How Stedi Uses Automated Reasoning for Access Control Policy Verification

Hye Woong Jeon, Olaf Conijn, Pavel Tcholakov*

January 17, 2023

Abstract

Stedi employees must be able to prove that customer data can never be accessed by Stedi-affiliated accounts or identities. Our in-house automated reasoning system, built on top of Microsoft's Z3 Satisfiability Modulo Theory (SMT) solver, encodes the semantics of the Amazon Web Services (AWS) Identity Access Management (IAM) Policy language, allowing it to reason about role-based access control. This system helps to provide assurance that customer data can only ever be accessed by its owner.

Keywords

Role-based access control, automated reasoning

1 Introduction

Stedi provides customers with software building blocks that can store and process data. Customers handling sensitive data would like to have definitive assurances about how their data can be accessed. Stedi would like to verify that its terms of service are upheld in our access controls. Specifically, we would like to show that customer data can only ever be accessed by the customer that owns the data, and never by an employee of Stedi. As Stedi Cloud is built on Amazon Web Services, the access controls are implemented using the AWS IAM Policy language. We can boil this down to the question: can a principal P perform action A on a given resource R? For Stedi, the principals are Stedi-affiliated identities, the actions are read-/write permissions, and the resources are customer-affiliated resources (Buckets, Functions, DynamoDB Tables, Logs & Roles).

1.1 Related work

Backes et al.[1] was our primary reference point, given that our problem statement concerns a subset of the IAM Policy universe. Consequently, we have adopted most of the encodings used in [1], and extended some of these encodings for our purposes e.g. Roles. Furthermore, [1] makes substantial contributions for encoding strings, as IAM Policies allow for both wildcard and regex expressions. However, Stedi does not use regex expressions in its IAM Policies, and we have taken a subtly different approach to encoding wildcards.

In contrast to [1], most of our paper's contributions derive from Role encodings and classifications of Roles depending on their permissions. In particular, we call the relationship between a Role R and a set of permissions P the *permission structure* of R and P. Our analyzer is able to reason about the permission structures of different varieties of Roles and permissions.

2 Approach

2.1 IAM overview

The AWS IAM Policy language, shown in Figure 1, is built around an object called a Statement. A Statement is a tuple that consists of a Principal, Action, Resource, Effect, and Condition, with Conditions an optional element. The Effect determines if the permission is to allow or deny, and the Principal, Action, and Resource determine who is allowed (or denied) to do which actions on what resources. Conditions further limit the scope of permissions by adding constraints surrounding the request context.[1] If Statements are the building blocks of the IAM Policy language, then Policies are its buildings; Policies are simply bundles of statements. Even further, Roles are bundles of Policies - though Roles do not have the same malleability that Policies do. Roles are a sub-class of Resources in which an identity can assume a Role, and hence be permitted

*Contact: hwjeon@uchicago.edu, olaf@stedi.com, pavel.tcholakov@stedi.com

[1]For our purposes, Conditions are outside of the scope of this paper, and Stedi plans to tackle these in due time.

to execute any permissions that the Role is allowed to perform.

IAM policies contain a sliding scale of permissions. Any permissions not directly written in the policy are *implicitly denied*. These implicit deny's are overwritten by policies directly allowed in the policy, otherwise known as explicit allows. Explicit allows are then overwritten by policies directly denied in the policy, similarly known as explicit deny's. This is to say that, in the IAM world, if action `s3:GetObject` is allowed *and* denied on all S3 Buckets, then the action is ultimately denied.

Taking a look at the example at Listing 1, we see that Principals that match with `principal1` and `principal2` are allowed to do any Action starting with `action` on any Resource starting with `resource`. However, no Principal is allowed to do Action `action2` on Resource `resource2`. Because explicit allows are always overwritten by explicit deny's, this means that Principals `principal1` and `principal2` are also not allowed to do `action2` on `resource2`. Every other permission not denoted in this example policy is implicitly denied. For example, no Principal is allowed to do Action `someAction` on Resource `someResource`.

Listing 1: Example Policy with two Statements

```
[
    {
        Action: action*,
        Resource: resource*,
        Principal: {
            AWS: principal1,
            Federated: principal2
        },
        Effect: Allow
    },
    {
        Action: action2,
        Resource: resource2,
        Principal: *,
        Effect: Deny
    }
]
```

2.2 Logical encoding

To reason about IAM security policies, Stedi uses an SMT solver called Z3. An SMT solver is an algorithm that determines the satisfiability of a propositional formula. For example, an SMT solver can take the following formula as input: $\varphi = (x_1 \vee x_2) \wedge \neg(x_2)$. A satisfying assignment for this example is $x_1 =$ `TRUE` and $x_2 =$ `FALSE`, since this assignment yields (`TRUE` $\vee$ `FALSE`) $\wedge \neg$(`FALSE`) = `TRUE` $\wedge$ `TRUE` = `TRUE`.

We define a few terms for precision. We say that a propositional formula φ is *satisfiable* if there exists an assignment of its variables such that φ evaluates to `TRUE`. Secondly, φ is called a *contradiction* if it is not satisfiable. Lastly, φ is *valid* if it evaluates to `TRUE` under all assignments of its variables. For example, $(x_1 \vee x_2) \wedge \neg(x_2)$ is satisfiable, $x_1 \wedge \neg(x_1)$ is a contradiction, and $\neg(x_1 \wedge \neg(x_1))$ is valid. Note that a formula is a contradiction if and only if its negation is valid. We use this fact liberally, as it is not enough to know that a formula is satisfiable, and we need absolute assurance that our policies function as we intend them to.

In words, we want to show that the formula `[Stedi accounts can access customer data.]` is a contradiction, or, equivalently, that `[NOT(Stedi accounts can access customer data)]` is valid. Unfortunately, Z3 cannot reason about formulas written in English, so we must encode the semantics of the IAM Policy language. Figure 1 shows the IAM Policy grammar and the corresponding encodings that we have used. Notice that Roles are encoded as large Policies; for each managed/inline Policy associated with the Role, we simply extract the allowed Statements and denied Statements, then put them all together.

There are two formulas that form the core of our automated reasoning system. Suppose we have two permissions called p_1 and p_2. The first formula is $\neg(p_1 \implies p_2)$, which we call the *allowed formula*. It asks if p_1 is less permissive than p_2 i.e. if p_2 is allowed to do anything that p_1 is allowed to do. From here on, we refer to the allowed formula $\neg(p_1 \implies p_2)$ with the notation `allowed(p1, p2)`. Accordingly, `allowed(p1, p2)` is `TRUE` if $\neg(p_1 \implies p_2)$ is unsatisfiable (i.e. a contradiction), and `FALSE` otherwise. We note a few observations. Firstly, it is largely the case that `allowed(p1, p2)` returns `TRUE` while `allowed(p2, p1)` returns `FALSE`, since one of `p1` or `p2` is likely to be more permissive than the other. Secondly, the allowed formula and its reverse can both be `TRUE`; this occurs if both policies entail exactly the same set of actions on the same set of resources, allowed by the same set of principals. Thirdly, the allowed formula and its reverse can also both be `FALSE`; this occurs if the set of permissions of `p1` intersects partially (i.e. is not a subset or superset) with the set of permissions of `p2`.

In our formulation of "allowedness", `p2` not being allowed to do the actions of `p1` is wildly different from `p2` being prohibited from doing the actions of `p1`. For instance, `p2` may be allowed to execute a strict subset of the permissions allowed by `p1`; `allowed(p1, p2)` would return `FALSE` since `p2` does not have all the permissions allowed by `p1`. However, it is also not the case that `p2` is prohibited from executing the permissions allowed by `p1`. This problem calls for a formula that checks for prohibition.

The second formula is $p_1 \wedge p_2$, which we call the *prohibitive formula*. It expresses if p_2 is prohibited from doing anything in p_1. Similarly, for the second formula, we check that $p_1 \wedge p_2$ is a contradiction if we are interested in proving that p_2 is prohibited from doing actions allowed by p_1 under all circumstances. From here on, we refer to the prohibitive formula as

Role = Policy*
Policy = Statement*
Statement = {Effect, PrincipalBlock?, ActionBlock, ResourceBlock}
PrincipalBlock = PrincipalType : PrincipalEntry*
PrincipalType = Principal | NotPrincipal
PrincipalEntry = PrincipalKey : string*
PrincipalKey = "AWS" | "Federated" | "Service" | "CanonicalUser"
ResourceBlock = ResourceType : string*
ResourceType = Resource|NotResource
ActionBlock = ActionType : string*
ActionType = Action | NotAction
Effect = "allow" | "deny"

$$\texttt{Role.encode} = \left(\bigvee_{\texttt{s}\in P(Allow)} \texttt{s.encode}\right) \wedge \neg \left(\bigvee_{\texttt{s}\in P(Deny)} \texttt{s.encode}\right)$$

$$\texttt{Policy.encode} = \left(\bigvee_{\texttt{s}\in S(Allow)} \texttt{s.encode}\right) \wedge \neg \left(\bigvee_{\texttt{s}\in S(Deny)} \texttt{s.encode}\right)$$

$$\texttt{Statement.encode} = \texttt{Principal.encode} \wedge \texttt{Resource.encode} \wedge \texttt{Action.encode}$$

$$\texttt{Principal.encode} = \begin{cases} \bigvee_{v\in P}(p=v) & \text{if } \texttt{PrincipalType} = \texttt{Principal} \\ \neg\left(\bigvee_{v\in P}(p=v)\right) & \text{if } \texttt{PrincipalType} = \texttt{NotPrincipal} \end{cases}$$

$$\texttt{Resource.encode} = \begin{cases} \bigvee_{v\in R}(r=v) & \text{if } \texttt{ResourceType} = \texttt{Resource} \\ \neg\left(\bigvee_{v\in R}(r=v)\right) & \text{if } \texttt{ResourceType} = \texttt{NotResource} \end{cases}$$

$$\texttt{Action.encode} = \begin{cases} \bigvee_{v\in A}(a=v) & \text{if } \texttt{ActionType} = \texttt{Action} \\ \neg\left(\bigvee_{v\in A}(a=v)\right) & \text{if } \texttt{ActionType} = \texttt{NotAction} \end{cases}$$

Figure 1: IAM Policy Grammar and its corresponding encoding

`prohibited(p1, p2)`. The function returns `TRUE` if the two policies are incompatible, and `FALSE` otherwise. Notice that this formula is symmetric: $p_1 \wedge p_2$ is equivalent to $p_2 \wedge p_1$. This is to say that $p_1 \wedge p_2$ is equivalent to putting together p_1 and p_2 into one large policy, and checking if there are any "collisions" between the statements.

Even with the prohibitive formula, we have not resolved the aforementioned problem of "not-allowed" and "prohibited". In fact, it is possible for `allowed(p1, p2)` and `prohibited(p1, p2)` to both return `FALSE` or both return `TRUE`. In this case, we call the permission structure of `p1` and `p2` *inconclusive*. Inconclusive policies where both `allowed(p1, p2)` and `prohibited(p1, p2)` return `FALSE` typically arise when the intersection between the two policies are partial i.e. one policy is neither a subset/superset nor completely separated from the other policy. On the other hand, both typically return `TRUE` in situations where "nothing" or "everything" is allowed. We expand on this particularity in the Examples subsection below.

In total, based on the allowed and prohibited formulas, we classify the permission structure of p_1 and p_2 into three categories: allowed, prohibited, and inconclusive. Representing `(allowed(p1, p2), prohibited(p1, p2)` as a tuple, the permission structure is allowed, prohibited, inconclusive if the tuple is `(TRUE, FALSE)`, `(FALSE, TRUE)`, `(TRUE, TRUE) / (FALSE, FALSE)`, respectively. The inquisitive reader may ask why the prohibited formula is even necessary, given that it did not solve the problem of "not-allowed" and "prohibited". Without the prohibited formula, we gain zero knowledge of whether the permission structure is prohibited or inconclusive. However, with the prohibited formula, we get further confidence of the correct permission structure.

At Stedi, the only identities that we use are Roles. This presents a challenging problem - a Stedi employee could assume a Role within a Stedi-affiliated account, then assume another Role within another account, then somehow end up assuming a Role inside a customer account that has read and write access to customer data. In order to circumvent this "graph search", Stedi replicates organization-level IAM policies to the customer local account. As such, we need only interpret customer-level IAM policies and check which Roles within the customer account have an "allowed" permission structure with respect to read/write permission policies. If we encounter any unexpected Roles (i.e. non-customer Roles) that have an "allowed" permission structure, then we will have identified a security risk. Roles that have an "inconclusive" permission structure are candidates for further investigation, while "prohibited" Roles are definitively proven to be denied access. Our automated reasoning system hence evaluates the customer local-level policies, and uses the resulting classification of Roles by their permission structures to further inform our IAM policy writing.

2.3 String encoding

The IAM Policy language makes rich use of wildcards and regular expressions. For wildcards, [1] uses a combination of prefix, suffix, and match operators. Because we only use wildcard expressions in our permissions, we take a simpler approach. The prefix-suffix approach was considered, but it was found to be too narrow in scope. For example, encoding `prefix*` or `*suffix` is quite simple; one need only check that the string in question has `prefix` and `suffix` as a prefix and suffix, respectively. However, for expressions with multiple wildcards, the matching becomes more complex. For example, using only prefix, suffix, and match operators on `s*s*s*s` may call `ss` a match, which is clearly incorrect. The problem arises from *overlaps*, since any overlaps between adjacent portions of the expression are treated as matches. Instead of having to exclude overlapped matches from the search space, we have opted for a simpler approach.

The Python API for Z3 provides several tools

to encode the wildcard expression; we utilize `Full(ReSort(StringSort))` for its representation. (An alternative encoding was considered, namely `Star(Range(chr(0), chr(127))`. However, this encoding performs magnitudes worse than the encoding written above.) We then construct regular expressions by concatenating our wildcard expression with other patterns. For example, suppose we wish to encode `s3:*`. This phrase is encoded as `InRe(Concat("s3:", Full(ReSort(StringSort))))`, where `InRe` is a regex constructor. Our method has the advantage of being simple to encode, while still being rigorous in pattern matching. However, this approach has performance disadvantages, as will be expanded on in the Evaluation section.

2.4 Examples

Listing 2: Straightforward policies

```
policy1 = {
    Action: s3:GetObject,
    Resource: *,
    Effect: Allow
}
policy2 = {
    Action: [s3:*, log:*],
    Resource: *,
    Effect: Allow
}
policy3 = {
    Action: [s3:GetObject, s3:PutObject],
    Resource: *,
    Effect: Deny
}
```

See Listing 2 for three straightforward policies. Clearly, `policy2` can do anything that `policy1` can do, and `policy2` is not prohibited from doing anything in `policy1`. We then expect `allowed(policy1, policy2)` to return `TRUE` and `prohibited(policy1, policy2` to return `FALSE`. For the opposite case, `policy3` cannot do what `policy1` is allowed to do. We then expect `allowed(policy1, policy3)` to return `FALSE` and `prohibited(policy1, policy3)` to return `TRUE`.

Listing 3: Implicit deny, explicit allow, explicit deny

```
policy1 = [
    {
        Action: action1,
        Resource: resource1,
        Effect: Allow
    },
    {
        Action: action1,
        Resource: resource1,
        Effect: Deny
    },
    {
        Action: action2,
        Resource: resource2,
        Effect: Allow
    }
]
policy2 = {
    Action: action1,
    Resource: resource1,
    Effect: Allow
}
policy3 = {
    Action: action2,
    Resource: resource2,
    Effect: Allow
}
policy3 = {
    Action: action3,
    Resource: resource3,
    Effect: Allow
}
```

See Listing 3 for examples of implicit deny, explicit allow, and explicit deny. Observe that `policy1` allows only Action `action2` on Resource `resource2`, and denies every other action-resource combination because of implicit deny's and explicit deny's. Accordingly, we then expect:

```
allowed(policy1, policy2) = FALSE
prohibited(policy1, policy2) = TRUE

allowed(policy1, policy3) = TRUE
prohibited(policy1, policy3) = FALSE

allowed(policy1, policy4) = FALSE
prohibited(policy1, policy4) = TRUE
```

See Listing 4 for an example of two policies that return `TRUE` for both `allowed` and `prohibited`. The first policy, `policy1`, does not allow anything; its set of permissions is the empty set. The second policy, `policy2`, allows everything; its set of permissions is the complement of the empty set. Clearly, `policy2` is allowed to do nothing, so `allowed(policy1, policy2)` return `TRUE`. There is also no contradiction in saying that `policy2` is prohibited from doing nothing; indeed, `prohibited(policy1, policy2)` returns `TRUE` accordingly. However, it is peculiar that `allowed(policy1, policy3)` and `prohibited(policy1, policy3)` both return `TRUE`. We do not expect `prohibited(policy1, policy3)` to return `TRUE`, since clearly `policy3` is prohibited from doing some Action on some Resource e.g. `action2` on `resource2`. Nonetheless, because the prohibitive formula essentially combines its two arguments into one large policy, combining `policy1` and `policy3` clearly create a formula that cannot be satisfied.

Listing 4: Both allowed and prohibited

```
policy1 = {
    Action: *,
    Resource: *,
    Effect: Deny
}
policy2 = {
    Action: *,
    Resource: *,
    Effect: Allow
}
```

```
policy3 = {
    Action: action1,
    Resource: resource1,
    Effect: Allow
}
```

Listing 4 also has an example of two policies that return `FALSE` for both `allowed` and `prohibited`. Notice that `policy3` is not allowed to do *everything* that `policy2` can do, but it is allowed to do some subset of `policy2`'s permissions. This "partial overlap" in permission spaces means that both `allowed(policy2, policy3)` and `prohibited(policy2, policy3)` return `FALSE`.

3 Evaluation

Our classification of roles by their permission structures have allowed us to make effective changes to our IAM policies. In particular, the inconclusive classification has led to the most improvements. When testing multiple Actions and Resources on Roles, most Roles do not fall neatly into the "allowed" and "prohibited" categories. Therefore, the inconclusive category allows us to make further adjustments to our policies, so that we get further assurance that our policies function as intended. For instance, we were interested in testing if any non-customer Roles are allowed to do Actions `s3:GetObject` and `s3:PutObject` on any Resource. The exact policy can be seen in Listing 5. We expected to see three "allowed" Roles, zero "inconclusive" Roles, and the rest with "prohibited" permission structures. Running the analyzer, the "allowed" Roles were returned as expected. However, there was an unexpected "inconclusive" Role; this Role was permitted to do `s3:Get*` on any Resource, but implicitly denied from doing `s3:PutObject`. Therefore, the analyzer correctly determined that there was insufficient evidence to make a definitive decision.

Listing 5: Tested policy for s3:GetObject and s3:PutObject

```
{
    Action: [s3:GetObject, s3:PutObject],
    Resource: *,
    Effect: Allow
}
```

As far as performance is concerned, the analyzer unfortunately leaves much left to be desired. In particular, the number of wildcards make a large impact on the analyzer's run-time - this is to be expected because every wildcard massively expands the search space. Fortunately, for our purposes, we do not expect to encounter more than four wildcards.

3.1 Statistics

For the purposes of collecting performance statistics on our analyzer, we created a IAM policy generator that generates IAM policies of different lengths and complexities. To control the complexity, we parametrize different components of the IAM Policy grammar. As seen, every Policy is a series of Statements, and every Statement contains an Effect and three similarly defined blocks (Action, Resource, and Principal blocks, which we call ARP blocks). Each ARP block contains a series of strings. Therefore, we define three parameters for the generator: the number of statements within each policy, the number of strings within each ARP block, and the number of characters within each string. Throughout this section, we represent these parameters as a tuple: (`numStatements`, `numStrings`, `stringLength`). For example, if we set our parameters to be $(1, 1, 5)$, then the output consists of policies with one statement and one string of length 5 within each of the ARP blocks.

Using randomly selected parameter values, we verified one thousand policies of varying lengths and complexities. Only the time it takes to run `allowed()` for each policy was measured (i.e. no reading time, no parsing time, etc). The top figure of Figure 2 shows the aggregate verification times for all trials. The bottom figure shows the aggregate verification times for trials under 264 milliseconds. Table 1 shows descriptive statistics - notice that 90% of cases are verified in less than 264 milliseconds. However, even in just one thousand trials, there are cases that take more than 10 seconds to verify, and we have encountered cases that take a minute or more.

To further investigate how string lengths affect verification times, we generated one hundred policies - each containing one statement with ARP blocks housing a wildcard-less string of length n, where $n \in \{10, 20, \cdots, 500\}$ (i.e. parameters $(1, 1, n)$). The results of running `allowed()` on these policies can be seen in Figure 3. As string length increases, the variance on verification times increases also, but the distribution is not smooth; several gaps can be seen, but all the distinct "bands" appear to grow at similar rates.

Furthermore, we studied how the number of statements within a policy affect verification times. Here, one hundred policies were generated, with each policy containing $n \in \{10, 20, \cdots, 200\}$ statements with ARP blocks housing a wildcard-less string of length 10 (i.e. $(n, 1, 10)$) . The benchmarking results can be seen in Figure 4. As the number of statements increases, the verification time increases as expected; given how our analyzer constructs propositional formulae, the number of components in the formula is a direct function of how many statements there are.

Finally, in order to examine the effect of wildcards, we added a wildcard parameter to the generator. Three hundred policies with generator parameters $(1, 5, 10)$ were created; zero, one, or two wildcard expressions were then randomly spliced into the strings, such that there would be one hundred policies each that contain zero, one, or two wildcards. Again, as can be seen in

Figure 5, the results are as expected - as the number of wildcards grow, the variance on verification times increases since the search space dramatically expands.

4 Conclusion

In this paper, we have introduced how Stedi encodes the IAM Policy language, and how these encodings are placed into propositional formulas to reveal gaps in security. Because of Stedi's exclusive use of Roles (as opposed to Users or other identity types), we have taken a Role-first approach, and investigated the different degrees to which Roles can be permissive. Furthermore, because we replicate organization-level IAM policies to the customer account-local level, Stedi customers can be assured that their data cannot be accessed by any Roles or identities other than themselves.

There are a number of improvements and developments that can be made. Firstly, we wish for more information regarding inconclusiveness. Currently, when a Role is determined to be inconclusive, no further information is provided. We would like to know which sections of the Role's policies induce the inconclusiveness; this would aid Stedi employees in creating robust policies without having to resort to a guess-and-check workflow. Secondly, we wish to bring resource-based policies into the domain; our analyzer can reason only about identity-based policies. However, a resource can be shared with identities outside of the containing account, and, as such, this requires our analyzer to reason about resources also. Fortunately, we expect no difficulties incorporating resource-based reasoning on the basis of our existing work. Lastly, as aforementioned in the Evaluation section, optimizing the analyzer (especially around wildcards) will conserve time greatly as the analyzer's usage becomes more frequent.

References

[1] Backes J, Bolignano P, Cook B, Dodge C, Gacek A, Luckow K, et al. Semantic-based automated reasoning for AWS access policies using SMT. IEEE; 2018.

A Figures

Statistic	Details (ms)
Minimum	40.131
Maximum	12643.917
Mean	207.105
Standard deviation	744.177
50th percentile	57.648
90th percentile	264.873

Table 1: Descriptive statistics for verification times

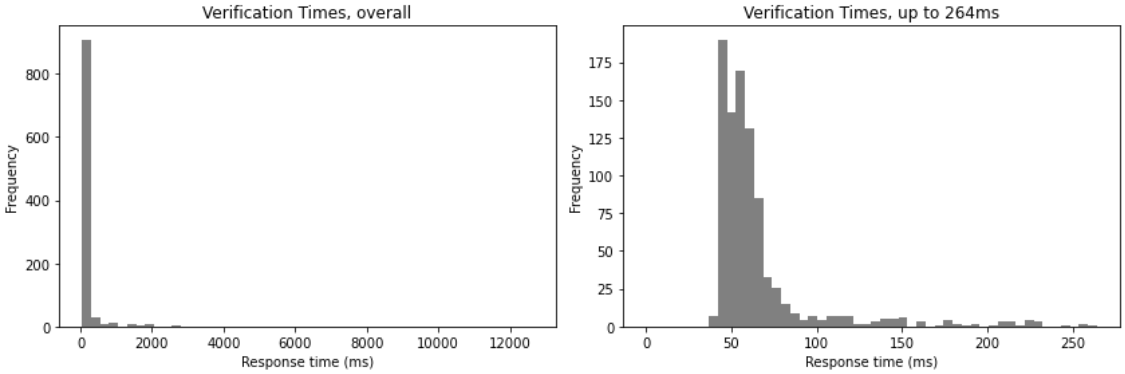

Figure 2: Verification times from 1,000 randomly generated policies

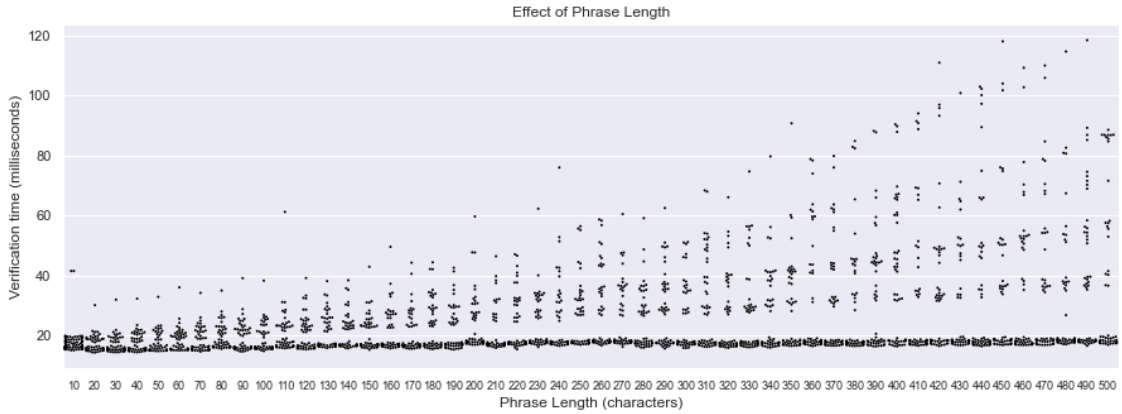

Figure 3: Verification times for policies with varying string lengths

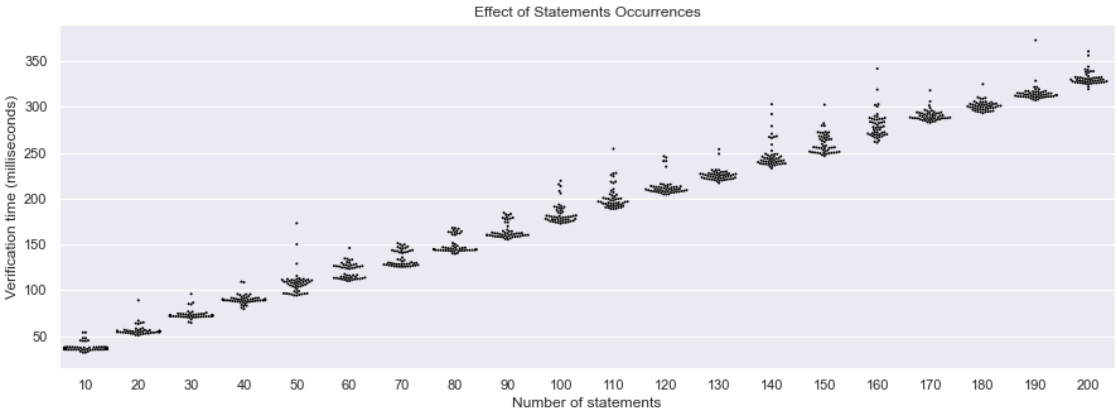

Figure 4: Verification times for policies with varying numbers of statements

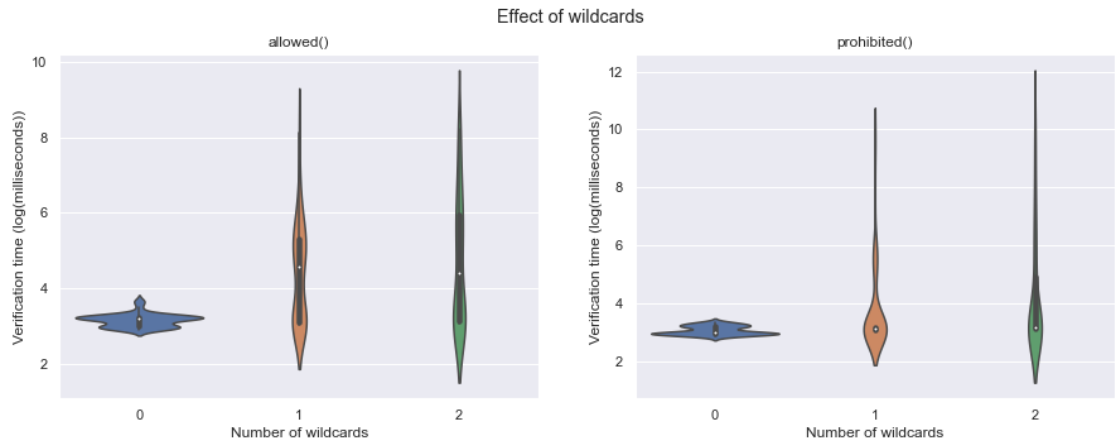

Figure 5: Violinplot of verification times with 0, 1, or 2 wildcard expressions