Logic, Leaves, and Labels: Visualizing SELinux Trees with ASP Chef

Mario Alviano*, Pietro Macrì

DeMaCS, University of Calabria, 87036 Rende (CS), Italy

Abstract

Visualizing security policies is no easy task, especially when they come from the labyrinthine world of SELinux. In this article, we explore how ASP Chef, a logic programming platform for structured reasoning and templated output, can be enhanced with tree visualization capabilities via the integration of the ApexTree framework. By introducing a new `tree/2` functor within the Mustache-based templating system of ASP Chef, we enable logic-based generation of structured tree data, seamlessly translatable into the JSON format of ApexTree. As a concrete and security-relevant use case, we focus on SELinux policy rules, which define access control through 4-tuples of subjects, objects, classes, and permissions. We demonstrate how to extract, filter, and interpret meaningful fragments of these policies using ASP and Lua scripting within ASP Chef. Nodes and links are programmatically generated to construct readable trees, enabling intuitive inspection of policy paths from user roles to allowed actions. We further enhance interpretability by integrating LLMs into the workflow, automatically generating natural language descriptions of the policy logic. The result is an expressive and interactive visualization of SELinux permissions.

Keywords

Answer Set Programming, ASP Chef, policy visualization, security policies

1. Introduction

Answer Set Programming (ASP) is a well-established declarative paradigm for solving complex combinatorial problems, rooted in nonmonotonic logic and stable model semantics [1, 2, 3, 4, 5]. Thanks to its expressiveness and robust reasoning capabilities, ASP has been successfully employed across domains such as security policy analysis [6] and configuration [7]. Despite the growing availability of efficient solvers like CLINGO [8], integrating ASP with user-friendly visual representations remains a practical challenge, particularly in domains that involve rich hierarchical structures. To bridge this gap, ASP Chef [9, 10] was introduced as a web-based environment for building interactive pipelines (recipes) over ASP computations. ASP Chef enables users to ground, solve, filter, and visualize answer sets through a browser-based interface that supports both logic-based reasoning and web-ready content generation. In particular, recent developments have introduced template-driven output generation via Mustache templates [11], allowing ASP users to declaratively render structured outputs in formats such as JSON, Markdown, or CSV, without the need for glue code or external scripts.

Among the visualization libraries now supported in ASP Chef is ApexTree (https://apexcharts.com/apextree/), a modern and lightweight JavaScript framework designed for rendering interactive tree structures from JSON-based specifications. ApexTree excels in presenting hierarchical data in a clear and dynamic way, supporting features such as collapsible branches, customizable labels, and animated transitions. These characteristics make it particularly attractive for exploring logical relationships and nested dependencies that naturally arise in domains such as security policies, data hierarchies, and structured documents. To enable seamless integration with ApexTree, ASP Chef introduces a novel `tree/2` functor within its Mustache-based templating system. This functor plays a central role in the declarative specification of tree structures from within ASP programs. The first argument of `tree/2`

CILC 2025: 40th Italian Conference on Computational Logic, June 25–27, 2025, Alghero, Italy

*Corresponding author.

mario.alviano@unical.it (M. Alviano)

https://alviano.net/ (M. Alviano)

0000-0002-2052-2063 (M. Alviano)

serves as a unique identifier for the tree, allowing multiple trees to be defined and distinguished within the same template, while the second argument captures the structural or configurational aspects of the tree. Specifically, it supports:

- Node declarations, represented as pairs of node identifiers and string labels, allowing each node to be uniquely referenced and visually rendered with meaningful content;
- Edge definitions, encoded as pairs of parent and child node identifiers, capturing the hierarchical topology of the tree;
- Root specification, where a node is designated as the starting point for rendering;
- Layout customization, i.e., how subtrees are inserted within node labels;
- Visual separators, which define how nodes or branches are delimited in the rendered output.

When a template is applied to an interpretation, ASP Chef automatically collects all instances of `tree/2` from the answers to the Mustache queries in the template, interprets them based on their type (node, edge, root, layout), and compiles a valid ApexTree-compatible JSON object (or a different object based on the provided elements). This process is fully encapsulated within the templating engine, and requires no imperative programming, no JavaScript coding, and no external transformation scripts. The user writes logic rules to define tree components as ASP facts, and the system takes care of generating the visualization-ready data structure. This integration exemplifies the design philosophy of ASP Chef: to empower users to express both logical content and presentation logic declaratively, without sacrificing modularity or expressiveness. It allows developers, researchers, and students to construct rich, navigable tree visualizations directly from answer sets, essentially turning abstract logic programs into concrete, interpretable structures. This is especially valuable in domains like security analysis, where hierarchical relationships (e.g., from user roles to permissions) are key to understanding system behavior, and where visual exploration can provide critical insights that textual output alone might obscure.

This paper also presents a case study demonstrating the use of the new tree visualization capabilities in the domain of SELinux policy analysis (a particularly intricate and security-critical context where reasoning over hierarchical structures is both natural and necessary). SELinux (Security-Enhanced Linux) policies govern access control by specifying which subjects (e.g., user or process types) are permitted to interact with which objects (e.g., files, sockets, services), and in what ways. These rules are typically expressed as 4-tuples in the form

`allow` *subject* *object*`:`*class* *permission*`;`

where:

- *subject* denotes the initiating entity, such as a process domain;
- *object* refers to the target entity being accessed;
- *class* defines the category of the object (e.g., file, system, service);
- *permission* specifies the allowed operation (e.g., read, write, execute).

While compact in syntax, these policies form dense, interconnected structures that are difficult to interpret without tool support, especially when multiple permissions apply across layered abstractions. To address this, we leverage ASP Chef to declaratively filter, structure, and visualize selected portions of an SELinux policy. The process begins with the specification of one or more subject–object pairs of interest. ASP rules are then used to traverse the space of policy rules, extracting the corresponding classes and permissions to construct a multilevel representation of access chains. This access graph is encoded using ASP facts that define the hierarchical relationships among policy elements. These facts are then interpreted through the `tree/2` functor to produce a structured representation of the policy segment as a tree, where the root corresponds to the initiating subject, and branches represent the propagation of access rights through object–class–permission links. The resulting tree is rendered in-browser via ApexTree, allowing users to interactively explore access chains, collapse or expand branches, and trace the permissions available to specific roles within the system.

To further improve interpretability, we augment this visualization pipeline with support for Large Language Models (LLMs). Using the Mustache-based templating system of ASP Chef, we can generate prompts for LLMs by automatically converting filtered policy facts into natural language sentences or SELinux-style statements (e.g., `allow sysadm_t init_t:system reboot;`). These statements are passed to the LLM, which returns human-readable summaries explaining the intent and implications of the selected rules. This natural language output is rendered using Markdown components within ASP Chef, providing a narrative interpretation alongside the interactive tree view. Together, these components offer a comprehensive, declarative, and human-friendly interface for exploring SELinux policies, transforming abstract rule sets into intuitive structures and readable explanations. This case study highlights how ASP Chef can be used not only as a logic programming platform, but also as a bridge between formal reasoning and interactive security analysis.

All in all, this work showcases how the integration of logic programming, templated content generation, and interactive web visualization can make complex security configurations more transparent and accessible. The declarative nature of the approach allows users to reason about policy subsets, construct trees, and generate human-readable explanations, all within a single, self-contained environment.

2. Background

2.1. SELinux Policies

Security-Enhanced Linux (SELinux) is a security architecture integrated into the Linux kernel that provides mandatory access control (MAC) mechanisms to restrict the actions that processes can perform on system resources. Unlike discretionary access control (DAC), where permissions are granted at the discretion of resource owners, SELinux enforces centrally defined policies that apply system-wide and cannot be overridden by individual users or applications. At the heart of SELinux lies a label-based policy model. Every subject (e.g., process) and object (e.g., file, socket, device) in the system is associated with a security context, typically composed of a user, a role, and a type. Among these, the type field plays a crucial role in access control decisions and is the primary focus of SELinux policies. Policy rules specify which subject types can access which object types, and under what conditions.

The core policy primitive in SELinux is the *allow rule*, which specifies explicit permissions granted by the security policy. These rules are written in the following form:

`allow` *subject object*`:`*class permission*`;`

Each such rule authorizes a process running under the specified subject type to perform the designated permission on a resource labeled with the given object type and class. The elements of the rule can be interpreted as follows:

- *subject* refers to the type associated with the process attempting the operation. In SELinux, this type (also called a *domain*) encapsulates the security identity of the process. For example, `sysadm_t` identifies privileged processes associated with system administration tasks.
- *object* is the type assigned to the target resource, such as a file, socket, or process. These types classify system resources according to their intended usage or sensitivity. For instance, `var_log_t` denotes log files under `/var/log`, while `init_t` may represent the system initialization process.
- *class* specifies the category of the object being accessed. This determines the set of valid permissions that can be applied. Common classes include `file`, `dir` (directory), `unix_stream_socket`, and `process`. Each class has its own permission vocabulary (e.g., `read`, `write`, `connectto`).
- *permission* defines the specific action that is allowed on the object. Permissions vary depending on the class of the object: for files, typical permissions include `read`, `write`, `execute`, `getattr`, and `open`; for sockets, they may include `connectto`, `sendto`, and `recvfrom`; for processes, permissions such as `sigkill` and `ptrace` are available.

Together, these four components define a fine-grained access control matrix that governs the interactions between labeled subjects and objects on the system. Each rule is a single entry in this matrix, permitting

a specific interaction between a domain and a resource. Because SELinux uses a default-deny model (i.e., denying all actions not explicitly allowed) these allow rules are essential to enabling the functional behavior of the system.

For example, the rule:

```
allow sysadm_t var_log_t:file read;
```

permits processes running in the `sysadm_t` domain to perform the `read` operation on any resource labeled with type `var_log_t` and class `file` (i.e., to read system log files).

Because SELinux policies can span thousands of such rules, they form a dense and hierarchical structure that can be difficult to interpret and audit manually. For instance, a single subject might be allowed to access dozens of object types, each with multiple classes and permissions. Moreover, alternative permission paths may exist to accomplish the same high-level operation, increasing the cognitive load on administrators and auditors.

Example 1. Let us suppose an administrator (subject: `sysadm_t`) wants to view the contents of system logs (object: `var_log_t`), which are typically stored in files under `/var/log`. Depending on how the logs are accessed, two alternative permission paths may apply:

(a) *Direct File Access:*

```
allow sysadm_t var_log_t:file read;
allow sysadm_t var_log_t:file open;
allow sysadm_t var_log_t:file getattr;
```

This path corresponds to using tools like `cat`, `less`, or `tail` to directly open and read log files from the filesystem (e.g., `cat /var/log/messages`).

(b) *Indirect Access via Journalctl:*

```
allow sysadm_t systemd_logind_t:dbus dbus;
allow sysadm_t journalctl_t:unix_stream_socket connectto;
allow sysadm_t journal_data_t:file read;
allow sysadm_t journalctl_exec_t:file use;
```

This path models access via a service like `journalctl` (e.g., `journalctl -xe`), where logs are obtained indirectly through inter-process communication, and require different permissions than raw file access.

Both plans achieve the same functional goal (i.e., viewing logs) but they operate at different abstraction levels (raw file vs. high-level service), involve different SELinux object types, and rely on different communication mechanisms (filesystem vs. IPC). ■

2.2. Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm centered on the representation of knowledge and reasoning over complex domains. An ASP program consists of a finite set of rules, each expressing how certain facts (known as atoms) can be derived from other conditions. A typical rule includes a head, which denotes a possible conclusion (either a single atom or a disjunction/choice among atoms), and a body, which specifies the conditions under which the head holds. These conditions are formed from a conjunction of literals, possibly including aggregates, comparisons, or arithmetic expressions. Formally, an ASP program Π gives rise to a set of answer sets (or stable models), each representing a self-consistent collection of atoms that satisfies all rules of Π under a specific stability condition [12]. This condition guarantees that each true atom is supported (i.e., justified by the program) and that no smaller subset of atoms forms a valid model. A program may admit zero, one, or multiple answer sets, depending on how non-determinism and constraints are used. This allows ASP to naturally encode and solve combinatorial search problems and explore alternative solutions.

To define the desired output of a program, ASP includes output directives, i.e., the `#show` directive. These serve to restrict the visible portion of each answer set, effectively projecting it onto a user-specified view. A show directive has the general form:

`#show p(`$\bar{t}$`) : conjunctive_query.`

where `p(`$\bar{t}$`)` is a term pattern, and `conjunctive_query` is a condition to obtain terms from the processed interpretation. This feature is crucial in practice, as ASP answer sets can be large and contain many auxiliary atoms used for intermediate derivations. Through `#show` directives, users can isolate the atoms that represent the meaningful outcomes of a computation, especially useful when such outcomes are to be rendered, visualized, or exported.

The expressiveness of ASP and its ability to handle incomplete knowledge, default reasoning, and constraints make it particularly well-suited for domains such as security policy analysis, where one must reason over what is permitted, forbidden, or undefined under a given specification. To illustrate the basics of ASP, consider the following example.

Example 2 (Continuing Example 1)**.** The scenario is modeled in ASP as follows:

```
% Subject attempting the operation
subject("sysadm_t").

% Policies granted to the subject
policy("sysadm_t", "journal_data_t", "file", read).
policy("sysadm_t", "journalctl_t", "unix_stream_socket", connectto).
policy("sysadm_t", "journalctl_exec_t", "file", use).
policy("sysadm_t", "systemd_logind_t", "dbus", communicate).
policy("sysadm_t", "var_log_t", "file", read).
policy("sysadm_t", "var_log_t", "file", open).
policy("sysadm_t", "var_log_t", "file", getattr).
% many more facts like these

% Permission paths: each represents a complete way to view logs
path(a, "var_log_t", "file", read).
path(a, "var_log_t", "file", open).
path(a, "var_log_t", "file", getattr).

path(b, "systemd_logind_t", "dbus", communicate).
path(b, "journalctl_t", "unix_stream_socket", connectto).
path(b, "journal_data_t", "file", read).
path(b, "journalctl_exec_t", "file", use).

% Choose exactly one path to validate
{selected_path(P) : path(P, _, _, _)} = 1.

% Derive required permissions from selected path
required(O,C,P) :- selected_path(Path), path(Path,O,C,P).

% Check which permissions are granted
granted(O,C,P) :- subject(S), policy(S,O,C,P).

% Valid path only if all required permissions are granted
:- required(O,C,P), not granted(O,C,P).

% Output the path and matched permissions
#show subject/1.
#show required/3.
#show selected_path/1.
#show granted/3.
```

If all permissions for a path are available, the program will return an answer set with the selected path and its granted permissions (in this case, either plan `a` or `b`). If multiple paths are valid (e.g., if both sets are granted), the ASP solver may return multiple answer sets, one per valid path. If no path is fully supported, the program yields no answer set, indicating that the subject cannot view the logs.

The above program can be extended with the following weak constraint

```
:~ granted(O,C,P). [1@1, O, C, P]
```

to prefer shorter paths (e.g., path `a` with 3 permissions vs. path `b` with 4). Alternatively,

```
:~ granted(O,C,P). [1@1, O, C]
```

to prefer paths involving less objects (within their classes), regardless of the number of permissions on the involved objects. ■

For further details on the syntax and semantics of ASP, including advanced rule types, aggregates, and standard directives, we refer the reader to the ASP-Core-2 standard [13], which defines the common specification adopted by modern solvers and development environments.

2.3. ASP Chef

An *operation* O is a function receiving in input a sequence of interpretations and producing in output a sequence of interpretations. Operations may produce side outputs (e.g., a graph visualization) and accept parameters to influence their behavior. An *ingredient* is an instantiation of a parameterized operation with side output. A *recipe* is a tuple of the form $(encode, Ingredients, decode)$, where $Ingredients$ is a (finite) sequence $O_1\langle P_1\rangle, \ldots, O_n\langle P_n\rangle$ of ingredients, and $encode$ and $decode$ are Boolean values. If $encode$ is true, the input of the recipe is mapped to **[[**`__base64__("`s`")`**]]**, where $s = Base64(s_{in})$ (i.e., the Base64–encoding of the input string s_{in}). After that, the ingredients are applied one after another. Finally, if $decode$ is true, every occurrence of `__base64__(`s`)` is replaced with (the ASCII string associated with) $Base64^{-1}(s)$. Among the operations supported by ASP Chef there are *Encode*$\langle p, s\rangle$ to extend every interpretation in input with the atom p`("`t`")`, where $t = Base64(s)$; *Search Models*$\langle\Pi, n\rangle$ to replace every interpretation I in input with up to n answer sets of $\Pi \cup \{p(\bar{t}).\mid p(\bar{t}) \in I\}$; *Show*$\langle\Pi\rangle$ to replace every interpretation I in input with the projected answer set $\Pi \cup \{p(\bar{t}).\mid p(\bar{t}) \in I\}$ (where Π comprises only `#show` directives. *Optimize*$\langle\Pi, n\rangle$ to replace every interpretation I in input with up to n optimal answer sets of $\Pi \cup \{p(\bar{t}).\mid p(\bar{t}) \in I\}$.

Example 3. The problem from Example 2 can be addressed in ASP Chef by a recipe comprising a single *Search Models*$\langle\Pi, 1\rangle$, where Π is the program in Example 2 (we will use *Optimize* if the weak constraint is included in Π). Alternatively, a recipe separating computational and presentational aspects would move the `#show` directives in a second ingredient *Show*. ■

Several operations in ASP Chef support expansion of *Mustache templates* [11]; among them, there are *Expand Mustache Queries*, *@vis.js/Network* (to visualize graphs), *Tabulator* (to arrange data in interactive tables), and *ApexCharts* (to produce different kinds of charts). A Mustache template comprises queries of the form `{{ `Π` }}`, where Π is an ASP program with `#show` directives—alternatively, `{{= `$p(\bar{t})$` : `*conjunctive_query*` }}` for `{{ #show `$p(\bar{t})$` : `*conjunctive_query*`. }}`. Intuitively, queries are expanded using one projected answer set of $\Pi \cup \{p(\bar{t}).\mid p(\bar{t}) \in I\}$, where I is the interpretation on which the template is applied on. Separators can be specified using the predicates `separator/1` (for tuples of terms), and `term_separator/1` (for terms within a tuple). The varadic predicate `show/*` extends a shown tuple of terms (its first argument) with additional arguments that enable repeating tuples in output and can be used as sorting keys (using predicate `sort/1`). Moreover, Mustache queries can use `@string_format(format, ...)` to format a string using the given format string and arguments, and floating-point numbers are supported with the format `real("NUMBER")`. Format strings can also be written as (multiline) *f-strings* of the form `{{f"..."}}`, using data interpolation `${`*expression*`:`*format*`}` to render *expression* according to the given *format*.

Example 4. Recipes from Example 3 can be further extended by including ingredients to visualize input and computed solution graphically. To this aim, the following Mustache template can be combined with *@vis.js/Network* to obtain the graph shown in Figure 1:

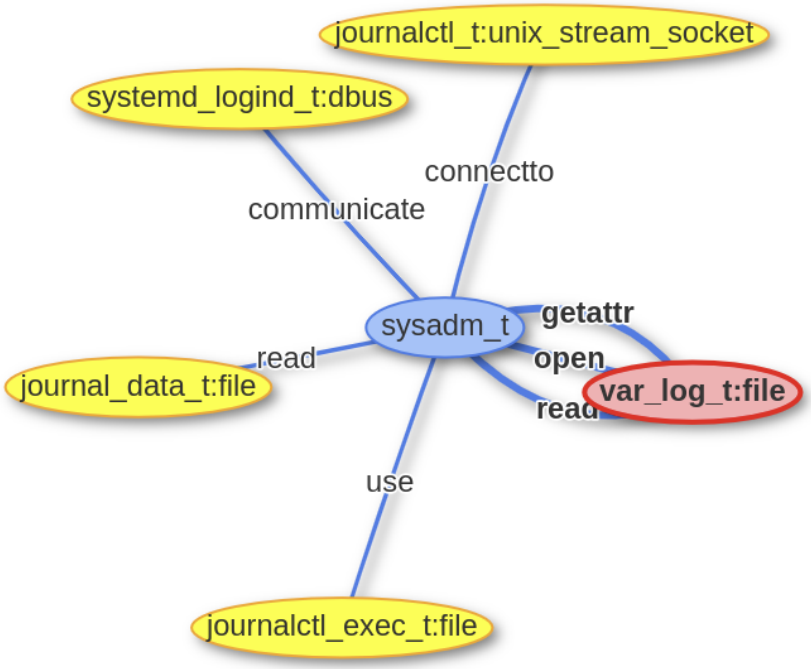

Figure 1: Graphical representation of input and selected path for the problem presented in Example 2

```
{ data: {
    nodes: [
      {{= {{f"{ id: "${S}", label: "${S}", group: "subject" }"}} : subject(S) }}
      {{= {{f"{
          id: "${O}:${C}",
          label: "${O}:${C}",
          group: "required"
        }"}} : required(O,C,P) }}
      {{= {{f"{
          id: "${O}:${C}",
          label: "${O}:${C}",
          group: "granted"
        }"}} : granted(O,C,P), not required(O,C,P) }}
    ],
    edges: [
      {{= {{f"{ from: "${S}", to: "${O}:${C}", label: "${P}" }"}} : subject(S),
   granted(O,C,P) }}
    ],
  },
  options: {
    nodes: { borderWidth: 2, shadow: true },
    edges: { width: 2, shadow: true },
  },
}
```

Note that Mustache queries are used to define nodes and links in the graph starting from facts in the computed answer set. A recipe addressing the selection problem and producing the visualization shown in Figure 1 is available at https://asp-chef.alviano.net/s/CILC2025/selinux-network. ■

3. Adding Trees to Mustache Templates

To enable the declarative construction of tree-like data structures in ASP Chef, we introduce the functor `tree(`*TreeID*`,` *Term*`)`, where *TreeID* is a symbolic identifier for the tree instance, and *Term* encodes either a structural element (i.e., `root`, `node` or `link`) or a configuration directive (i.e., `children_on` or `separator`). All `tree/2` atoms belonging to the same *TreeID* are collected during template expansion and processed together to produce a hierarchical tree representation. The processing involves three key steps, described below, which are applied to every Mustache query within the expanded template.

Collection of Facts. All answers of the form `tree(`*TreeID*`,` *Term*`)` are extracted. For each *TreeID*, we collect:

- *Node declarations:* `node(ID, Label)` to assign a string label to node `ID`;
- *Link declarations:* `link(Parent, Child)` to define directed edges from `Parent` to `Child`;
- *Root declaration:* `root(ID)` to define the starting node for the tree;
- *Rendering format:* `children_on(Placeholder)` to specify how children are included in the node label (optional);
- *Separator:* `separator(SEP)` to specify the delimiter used to join multiple children (optional).

Recursive Tree Construction. The tree is constructed starting from the declared root node, say `id_0`. The construction proceeds recursively as follows: Let `label_0` be the string associated with node `id_0`—i.e, the answer includes `node(id_0, label_0)`. Let `c_1`, . . ., `c_n` be the identifiers of all children of `id_0`—i.e., the answer includes `link(id_0, c_i)`, for `i = 1..n`. The string representation of the root node is constructed by embedding the recursively rendered children into the placeholder specified by `children_on(Placeholder)`. If no `children_on` directive is provided, the default placeholder is `{CHILDREN}`. Similarly, the children of a node are joined using the value of the `separator` directive, which defaults to ", " (comma followed by space). Each child `c_i` is recursively processed using the same procedure.

Example 5. Consider the following set of ASP facts:

```
expr(root(1)).
expr(children_on("@")).
expr(separator("")).

expr(node(1, "x @")).
  expr(node(2, "+ @")).
    expr(node(3, "3 @")).
    expr(node(4, "5 @")).
  expr(node(5, "- @")).
    expr(node(6, "2 @")).
    expr(node(7, "1 @")).

expr(link(1,2)).
expr(link(1,5)).
expr(link(2,3)).
expr(link(2,4)).
expr(link(5,6)).
expr(link(5,7)).
```

They represent the expression $(3 + 5) \times (2 - 1)$ (infix notation) using Polish notation (prefix notation): `x + 3 5 - 2 1`. We can easily obtain the Polish representation with the following Mustache template:

```
{{= tree(mytree, Term) : expr(Term) }}
```

Note that we are using `@` as a placeholder for children, and the empty string as separator. A recipe is available at https://asp-chef.alviano.net/s/CILC2025/polish-notation. ■

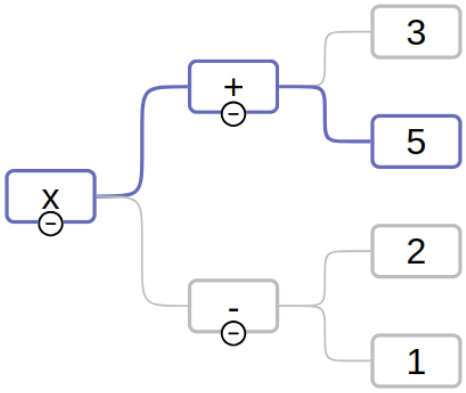

Figure 2: Arithmetic expression from Example 5 represented in Polish notation and visualized with ApexTree. The paths from a node to the root is highlighted when the mouse pointer is moved on the node (here, node 5).

Integration with ApexTree. To support interactive tree visualizations within ASP Chef, we integrated the ApexTree framework, a lightweight JavaScript library for rendering tree structures from JSON data. ApexTree expects a configuration object that includes both display settings and a hierarchical data structure representing the tree. The core configuration object passed to ApexTree typically follows this structure (simplified for illustration):

```
{ "data": {
    "name": "Root Label",
    "children": [
        { "name": "Child A", "children": [...] },
        { "name": "Child B", "children": []    } ]
  } }
```

Each node in the tree is represented by a JSON object with at least a `"name"` field (string label), and optionally a `"children"` array listing its subtrees. ASP Chef allows users to define the `data` field of this object declaratively by writing logic programs that emit facts of the form `tree(TreeID, Term)`. During template expansion, ASP Chef processes all such facts and assembles them into the corresponding JSON tree as described in the previous paragraphs. A typical Mustache template to configure ApexTree within ASP Chef may look like the following:

```
{ data: {{
    #show tree(my_tree, root(Node)) : tree(Node,_), not tree(_,Node).
    #show tree(my_tree, node(Node, @string_format("{id: '%s', name: '%s',
    children: [{CHILDREN}] }", Node, Name))) : node(Node,Name).
    #show tree(my_tree, link(A,B)) : tree(A,B).
  }},
  options: ... }
```

Above, the tree is retrieved from the interpretation in input, specifically from the binary predicate `tree` representing parent-child edges. Note that the string representation of nodes is obtained using format strings, and children are placed as expected by ApexTree.

Example 6 (Continuing Example 5)**.** An interactive view on the arithmetic expression is obtained with the following Mustache template:

```
{ data: {{
    #show tree(my_tree, root(Node)) : expr(root(Node)).
    #show tree(my_tree, node(Node, @string_format("{id: '%s', name: '%s',
    children: [{CHILDREN}] }", Node, @string_sub(Name,1,-3)))) :
    expr(node(Node,Name)).
    #show tree(my_tree, link(A,B)) : expr(link(A,B)).
  }},
  options: {
```

```
    width: 800, height: 700, siblingSpacing: 30,
    direction: 'left', borderWidth: 2, enableExpandCollapse: true,
    fontSize: '20px', fontFamily: 'sans-serif' }
}
```

The JSON object resulting by the application of the above template is interpreted by the new *ApexTree* operation of ASP Chef to obtain the visualization shown in Figure 2. ■

4. Application: Visualizing SELinux Policies

To demonstrate the practical utility of the `tree/2` functor and ApexTree integration, we developed a complete ASP Chef recipe (https://asp-chef.alviano.net/s/selinux/visualizing-policies#pit0500/asp-chef-short-links) aimed at visualizing SELinux policies as tree structures. The input to this recipe consists of the set of standard policy rules available in SELinux-enabled systems (such as those found in a typical Fedora distribution). Each rule is modeled as a fact of the form

```
policy(S, O, C, P).
```

representing an `allow` rule where subject type `S` is granted permission `P` on object type `O` and class `C`. The recipe unfolds in several declarative steps, illustrated in the following.

Filtering the Policy Space. The first ingredient filters the policy set to retain only those rules that involve a selected pair of subject and object types. This is achieved using a *Search Models* operation:

```
{ select(S,O) : subject(S), object(O) } = 1.
needed_policy(S,O,C,P) :- select(S,O), policy(S,O,C,P).
```

Users can choose from predefined facts of the form `subject("sysadm_t")` and `object("init_t")` to explore the permissions of interest. This filtering focuses the analysis and visualization on a manageable subset of the overall policy.

Assigning Unique Node Identifiers. To encode the policy structure as a tree, each relevant entity (subject, object, class, permission) must be uniquely identified. This is handled via a Lua *@-term* script:

```
ids = {}
counter = 2

function get_id(name, parent)
    key = tostring(parent) .. "_" .. tostring(name)
    if ids[key] == nil then
        ids[key] = counter
        counter = counter + 1
    end
    return ids[key]
end
```

This function ensures that each node receives a consistent and unique identifier, even when the same name appears under different parents.

Defining Tree Nodes. A second *Search Models* ingredient constructs the hierarchical structure by associating each node with its parent:

```
node(root, none, Subject) :- needed_policy(Subject, _, _, _).
node(ID, Subject, Object) :- needed_policy(Subject, Object, _, _),
    ID = @get_id(Object, Subject).
node(ID, Object, Class) :- needed_policy(_, Object, Class, _),
    ID = @get_id(Class, Object).
node(ID, Class, Permission) :- needed_policy(_, _, Class, Permission),
    ID = @get_id(Permission, Class).
```

The tree starts at the subject, which is labeled as the `root` node. Subsequent nodes represent object types, classes, and permissions, forming a logical path through the policy rule.

Establishing Tree Links. Using the derived nodes, the edges of the tree are defined with:

```
tree(root, Node) :- node(root, none, Subject), node(Node, Subject, Object).
tree(Node1, Node2) :- node(Node1, Subject, Object), node(Node2, Object, Class),
    node(root, none, Subject).
tree(Node1, Node2) :- node(Node1, Object, Class), node(Node2, Class, Permission).
```

These rules connect nodes in a layered fashion: subject → object → class → permission.

Selecting Relevant Output. To keep the output concise, only node declarations and tree links are shown using a *Select Predicates* operation, which is essentially equivalent to a *Show* ingredient with the following directives:

```
#show needed_policy/4.
#show node/3.
#show tree/2.
```

Encoding the Tree for ApexTree. Finally, the tree is rendered using an *Encode* operation with a Mustache template. The template is similar to the one reported in Example 6, and produces a JSON tree structure ready to be consumed by ApexTree. The result is an interactive, navigable tree that shows how a subject type (e.g., `sysadm_t`) can traverse the policy space to reach various permissions over an object type (e.g., `init_t`). This hierarchical view enables users to explore SELinux policies in an intuitive way. We refer Figure 3 for an example comparing the permissions of two subjects, namely users with the roles `sysadm_t` and `dbadm_t`, on the object `mysqld_t`: We observe that `sysadm_t` is granted fewer permissions than `dbadm_t`, reflecting the core principle of SELinux that even administrative domains such as the traditional `root` user do not possess unrestricted authority by default.

4.1. Enhancing Interpretability with LLMs

SELinux policies define granular access control rules that regulate interactions among processes, files, sockets, and other system entities. Due to their sheer size and complexity, these policies are often difficult to inspect manually, especially when attempting to understand the precise set of permissions granted to a specific subject in a given security context. While the tree visualizations described earlier provide a structural view of access relationships, they do not directly explain the implications of the policies in natural language. To bridge this gap, we integrate Large Language Models (LLMs) into ASP Chef recipes to generate human-readable explanations of selected policy fragments.

The approach involves constructing messages dynamically using ASP rules. We specify a *system prompt* to instruct the LLM on its role:

> You are an expert in SELinux policy management and your task is to create a detailed text starting from a specific set of policies.

Then, using `@string_format` function and Mustache templating, we construct the LLM input based on the filtered policy data:

```
message(user(@string_format( "allow %s %s:%s %s;", S,O,C,P)) ) :-
  needed_policy(S,O,C,P).
```

Here, `needed_policy/4` captures the subset of policies relevant to the subject–object pair selected by the user. For the application discussed in this section, and the permissions shown in Figure 3, the rendering of the template is the following:

```
allow sysadm_t mysqld_t:unix_stream_socket connectto;
allow dbadm_t mysqld_t:unix_stream_socket connectto;
```

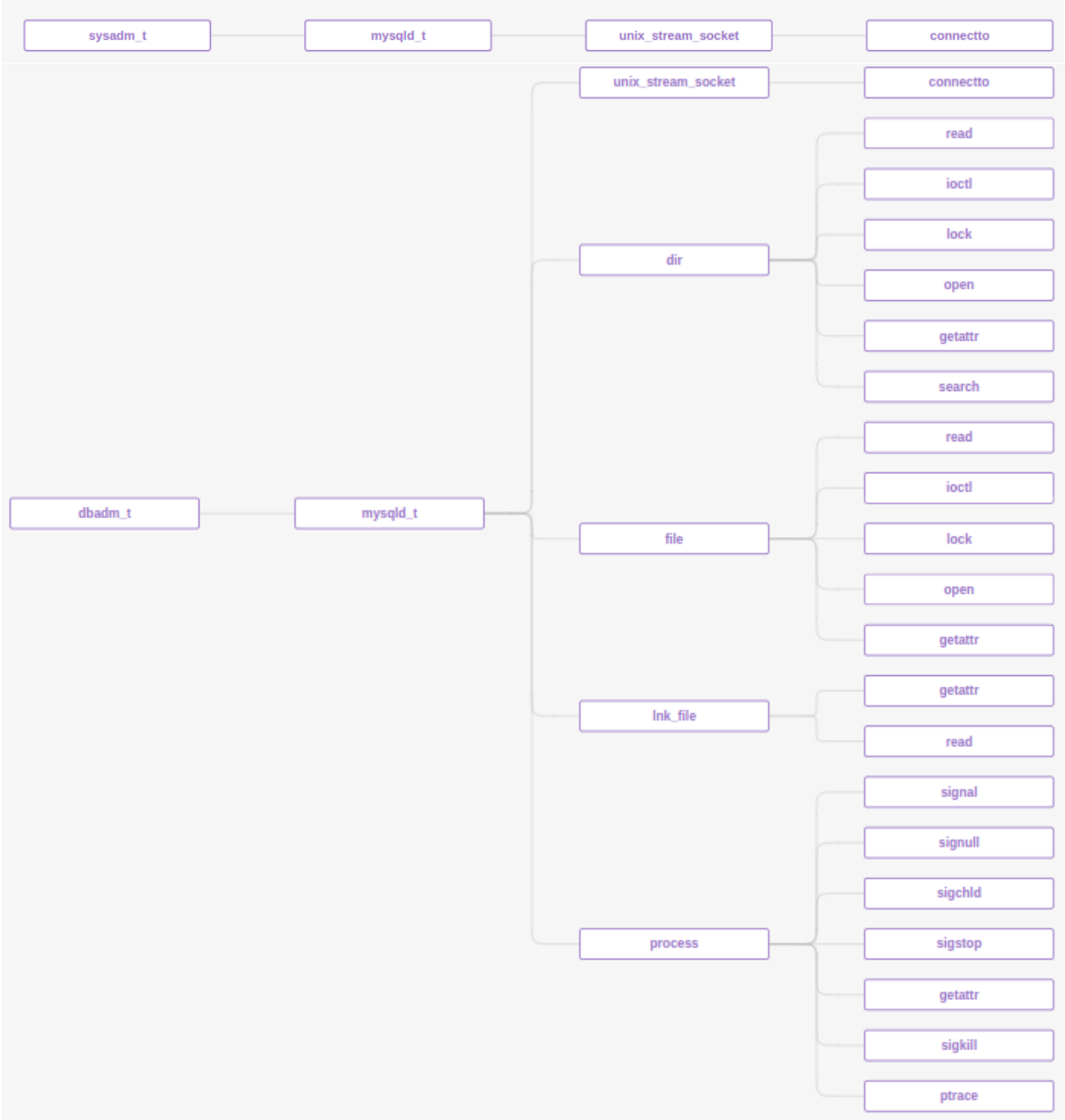

Figure 3: Tree view on the permissions of `sysadm_t` and `dbadm_t` on the object `mysqld_t` obtained with ApexTree and ASP Chef

```
allow dbadm_t mysqld_t:dir read;
allow dbadm_t mysqld_t:dir ioctl;
allow dbadm_t mysqld_t:dir lock;
allow dbadm_t mysqld_t:dir open;
allow dbadm_t mysqld_t:dir getattr;
allow dbadm_t mysqld_t:dir search;
allow dbadm_t mysqld_t:file read;
allow dbadm_t mysqld_t:file ioctl;
allow dbadm_t mysqld_t:file lock;
allow dbadm_t mysqld_t:file open;
allow dbadm_t mysqld_t:file getattr;
allow dbadm_t mysqld_t:lnk_file getattr;
allow dbadm_t mysqld_t:lnk_file read;
allow dbadm_t mysqld_t:process signal;
```

Figure 4: Natural language description of some SELinux policy rules obtained with ASP Chef by interacting with LLMs hosted at groq.com

```
allow dbadm_t mysqld_t:process signull;
allow dbadm_t mysqld_t:process sigchld;
allow dbadm_t mysqld_t:process sigstop;
allow dbadm_t mysqld_t:process getattr;
allow dbadm_t mysqld_t:process sigkill;
allow dbadm_t mysqld_t:process ptrace;
```

The resulting messages (i.e., the fixed system message, and the user message generated by expanding the Mustache template) are sent to the LLM using the `@LLMs/Chat Completion` operation of ASP Chef, and the response is rendered as Markdown within the recipe interface.

This integration enables users to obtain high-level summaries of the selected policy logic (e.g., what permissions a process in the `sysadm_t` domain has when interacting with `mysqld_t` processes) without requiring deep knowledge of SELinux syntax or semantics. An excerpt is shown in Figure 4. It also facilitates explanation-based auditing and helps bridge the gap between low-level rule inspection and practical security understanding.

5. Related Work

Efforts to improve the interpretability and usability of ASP have long emphasized the need for visualization tools capable of rendering the structure and meaning of answer sets. Early systems such as ASPViz [14], IDPD3 [15], and Kara [16] pioneered the idea of encoding visual elements directly as ASP facts, which could then be interpreted by external renderers. These tools enabled users to describe shapes, layouts, colors, and animations in a logic-based format, bridging symbolic reasoning and graphical output.

More recent tools such as clingraph [17] and ASPECT [18] have continued this trend, offering high-level declarative languages to describe visualizations of ASP outputs. While these systems significantly enhance expressiveness and produce exportable graphics, they typically rely on auxiliary rendering engines and may require complex local installations.

ASP Chef [9] introduced its first visualization mechanism in [10] through the *Graph* ingredient, enabling users to generate network-like visualizations directly from ASP recipes. Inspired by ASPViz and its successors, this feature allowed users to express graph structures as logic programs and produce visual feedback without imperative code.

In contrast to these systems, recent developments in ASP Chef adopt a template-based approach to visualization, grounded in the use of Mustache templates. Rather than embedding visualization logic in the ASP program, the user defines the output format declaratively, and the template engine fills in the data extracted from the answer sets. This model-driven strategy allows ASP Chef to interface directly with modern browser-based visualization libraries, including:

- *@vis.js/Network* for dynamic graph rendering,
- *Tabulator* for interactive data tables,
- *ApexCharts* for plotting time series, bar charts, and more,

- and now, *ApexTree* for hierarchical tree structures.

The work presented in this paper builds on this foundation by introducing a new `tree/2` functor, which enables users to define tree-structured data declaratively using ASP. By embedding structural and layout information in logic programs, users can automatically generate interactive tree visualizations without writing any JavaScript or low-level JSON. This stands in contrast to earlier systems that either required explicit visual encoding logic or used tightly coupled visual interpreters.

To the best of our knowledge, no previous ASP visualization system supports hierarchical tree rendering through templated JSON generation. Furthermore, the application of these visualizations to SELinux policy exploration is novel. Prior work has addressed logic-based access control analysis [6, 7], but none have leveraged ASP to construct interactive, layered visualizations of security policies, let alone integrated them with LLMs for natural language explanation.

6. Conclusion

We introduced a new extension to ASP Chef for declarative tree visualization, centered around the `tree/2` functor and Mustache-based JSON templating. This approach enables ASP programs to produce structured tree data, rendered interactively using the ApexTree framework. Our solution supports browser-native visualizations without requiring manual scripting, and we demonstrated its effectiveness through a case study on SELinux policy analysis. By representing access control rules as tree structures, we made complex policies easier to explore and interpret. In summary, this work enhances ASP Chef with:

- Declarative tree construction via `tree/2`;
- Templated JSON generation through Mustache;
- Native integration with ApexTree for visualization;
- Application to real-world hierarchical domains like SELinux.

These contributions make ASP outputs more interpretable and lay the groundwork for future integrations with richer interactive tools.

Acknowledgments

This work was supported by the Italian Ministry of University and Research (MUR) under PRIN project PRODE “Probabilistic declarative process mining”, CUP H53D23003420006, under PNRR project FAIR “Future AI Research”, CUP H23C22000860006, under PNRR project Tech4You “Technologies for climate change adaptation and quality of life improvement”, CUP H23C22000370006, and under PNRR project SERICS “SEcurity and RIghts in the CyberSpace”, CUP H73C22000880001; by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN “Assistente Virtuale Intelligente di Negozio” (CUP B29J24000200005); and by the LAIA lab (part of the SILA labs). Mario Alviano is member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-4o for grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

[1] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103. doi:10.1145/2043174.2043195.

[2] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68.

[3] V. Lifschitz, Answer Set Programming, Springer, 2019.

[4] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own asp-based system?!, Theory Pract. Log. Program. 23 (2023) 299–361.

[5] M. Alviano, C. Dodaro, S. Fiorentino, A. Previti, F. Ricca, ASP and subset minimality: Enumeration, cautious reasoning and muses, Artif. Intell. 320 (2023) 103931.

[6] S. Sartoli, A. S. Namin, Modeling adaptive access control policies using answer set programming, J. Inf. Secur. Appl. 44 (2019) 49–63. URL: https://doi.org/10.1016/j.jisa.2018.10.007. doi:10.1016/J.JISA.2018.10.007.

[7] J. Hu, K. M. Khan, Y. Bai, Y. Zhang, Constraint-enhanced role engineering via answer set programming, in: H. Y. Youm, Y. Won (Eds.), 7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012, ACM, 2012, pp. 73–74. URL: https://doi.org/10.1145/2414456.2414499. doi:10.1145/2414456.2414499.

[8] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[9] M. Alviano, D. Cirimele, L. A. Rodriguez Reiners, Introducing ASP recipes and ASP Chef, in: ICLP Workshops, volume 3437 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023.

[10] M. Alviano, L. A. Rodriguez Reiners, ASP chef: Draw and expand, in: P. Marquis, M. Ortiz, M. Pagnucco (Eds.), Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024, Hanoi, Vietnam. November 2-8, 2024, 2024. URL: https://doi.org/10.24963/kr.2024/68. doi:10.24963/KR.2024/68.

[11] M. Alviano, W. Faber, L. A. Rodriguez Reiners, ASP Chef grows Mustache to look better, 2025. URL: https://arxiv.org/abs/2505.24537. arXiv:2505.24537.

[12] M. Gelfond, V. Lifschitz, Logic programs with classical negation, in: D. Warren, P. Szeredi (Eds.), Logic Programming: Proc. of the Seventh International Conference, 1990, pp. 579–597.

[13] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309. URL: https://doi.org/10.1017/S1471068419000450. doi:10.1017/S1471068419000450.

[14] O. Cliffe, M. D. Vos, M. Brain, J. A. Padget, ASPVIZ: declarative visualisation and animation using answer set programming, in: ICLP, volume 5366 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 724–728.

[15] R. Lapauw, I. Dasseville, M. Denecker, Visualising interactive inferences with IDPD3, CoRR abs/1511.00928 (2015).

[16] C. Kloimüllner, J. Oetsch, J. Pührer, H. Tompits, Kara: A system for visualising and visual editing of interpretations for answer-set programs, in: INAP/WLP, volume 7773 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 325–344.

[17] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, *Clingraph*: A system for asp-based visualization, Theory Pract. Log. Program. 24 (2024) 533–559. URL: https://doi.org/10.1017/s147106842400005x. doi:10.1017/S147106842400005X.

[18] A. Bertagnon, M. Gavanelli, ASPECT: answer set representation as vector graphics in latex, J. Log. Comput. 34 (2024) 1580–1607. URL: https://doi.org/10.1093/logcom/exae042. doi:10.1093/LOGCOM/EXAE042.