by an “and”. There is a small problem with “or”: unlike “and”, it is not transitive. And we need transitivity to make things equivalent.

A workaround that we are going to take throughout this chapter is to first define SP-equivalence implicitly, rather than explicitly. SP-equivalence will be partially interpreted, and it won’t be an equivalence relation at this stage (it will just be a relation).

Then, we will propose a constructive implementation of SP-equivalence using a SAT solver supporting Equality Theory by embedding this relation into an equality.

Azure scoped permissions

In chapter 1, we defined an equivalence relation which let use compare simple roles made of permission sets. Roles were grouped into three equivalence classes: W, A and R and we could compare them based on an ordering that we set as W > A > R.

Azure roles are more complex: we need to account for scopes.

To reason about the reality of Azure RBAC design, let's define a few concepts:

An Azure scoped permission "a" is a tuple (a1,a2) where "a1" is a resource Id (eg: /subscriptions/UUID/resourceGroups/myRG/providers/Microsoft.Network/loadbalancers/someLB) and "a2" is a permission (eg: Microsoft.Network/loadbalancers/write)

Concretely, a scoped permission is nothing more than an Azure activity log split into its two fundamental components.

Now let's call $S\_P_{cluster,period}$ the set of all azure scoped permissions found by scanning Azure activity logs of a SPN cluster over a period. For short, we'll just say S_P.

A scope filter f(resource permission, level) is an uninterpreted function operating on strings which, when it is defined, extracts the resource container name of a resource Id string at a given scope level and returns it as a string. For example,

f(("/subscriptions/UUID/resourceGroups/myRG/providers/Microsoft.Network/loadbalancers/someLB,loadbalancers/write"),"subscriptions")= "/subscriptions/UUID"

Observe that all inputs and outputs are pure strings.

When the level we are talking about is obvious ("subscriptions" in our example), it can be omitted in the signature.

In practical terms, when the uninterpreted function *f*() is defined, it essentially limits the first input string based on a criterion specified by the second input. For that, it performs a string truncation: the truncation of string x will be noted $\lfloor x \rfloor$.

Formally, f() is defined as such:

For any a=(a1,a2) and b=(b1,b2) in S_P,

Axiom 1 $\lfloor a1 \rfloor = \lfloor b1 \rfloor \Rightarrow f(a1)=f(b1)$

Axiom 2 a2=b2 => f(a1)@f(b1)

Axiom 1 states that, if truncated strings a1 and b1 are literally equal, so are their scope filters.

Axiom 2 states that, if permissions a2 and b2 are literally equal, then their scope filters (relating to a1 and b1) are in a relation @.

Relation @ is defined as such:

For any a=(a1,a2) and b=(b1,b2) in S_P,

Axiom 3 f(a1)=f(b1) => f(a1)@f(b1)

Axiom 2' a2=b2 => f(a1)@f(b1)

Axiom 3 and 2' state that if two tuples have the same scope filter or the same permission, they are in the relation.

Axiom 2' is just the same as axiom 2. The prime can be omitted.

Concretely, two tuples in S_P are scope equivalent if their scopes are literally equal, or their permissions are identical.

Note that $\lfloor a1 \rfloor \neq \lfloor b1 \rfloor$ doesn't mean that $f(a1) \neq f(b1)$ because axiom 2 says we could have $\lfloor a1 \rfloor \neq \lfloor b1 \rfloor$ and $f(a1)=f(b1)$ simultaneously. $f(a1)=f(b1)$ is just a special case of $f(a1)@f(b1)$.

Conversely, $a2 \neq b2$ doesn't mean that $f(a1)$ isn't in relation @ with $f(b1)$.

At this stage, @ is not an equivalence relation: it is nonsymmetrical and intransitive.

Non symmetry

Obviously, all tuples implied by axiom 3 or axiom 2 are symmetrical with regards @.

But because @ is uninterpreted, there might be tuples in relation which are neither implied by axiom 3 nor axiom 2. These tuples have no reason to be symmetrical with regards @.

Intransitivity

Unless tuples are directly connected, nothing states in the axioms that there exists a chain of tuples connecting any tuple pairs.

Implementation of @ using a SAT solver

Since @ is uninterpreted, we cannot leverage it readily to generate a partition.

We propose to embed @ into an equality relation. This approach brings several benefits:

- @ becomes interpreted
- Equalities are efficiently implemented into SAT/SMT solvers supporting Equality Theory
- Equalities are reflexive, symmetrical and transitive
- Equalities generate the finest possible partitions
- Equalities are auditable and explainable (see next chapter)

Consider the following extremely simple equality "≅":

For all tuples $x=(x1,x2)$ in S_P, $\lfloor x1 \rfloor \cong x2$

Note that the equality doesn't operate on S_P terms, but on a flattened set where scopes and permissions are intermixed. We'll call this set 'FLAT'.

Now let's replace the standard equality "=" by this new equality in all 3 axioms.

@ becomes:

For any a=(a1,a2) and b=(b1,b2) in S_P,

Axiom 3 $f(a1) \cong f(b1) \Rightarrow f(a1)@f(b1)$

Axiom 2 $a2 \cong b2 \Rightarrow f(a1)@f(b1)$

Axiom 1 $\lfloor a1 \rfloor \cong \lfloor b1 \rfloor \Rightarrow f(a1) \cong f(b1)$

We call this embedded relation @ SP-equivalence.

Let's prove that ≅ embeds @ by proving that axioms of @ are met:

Pick any two terms x and y in 'FLAT' so that x ≅ y.

We have 3 cases to consider:

Case 1: Suppose one term is a scope (say, 'x') and the other term is a permission (say, 'y'). Since 'x' is a scope, let's rewrite it $\lfloor x1 \rfloor$. Since 'y' is a permission, let's rewrite it y_2. (These are simple syntax changes to clarify the proof).

If $\lfloor x1 \rfloor \cong y_2$, it means that there exists a tuple $x=(x_1,x_2)$ and a tuple $y=(y_1,y_2)$ and a chaining of equalities $c_i \cong c_j$ in S_P so that

$\lfloor x1 \rfloor \cong x2$, $\lfloor y1 \rfloor \cong y2$ and

$\lfloor x1 \rfloor \cong c_i \cong c_j \cong \lfloor y1 \rfloor \cong y2$ or $x2 \cong c_i \cong c_j \cong y2$

The 3 axioms are met, so x @ y.

Case 2: Suppose both terms are scopes. We may rewrite them as $\lfloor x1 \rfloor$ and $\lfloor y1 \rfloor$.

If $\lfloor x1 \rfloor \cong \lfloor y1 \rfloor$, axioms 1 and 3 are met, so x @ y.

Case 3: Suppose both terms are permissions. We may rewrite them as x2 and y2.

If x2 ≅ y2, axiom 2 is met, so x @ y.

In its interpreted version, @ is an equivalence relation (because ≅ is an equivalence relation and because @ is embedded in ≅), so the effect on S_P is to generate a partition.

We call the finest partition induced by @ a condensate.

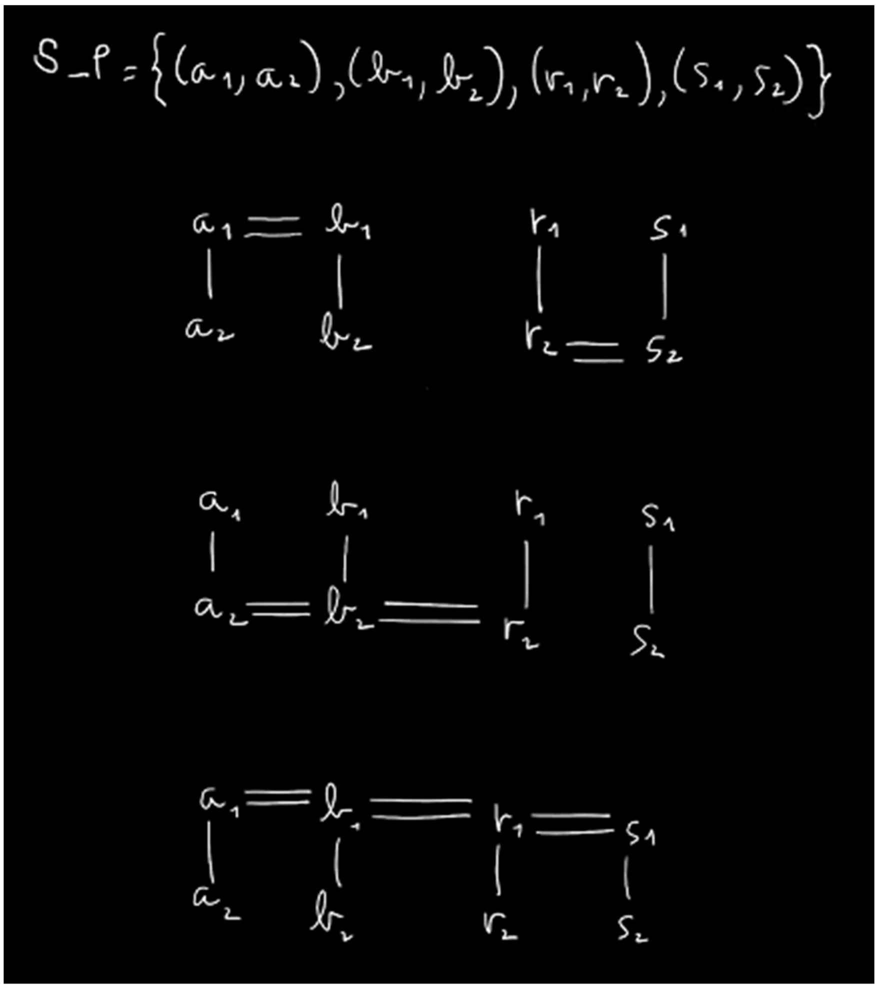
Some outstanding differences between a condensate and a WAR partition defined in chapter 1 are:

- terms are not just atomic permissions, but (scope,permission) tuples
- equivalence classes correspond to tuple bins, not Write, Action or Read
- the number of equivalence classes is not 3, it is variable: condensates may hold one single equivalence class (grouping all scoped permissions together) or as many equivalence classes as there are scoped permissions in cluster logs, or number of classes in-between (more of that in the Verification chapter).

SP-equivalence is implemented using ≅ in Azure silhouette, an open source SPN rights minimization tool available from https://github.com/labyrinthinesecurity/silhouette

Visual examples

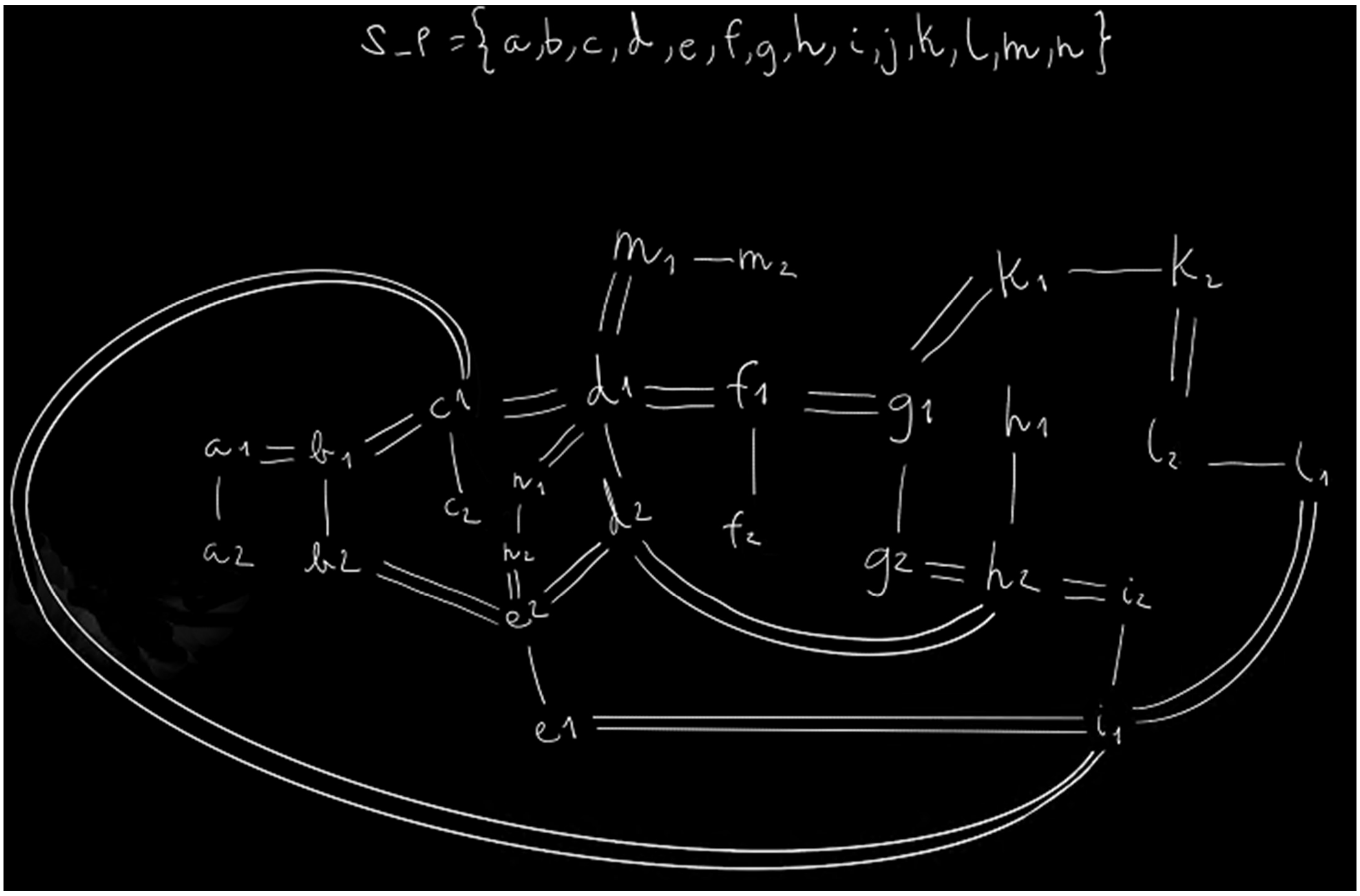
Here is an example of a S_P set with only 4 tuples: a,b,r and s. Depending on the pairing induced by ≅, we provide 3 different partitions:



The first partition features two equivalence classes, with two tuples each. The second one also features two classes, but the first one has 3 tuples whereas the second one only has one.

The last partition features a single equivalence class, encompassing all tuples.

Here is a more complex S_P set, showing how equality chains can be highly intertwined:



Molten condensates

In clusters where independent activities operating in independent scopes feature overlapping permissions, SP-equivalence will join the independent scopes to form the condensate, which breaks the least privileges principle.

We call this invalid configuration a *molten condensate*, and it cannot be prevented by using SP-equivalence alone. We need to refine "@" to make a finer partition of S_P so that we can make a distinction between segregated scopes.

Let's introduce a new function v() operating over S_P:

For any a and b in S_P, v(a1)≠ with regards @ if a and b belong to different segregated scope.

There are different ways to build v():

- if scopes are subscriptions, we can tag similar subscriptions and compare tags
- if scopes are resource groups, we can use tag as above, but also regular expressions if they follow a well-defined naming convention

In all cases, we must ensure that the scheme defining v() is complete and consistent (i.e., segregated scopes are non-overlapping). To that end, let's call "≡" the enriched version of "@":

Axiom 4: For any a and b in S_P, a ≡ b iff a @ b and v(a) = v(b)

Suppose that we set the following regular expressions to capture segregated resource groups:

RG0="/subscriptions/.+/resourcegroups/(app1_.+)$"

RG1="/subscriptions/.+/resourcegroups/(.+zone{s}?)$"

RG2="/subscriptions/.+/resourcegroups/(app{2,3}_.+)$"

If you naming convention always ensure that resource group name always begin with app1_,app2_app_3 or else that they include the words "zone" or "zones", then v() is well defined and always return a valid term c in S_P.

At that point, we are able to define special class representatives, one for each segregated resource groups (symbol _ is a don't-care):

<'RG0',_>

<'RG1',_>

<'RG2',_>

We use special brackets to distinguish these class representatives to usual tuples in S_P.

To make sure they are segregated, we add the following conditions:

Axiom of verticality: <'RG0',_> ≠ <'RG1',_> ≠ <'RG2',_> ≠ <'RG0',_>

In relation "≡" enriched with v() and the extra terms in bracket, v() is a congruence:

For any tuples a and b, a ≡ b implies v(a)=v(b)

The additional restriction ("and" condition) imposed on S_P terms in Axiom 4 ensures that the equivalence classes of ≡ are included in the equivalence classes of @. This is why we say that the partition of ≡ is finer than the partition of @.

Of course, v() is not a congruence of "@".

When defined properly, the vertical congruence can prevent molten condensates. It breaks them down into smaller independent components.

Example 1

Consider the following scoped permissions:

a=('/subscriptions/abcd/resourcegroups/app1_dev_northeurope',' 'microsoft.compute/virtualmachines/extensions/write')

b=('/subscriptions/abcd/resourcegroups/app1_prd_westeurope',' 'microsoft.compute/virtualmachines/extensions/write'

One would get v(a)=<'RG0', _>

And v(b)=<'RG0', _>

so v(a)=v(b)

Observe that, in Axiom 4, v(a)=v(b) may or may not be compatible with a @ b. Both conditions must be met so that a ≡ b

Example 2:

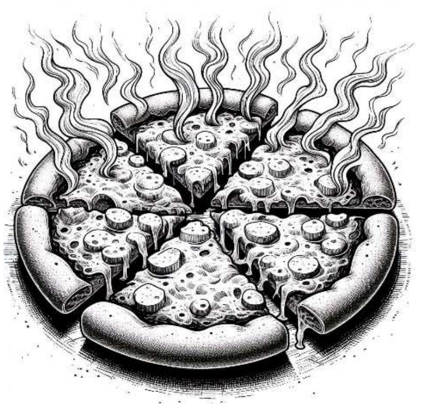
v(('/subscriptions/abcd/resourcegroups/app1_dev_northeurope',' 'microsoft.compute/virtualmachines/extensions/write'))=
(('/subscriptions/abcd/resourcegroups/app1_prd_westeurope',' 'microsoft.compute/virtualmachines/extensions/write')

v(('/subscriptions/abcd/resourcegroups/rbt2g_zones',' 'microsoft.compute/virtualmachines/extensions/write'))=
(('/subscriptions/abcd/resourcegroups/veztx_zone3',' 'microsoft.compute/virtualmachines/extensions/write')

In this case, v(a)='<'RG0', _> and v(b)=<'RG1', _> so v(a) ≠ v(b)

The axiom of verticality ensures that a and b are not SP-equivalent. The molten condensate they could have formed due to the common 'microsoft.compute/virtualmachines/extensions/write' permission is prevented.

Vertical congruence is similar to slicing a pizza. The pizza (condensate) is vertically separated into segregated slices. Feel free to call it "pizza congruence"!



A pizza vertically sliced into 6 segregated scopes.

Implementation of ≡ using a SAT solver

We follow the same line of thoughts as before, and embed ≡ into an equality, adding two more constraints or RG_i:

Definition of ≅:

For all tuples x=(x1,x2) in S_P and for all i,j in scoped resources RG,

$\lfloor x1 \rfloor \cong x2$

$RG_i \neq RG_j$ if $i \neq j$ with regards ≅

$RG_i \cong RG_j$ if $i = j$

Vertical congruence, like SP-equivalence, is also implemented in Azure silhouette

https://github.com/labyrinthinesecurity/silhouette

Accounting for the desired silhouette

So far, SP-equivalence hasn't taken desired silhouette requirements into consideration. To fix this, we're going to look at 2 situations:

1) the desired silhouette is the same for all 3 axis (W, A and R)
2) the desired silhouette can differ from one axe to another.

Case 1: same upper bound scope for W, A and R

In this case, meeting the desired silhouette upper bound is built-in into SP-equivalence, because filter f() will operate at the same level consistently for all kinds of permissions, and this level is the single one provided by the desired silhouette, regardless of permission type W, A or R.

Case 2: different upper bounds for W, A and R

We need to refine "@" to make a finer partition of S_P so that we can make a distinction between different upper bounds.

Horizontal congruence

Let’s introduce a new function h() operating over S_P tuples given a desired silhouette (W^+, A^+,R^+), where:

- W^+ is the upper scope set for Write operations (“resource group”, for instance)
- A^+ is the upper scope set for Write operations (“subscription”, for instance)
- R^+ is the upper scope set for Write operations (“resource group”, for instance)

For any “a” in S_P, h(a)=

- <‘RG’,’W’> if a2 is a Write operation and the desired silhouette for W is set at resource group level
- <‘SUB’,’A’> if a2 is an Action operation and the desired silhouette for A is set at subscription level
- <‘RG’,’R’> if a2 is a Read operation (recall the read operations cannot be retrieved from Azure Activity Logs, so this is just a theoretical result to make h() well-defined) and the desired silhouette for R is set at resource group level

The tuples returned by h() are not the usual tuples in S_P. This is why we use a bracket notation.

As it turns out, S_P must be enriched with a few new symbols: the h() function itself must be added to the set of S_P terms, as well as the 9 following “bracket”tuples:

Scope / Permission type	Write	Action	Read
Resource Group	<’RG’,’W’>	<’RG’,’A’>	<’RG’,’R’>
Subscription	<’SUB’,’W’>	<’SUB’,’A’>	<’SUB’,’R’>
Management Group	<’MG’,’W’>	<’MG’,’A’>	<’MG’,’R’>

The refined version of “@”, the horizontal congruence, is defined as such:

a is horizontally congruent to b iff a@b and h(a)=h(b)

We can check that it is reflexive, symmetric, and transitive, and that it is a proper refinement of @

In this set up, we could implement a desired silhouette 337 by saying that filter(<’RG’,’W’>,rg)=filter(<’RG’,’R’>,rg), for example. So <’RG’,’W’> and <’RG’,’R’> would be merged into the same equivalence classes but <’SUB’,’A’> wouldn’t.

Horizontal congruence features more equivalence classes than @ (because we used to have only one class for all resource groups, now we have potentially one for W+R, and one for A). and the equivalence classes are finer (they contain less members).

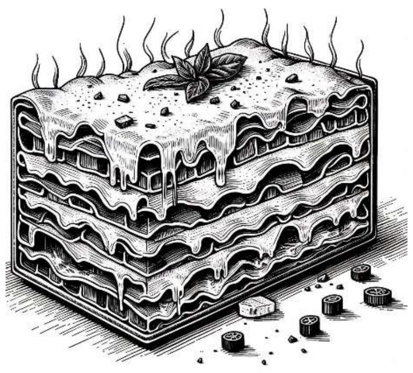
Note that h() is congruent with respect to this new relation.

Limitations of @ and horizontal congruence when upper silhouette bounds are too high

Using @ or the horizontal congruence with a desired silhouette standing above “resource group” is not guaranteed to meet least privileges criteria, because the grouping process will promote all permission to the maximum allowed upper bound scope.

Let’s keep our example of a desired silhouette set to 337. Here, horizontal congruence will merge all ‘Action’ permissions at subscription level, even though ground truth (Azure activity logs) only provide evidence that actions are used, say, in a single resource group.

Horizontal congruence is similar to layering a plate of lasagna. It may be called lasagna-congruence!



Lasagna horizontally sliced into multiple layers.

As of today, due to these security limitations, there is no plan to implement Horizontal congruence in Azure silhouette.

Condensates

A condensate is a convenient one-place representation of all Azure RBAC role definitions and role assignments granted to a cluster of similar SPNs.

It is the output of a role miner (an SMT solver), given the following inputs:

1. a desired silhouette as an input,
2. the Azure activity logs of a SPNs cluster, over a period.

In a condensate, each equivalence class corresponds to an Azure role: it contains all necessary and sufficient information to build the role definition and its multiple role assignments.

An Azure role definition is built by collecting all permissions found in the tuples of an equivalence class, discarding resource Id information.

Azure role assignments sustained by an Azure role definition are built by collecting scopes found in the tuples of the equivalence class, at the level determined by filter f().

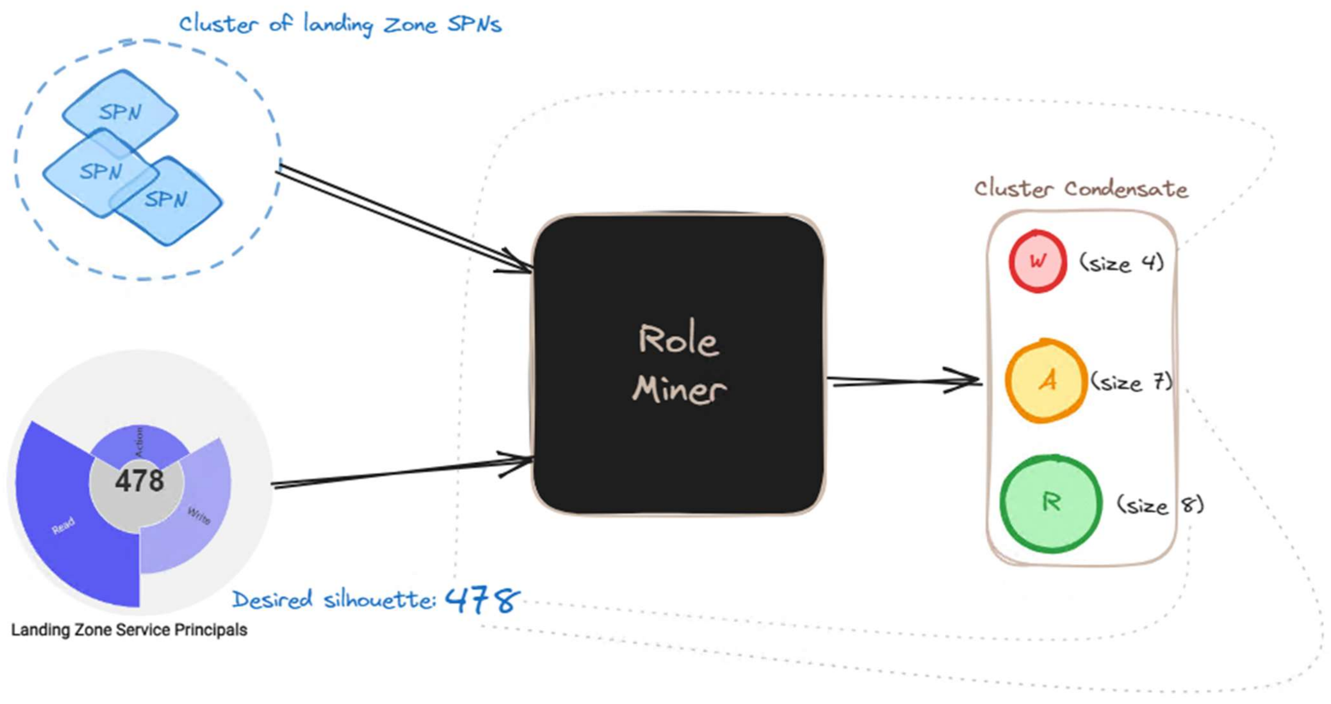
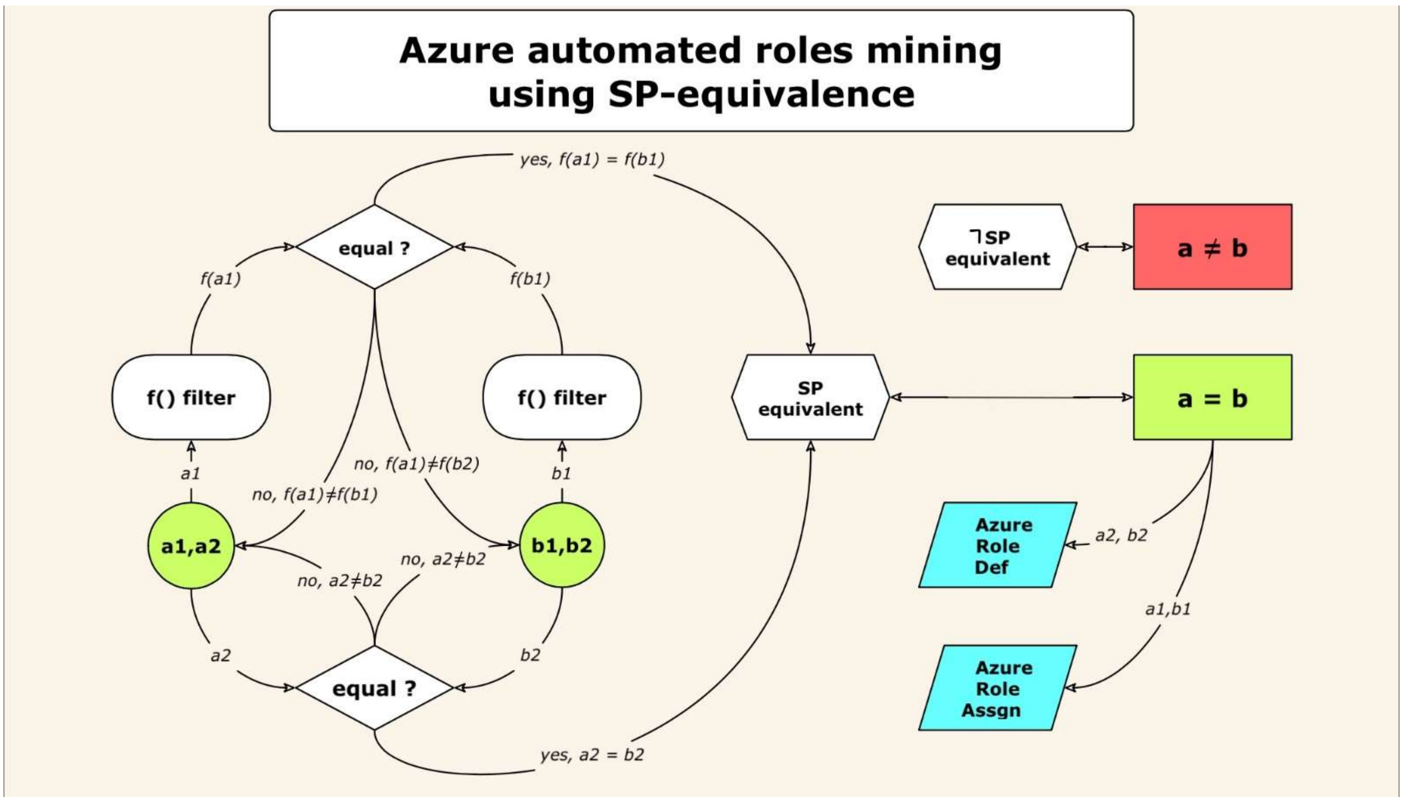


Figure 1: a condensate is the output of a role miner reasoning on two inputs: the Azure activity logs of a group of SPNs and a desired silhouette.

“@” generates a variable number of equivalence classes containing scoped permissions. Here is an example to see how this is done.

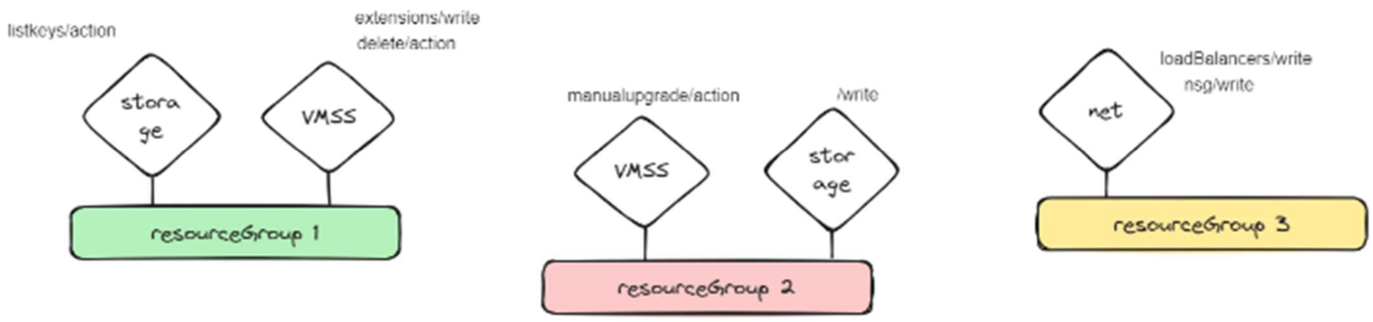
Here's a visual overview of condensation (the process of building up Azure role assignments and role definitions with SP-equivalence):



In this picture, a solver attempts to equate tuples a=(a1,a2) and b=(b1,b2) using SP-equivalence. If it works, both tuples will end up in the same azure role definition. They will also end up in an Azure role assignment, potentially the same but not necessarily.

Condensation in practice

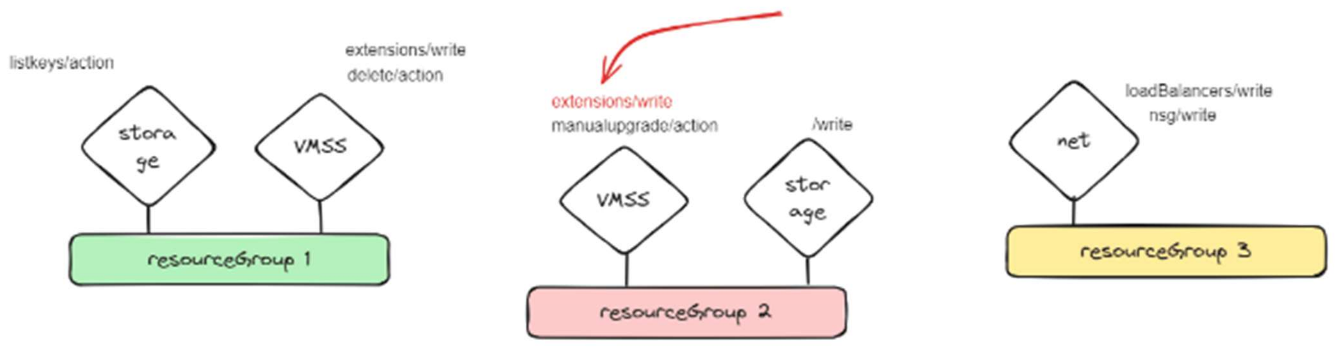
Suppose the solver has identified the following permissions at some point during the logs scan:



The solver has outlined 3 equivalence classes so far (resource providers omitted for brevity):

1. listkeys/action = extensions/write = delete/action = resourceGroup1
2. manualupgrade/action = /write = resourceGroup2
3. loadBalancers/write = nsg/write = resourceGroup3

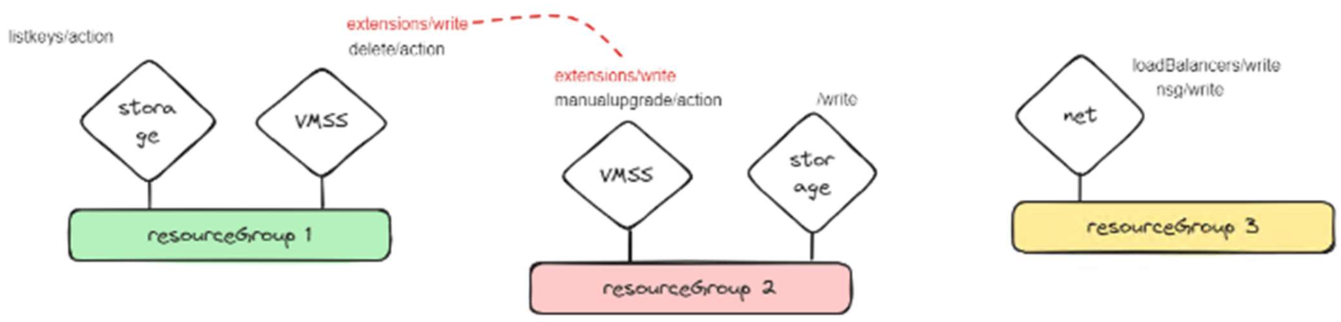
To illustrate the progressive nature of SP-equivalencing, suppose the scanner finds a new "extensions/write" permission assigned to resourceGroup2 (red arrow, below):



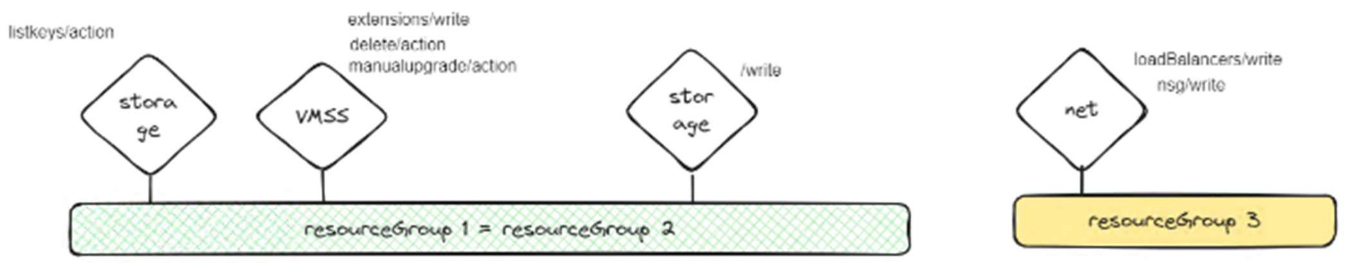
This results in the following equivalencing (to the right, in bold):

manualupgrade/action = /write = resourceGroup2 = **extension/write**

Since extensions/write is also part of resourceGroup1, SP-equivalence performs a **transitive merge** between resourceGroup1 and resourceGroup2:



Because of the merge, we end up with only two equivalence classes:



The two equivalence classes are:

1. listkeys/action = extensions/write = delete/action = manualupgrade/Action = resourceGroup1 = resourceGroup2
2. loadBalancers/write = nsg/write = resourceGroup3

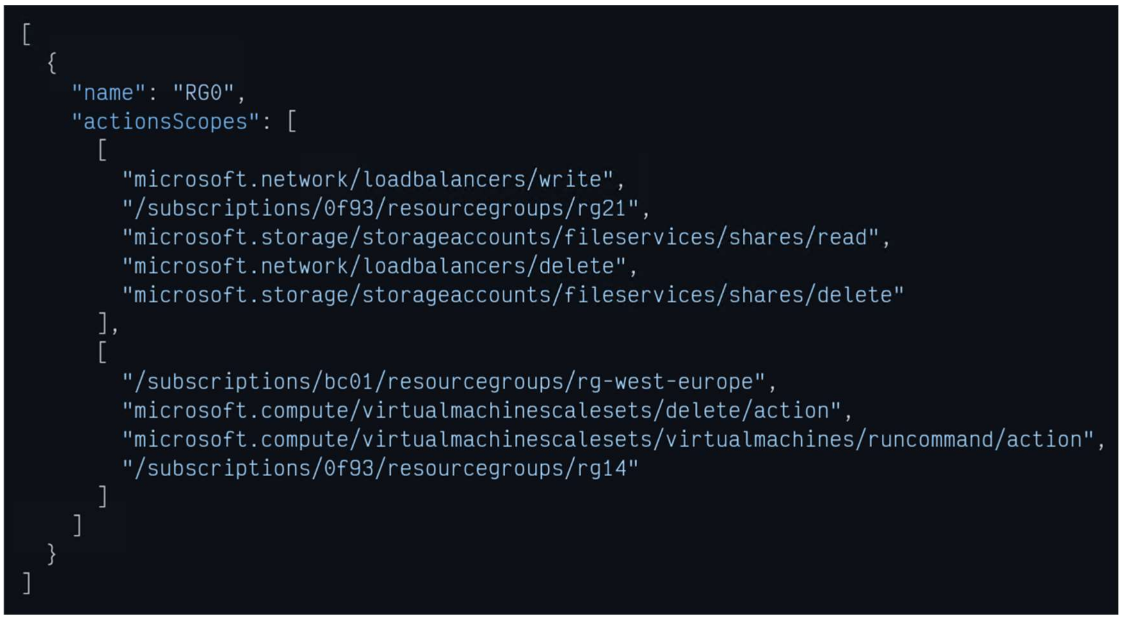
From condensates to Azure roles

To turn equivalence classes into actionable role assignments, some simple post-processing is required:

```
For each equivalence class in the condensate,
    for each scoped permission (s1,p1),
        for each s1,
            add p1 to the role assignment at scope s1 (create the role assignment if it doesn't exist)
```

Example

Here is a simple condensate in JSON format featuring two equivalence classes:



```json
[
  {
    "name": "RG0",
    "actionsScopes": [
      [
        "microsoft.network/loadbalancers/write",
        "/subscriptions/0f93/resourcegroups/rg21",
        "microsoft.storage/storageaccounts/fileservices/shares/read",
        "microsoft.network/loadbalancers/delete",
        "microsoft.storage/storageaccounts/fileservices/shares/delete"
      ],
      [
        "/subscriptions/bc01/resourcegroups/rg-west-europe",
        "microsoft.compute/virtualmachinescalesets/delete/action",
        "microsoft.compute/virtualmachinescalesets/virtualmachines/runcommand/action",
        "/subscriptions/0f93/resourcegroups/rg14"
      ]
    ]
  }
]
```

The first equivalence class contains delete and read operations. Also, it holds only one resource group from subscription "0f93".

The second equivalence class contains action ops and holds two resource groups: one from subscription "bc01", and one from "0f93".

To produce this condensate, the solver reasoned with a filter set at resource group level for Write and Action operations. Although we do not know for Read, we can reasonably suppose the solver reasoned without horizontal congruence. Since 'RG0' is the only special symbol mentioned in this condensate, we can also suppose that the solver reasoned without vertical congruence. So, it's likely that the solver resorted to vanilla SP-equivalence.

The reason why Write and Action operations are not merged into one and the same equivalence class is backed by ground truth: when scanning Azure activity logs, the solver could not find evidence that deleting a VMSS could happen in the same resource group as reading a shared file, for example.

See how scopes and resource groups are intermixed in each equivalence class, easing automated processing and human auditability.

Verification phase

Automation of condensates production saves a lot of time, but it is not completely error-free. As we've seen in the previous section, two kinds of least privilege violations may happen:

1) when setting a desired silhouette axe (W, A or R) at subscription level or management group level, permissions that could be pinned to resource group level will get an extended scope,
2) when permissions are used in very different contexts, scope-equivalence may merge unrelated resource containers by similarity, breaking segregation of duties. The outcome is a molten condensate.

While both problems may be solved by working on the root cause (e.g. lowering desired silhouettes in the first case, splitting SPNs by duties), the design change commanded to fix it might be too disruptive to implement as corporate scale.

If we can't work on the root cause, the automated roles will contain privileges violations that must be detected and mitigated. This requires security officers to audit all condensates as efficiently as possible.

Take the sample JSON condensate from the previous chapter and say you want to audit it. As a security officer, it might strike you that a cluster needs to create loadbalancers and read shared files. Depending on the organization it may make sense, or it might be a molten condensate.

To determine whether two anomalous scoped permissions are true positives or false positives, we need a way to understand why the solver took the decision to equal them under SP-equivalence.

The first part of this section will describe what we need to explain. The second part will cover how to explain.

Part 1: an analysis of equivalencing.

Equivalencing is a very generic process: our analysis is not dedicated to SP-equivalencing, it may apply to any equivalence relation.

When a solver analyzes an activity log, the algorithm used for SP-equivalence, called union-find, has three possible outcomes:

1) An equivalence class creation, to acknowledge a scoped permissions which cannot be matched with anything it's learnt so far. (That is to say, the scoped permissions "x" is only related to itself: x @ x by reflexivity)
2) An equivalence class growth, when the solver equalizes a scoped permission "x" with another scoped permission "y", then "x" is added to the class of "y": x @ y
3) A merger of two equivalence classes, when the solver realizes that a scoped permission "x" must be equalized to permission "y" in some class, yet "x" is already member of another class because it was previously equalized with "z": x @ y and x @ z implies y @ z (by symmetry and transitivity)

Obviously, class creations always precede growths or mergers, but fundamentally, the process is unordered.

Typical implementations of the union-find algorithm keep track of the equivalencing progress in an internal graph which is difficult to understand by a human reviewer. What's more, this graph can be opaque and not natively exportable.

To clarify things, let's trace the condensation process using a 2D grid, with time passing from left to right: the number terms (i.e. unique scoped permissions) scanned so far show up on the X axe, and the number of equivalence classes produced so far show up on the Y axe.

At t0, the solver has scanned 0 activity logs and has built 0 equivalence classes.

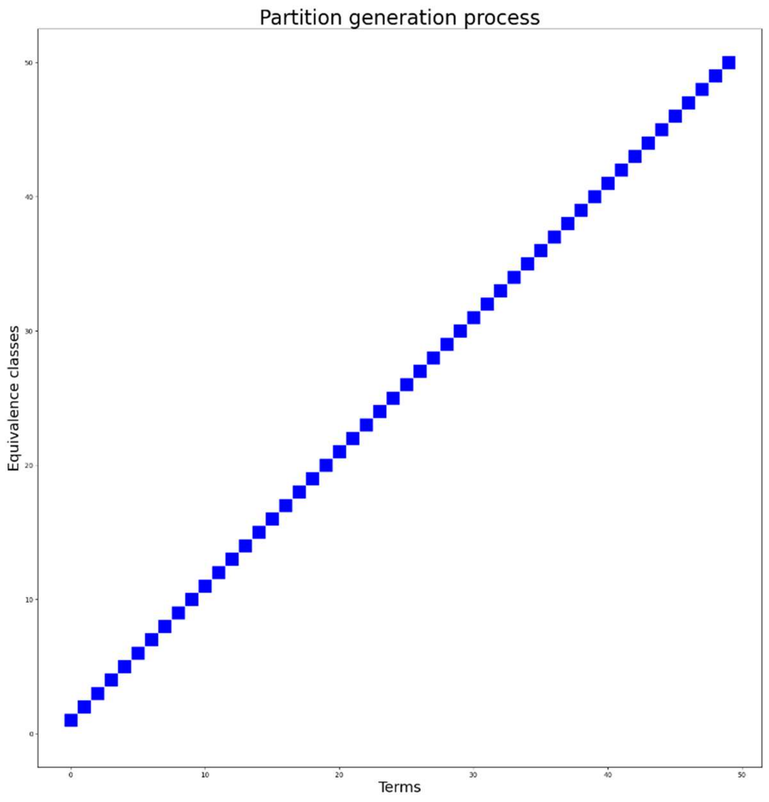
As time moves forward and logs get analyzed, creations, growths and mergers start to unwind.

1) creation operations increase the number of equivalence classes by one, so the walk moves up one step on the Y axe when X steps to the right,
2) growth operations have no effect on the number of equivalence classes, so Y remains unchanged when X steps to the right,
3) merger operations reduce the number of equivalence classes by one, so the walk moves down one step on the Y axe when X steps to the right.

Let's examine two extreme situations:

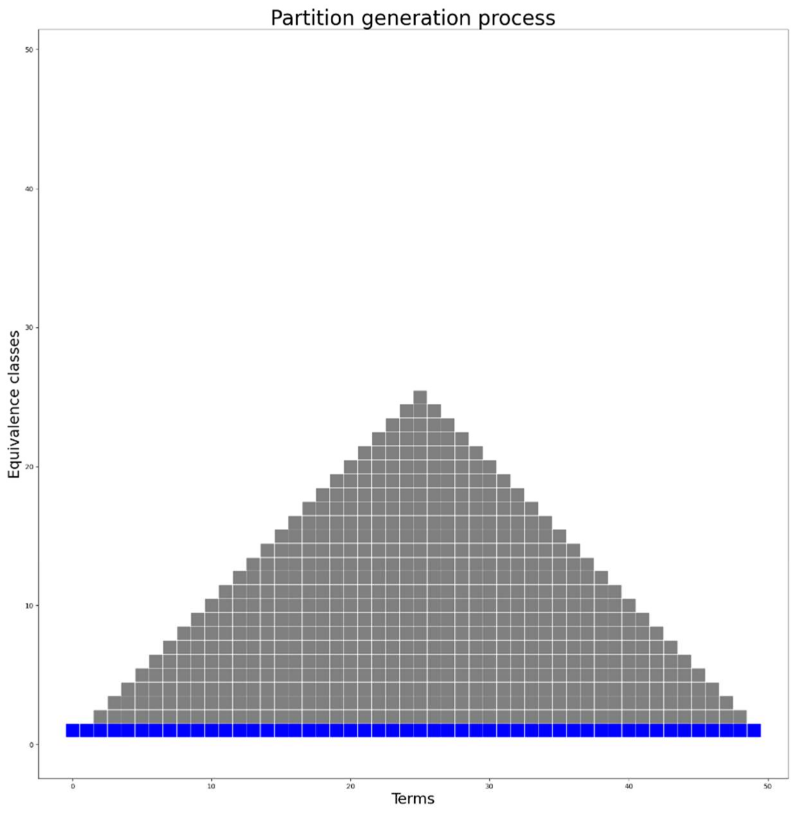
First situation: in a SPN cluster, every single term is segregated into its own equivalence class.

Here, the walk has no free choice but to perform creation ops exclusively, at each step. The walk will look like a diagonal:



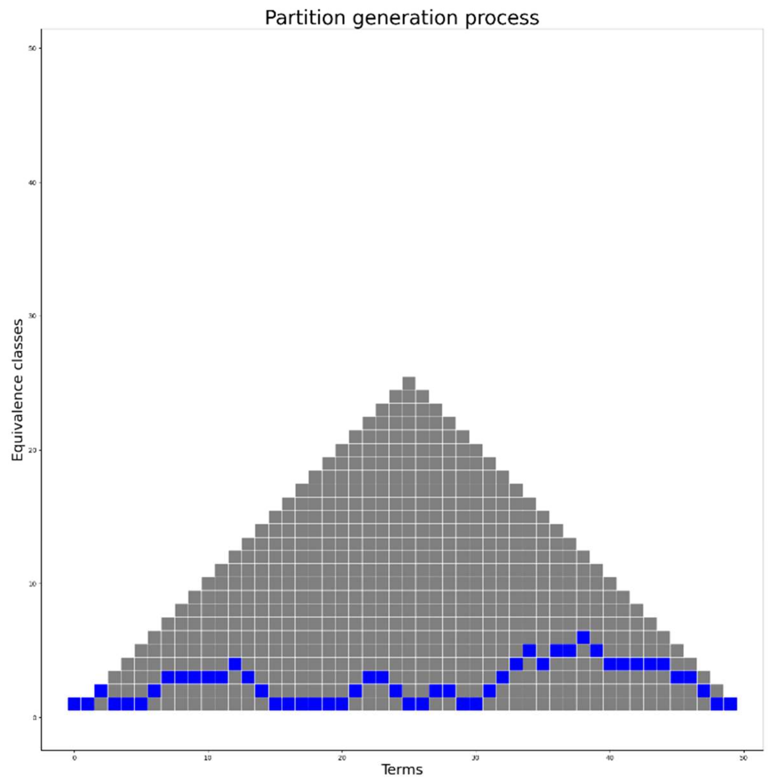
The second situation is opposite: all terms are grouped into a single equivalence class, right from the beginning of the scan and until the end.

It means that the walk performs just a single creation at t zero, and only merges subsequently. The walk (blue) will look flat:



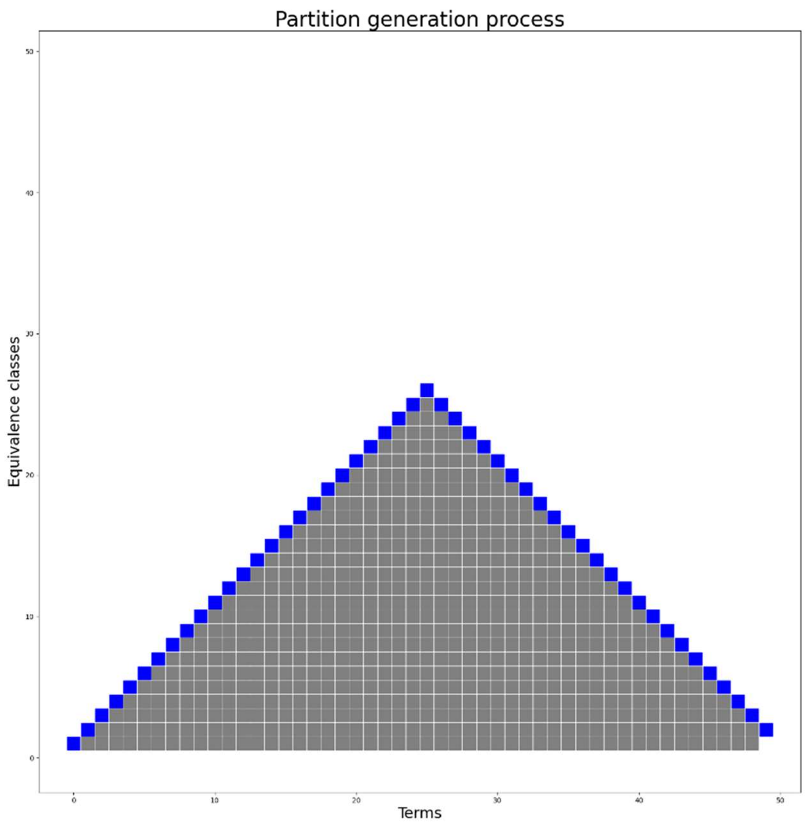
This second situation is very different from the first, because to get to the last square, the walk can behave wildly: any grey square can be taken along the way. Growths will slant the walk upwards, merges will slant it downwards. These grey squares define the convex envelope of all possible paths.

Here is an example of a walk ending up with all terms grouped into a single equivalence class:



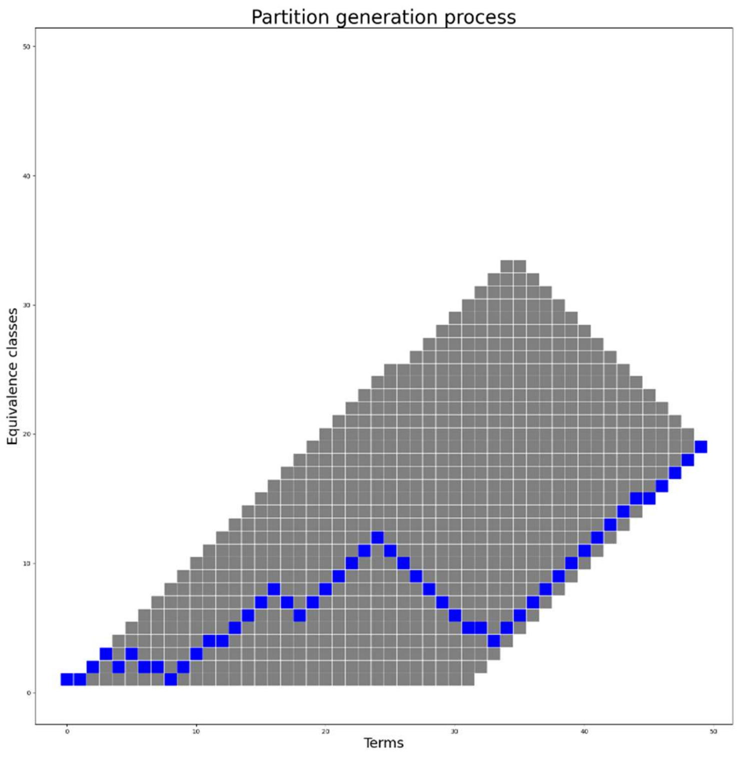
One can clearly see the zigzag trace left by “pseudo-random” sequences of growths and merges.

In fact, this pattern depends on the order by which the sorter processes activity logs from Azure. In the very unlikely case where all growths happen first and merges last, the pattern will look like a chevron:



The midpoint (the chevron’s tip) is a hard constraint imposed by the walk’s envelope: would the walk continue to move up diagonally, there wouldn’t be enough time (or, more accurately, enough terms) for it to “land” at the final square made of only one equivalence class.

Now that we have gained an understanding of extreme situations, we can ask what a typical walk, sitting in-between, looks like? The final square can now be located anywhere on the vertical axe, so the convex envelope of the walk doesn’t look like a triangle but a trapezoid (a quadrilateral with 2 parallel sides):



The 4 vertices of the trapezoid are:

1) the initial square (no terms, no equivalence classes)
2) an inflexion point required for a diagonal walk to turn to get to the final square “on time”
3) another inflexion point required for a horizontal walk to turn to get to the final square “on time”
4) the final square

It should become clear that extreme situations 1 and 2 are degenerated trapezoids:

- the diagonal line occurs when the initial square and second inflexion points are the same, and when the first inflexion point and final square are the same, yielding only two points
- the triangle occurs when the final square and second inflexion point are the same, yielding only three points

Part 2: feathers and condensates

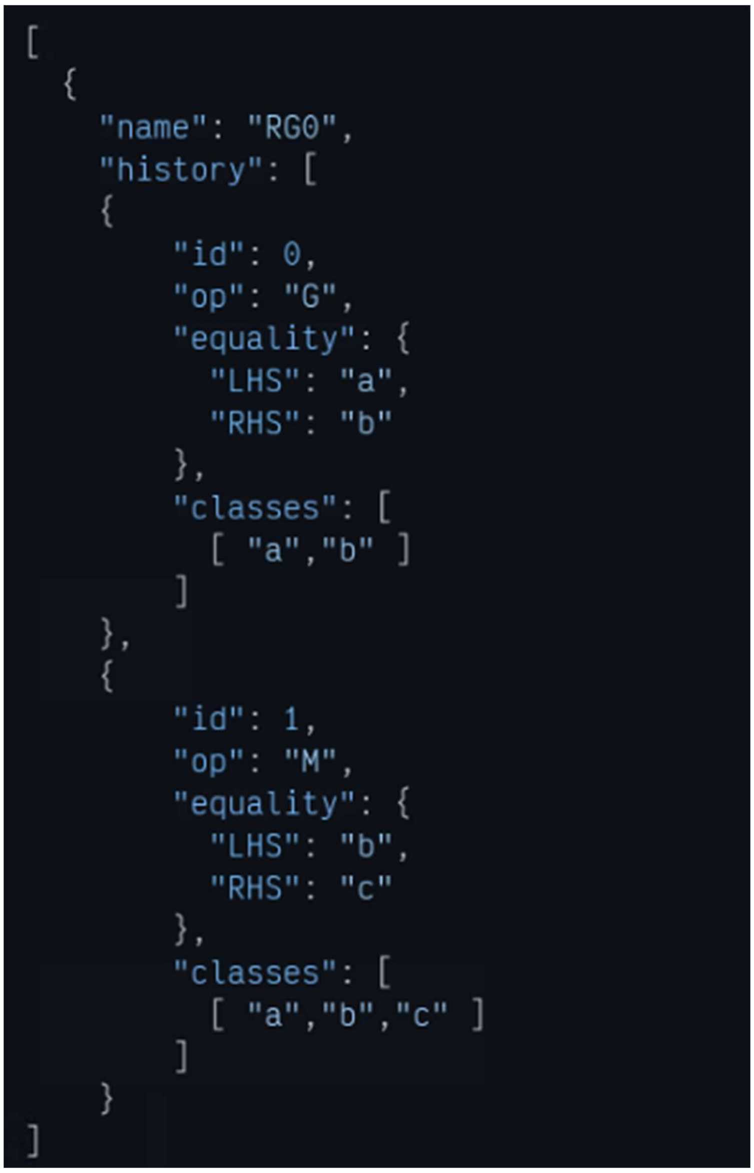
Let’s relate condensation to what we have just said: when we look at a condensate, we have access to the final square of its walk. We also have access to the initial square, obviously. From these two points, we can infer the shape of the (potentially degenerated) trapezoid.

The path in-between lies in the trapezoid’s grey convex envelope.

Suppose that we have a suspicion our condensate is actually molten, because we’ve identified to seemingly incompatible scoped permissions. To confirm our suspicion, we need to find a specific step in the pseudo-random walk that yields to the merge or the growth of the pair.

What is missing to accomplish this task? The condensate itself yields a trapezoid, but it doesn’t contain any walk. For that, we need to ask the solver to build a timeline of sequence of create/growth/merge operations. This timeline is called a feather (Format Explaining Automation by Tracing the History of Equivalencing in Reasoning)

Here is an example of a very “short” Feather:



```
[
  {
    "name": "RG0",
    "history": [
    {
        "id": 0,
        "op": "G",
        "equality": {
          "LHS": "a",
          "RHS": "b"
        },
        "classes": [
          [ "a","b" ]
        ]
    },
    {
        "id": 1,
        "op": "M",
        "equality": {
          "LHS": "b",
          "RHS": "c"
        },
        "classes": [
          [ "a","b","c" ]
        ]
    }
]
```

This feather relates to a set of resource groups belonging to the same segregation scope RG0. The timeline is made of only two entries:

- A growth operation 'G' at time 0: b is added to the equivalence class of a
- A merge operation 'M' at time 1: equivalence classes [a,b] and [c] are merged into a single class [a, b, c] because b is the same as c. The reason why c was equated to be might be that the permission c2 held in tuple c=(c1,c2) is the same as permission b2 in b=(b1,b2). Or because the resource groups c1 and b1 are identical and belong to the segregated scope RG0.

We have omitted creation operations, but they are typically present in a feather, denoted as operation 'C'.

So basically, a feather is an instance of a walk yielding to a condensate: the walk itself is characterized by a unique timeline of C, G and M operations, whereas a walk instance assigns specific terms to each operation (like "b" and "c" in the M operation example above).

A feather timeline corresponds to the very sequence chosen by the solver to produce the condensate, among all possible walk instances among all possible walks fitting within the envelope.

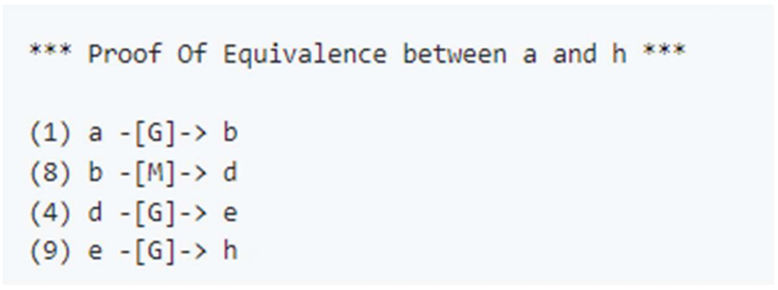
Part 3: investigation of molten condensates with proofs of equivalence

When a timeline is attached to every condensate, it is now possible to investigate any dubious SP-equivalences in a human-friendly way.

If a security reviewer identifies a couple of suspicious scoped permissions "a" and "b", one can explore the feather automatically to identify the unique sequence of equalities originating from "a" end terminating at "b".

This sequence is called a proof of equivalence.

Here is a real-world example:


```
*** Proof Of Equivalence between a and h ***

(1) a -[G]-> b
(8) b -[M]-> d
(4) d -[G]-> e
(9) e -[G]-> h
```

In this screenshot, the sequence of SP-equivalent equalities between a and h can be written a @ b @ d e @ h

The first equality, a @ b, comes from a 'G' operation occurring at time 1: "a" is added to the equivalence class of b.

The second equality, b @ d, comes from a 'M' operation. It involves two equivalence classes: a first one represented by b and a second one represented by d. The resulting merged class contains both b and d.

A human reviewer can easily determine if each point in the timeline corresponds to a semantically valid decision from the solver. If "a" and "h" shouldn't be equivalenced, there must be a point somewhere in the timeline that violates segregation of duties.

Part 4: tool for generating proof of equivalence

Zircuit is an open source tool that takes a feather, a couple of scoped permissions “a” and “b” as inputs and attempts to find a proof of equivalence joining them.

It is available from github: https://github.com/labyrinthinesecurity/zircuit

The following command will generate the proof of equivalence discussed in part 3:

```
./z.py --filename test_samples/test2.json --name RG0 --origin a --endpoint h
```

--filename is the name of a feather containing several timelines generated with the vertical congruence acting on several segregated resource group scope RG0, RG1, etc.

--name is the name of the resource scope to be used as a timeline

--origin and –endpoint define the scoped permissions we wish to investigate

Appendix: Azure RBAC mini world

RBAC mini-world is a formal verification tool aimed at analyzing the *consistency* of uninterpreted relation @, as well as its refined vertical variation. It reasons on a simplified version of Azure scoped permissions, where scopes are integers and permissions are strings.

Tuples, scope filter f() and relation @ are defined as Z3 uninterpreted functions:

IsTuple = Function('Is a tuple', scopes, permissions, BoolSort())

f = Function('scope filter', scopes, FLAT)

SP = Function('SP_equivalence', FLAT, FLAT, BoolSort())

f() takes "scopes" strings as input and strings in f()'s codomain 'FLAT' as output.

@ takes two strings in 'FLAT' as input, and a Boolean as ouput. The relation is True if and only if both strings are SP-equivalent.

The 3 axioms defining f() and @ are formalized as such:

Axiom 1: ForAll([a1,b1,a2,b2],Implies(And(a1 == b1,IsTuple(a1,a2),IsTuple(b1,b2)), f(a1) == f(b1)))

Axiom 2: ForAll([a1,b1,a2,b2],Implies(And(a2 == b2,IsTuple(a1,a2),IsTuple(b1,b2)), SP(f(a1), f(b1))))

Axiom 3: ForAll([a1,b1,a2,b2],Implies(And(f(a1) == f(b1),IsTuple(a1,a2),IsTuple(b1,b2)),SP(f(a1),f(b1))))

Analyzing properties of @

Mini-world proves that @ is reflexive by showing the following statement is UNSATisfiable:

IsTuple(a1,a2),Not(SP(f(a1),f(a1)))

It proves symmetry of @ when limited to axioms 2 and 3 by showing the following statement is UNSATisfiable:

Local symmetry induced by axiom 3: And(IsTuple(a1,a2),IsTuple(b1,b2), f(a1) == f(b1),SP(f(a1),f(b1)),Not(SP(f(b1),f(a1))))

Local symmetry induced by axiom 2: And(IsTuple(a1,a2),IsTuple(b1,b2),a2==b2, ,SP(f(a1),f(b1)),Not(SP(f(b1),f(a1))))

It proves non-symmetry of @ by finding SATisfable counter examples:

Non-symmetry: And(IsTuple(a1,a2),IsTuple(b1,b2),SP(f(a1),f(b1)),Not(SP(f(b1),f(a1))))

It proves intransitivity of @ by finding SATisfable counter examples:

And(IsTuple(a1,a2),IsTuple(b1,b2),IsTuple(c1,c2),SP(f(a1),f(b1)),SP(f(b1),f(c1)),Not(SP(f(a1),f(c1))))

Analyzing properties of ≡ with common prefixes (cylinders)

Recall that ≡ is a refinement of @ equipped with a congruent vertical slicing function v().

We simulate v() by resorting to common prefixes. Consider an alphabet Σ made of letters 0 to 9. Any word w in this alphabet belongs to Σ^*.

Segregated scopes are represented as cylinders. The cylinder of w is cyl(w)={ wx | x ∊ Σ^*}

We say that two resources groups belong to the same segregated scope 'RG0' is they are in cyl(10), they belong to 'RG1' if they are in cyl(20), and so on.

To implement cylinders in z3, we add the following axioms:

ForAll([a1,b1],Implies(And(IsTuple(a1,a2),IsTuple(b1,b2),a1>=10,a1<=19,b1>=10,b1<=19),SP(f(a1), f(b1))))

ForAll([a1,b1],Implies(And(IsTuple(a1,a2),IsTuple(b1,b2),a1>=20,a1<=29,b1>=20,b1<=29),SP(f(a1) , f(b1))))

ForAll([a1,b1],Implies(And(IsTuple(a1,a2),IsTuple(b1,b2),a1>=30,a1<=39,b1>=30,b1<=39),SP(f(a1) , f(b1))))

With these additional constraints, miniworld can reason on @ refined with v() in a simplified world of integer-named resource groups spread across three cylinders.

Azure RBAC mini world can be found in Azure silhouette's repository:

https://github.com/labyrinthinesecurity/silhouette/blob/main/RBACminiworld.py