Using ILP to Learn AppArmor Policies

Lukas Brodschelm [1], Marcus Gelderie[1]
[1]*Aalen Univeristy of Applied Sciences, Beethovenstr. 1,73430 Aalen, Germany*
{*firstname.lastname*}*@hs-aalen.de*

Keywords: ILP, Inductive Logic Programming, AppArmor, Access Control, Access Control Policies

Abstract: Access control has become ubiquitous in contemporary computer systems but creating policies is an costly and errorprone task, thus it is desirable to automize it. Machine learning is a common tool to automate such tasks. But typical modern machine learning (ML) techniques require large example sets and do not give guarantees which makes it hard to learn policies with them. Inductive logic programming (ILP) is a symbolic form of ML that addresses these limitations. We show how ILP can be used to create generalized file access policies from examples. To do so we introduce two strategies to use the ILASP ILP framework to create file access rulesets for AppArmor. Further, we introduce concepts to generate negative examples for the learning tasks. Our evaluation shows the feasibility of our strategies by comparing them with AppArmor's default tooling.

1 INTRODUCTION

Access control systems are ubiquitous in contemporary computer systems. From file access rules to process permissions like capabilities to permissions in distributed systems, there are various forms of access control systems (Anderson, 2020). What they all have in common is the use of some sort of policy to describe the permissions that are granted to subjects.

The policy description language (PDL), and what a policy expresses depends on the system. Especially for large and complex systems, defining a policy is a time-intensive and error-prone task (Nobi et al., 2022). Hence, automating this task seems desirable and ML (Machine Learning) is applied to do so. However, generating policies with typical ML methods poses two significant challenges. First, common ML approaches often require large example sets. Yet, for many policy application domains such policy data sets either do not exist or are not publicly available (see, e.g., (Nobi et al., 2022) for a survey on the topic). Second, many common ML techniques do not offer hard guarantees on the properties of the policies they generate. Modern ML relies heavily on probabilistic methods which usually fail to give robust guarantees on the behavior of its output in every edge case (Rechkemmer and Yin, 2022). However, especially for policies, certainty that some actions are always (or never) allowed is desirable.

Inductive Logic Programming (ILP) (Muggleton, 1991; Cropper and Dumančić, 2022) is an symbolic ML technique that addresses these challenges, therefore it was already used in contemporary research to learn policies (Law et al., 2020a; Calo et al., 2019; Cunnington et al., 2019b). ILP relies on logic, rather than statistics which leads to rather different strengths and weaknesses from those seen in ML systems built on neural networks (Cropper and Dumančić, 2022).

The present paper takes the first step in using ILP to generate access control policies for common mandatory access control (MAC) systems. We use the ILASP (Law et al., 2014) ILP framework to learn file access policies for AppArmor. To investigate how ILP can be used to learn such policies, we introduce and evaluate two distinct strategies for applying ILP to policy learning. The first strategy aims to craft a learning task in such a way that the resulting solutions can be directly transpiled to AppArmor's PDL, while the second approach learnes pattern that can be applied to distinct parts of the file system (FS). We then compare the two strategies in terms of the policies they generate and the resources they require. To do so, we introduce a method for generating examples, and in particular negative examples, for an arbitrary application. Finally, we learn policies for a custom test application and compare the strategies with AppArmor's `aa-logprof`.

We choose to use a custom test application, rather than a real-world application or even benchmark suites. This has several reasons: First, as we will show below, the current ILP frameworks do not scale to the complexities involved in writing policy for most

real-world applications. This is particularly true for the first learning strategy. In order to compare our two strategies in terms of the quality of the policies they produce, we therefore need to stick to small applications for now. Second, for most real-world applications we run into at least one of the following problems: (A) There is no comprehensive set of examples and no easy way to automatically create them (indeed, this is an active area of research; see Related Work in section 3). (B) There is no reference policy to compare with. Even if such a policy exists, it is often written with a very particular use case in mind (e.g. running an nginx for a chat server inside a snap container on Ubuntu). By using a custom application, we can ensure that we can verify the learned policy against the actual behavior of the application.

2 PRELIMINARIES

Terminology for logic programming is used in the remainder of this work. We assume basic familiarity with logic programming concepts, such as *predicates*, *constant symbols*, *arity*, and *literals*. For an overview, the reader is referred to (Apt et al., 1997). Further we discuss *rules*, each rule has the form `head :- body`, where the head applies if the condition in the body is true. Rules without a body, i.e., those who allways apply are called *facts*.

A *model* of a program is a set of ground facts involving the predicate symbols and constants occurring in the program that "satisfies every rule". An *answer set* (AS) is a very particular kind of model; one which satisfies a certain minimality property. Logic programming with AS semantics is called *Answer Set Programming* (ASP) (Lifschitz, 2019), and it is the basis for the ILASP framework (described below) that we use in this paper. Note that ASP is different from the better known Prolog, which works in a fundamentally different way. However, the details are not relevant to this paper. It is noteworthy that a logic program may have many possible ASs in general. We cannot give a full introduction to ASP in this paper and refer the reader to (Lifschitz, 2019). However, no in-depth knowledge beyond what we introduced here should be necessary.

Inductive Logic Programming (ILP) is a symbolic Machine Learning approach, that first became popular in the 1990s but is an ongoing research topic (Muggleton, 1991; Cropper and Dumančić, 2022). There are multiple modern ILP frameworks (for an overview see (Cropper and Dumančić, 2022)). While most ML systems are based on statistics, ILP is based on first-order logic. This is the main reason, why many ILP systems, can learn meaningful rules from small example sets (Law et al., 2014; Cropper and Dumančić, 2022). In this paper, we use an ILP framework called ILASP (Law et al., 2014) because it is publicly available and well-documented.

ILP tasks typically involve three input components (Cropper and Dumančić, 2022): (i) A background knowledge B, describing the FS structure. (ii) A search space S, defining the structure of the inducted rules. (iii) Positive E^+ and negative E^- examples, consisting of facts that should be true, respectively false. For these input components the ILP framework searches an hypothesis H that: (i) is contained in the provided search space $H \subseteq S$. (ii) respects all given examples, positive ($\forall e \in E^+ : e$ is *true* in the program $H \cup B$) and negative ($\forall e \in E^- : e$ is *false* in the program $H \cup B$) (iii) is optimal in a framework-specific metric. It is common that ILP frameworks aim for the shortest hypothesis (e.g. (Law et al., 2014)). Since the three components are used together to induct a program they are not independent from each other. This especially applies to the used symbols.

Inductive Learning from Answer Sets (ILASP) is an ILP framework first introduced in 2014 by Mark Law (Law et al., 2014; Law et al., 2020b). While most ILP frameworks learn Prolog programs, ILASP learns answer set programs (ASPs). As mentioned before, an ASP can have multiple ASs. ILASP ensures that the output ASP is such that (a) for *every* positive example, there is at least one AS of the program that *covers* it and (b) no AS covers *any* of the negative examples. The first point coincides with *brave* learning, while the second point coincides with *cautious* learning. ILASP incorporates both.

The notion of *covering* an example is subtle. Each ILASP example consists of a set of *includes* (facts that the AS covering the example must contain) and *excludes* (facts that the AS covering the example must *not* contain). So even a positive example can *prevent* certain facts from appearing in the AS that covers it. As we will explain below, we do not use the excludes.

AppArmor is a MAC system for the Linux kernel (Wright et al., 2002). Next to SELinux (Smalley et al., 2001), AppArmor is one of the two popular MAC systems for Linux (Zhu and Gehrmann, 2021). Like all MAC systems, it provides a system-wide policy. Unlike SELinux, an AppArmor policy works based on file-paths and is specific to an application.

In this paper, we focus on AppArmor PDL file access rules, they follow the pattern: `<owner flag>`

<path> <access modes>. AppArmor follows a default-deny approach, so each rule typically *grants* access to a file. Rules, containing wildcards *, or ** apply to all files matching the wildcard. While * matches only files of the current folder, ** applies recursively to all files and folders below. AppArmor's *abstractions* allow to define and include predifined sets of rules into the policy. Predefined abstractions exist for common usecases, e.g., console access.

Of particular interest to this work, AppArmor provides a set of utilities to aid in policy generation. It supports a *complain mode* that can be used to audit if a program would violate a policy without actually enforcing it. Moreover, AppArmor provides `aa-logprof`, which can be used to generate policies from these logs. We use it in our evaluation to compare our approach with more established tooling.

Threat Model: To clarify our assumption about the abilities of the attacker, we state our threat model. We assume that the adversary is able to compromise an application *A* at some point, to the effect that she can perform any task within the process running *A* (subject to the policy and other OS security features constraining *A*). In particular, we assume, that during a first example gathering stage, the application *A* is trusted and that any file access the application makes during that stage is acceptable.

3 RELATED WORK

There has been prior research on automated policy generation for AppArmor. Lic-Sec (Zhu and Gehrmann, 2021; Zhu et al., 2023) is a cloud tool that automatically generates AppArmor profiles for Docker. Lic-Sec combines LiCShield (Mattetti et al., 2015) and Docker-sec (Loukidis-Andreou et al., 2018) to trace applications and generate corresponding AppArmor profiles. Similarly, Kub-Sec (Zhu and Gehrmann, 2022) is a utility that generates AppArmor policies for Kubernetes. Different from our work all these tools focus on tracing activities inside a container, and generate a policy from that input. By contrast we focus we focus on learning *generalized* policies. The *negative* examples we use for our learning task can not be obtained by tracing, which is only suited to generate positive examples.

ASPgen (Li et al., 2020) and its docker-specific twin ASPgen-D (Huang et al., 2022) are AppArmor-specific frameworks that generate rules using an expert system. Both are based on a role based access control (RBAC) system, for which file accesses are assigned to roles. This may be viewed as extending AppArmor with an RBAC, which is rather different from our approach, that intends to learn generalizations without applying a given access control scheme.

ILP, and especially ILASP has been used in policy learning a few times (Law et al., 2020a; Cunnington et al., 2019b; Calo et al., 2019; Bertino et al., 2019; Drozdov et al., 2021; Cunnington et al., 2019a). However, there are general differences to our approach. First, these approaches neither target MAC systems like AppArmor, nor they deal with a tree representation of a file system. Instead those solutions learn policies that depend on given attributes or roles. To do so they use prelabeled data sets, where attributes or roles are assigned to entities. Neither AppArmor's PDL nor the Linux FS provide an obvious way to express such context information. Thus, learning rules depending on the tree structure of the FS makes perfectly sense, but bears rather different challenges. Further, the other approaches learn from different kinds of example sets, in general they are larger, noisy, or observed over time. This different from this work, where we use a small example set for a single application. Finally our learned rules are translated into the AppArmor PDL, which is rather limited in its feature set. This brings different challenges from the other approaches, where either the inducted program is used as PDL or a more power full generic PDL is used.

4 LEARNING APPARMOR POLICIES

Below, we outline two strategies on how ILASP can be applied to policy learning. But first, we introduce some common building blocks.

In the background knowledge of the ILASP task we provide a tree representation of the Linux FS. We represent the tree using the predicates `fso` and `parent`. `fso(fso1)` declares the existence of an file system object (FSO) with the symbol "fso1". The predicate `parent(fso1, fso2)` declares that "fso1" is the parent of "fso2". Using such parent declarations, the complete FS tree structure is described. We do not distinguish between files and folders and refer to both only as FSO. Aside the file system we include rules for the `ancestor` predicate, that is required to express wildcards of arbitrary depth, i.e., the ** wild card. This is defined recursively with the rules `ancestor(A, B) :- parent(A, B)` and `ancestor(A, B) :- ancestor(A, C), ancestor(C, B).`

Next, we briefly sketch how examples in ILASP work and how we use them (for a detailed explana-

Listing 1: Small Example Declaration

```
#pos({read(x2Ff1),write(x2Ff1)},{})
#neg({read(x2Ff2)},{})
#neg({write(x2F)},{})
```

Listing 2: Example Rules

```
execute(V1) :- ancestor(x2Fbin,V1).
read(V1) :- parent(x2Fvar,V2),
    ↪ parent(V2,V1).
```

tion, see (Law et al., 2014)). listing 1 shows several such examples. There are *positive* and *negative* examples. Each example, whether positive or negative, consists of up to three sets, an *include set*, an *exclude set*, and a *context* (example-specific background knowledge) (Law et al., 2014; Law et al., 2020b). We neither use excludes nor contexts and rely only on the include set to express our examples.

Recall that ILASP treats positive and negative examples different. For positive examples it requires that at least one AS of the resulting hypothesis covers it. Thus, we use a single positive example that contains every allowed file access to ensures that at least one AS of the learned hypothesis contains all facts. Using multiple positive examples would theoretically allow solutions where, multiple AS exist but none of them covers all positive accesses. We use a negative example per denied file access since for negative examples ILASP ensures that no AS contains all includes of *any* negative example. Thus, using an example per access ensures that none access is contained in any AS. We do not use the exclude set, nor the context as they are not required for our approach to model the learning task.

The file accesses, whether positive or negative, are expressed as logical atoms of the form `mode(fsoSymbol).` Where `mode` is one of the supported access modes and `fsoSymbol` is a constant symbol of type `fso` that is defined in the background knowledge. We learn rules for the three access modes read, write, and execute. In addition, AppArmor supports the *owner* flag, which reflects that the process must be the owner of the resource. To embed this information in the learning task, owner-specific variations of the access modes are used. Thus in total, the learning task deals with six access modes, read, write, execute, and their owner-specific variants.

Although the two strategies described below use different search spaces, they rely on the same search space structure. We want to learn rules of the form `accessMode(V) :- condition` that can easily be expressed with AppArmor's PDL.

4.1 GENERATING EXAMPLES

Just like AppArmor's aa-logprof we utilize the complain mode to obtain positive example accesses. Please note that covering all features of an application might be challenging, but this is considered to be out of scope. The examples we obtain from the log contain: a) the accessed file, b) the used access mode, c) if the file is owned by the accessing process.

The examples are stored as a tree structure, called *example tree*:, a labeled tree where the nodes represent FSOs labeled with a list of allowed access modes and explicitly denied access modes. Of course, up to now, no denied accesses have been recorded.

To extend the example tree with negative examples, we rely on a human supervisor. This is done via an interactive dialog, that is rather similar to what *aa-logprof* does. However, to minimize the number of asked questions our script only suggests *relevant* files, that are likely to make a difference in the resulting policy. A negative example for an access mode M is considered *relevant* for a sub-tree T if, within T, M is not used as a negative example for any node, but is listed as a positive example for at least one node. Of course, this approach requires that the filesystem contains the same files during learning as during deployment.

4.2 APPROACH 1: ASP AS POLICY

The first strategy we present is to induct one hypothesis that can be directly translated into an AppArmor policy. We dub this the "ASP as policy". This means, that the ILASP task should learn an ASP program that expresses the policy. To do so, the search space must contain rules that allow access depending on generic patterns and constant symbols. Recall that we define two predicates, `parent` and `ancestor`, that can be used in the rule body. These predicates dictate what patterns can be discerned. Both are already used in the background knowledge and have arity 2. The rule head contains the access mode which is an 1-ary predicate. Examples for such rules are shown in listing 2.

Remark 1. Predicates other than `parent` or `ancestor` are conceivable, e.g. `is_devicefile`. We choose to omit them for now, as the resulting ILASP task will not scale well (as explained below in section 5).

ILASP supports a short hand declaration of the search space called *mode bias*, but we define the search space explicitly. This allows to optimize the size of the search space for performance. As further

discussed in the evaluation (c.f. section 5), for "ASP as policy" the performance is crucial.

To create the search space, we combine all required rule heads (access modes) with all rule bodies. While there are only 6 rule heads, there infinite thinkable combinations for the rule body. To limit the size of the search space, we limit the length of the rule bodies to five predicates. The rule bodies are combinations of the `ancestor` and `parent` predicate. To connect these predicates, as many variable names as needed are introduced. Constraining the length like this puts a limit on the length of rules like `allow /a/*/*/*/*`. We believe this is a reasonable limit.

To reference a certain FSO within a policy rule we require constant symbols. The symbols used here are the same, already defined in the background knowledge. Since the symbols reference FSO objects using an absolute path, we can limit the amount of constant symbols to one per rule, as referencing two would not make sense. E.g., consider rules like `/etc/app.conf*/var/app/** rw`. This leads to a search space, that grows *linear* to the amount of defined FSO constant symbols.

Remark 2. The choice to use absolute paths and reference FSOs limits the rules that can be learned. We are not able to learn rules like `/etc/*.conf` that use wild cards in the middle of the rule. To be able to learn such rules, requires additional characteristic predicates and further increases the size of the search space, we leave this open to future work.

For the "ASP as Policy" approach, a single learning task with all example file accesses is performed. The examples are derived from the example tree, that is already introduced in section 4.1. Each node in this tree contains a set of positive and negative access modes, we add a node specific negative example for each element of the negative set and a positive example for each element of the positive set. There will be combinations of access mode and node, for which no example is added.

The output of this learning task is an ASP program that defines which FSOs can be access under which mode. It is straightforward translated into the AppArmor PDL. For example, the program in listing 2 would translate to listing 3. The predicate `parent` is translated into the "*", while `ancestor` is represented with "**". When there are two rules dealing with the same path but with divergent access modes they are merged into a single AppArmor line.

An issue with this approach is, that the search space scales with the FSOs. For large policies with many FSOs this leads to scalability issues, since the size of the search space is crucial for the complexity of the learning task. Remembering that the inducted hypothesis is a subset of the search space, points out why, since choosing a subset of a set equals selecting an entry of its power set, and the size of the power set scales exponentially to the size of the set. Further, ILASP aims for a short hypothesis and becomes less efficient if a longer hypothesis is required (Law et al., 2014).

Listing 3: Translated Example Policy

```
/bin/** x
/var/* r
```

4.3 APPROACH 2: ASP AS PATTERN

The "ASP as Policy" approach constructs one monolithic learning task that yields one ASP describing the entire policy. While this strategy is conceptually simple, this approach does not scale well (cf., section 5). We therefore introduce a second strategy that addresses some of these problems. For the second strategy, which we dub "ASP as Pattern", we utilize ILASP to learn patterns only for sub-trees of the example tree and later combine them into one coherent policy. In particular, we run multiple ILASP tasks of considerably smaller size.

For the "ASP as Pattern" strategy, we construct a search space that does not contain any constant symbols that are referring to a certain file or folder. Policies that are inducted with such a space describe relational patterns only but do not refer to any base folder (e.g. `read(X) :- parent(Y,Z),parent(Z,X).`). If applied to the entire FS, they will most likely be too permissive: We would obtain generic patterns like `/*/*` which are of very limited use. Instead, we would like to obtain rules of the form `/etc/ * r`. We will explain below how to remedy this. For now, note that the search space now does not contain any references to filenames, thus it has *constant* size in the number of filenames.

ILASP always tries to compute solutions that cover all examples. With a given hypothesis space that lacks constant symbols (and therefore facts), this may be impossible. In this case, ILASP will recognize the learning task as unsatisfiable. We now introduce an algorithm that nevertheless learns complete policies that cover all examples.

We use the same file access tree as before in section 4.2. The algorithm starts at the root of the tree. It runs an ILASP pattern learning task using all available examples. If this learning task is successful, it prefixes the resulting patterns with / and has found a valid policy. Otherwise, the task is unsatisfiable and the algorithm splits up the tree. It creates a sub-tree

for each child of the root node and recursively works on those sub-trees. Eventually, either a pattern for the sub-tree is found or we reach a leaf node. Learning patterns for a leaf node does not make sense. But leaf nodes are trivial to cover with a specific rule, like "allow `/etc/passwd`". Moreover, we know that no pattern could have subsumed this explicit rule, or ILASP would have already found it.

The learned patterns are stored with the nodes for which they are learned. After all patterns are learned, they are translated to AppArmor rules. To do so the file patterns are prefixed with the path of the corresponding node. This limits the scope of the pattern to a certain sub-tree. For now the patterns only consist of the parent and ancestor relations, thus all rules obtained using this concept start with a file path and optionally end with wild cards. However, by extending the learning task to support further characteristics more sophisticated pattern would be learned.

5 EVALUATION

The following evaluation aims to compare the policies that are learned by the two approaches described above and to compare the results with policies that can be obtained by `aa-logprof`. We focus on showing the strengths and weaknesses of our strategies.

5.1 METHODOLOGY

To evaluate the introduced learning strategies, we generate AppArmor policy rules for a custom target application using both our strategies and the AppArmor utility `aa-logprof` to compare ourselves with an established tool. We then compare the resulting policies, and the creation process itself, using the evaluation criteria described in section 5.2.

For the evaluation we introduce a benchmark application. Compared to using an existing application this brings two major advantages: First, this avoids the issue of obtaining complete example sets. Second, it allows to construct an example that allows us to verify that patterns can be learned, without consisting of too many file accesses. Unfortunately, the "ASP as Policy" strategy from section 4.2 scales rather poorly and consumes enormous amounts of memory (e.g., attempting to learn policies for `apache2` failed, because the learning task consumed more than 200 GiB of RAM). Thus, using an application with a reduced scope allows us to nevertheless compare with the results of this strategy.

The benchmark application accesses 10 files in total, while most of the files are accessed in the users home directory it also reads two configuration files at `/etc/` and executes the binary `/bin/true`. It is meant to be run by a regular Linux desktop user, with a home directory that is owned by that user.

Consistent with our threat model (c.f., section 2), all accesses the evaluation application makes are granted and included as positive example accesses. The negative examples require some supervisor interaction. The supervisor answers them according to the following decisions: a) Read and write for all files in `$HOME/Documents/` is allowed. b) Read access for all files in `$HOME`, but not for files in `$HOME/.config` is allowed. c) Read access to `$HOME/.config/testapp`, and the folders below it is allowed. d) Read and write access to `$HOME/.config/testapp/log` is allowed. e) No further access is granted apart from the files that are used by the application. The resulting example data set consists of 12 positive and 6 negative examples.

5.2 Evaluation criteria

Different quality criteria for access control policies (Beckerle and Martucci, 2013; Bertino et al., 2019) are used. But in general they make similar claims. In this work we group these quality criteria into three categories that are relevant for our evaluation.

Correctness: A policy must reflect the permissions described in section 5. It must neither allow further access nor deny required accesses.

Understandability: A common claim is that a policy should neither contain redundancy nor contradictions (Beckerle and Martucci, 2013; Bertino et al., 2019). However, neither ILASP nor `aa-logprof` are likely to include this. Thus we only focus on the length of policy, the rule set should be as concise as possible.

Adaptability: AppArmor policies, should contain wild cards so less is not necessary to modify them, when files are added or removed.

In addition we also consider the computational resources required. We do not conduct a performance benchmark for the learning tasks, but note significant differences in runtime and memory consumption and highlight scalability differences between the policy generation methods.

5.3 Evaluation results

aa-logprof: Different from the strategies introduced in this work, `aa-logprof` does not learn a policy. Thus, the profile generation with "aa-logprof" does not require notable computation time or consume relevant amounts of main memory.

The policy generated by `aa-logprof` is correct,

except for the fact that the "nameservice" abstraction includes additional files that are not part of our definition. However, `aa-logprof` asks a human supervisor before it introduces an abstraction.

The profile generated by `aa-logprof` is nine lines long. Among these lines, one is an abstraction that covers the access to `/etc/passwd` and `/etc/nsswitch.conf`. The other lines allowing access to a required FSO in the user's home directory each. For all of these rules logprof replaced the name of the user with a `*` wild card. E.g., `/home/*/file4`.

"ASP as Policy": As already mentioned, the "ASP as Policy" learning strategy requires more resources and suffers from scalability issues. The learning task for the provided example application took hours to compute and required more than 100 GiB of main memory. The resulting ASP program allows FSO specific access for most examples. Further, it uses the rule `ownerMread(V1) :- parent(x2Fhomex2Ftux78, V1).` to allow owner read access to all files directly in the users home directory and the rule `ownerMread(V1) :- ancestor(x2Fhomex2Ftux78x2FDocuments, V1).` to allow owner read access to all files below the documents directory.

The rules fulfill our correctness definition and consists of ten rules in total. This is longer than the profile generated by `aa-logprof`. The reason for this is, that the learning task learns disjunct rules for each access mode and it is not optimized to learn rules, that can be merged.

Note that the learned rules do not replace the username with a wild card, which makes the rules specific to a user. This is a clear adaptability flaw. However, `aa-logprof` applies a static rule to achieve this, an future version could enhance learned rules by applying such static modifications.

A major advantage of the learned rule set is its use of wild cards below the user's home directory. This is a massive adaptability advantage as it is to be expected that files under `Documents` will change frequently. Note that the rules for write access remain file-specific. Thus there is potential for further improvements.

"ASP as Pattern": Unlike the first strategy, this strategy invokes multiple learning tasks. As already explained, these learning tasks are often unsatisfiable, however they do not require much time to compute. For our example application the complete learning process was done in about one to two minutes. Further it did not show the memory issues we observed for the "ASP as Policy" strategy.

The profile obtained with this strategy has eight lines in total, which is shorter than the other profiles. Further, this rule set allows read and write access to all files in the `Documents` directory. This is the only contained generalization.

Comparing the learning tasks of both strategies it is reveals why this strategy introduces a generalization for both access modes, while the other strategy does not. The "ASP as Pattern" strategy uses patterns, i.e., wild cards where possible. But the "ASP as Policy" strategy uses them only if they lead to a shorter hypothesis. Compared with the other learning strategy, the rule set of this strategy is missing the wild card at the user's home directory. This can be explained by how the learning task is created. A learning task must find a solution that covers a complete tree below a certain folder. Patterns of smaller scope can not be found. A possible approach to improve this might be to run multiple learning tasks per node, each of them considering only a certain depth below the current node. We leave this for future work.

Just like the profile obtained with "ASP as Policy", the rules in the profile are specific to a certain user, i.e., the user's home directory is not replaced with a wild card.

6 CONCLUSION

In this work, we show two novel strategies to utilize the ILASP ILP framework to learn AppArmor policies. Doing so we have shown how to learn file access policies with ILP and that it is possible to use ILP to learn generic rules from very small example sets.

Our work indicates that scalability is a major challenge when we learn policies with ILASP. The "ASP as Policy" strategy we present does not scale for more complex applications. The "ASP as Pattern" strategy, on the other hand, appears to scale rather well. This strategy splits up the file tree and conducts smaller learning tasks, which, for performance reasons, seems to be necessary when utilizing ILASP to learn file access policies. While the "ASP as Policy" strategy can not be applied to real world scenarios, the pattern based strategy clearly has the potential to be used with common applications. However, still the complexities inherent in ILP mean that great care must be taken when crafting learning tasks.

Further, the evaluation indicates that learned AppArmor policies can be shorter than those created using the AppArmor board utility `aa-logprof`. But more importantly, we observe that a learned policy contains other generalizations than those that are introduced by aa-logprof. Thus, to obtain a policy that is as adaptable as possible, it seems to make sense to combine rule-based generalizations (as used by

`aa-logprof`) with learned policies. We leave this to future work.

Future work should consider combinations of rule-based generalizations and learned policies. Addressing limitations of the introduced strategies, such as the inability to generalize home directories, also seems promising. Moreover, expanding beyond AppArmor file access policies seems a natural next step. Thus, future work should extend the scope and consider further application domains, e.g., other MAC systems like SELinux, or completely different applications like web-APIs or firewall rules.

REFERENCES

Anderson, R. (2020). *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons.

Apt, K. R. et al. (1997). *From logic programming to Prolog*, volume 362. Prentice Hall London.

Beckerle, M. and Martucci, L. A. (2013). Formal definitions for usable access control rule sets from goals to metrics. In *Proceedings of the ninth symposium on usable privacy and security*.

Bertino, E., Russo, A., Law, M., Calo, S., Manotas, I., Verma, D., Jabal, A. A., Cunnington, D., de Mel, G., White, G., et al. (2019). Generative policies for coalition systems-a symbolic learning framework. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

Calo, S., Manotas, I., de Mel, G., Cunnington, D., Law, M., Verma, D., Russo, A., and Bertino, E. (2019). Agenp: An asgrammar-based generative policy framework. *Policy-based autonomic data governance*.

Cropper, A. and Dumančić, S. (2022). Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, 74.

Cunnington, D., Law, M., Russo, A., Bertino, E., and Calo, S. (2019a). Towards a neural-symbolic generative policy model. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE.

Cunnington, D., Manotas, I., Law, M., de Mel, G., Calo, S., Bertino, E., and Russo, A. (2019b). A generative policy model for connected and autonomous vehicles. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE.

Drozdov, A., Law, M., Lobo, J., Russo, A., and Don, M. W. (2021). Online symbolic learning of policies for explainable security. In *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE.

Huang, C., Wang, K., Li, Y., Li, J., and Liao, Q. (2022). Aspgen-d: Automatically generating fine-grained apparmor policies for docker. In *2022 IEEE ISPA/BDCloud/SocialCom/SustainCom*. IEEE.

Law, M., Russo, A., Bertino, E., Broda, K., and Lobo, J. (2020a). Fastlas: scalable inductive logic programming incorporating domain-specific optimisation criteria. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34.

Law, M., Russo, A., and Broda, K. (2014). Inductive learning of answer set programs. In *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings 14*. Springer.

Law, M., Russo, A., and Broda, K. (2020b). The ilasp system for inductive learning of answer set programs. *arXiv preprint arXiv:2005.00904*.

Li, Y., Huang, C., Yuan, L., Ding, Y., and Cheng, H. (2020). Aspgen: an automatic security policy generating framework for apparmor. In *2020 IEEE ISPA/BDCloud/SocialCom/SustainCom*. IEEE.

Lifschitz, V. (2019). *Answer set programming*. Springer Heidelberg.

Loukidis-Andreou, F., Giannakopoulos, I., Doka, K., and Koziris, N. (2018). Docker-sec: A fully automated container security enhancement mechanism. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

Mattetti, M., Shulman-Peleg, A., Allouche, Y., Corradi, A., Dolev, S., and Foschini, L. (2015). Securing the infrastructure and the workloads of linux containers. In *2015 IEEE Conference on Communications and Network Security (CNS)*. IEEE.

Muggleton, S. (1991). Inductive logic programming. *New generation computing*, 8.

Nobi, M. N., Gupta, M., Praharaj, L., Abdelsalam, M., Krishnan, R., and Sandhu, R. (2022). Machine learning in access control: A taxonomy and survey. *arXiv preprint arXiv:2207.01739*.

Rechkemmer, A. and Yin, M. (2022). When confidence meets accuracy: Exploring the effects of multiple performance indicators on trust in machine learning models. In *Proceedings of the 2022 chi conference on human factors in computing systems*.

Smalley, S., Vance, C., and Salamon, W. (2001). Implementing selinux as a linux security module. *NAI Labs Report*, 1(43).

Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. (2002). Linux security modules: General security support for the linux kernel. In *11th USENIX Security Symposium (USENIX Security 02)*.

Zhu, H. and Gehrmann, C. (2021). Lic-sec: an enhanced apparmor docker security profile generator. *Journal of Information Security and Applications*, 61.

Zhu, H. and Gehrmann, C. (2022). Kub-sec, an automatic kubernetes cluster apparmor profile generation engine. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE.

Zhu, H., Gehrmann, C., and Roth, P. (2023). Access security policy generation for containers as a cloud service. *SN Computer Science*, 4(6).