Hapi: A Domain-Specific Language for the Declaration of Access Policies

Vinícius Juliãoindica Ramos

UFMG, Brazil

Alexander Holmquist

UFMG, Brazil

Fernando M. Quintão Pereira

UFMG, Brazil

Abstract

Access policies specify what are the actions that different actors can perform on available resources. Access policies are a core notion in multiuser environments, such as operating systems and distributed databases. Currently, most of these systems use general data specification languages, such as JSON, XML and YAML to describe access policies. Yet, domain-specific languages are also available for this task. One of such languages is Legalease, from Microsoft. This paper presents a new version of Legalease, called Hapi. Hapi replaces Legalease's notion of a lattice with a partially ordered set (poset). Posets improve the expressivity of Legalease, at the expenses of a more expensive verification algorithm. This poset-based representation generalizes the notion of actors, actions and resources to user-defined entities. Hapi is publicly available. Its distribution includes a policy visualizer and a code-compression tool to efficiently store specifications.

Email addresses: viniciusjuliao@dcc.ufmg.br (Vinícius Juliãoindica Ramos), alexmol@ufmg.br (Alexander Holmquist), fernando@dcc.ufmg.br (Fernando M. Quintão Pereira)

1. Introduction

Access policies are a set of rules that determine the actions that users (actors) can perform on resources. Access policies are typical in multiuser systems, such as shared operating systems and distributed databases [1, Ch.14]. Enterprises tend to use general data specification languages, like JSON, YAML or XML to write access policies. These formats are employed in very popular applications, including Amazon's AWS CloudFormation [2], Microsoft's Azure [3] and Google's Firebase [4]. Nevertheless, the research community recognizes the need of having domain-specific languages (DSLs) for this task [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17].

Legalease. Among the many domain-specific languages that have been designed and implemented for the specification of access policies, one is of interest to us: Legalease [18]. Legalease is a language for *Role-Based Access Control* [19]. It supports the specification of hierarchies of access policies in lattices, much like in an object-oriented language with multiple inheritance. The beauty of this approach is that new policies can be easily derived from old ones by either specialization or generalization. In the former case, users add more elements (actors, actions, etc) to the bottom of an already well-established lattice; in the latter, new elements are added to the top of that lattice. Therefore, actors might have their rights either increased or decreased as we move around this hierarchy.

Unfortunately, since its original conception, Legalease has never seen wide usage. The language specification has never been explicitly given, and its reference implementation is nowhere to be found. This fact is unfortunate because we believe that Legalease has many virtues that, once properly engineered, would greatly outperform general data description languages in terms of expressivity.

Hapi. This paper provides our vision of a correct and efficient implementation of Legalease. throughout this implementation process, we had to make several

assumptions about the semantics of LEGALEASE. Said semantics has never been made explicit by its authors. We believe that the language that emerged from these assumptions is different from LEGALEASE. Therefore, we shall call it HAPI, short for *Hierarchical Access Policy Implementation*. The key differences to LEGALEASE are as follows:

Semantics: LEGALEASE uses a lattice to define the hierarchy of access policies. While implementing such lattices, we eventually realized that a simpler abstraction would suffice for all the verifications that could be performed onto programs. Therefore, HAPI uses a partially ordered set (poset) instead; hence, removing from its specification the need that data elements have lowest upper bounds or greatest lower bounds.

Syntax: the original specification of LEGALEASE did not provide a complete grammar. A reference implementation is not available either. From the examples provided by Sen *et al.* [18], we could infer that LEGALEASE would rely on indentation for creating access hierarchies. However, indentation only would make it very difficult to support multiple definitions of policies at the same level of the hierarchy. Therefore, we decided to leave indentation aside, replacing it with a block-oriented syntax with explicit delimiters.

Implementation: HAPI comes with a reference implementation, written in Kotlin. HAPI also provides users with an online interface in which programs can be converted into YAML specifications, or visualized as in an access matrix. This reference implementation is accompanied by an e-book, where several examples of specifications can be found and expanded.

The design and implementation of HAPI was sponsored by `Cyral Inc`, a California-based company specialized in the protection of data. Originally, `Cyral` products would use YAML to specify access policies. HAPI is meant as a replacement to these specifications. The first description of HAPI appeared in an early work of ours [20]. Since then, we have released an online book

about the language [21], and have augmented its implementation with a text compression tool, to facilitate the storage of specification policies. This paper complements the e-book, and extends our first account of the language. For completeness, we include in this paper material present in that first description. Thus, Sections 3-5 have been taken from our earlier publication, except that we take advantage of the extra space available in this new work to provide a more comprehensive overview of Hapi. Section 6 includes original material, covering code compression and presenting experimental results concerning the implementation of the language. Additionally, Section 7 has been greatly expanded to explain how Hapi differs from policy specification languages in general, and from Legalease in particular.

2. Data Protection asks for Readable Access Specifications

The main force driving the implementation of Hapi is the advent of different data-protection laws in the second decade of the 2000's. Although legislation differs across countries and regions, such resolutions have a common goal: to regulate the usage of private information concerning individuals and organizations. Examples of data-protection laws include:

- The *General Data Protection Regulation* (GDPR)[1], valid in the European Economic Area since 2016.
- The *California Consumer Privacy Act*[2], effective since January of 2020 in the American state of California.
- The *General Law of Personal Data Protection* [22], (*Lei Geral de Proteção dos Dados Pessoais*) taking effect in August of 2020 in Brazil.
- The *Law on the Protection of Personal Data* (LPPD) N. 6698, valid in Turkey since April 7th, 2016 [23].

[1] `https://eugdpr.org/`
[2] AB-375 Privacy: personal information: businesses.(2017-2018)

These rules and regulations must be not only understood by law experts, but also enforced at the technology level. As well argued by Morel and Pardo [24], lawyers and programmers need a common ground to express requirements that must be mutually understood across their respective circles. HAPI has been designed with this goal in mind. This section explains how this design achieves readability, supports data categorization and provides deterministic semantics.

Readability. Data protection policies must conform to some *Auditable* specification. In other words, when establishing contracts, privacy must be well-understood between all the involved parties. Yet, natural languages, like English, offer much room for ambiguities [24, Sec.3]. On the other hand, typical data-exchange formats, such as JSON and YAML, might impair an auditor's ability to accurately understand the terms in a contract, depending on her familiarity with such representations [24, Sec.3]. HAPI's syntax is textual, relying almost exclusively on alphanumeric characters. Special symbols—parentheses, curly braces, commas and semicolons—are used only as punctuation. Example 1 shows how HAPI could be used to describe a piece of legal information.

Example 1. *Taking the Paragraph 3 from the Article 3 of the European GDPR, we have that Member States from Europe Union may have its own public international laws. In addition, in Article 9, paragraph 4, we have that there are some categories of data that are considered special, and Member States are free to augment them. One of these special data categories includes genetic information. In this context, for the sake of illustration, let us assume that German companies cannot read genetic data from countries outside the European Union. This restriction could be modeled by the following English sentence:*

> "Companies based on every EU State can store any data from countries outside the European Union, except Germany, for German-based companies cannot store genetic data."

Figure 1 shows how the above sentence could be translated in HAPI. *Section 3 will provide some rational for that syntax. Nevertheless, we hope that*

the reader can see a correspondence between the English sentence and its Hapi *representation, based solely on the clarity of the latter.*

```
1  data Countries =
2    Africa(Algeria, Angola, ....),
3    Asia(Afghanistan, Armenia, ...),
4    EU(Austria, Belgium, …),
5    Europe(Albania, EU, Andorra, …);
6
7  data Action =
8    Read, Write, Store;
9
10 data Resources =
11   PersonalData(
12     GeneticData, CreditCard),
13   ...,
14   WebTracking;
```

(a)

```
15 main = ALLOW {
16   Countries: EU
17   Action: Store
18   Resources
19 } EXCEPT {
20   DENY {
21     Countries: Germany
22     Action: Store
23     Resources: GeneticData
24   }
25 };
```

(b)

Figure 1: (a): Data categories defined in Hapi for Example 1. (b): The partial relations defined by the specification in part (a).

Categorization. A common trait of data protection laws is *categorization*: data must be separated into categories, so that different data groups are subject to distinct access policies. To separate data, Hapi uses the notion of a poset, a partial order, whose formal definition is yet to be presented in this paper (see Def. 1). Nevertheless, for now, the reader can know that categories of data in Hapi might be grouped into hierarchies. Policies that apply to upper elements in such a hierarchy are also valid for sub-categories. Example 2 illustrates how Hapi supports data categorization.

Example 2. *Figure 1 (a) is a* data declaration*: the part of a* Hapi *specification where data is defined. This example policy uses three main categories of data:* `Countries`*,* `Action` *and* `Resources`*. Each one of these data groups is further divided into subcategories. For instance, the category* `Countries` *is divided into four subgroups:* `Africa`*,* `Asia`*,* `EU` *and* `Europe`*. Notice that the last group,* `Europe`*, includes every data under the* `EU` *set.*

A semantic principle of Hapi is that elements within the same data category are governed by the same access rules. However, there are situations in which

it is necessary to create exceptions within categories. Thus, following a pattern originally invented by LEGALEASE designers, HAPI lets users specify exceptions to general rules explicitly via either `ALLOW` or `DENY` clauses. Example 3 introduces the syntax used to revert access policies.

Example 3. *Figure 1 (b) is an* access specification. *It defines a general access rule over triples formed by the categories* `Countries`, `Action` *and* `Resources`. *The intention of the initial rule, e.g.,* `main = ALLOW {...}` *is to inform an enforcer (regardless of the meaning of enforcement) that countries from the subcategory* `EU` *are allowed to* `Store` *any resources. However, an exception to this general rule exists:* `Germany`, *a subcategory of* `EU` *is not allowed to perform the* `Store` *action on a particular category of resources:* `GeneticData`. *The* `EXCEPT DENY {...}` *statement in Figure 1 (b) sets this constraint up.*

Determinism. A concern that permeates the design of HAPI is determinism: the semantics of a HAPI specification is deterministic. A HAPI program, as we will see in Section 4.3 represents a set of tuples, which is immutable. As a consequence, access policies cannot be enforced `ex post facto`; that is, before the policy was in place. Additionally, once in place, policies cannot be updated—any changes will imply the replacement of a policy specification with a completely new one.

Nevertheless, policies must be *enforced*. In our vision, enforcement is orthogonal to specification. The specification—a HAPI file—is an immutable text. Enforcement, in turn, requires computation. HAPI can be used in tandem with any program to evaluate and validate access policies, as Example 4 illustrates. This paper concerns exclusively the design of HAPI. Thus, the enforcement of HAPI policies is immaterial to our presentation.

Example 4. *Figure 2 shows a* HAPI *specification of a simple access policy. This policy is given by two immutable pieces of text, Fig. 2 a and Fig. 2 b. How this policy is enforced is left up to the system that uses* HAPI. *Figure 2 (c) shows a program that enforces this policy. We ask the reader to assume the method*

`policy.allowed` verifies if some access is valid, following the semantics yet to be presented in Section 4.3.

(a)

```
data Actor = Alice;
data Action = TransferMoney;
data Day =
  WeekDay(Mon, Thu, Wed, Thu, Fri),
  WeekEnd(Sat, Dom);
```

(b)

```
main =
ALLOW {
  Actor: Alice
  Action: TransferMoney
  Day
} EXCEPT {
  DENY {
    Actor: Alice
    Action: TransferMoney
    Day: WeekEnd
  }
};
```

(c)

```
import datetime, calendar, hapi
date = datetime.datetime.today()
day = calendar.day_name[date.weekday()]
policy = hapi.read_policy('file.hapi')
allowed = policy.allowed(Actor='alice', Action='TransferMoney', Day=day)
if (allowed):
  TransferMoney(alice, value)
```

Python

Figure 2: (a-b) Hapi specification of the policy "Alice can transfer money, except on weekends". (c): Enforcement of the policy, via a `Python` program.

3. Preliminary Definitions

Hapi programs consist of two parts: data declaration and policy specification. The former determines the data that a program might refer to. The latter determines how said data relates in terms of access rights. Throughout this section, we shall clarify these notions via examples.

3.1. Data Declaration

Data is represented in Hapi by the concept of a partially ordered set (poset), a relation that we define as follows:

Definition 1 (Poset). *A partially ordered set $(S, \leq)$ is formed by a set S, plus a partial less-than relation $\leq$ between the elements of S that is reflexive, antisymmetric, and transitive.*

The ordering mentioned in Definition 1 is called partial because it is not mandatory that every pair of elements in S has a defined relationship. The next example shows how posets can be specified in Hapi.

Example 5. *Figure 3(a) shows the specification of three partially ordered sets in* HAPI*:* ***Actions****,* ***Resources*** *and* ***Actors****. The syntax* `a(b1,...,bn)` *determines* ***n*** *less-than relations like* $\texttt{bi} < \texttt{a}, 1 \leq \texttt{i} \leq \texttt{n}$.

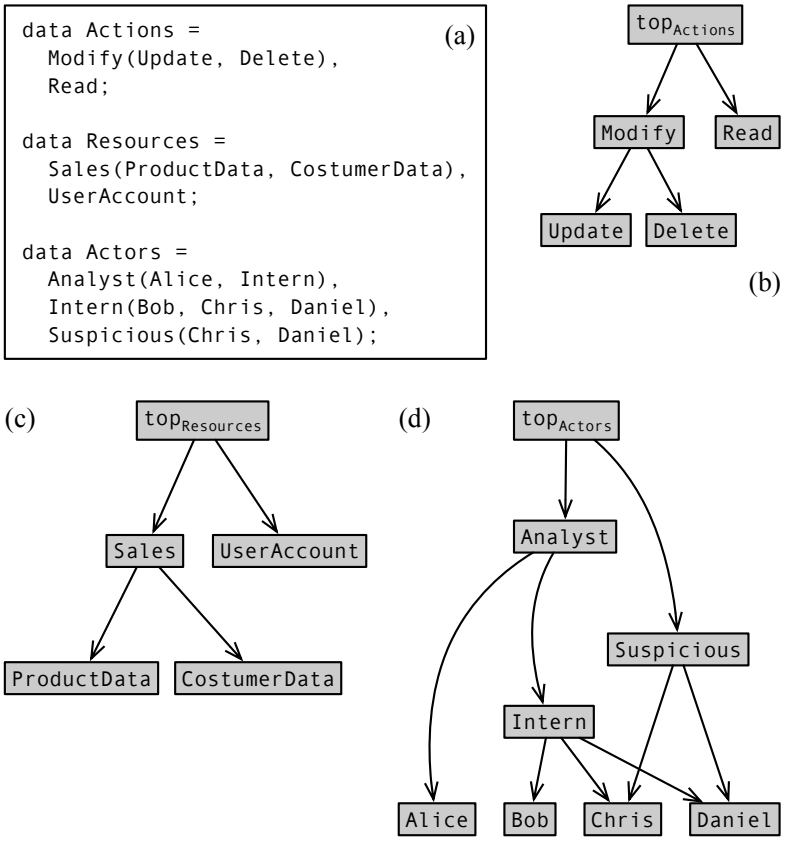


Figure 3: (a): An example of data declaration in HAPI. (b-d): The partial relations defined by the specification in part (a).

Posets define acyclic graphs. Figures 3(b-d) show the graphic representation of the three posets earlier mentioned in Example 5. The data declaration is open, meaning that users can create new elements, and specify the hierarchical relation between them. Users are free to create data, and to determine where they fit in the hierarchy. Notice that by construction, every poset defined in HAPI contains a topmost element. Syntactically, this topmost element is determined by the left side of a data specification. In Figure 3, these elements are: `Actions`,

`Resources` and `Actors`.

3.2. Access Specification

Policy specifications in HAPI range over the elements defined in the data declaration. Syntactically, these specifications follow a nesting structure. Nested rules either expand or constrain the access rights specified in the encircling rules. In Section 4.3 we shall dive into the minutia of this semantics. Meanwhile, Example 6 provides the reader with an informal understanding of it.

Example 6 (Specification). *Figure 4(a) defines possible access rights on the data discussed in Example 5. Access specifications start when the parser encounters an arbitrary label not preceded by* ***data****. We choose the label* ***main****, as seen in Figure 4(b), Line 9. Notice that* ***main*** *is not a reserved word (see Sec. 4.2). In this example, every triple (****Actors****,* ***Actions****,* ***Resources****) has all its access denied by default. The* ***ALLOW*** *clause in Lines 12-15 gives back access to any triple where* `Actors` = `Analyst` ∧ `Resources` = `Sales`*. Then, the next two* ***DENY*** *clauses remove accesses to triples where either* `Actors` = `Intern` ∧ `Actions` = `Modify`*, or* `Actors` = `Suspicious`*. Notice that triples can be specified independently (as in Figure 3(a)), and then referred to in the specification (as in Line 18).* HAPI *also supports separate compilation units. In other words, the specifications in Figures 4 (a) and (b) are written in separate files.*

Access rules range over products of posets. A product poset is a standard algebraic notion. However, given the importance of this notion to this presentation, Definition 2 restates it.

Definition 2 (Product Poset). *If* $(P_0, \leq_0)$ *and* $(P_1, \leq_1)$ *are posets, then the product poset* $(P_0 \times P_1)$ *is given by all the tuples* $(e_0, e_1), e_0 \in P_0, e_1 \in P_1$*. We say that* $(e'_0, e'_1) \leq (e''_0, e''_1)$ *if, and only if,* $e'_0 \leq_0 e''_0$ *and* $e'_1 \leq_1 e''_1$*.*

Ultimately, HAPI's specifications produce a set of tuples known as an *Access Policy* (see Definition 4 in Page 16). These tuples are formed by *atoms*. Atoms are the smallest elements in the poset specification. For instance, data `Actors`

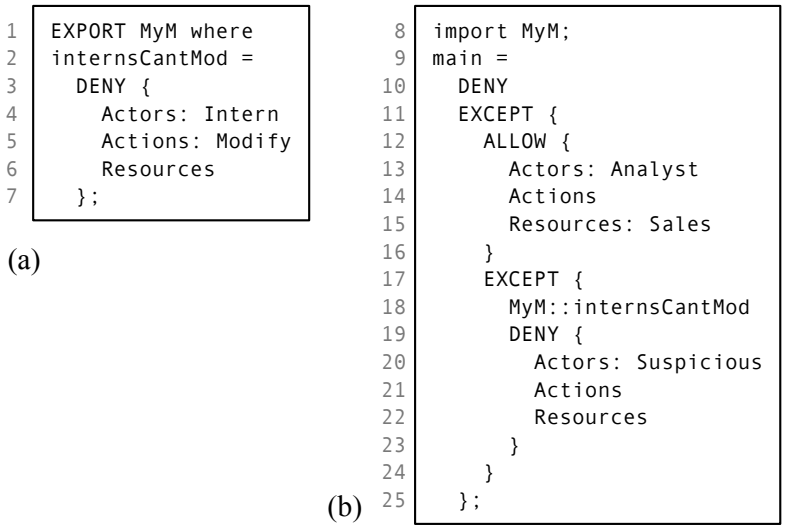


Figure 4: Example of policy specification using the posets defined in Figure 3(a).

in Figure 3(a) defines four atoms: `Alice`, `Bob`, `Chris` and `Daniel`. We can see the tuples defined by a HAPI program as elements that form a multi-dimensional space. The next example illustrates these notions.

Example 7. *Figure 5 shows all the atoms defined by the declarations seen in Figure 3. Visually, atoms are the leaves in the poset graphs in Figures 3(b-d). Because Figure 3 contains three data declarations, it forms the three-dimensional space of Figure 5. This space is defined by the product poset* `Resources` $\times$ `Actions` $\times$ `Actors`.

Access Matrices. HAPI specifications can range on spaces with any number of dimension—a new dimension being created for each new data declaration. Nevertheless, policies with three dimensions are specially useful, for they can be nicely visualized as an *Access Matrix*. Definition 3 recalls the notion of an access matrix, a concept originally introduced by Butler Lampson in the early 70's [25].

Definition 3 (Access Matrix). *Given the product poset $P = D_1 \times D_2 \times D_3$, formed by three* `data` *declarations D_1, D_2 and D_3, a* HAPI *specification S de-*

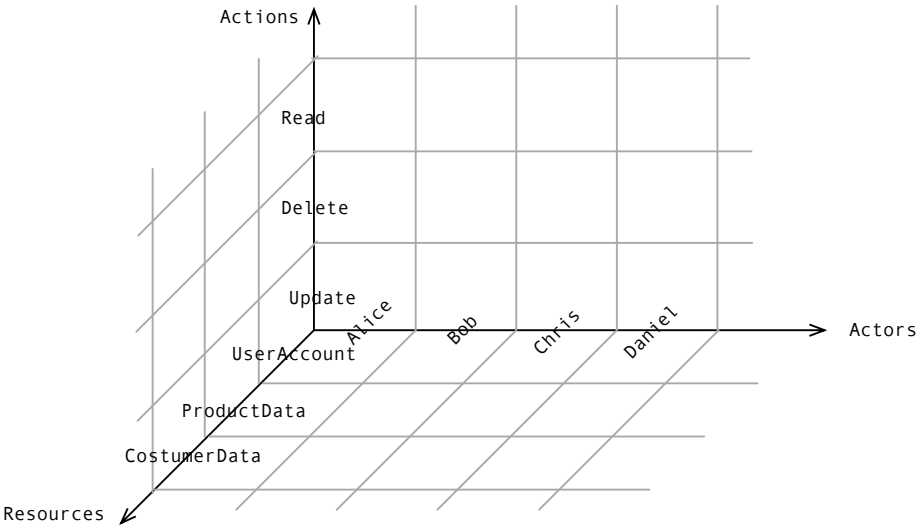


Figure 5: The tuple space formed by the atoms defined in Figure 3. HAPI programs determine points in this space.

fines, without loss of generality, an access matrix M having as rows the atoms from D_1 and as columns the atoms from D_2. The element $M[i, j]$ contains the atoms from D_3 that are allowed by S.

Example 8. *Figure 6 shows the access matrix defined by the program in Figure 4. This data structure encodes, for instance, the fact that the tuple* (`Alice`, `UserAccount`, `Delete`) *is present in the specification. On the other hand, the tuple* (`Bob`, `UserAccount`, `Delete`) *is not.*

4. Language Specification

As already hinted in Section 3, a HAPI program ultimately defines a set of tuples. Each tuple is formed by elements of a product poset. We refer the reader to Example 8 for a concrete illustration. Therefore, the specification of HAPI amounts to showing how, from a program, we can derive the tuples that said program entails. In this section, we approach this task in three phases. First, in Section 4.1, we show the syntax of HAPI. However, instead of going directly from syntax to semantics, we found it easier to provide HAPI with a

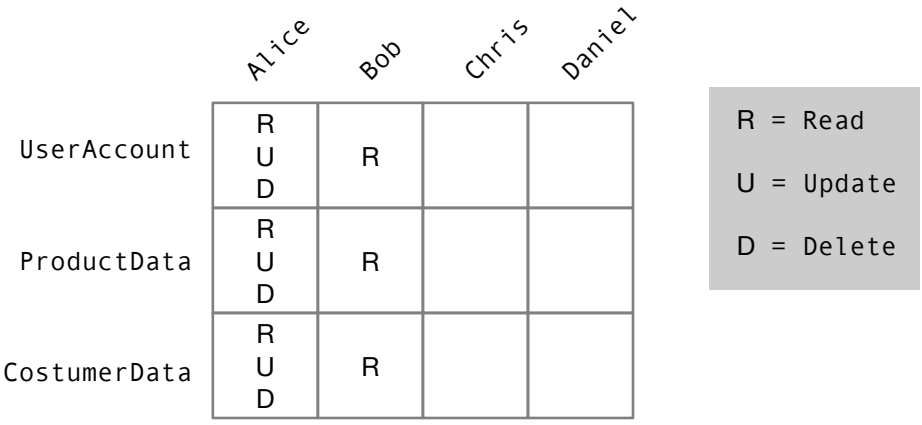


Figure 6: The access matrix defined by the specifications from Figure 4.

kind of "*intermediate representation*". This intermediate representation is what we call a *normalized program*. Thus, in Section 4.2 we explain how programs can be normalized. Finally, in Section 4.3 we show the semantics of normalized programs.

4.1. Syntax

The syntactic specification of Hapi is very short. Figure 7 shows the grammar that defines such syntax. Any Hapi program alternates restrictive (`DENY`) and permissive (`ALLOW`) clauses. This structure follows closely what is our understanding of Legalease's syntax (Legalease's grammar has never been publicly disclosed). Notice that the alternation between `DENY` and `ALLOW` clauses is enforced at the syntactic level. In other words, Hapi's grammar does not permit nesting a `DENY` clause into another.

According to Gerl *et al.* [26], "*A Privacy Language must be Human-Readable*". Thus, readability has been a driving force behind the syntax of Hapi, as discussed in Section 2. We believe that the present materialization of Hapi's syntax is sufficiently readable to be understood, even by non-programmers. Example 9 further supports this perception of ours.

Example 9. *The specification at Lines 2-7 of Figure 4 can be read as "*You can allow everything, except that you must deny Interns Modifying any Resource.*"*

S	::=	`data` $L = E_1, \ldots, E_n$ `;`	Data Statement
P	::=	L `=` C `;`	Policy Statement
C	::=	$D \mid A$	Policy Clause
D	::=	`DENY EXCEPT` $\{ A_1, \ldots, A_m\}$	Deny Clause
		| `DENY` $\{ T_1, \ldots, T_n\}$	
		| `DENY` $\{ T_1, \ldots, T_n\}$ `EXCEPT` $\{ A_1, \ldots, A_m\}$	
A	::=	`ALLOW EXCEPT` $\{D_1, \ldots, D_m\}$	Allow Clause
		| `ALLOW` $\{T_1, \ldots, T_n\}$	
		| `ALLOW` $\{T_1, \ldots, T_n\}$ `EXCEPT` $\{D_1, \ldots, D_m\}$	
E	::=	L`(`$L_1, \ldots, L_n$`)` $\mid L$	Poset Element
T	::=	L`:` $L_1, \ldots, L_n \mid L$	`Attribute`
L	::=	`[a-zA-Z][a-zA-Z0-9]*`	`Label`

Figure 7: Hapi context free grammar.

Block Delimiters. The main difference between Hapi's grammar and Legalease's concerns the semantic meaning of indentation. Legalease, similarly to Python, uses indentation to open up new blocks of code. Our initial implementation of Hapi would do it as well. However, currently Hapi uses braces as a block delimiter. The reason for this departure is the fact that Hapi does not reserve keywords, and allows multiple blocks of policies at the same nesting level—something absent from Legalease[3]. Thus, except for `DENY` and `ALLOW`, users are free to define posets containing any identifier. This flexibility has complicated the incorporation of indentation onto Hapi's parser.

4.2. Normalized Representation

Before we explain the semantics of Hapi, we shall introduce a normalized representation for its programs. The goal of this normalized representation is to simplify the statement of semantic rules. Any Hapi program can be normalized, and any program in the normal representation is a valid Hapi program. Normalized programs meet the following properties:

1. Every `ALLOW` or `DENY` clause refers to a single element;

[3]Python allows multiple functions defined at the same nesting level, but they all start with the `def` reserved keyword.

2. Programs start with either `ALLOW` $\top$ or `DENY` $\top$, where $\top$ is the topmost element in the product poset.

Figure 8 shows a suite of rewriting rules that convert a program to normal form. Rewriting is implemented by a function `nm`, that selects the right transformation by pattern matching. In total, Figure 8 defines 4 patterns. These rules assume that policy definitions start with a special identifier: `main`. In practice, Hapi's syntax makes no provision for such a keyword: any identifier can be the starting point of a specification. Assuming the existence of `main` in Figure 8 simplifies the definition of the rewriting rules, because it gives us the means to differentiate the treatment of the first rule in a specification from the rules that follow it.

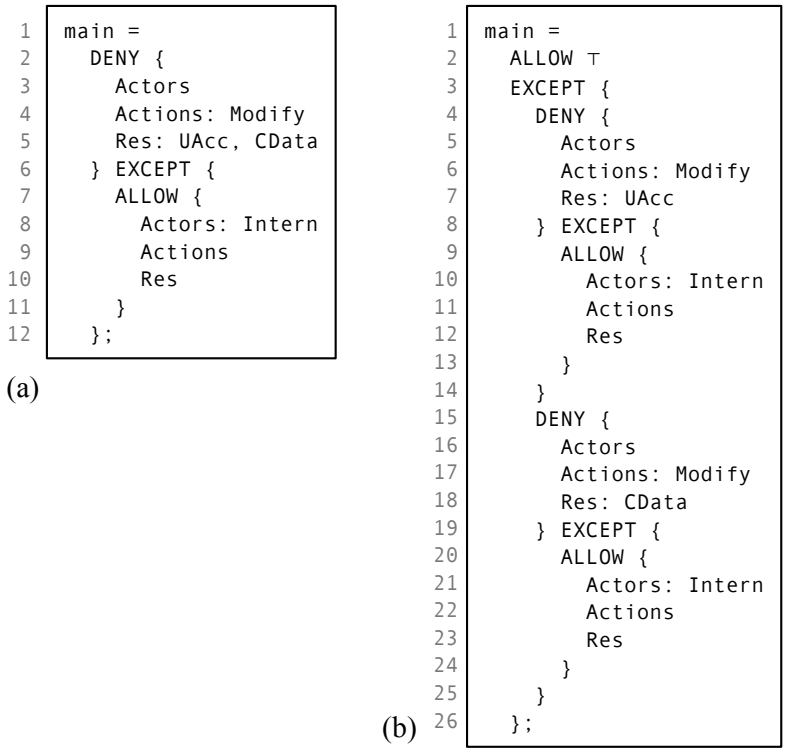
Aliases

C	=	`ALLOW | DENY`
T	=	$\{L\texttt{:}\ \ L_1, \ldots, L_n\}$
˜`ALLOW`	=	`DENY`
˜`DENY`	=	`ALLOW`

#	X		nm(X)
0	main = C EXCEPT ˜C+	→	main = C $\top$ EXCEPT nm(˜C+)
1	main = C T	→	main = ˜C $\top$ EXCEPT {nm(C T)}
2	C {T1: L1, ..., Lm}	→	C {T1: L1} ... C {T1: Lm}
3	C {T1: L1, ..., Lm} EXCEPT ˜C+	→	C {T1: L1} EXCEPT nm(˜C+) ... C {T1:Lm} EXCEPT nm(˜C+)
4	{C1 T1, ..., Cn Tn}	→	{nm(C1), ..., nm(Cn)}

Figure 8: Normalization rules.

Example 10. *Figure 9(a) shows a policy specification that is not in normal form: the* `DENY` *clause at Line 2 does not explicitly refer to any element, and the* `Res` *element at Line 5 refers to two terms. Figure 9(b) shows the same program, this time in normalized form. We use the symbol* $\top$ *as a surrogate to the topmost element in the underlying poset.*



```
main =
  DENY {
    Actors
    Actions: Modify
    Res: UAcc, CData
  } EXCEPT {
    ALLOW {
      Actors: Intern
      Actions
      Res
    }
  };
```

(a)

```
main =
  ALLOW ⊤
  EXCEPT {
    DENY {
      Actors
      Actions: Modify
      Res: UAcc
    } EXCEPT {
      ALLOW {
        Actors: Intern
        Actions
        Res
      }
    }
    DENY {
      Actors
      Actions: Modify
      Res: CData
    } EXCEPT {
      ALLOW {
        Actors: Intern
        Actions
        Res
      }
    }
  };
```

(b)

Figure 9: (a) Example of specification in Hapi. (b) Normalized counterpart of the specification.

4.3. Semantics

As already explained in Section 3.2, any Hapi program is equivalent to a set of tuples. Thus, interpreting such a program yields a finite set of tuples—each one of these tuples being an element of the product poset defined in the program's data declaration segment. This section explains how such interpretation can be carried out. To enable this explanation, we formally state the notion of an *access policy*: the result of the interpretation of a Hapi program.

Definition 4 (Access Policy). *An* access policy *is a subset of a product poset in which every element is formed exclusively by a combination of* atoms. *Given a poset P, an atom is any smallest element $a \in P$. Hence, if a is an atom, then, for any other element $e \in P$, it is not the case that $a > e$.*

4.3.1. From Programs to Access Policies

Syntactically, a normalized HAPI program is a set of *clauses*. Each clause is formed by three elements:

- *Type* $\in \{\texttt{DENY}, \texttt{ALLOW}\}$.
- *Elements* (*Elems*): any label in the product poset defined in the program's data declaration segment.
- *Exceptions*: a list of labels in the poset defined in the program's data declaration segment.

Example 11. *Figure 10(a) shows the five clauses that form the normalized* HAPI *program earlier seen in Figure 9. Notice that clauses* `D` *and* `E` *have empty* Exception *components.*

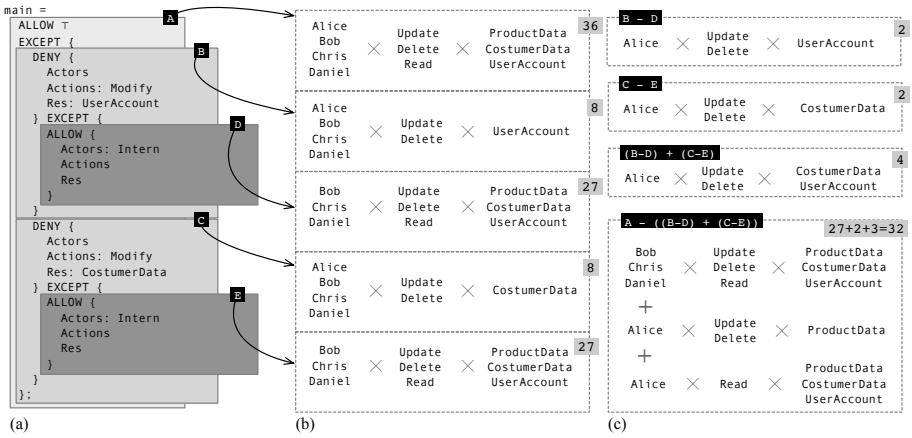


Figure 10: (a) The five clauses that constitute the normalized HAPI program earlier seen in Figure 9. (b): The tuples encoded by each clause. Gray boxes denote the cardinality of each set. (c): The access policy produced by the function `apply`.

If we let U denote the universe of atomic elements in the product poset, then a clause defines a subset of U. For clauses with empty *Exception* blocks, this subset if formed by the atomic elements less than or equal to the labels mentioned in the clause's *Elements* list. Example 12 illustrates this observation.

Example 12. *Figure 10(b) shows the elements that belong into each one of the five different clauses seen in the program in Figure 10(a), if we do not take into account the* Exception *blocks. Thus, for clauses* `D` *and* `E`*, which do not contain exceptions, the list of elements is complete. The numbers in grey boxes next to each product set gives the cardinality of that set.*

$$
\begin{array}{lcl}
\texttt{apply}(\text{Type:}\texttt{ALLOW}, \text{Elems:}\top, \text{Exceptions:}[C_i]_n) & \rightarrow & U - \bigcup_{i=1}^{n} \texttt{apply}\ (C_i) \\
\texttt{apply}\ (\text{Type:}\texttt{DENY}, \text{Elems: }\top, \text{Exceptions:}[C_i]_n) & \rightarrow & \bigcup_{i=1}^{n} \texttt{apply}\ (C_i) \\
\texttt{apply}\ (\text{Type:\_}, \text{Elems: L}, \text{Exceptions:}[C_i]_n) & \rightarrow & A(L) - \bigcup_{i=1}^{n} \texttt{apply}\ (C_i) \\
\texttt{apply}\ (\text{Type:\_}, \text{Elems: L}, \text{Exceptions:}[\,]_0) & \rightarrow & A(L)
\end{array}
$$

Figure 11: The function `apply`, that translates a normalized Hapi program into an access policy. We let U denote the universe of atomic elements in the product poset, and $A(L)$ denote all the atoms that are less than the elements in the list L.

As example 12 shows, computing the access policy of a clause that does not contain an *Exception* block is trivial. To find the access policy denoted by the other clauses, Figure 11 defines the `apply` function. This function translates a normalized Hapi program into an access policy. The first two cases in Figure 11 match the two forms possible for the initial clause of a normalized Hapi program. The initial clause of a normalized program refers, necessarily, to the top element in the poset ($\top$), and can be either `ALLOW` $\top$ or `DENY` $\top$. In the former case, the access policy includes the entire universe (U) of tuples, minus the exceptions. In the latter, the access policy includes only the tuples in the exceptions. The third clause yields the same set of tuples, regardless of its type (`ALLOW` or `DENY`). This clause aggregates all the atoms that are less than or equal the elements specified in the list L. Finally, the fourth clause aggregates elements from clauses without *Exceptions*. This fourth clause is the stop condition of the `apply` function.

Example 13. *Figure 10(c) shows the result of evaluating the program in Figure 10(a) with the* `apply` *function. Essentially, let C be a clause with exceptions given by subclauses C_1 and C_2. Assume that S is the set denoted by the elements of C (not considering exceptions). Likewise, let S_1 and S_2 denote the elements*

produced by `apply`(C_1) *and* `apply`(C_2)*, respectively. In this case,* $S \setminus (S_1 \cup S_2)$ *are the elements produced by* `apply`(C)*.*

We emphasize that checking if a given tuple belongs into a HAPI specification does not require the generation of all the tuples encoded by that specification—something that the `apply` function from Figure 11 does. A linear traversal of the specification suffices in this case. To see why such is the case, consider an access query, given by a certain tuple $t = (d_1, d_2, \ldots, d_n)$ from a product poset $D_1 \times D_2 \times \ldots \times D_n$. For any element $d_i \in T, 1 \leq i \leq n$, in the worst case, we can verify the status of d_i (*allowed* or *denied*) by traversing the HAPI access specification once. Considering that access policies tend to be short, performing a single query on a modern processor is an action that takes microseconds.

4.4. Properties

The semantics of HAPI ensures a few properties: *Commutativity*, *Termination*, *Totality* and *Consistency*. We state these properties below as theorems of trivial demonstration:

Theorem 1 (Commutativity). *The order in which clauses at the same nesting level appear in a* HAPI *program bear no consequence on the semantics of said program.*

> Follows from the commutativity of set union, the operator used by the `apply` function to join the information evaluated from different clauses at the same nesting level. □

Theorem 2 (Termination). *Function* `apply` *(Fig. 11) evaluates any* HAPI *program in a finite number of steps.*

> The proof of this theorem is trivial, as every recursive clause that makes up the `apply` function invokes recursion on a smaller part of the HAPI program. □

Theorem 3 (Totality). *For each tuple* t *of the product poset* P*, and any* HAPI *specification* C *ranging over* P*, either* C *allows or* C *denies* t*.*

> The application of the policy C over the product poset P yields a set of tuples T. If $t \in T$, then C allows t. Otherwise, C denies t. □

Theorem 4 (Consistency). *For any tuple t of the product poset P, and any* Hapi *specification C ranging over P, C cannot both allow and deny t.*

> The order in which the rules in Figure 11 are evaluated on C is deterministic: top-to-bottom, depth-first traversal on nesting order. Therefore, only one outcome can result of such an application. □

5. Tooling

There exists a small ecosystem of tools built around Hapi. These tools are publicly available, together with an e-book that describes the language specification. The tools are listed below, and explained in the rest of this section.

- **Parser**: a parser that translated Hapi specifications into an intermediate representation.
- **Translator**: an algorithm, implemented after the visitor design pattern [27], that traverses Hapi's intermediate representation and translates it to YAML files.
- **GUI**: an online graphic user interface featuring different visualization tools for Hapi programs.

5.1. Parser

We have developed a parser for Hapi. This parser is implemented on ANTLR [28], and invoked by a Kotlin driver. Kotlin was chosen for no special reason, other than the authors' personal preference for it. Parsing happens in two phases: the first pass builds the posets after reading the data declarations found in every Hapi specification. The second pass builds the Intermediate Representation (IR) denoting the Hapi specification. This intermediate representation—essentially an abstract syntax tree—can be processed by different visitors. Examples of visitors include pretty-printers, policy evaluators, translators and semantic checkers. Semantic verification detects, for instance, cyclic dependencies such as `data Foo = A(B), B(A)`.

Interoperation: Kotlin code inter-operates with Java's; hence, users willing to extend the Hapi framework are not restricted to a particular programming

language. In other words, code that traverses the data structures that represent Hapi programs can be written in Java, Scala, Groovy or any other language compatible with the Java Virtual Machine. We also provide mechanisms to serialize and deserialize Hapi's IR, so that even non-Java related languages can read and manipulate it, if necessary.

5.2. Translator

Next, seeing that YAML is currently used by a host of companies as a configuration file, we provide a translator that takes as input a Hapi (`.hp`) program, and outputs its YAML (`.yaml`) encoding. YAML follows `Cyral`'s protocol, which is the format that it was originally meant to replace. The YAML generator is currently restricted to policy specifications with a fixed declaration of data elements. Thus, input files (to the YAML generator only) must specify exactly three posets: Actors, Actions and Resources. Example 14 shows the YAML specification derived from a Hapi program. The decision to limit the translator was taken to satisfy the constraints imposed by `Cyral` on the layout of the generated YAML. Nevertheless, nothing hinders Hapi users from translating specifications into more complex encodings, be they written in YAML or in any other format.

Example 14. *Figure 12 shows the YAML file that results from translating the* Hapi *program first seen in Example 5. This figure is a snapshot of* Hapi*'s graphic user interface, yet to be discussed at Section 5.3.*

5.3. The Graphical User Interface

Hapi can be used through an online interface publicly available. Figure 13 shows the first page of this interface. Users may type in the code of the whole policy specification. They might also input specification files separately, along with a set of Hapi data declarations. This online interface then provides users with three different data-visualization tools:

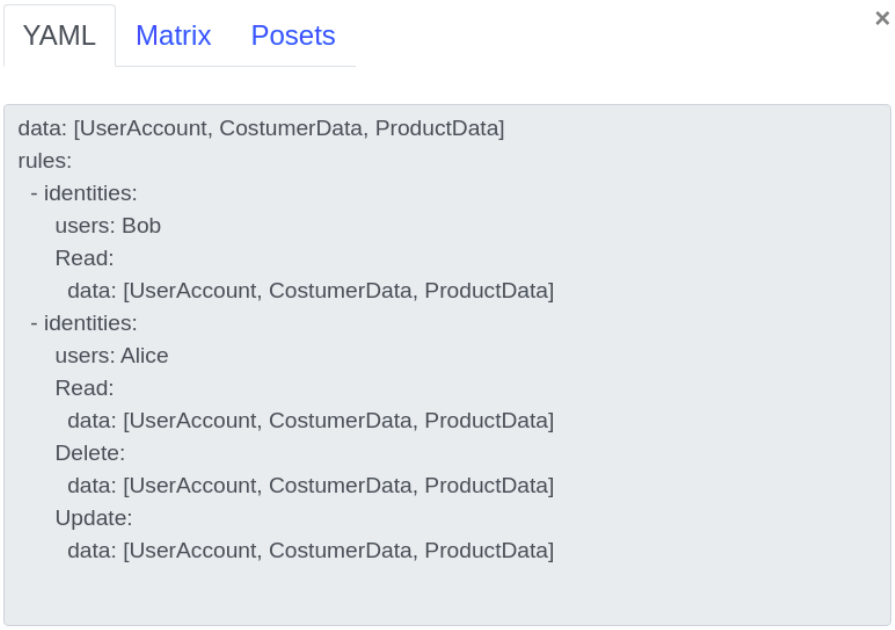


Figure 12: The YAML specification generated by the translator for the program earlier seen in Example 6.

- **Matrix**: the access matrix that represents an access policy. Figure 14 shows an example of an access matrix. Users can customize the information displayed via a search box (not shown in the figure).
- **Graph**: the different posets that constitute the data declaration. These posets are organized as graphs. The vertical height of nodes denote their ordering within the poset. Figure 15 shows the window of the graph visualizer, with three different posets.
- **YAML**: the YAML encoding translated from a Hapi specification (see Sec. 5.2) into `Cyral`'s protocol.

6. Code Compression

The number of different policy specifications necessary to establish all the valid relations between users of a distributed system can be very large. These

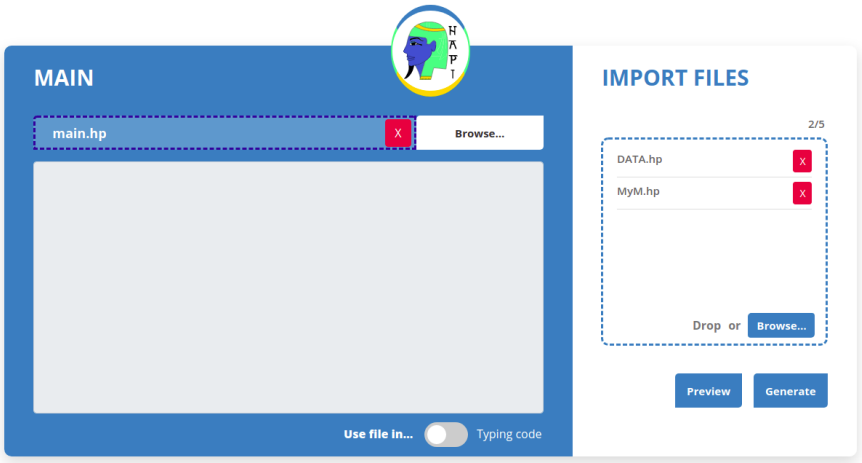


Figure 13: The main window used in the HAPI online interface. Users can type programs in the text box on the left, or they can input HAPI files using the search box on the right side of the window.

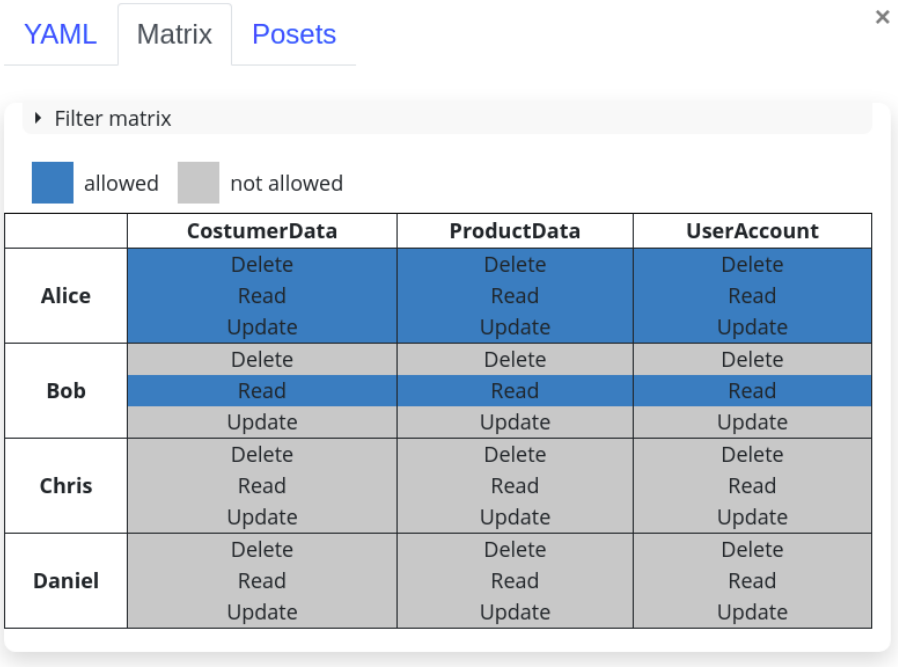
systems might comprise thousands of users, each one with particular types of access to different resources. The graphs formed by users, actions and resources can quickly achieve billions of edges. Therefore, storing policies in a compact way is of capital importance to these systems. To this end, HAPI provides users with a compressed format. This format consists of a YAML specification that meets two properties:

Readability Once in textual format, it can be conveniently read by human beings.

Denseness Once reduced via typical text compression algorithms, it can be compactly stored.

The compressed format of HAPI specifications resembles the YAML descriptors that Amazon uses to define access policies in the *AWS Identity and Access Management* (IAM) system[4]. However, keywords like `Type` and `Properties` have been eliminated, because HAPI itself contains very few reserved identifiers.

[4]As described at `https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-iam-policy.html`, last visited on January 26^{th}, 2022.



	CostumerData	ProductData	UserAccount
Alice	Delete Read Update	Delete Read Update	Delete Read Update
Bob	Delete Read Update	Delete Read Update	Delete Read Update
Chris	Delete Read Update	Delete Read Update	Delete Read Update
Daniel	Delete Read Update	Delete Read Update	Delete Read Update

Figure 14: An example of an access matrix as seen from the GUI. Users can customize the matrix, eliminating or adding rows and columns. Users can also choose which elements in the Hapi posets should be displayed.

Before explaining our rational behind this YAML version of a Hapi program, Example 15 will illustrate it.

Example 15. *Figure 16 shows a* Hapi *specification, and the corresponding YAML program. The YAML file in Figure 16 (b) yields the set of relations that the policy in Figure 16 (a), Lines 19-33 specifies.*

Semantics of YAML Encodings. The YAML translation of a Hapi specification encodes the access policy (Definition 4 in Page 16) denoted by that specification. In other words, the YAML version of a Hapi program contains the information necessary to list all the tuples that are ultimately generated by the interpretation of a Hapi specification. Figure 16 (c) shows the grammar of YAML files. Each rule (starting with a dash) denotes a cartesian product formed by all the combinations of elements in the nested lists. Example 16 clarifies this semantics.

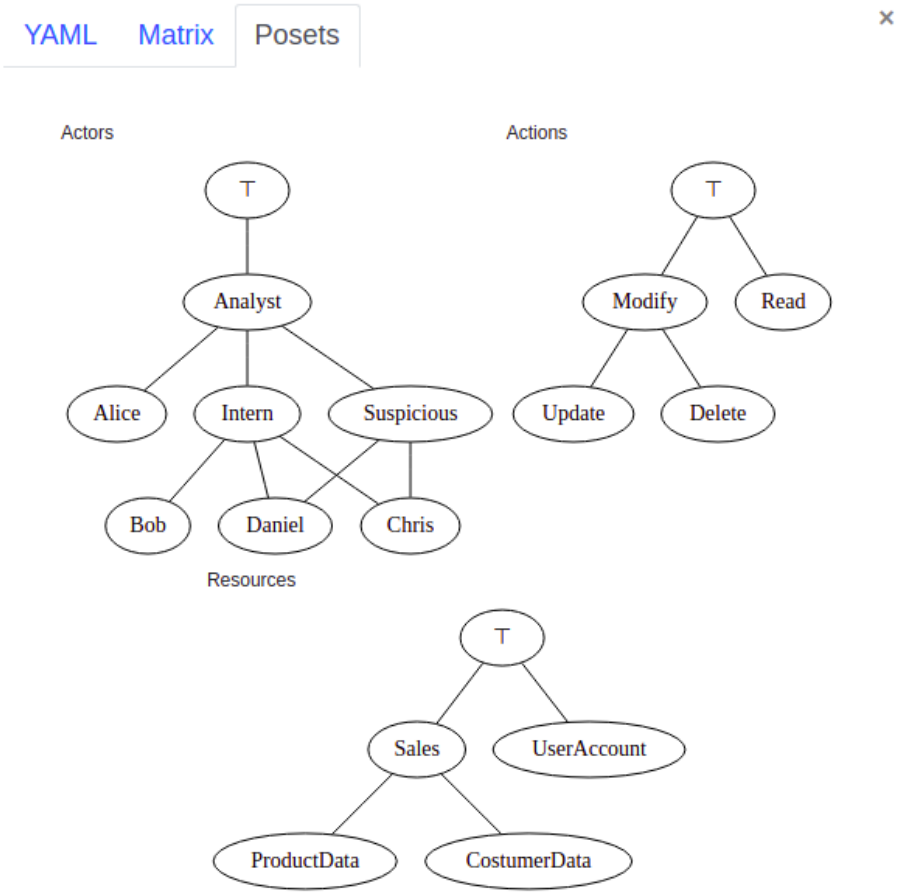


Figure 15: Graphic representation of three HAPI posets.

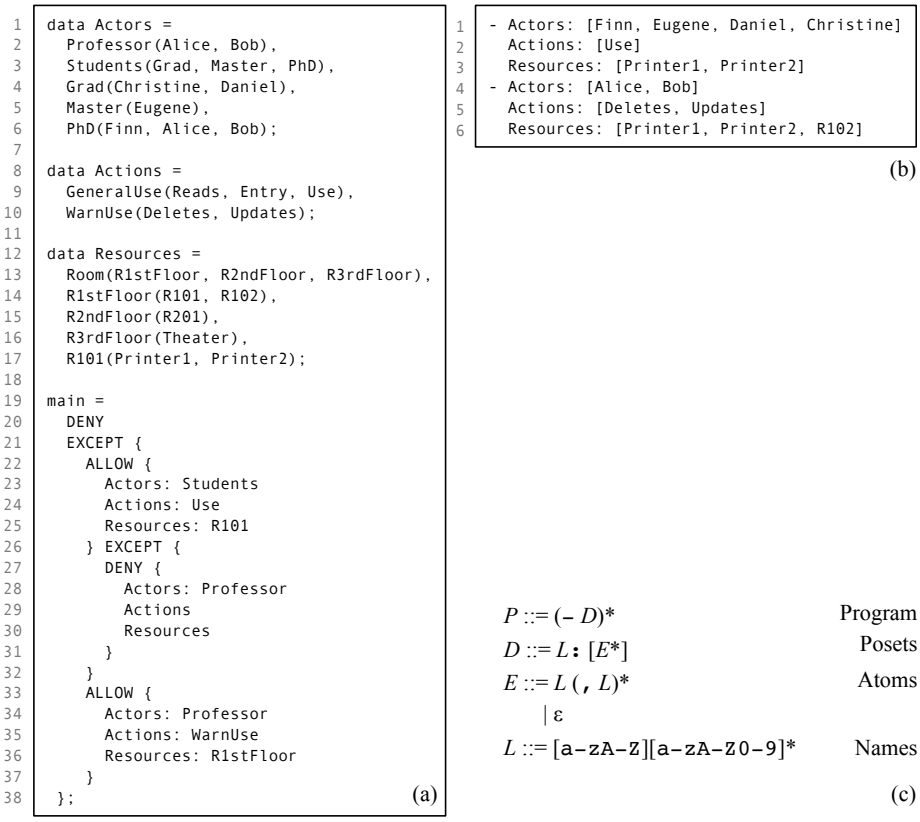
Example 16. *Figure 16 (b) represents 20 tuples—eight formed by the product:*

$\{\texttt{Finn}, \texttt{Eugene}, \texttt{Daniel}, \texttt{Christine}\} \times \{\texttt{Use}\} \times \{\texttt{Printer1}, \texttt{Printer2}\}$

The other twelve are formed by the product:

$\{\texttt{Alice}, \texttt{Bob}\} \times \{\texttt{Deletes}, \texttt{Updates}\} \times \{\texttt{Printer1}, \texttt{Printer2}, \texttt{R102}\}$

The YAML representation encodes only the *policy specification* of a HAPI program—it says nothing about the data declarations in that program, which must be stored separately. The rational behind this omission is that most of the textual redundancies in a HAPI program exist in the policy specification, not in the data declaration. In other words, small variations in the relations between



```
data Actors =
  Professor(Alice, Bob),
  Students(Grad, Master, PhD),
  Grad(Christine, Daniel),
  Master(Eugene),
  PhD(Finn, Alice, Bob);

data Actions =
  GeneralUse(Reads, Entry, Use),
  WarnUse(Deletes, Updates);

data Resources =
  Room(R1stFloor, R2ndFloor, R3rdFloor),
  R1stFloor(R101, R102),
  R2ndFloor(R201),
  R3rdFloor(Theater),
  R101(Printer1, Printer2);

main =
  DENY
  EXCEPT {
    ALLOW {
      Actors: Students
      Actions: Use
      Resources: R101
    } EXCEPT {
      DENY {
        Actors: Professor
        Actions
        Resources
      }
    }
    ALLOW {
      Actors: Professor
      Actions: WarnUse
      Resources: R1stFloor
    }
 };
```

(a)

```
- Actors: [Finn, Eugene, Daniel, Christine]
  Actions: [Use]
  Resources: [Printer1, Printer2]
- Actors: [Alice, Bob]
  Actions: [Deletes, Updates]
  Resources: [Printer1, Printer2, R102]
```

(b)

$P ::= (\texttt{-}\ D)^*$ Program

$D ::= L\texttt{:}\ [E^*]$ Posets

$E ::= L\ (\texttt{,}\ L)^*$ Atoms

$\quad\quad \mid \varepsilon$

$L ::= \texttt{[a-zA-Z][a-zA-Z0-9]}^*$ Names

(c)

Figure 16: (a): A complete Hapi specification. (b): The corresponding YAML specification. (c): The grammar of YAML encodings.

users and resources in a distributed service lead to new policy specification; however, they leave the data declaration unchanged. The YAML file is produced by a direct interpretation of function `apply` (Figure 11, Page 18).

Even if we disregard the data declaration module of a Hapi program, the function that converts this program into the YAML format is not a bijection. In other words, from a given Hapi policy specification, only one YAML representation can be derived—as long as the order in which relations are declared in the YAML file are not considered. However, it is possible to write the same YAML file with different Hapi specifications. That is to say that it is possible to build the same set of tuples by combining different sequences of `ALLOW` and `DENY` clauses applied onto different levels of the same poset.

Compression Efficiency. Figure 17 compares the—compressed—sizes of different HAPI files and their corresponding YAML encodings. The figure uses a set of synthetic benchmarks, produced out of different combinations of three posets. The YAML encoding does not include the data declaration of an access policy. Thus, for fairness, Figure 17 measures only the size of the access specification in the HAPI files. The synthetic benchmarks used in Figure 17 vary along the following dimensions:

- Number of *Elements* within each poset (2, 3, ..., 6).
- *Depth* of the most nested rule (2, 3, ..., 5).
- Number of rule *sequences* at the same nesting level (4, 5).

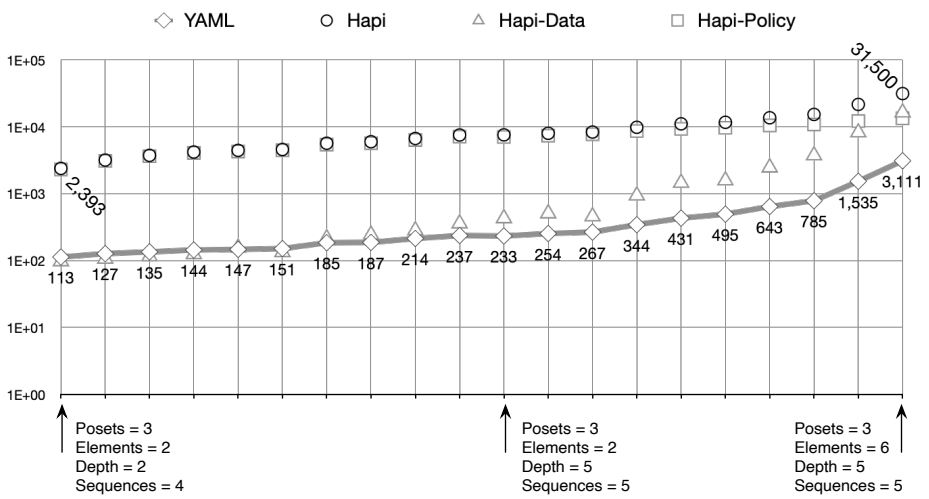


Figure 17: Size (in bytes) after compression using 21 different access policies specified onto the same set of data declarations.

Discussion. The compressed size of the YAML file is substantially smaller than the compressed size of the HAPI specification. Ratios vary from 10 to 32 times. For compression, we are using the `bzip2` algorithm for all—YAML and HAPI—files shown in Figure 17. Interestingly, the YAML format does not explore poset properties: it does not use the common ancestor of elements; instead, listing all the elements explicitly. However, we recall that this encoding has been designed

for compression. Thus, multiple instances of the same element end up replaced by a few bits of information. We found this approach simpler than using a more complex algorithm to replace elements with their common ancestor whenever all the offspring of this ancestor would be present in the specification.

7. Related Work

Many different languages can be used to enforce security or privacy policies [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. This section situates Hapi among them, in terms of syntax and semantics. Regarding the former, as we shall discuss in Section 7.1, Hapi uses a grammar that is independent from other data-specification formats, such as XML, JSON or YAML. Concerning the latter, Hapi is a role-based access control model with support for hierarchies. A Hapi specification allows the creation of roles in its data declaration part, and allows the assignment of permissions to these roles in its access specification part.

Hapi fits the second level of Sandhu *et al.*'s taxonomy [19], which they have named $RBAC_1$. Models in this category assign roles and subroles to users. Thus, the access rights of users are restricted according to the set in which these users belong. This model is *static*, as we shall discuss in this Section 7.2. In contrast, *Attribute-Based Access Policies* associate users with dynamic attributes, which are expected to vary according to time, space or other constraints. Section 7.3 closes our discussion about related work by reviewing how Hapi differs from Legalease, its main inspiration.

7.1. Syntax

Madani *et al.* [29] provides a comprehensive survey on languages used to specify access rights, covering the most well-known languages up to 2015. A more recent overview on the topic has been made available by Morel and Pardo [24] in the context of data-protection laws. Madani *et al.*'s work indicates that there are two main directions in which such access policies can be described: either via general data description languages (XML, JSON, YAML),

or via domain-specific languages. With regards to the former approach, XML is the most widely used format for the description of security policies. Testimony to this success is the prominence of languages like XACML, widely used to enforce access control in distributed systems [15], SAML [30] and many other languages built onto the XML syntax [5, 11, 12, 31, 13, 17]. Currently, JSON joins XML as *lingua franca* in the world of distributed services [32]. Several large service providers presently rely on this format to define access control policies[5].

Madani *et al.* [29] and Morel/Pardo [24] also present several examples of domain-specific languages (DSL) tailored for the declaration of access policies. DSLs tend to be more user friendly than general data description formats. In other words, DSLs generally seek to emulate the syntax of natural languages. Example 17 illustrates this aspect of two of them.

Example 17. *Figure 18 shows snippets written in two DSLs used to specify access rights:* SecPal *[10] and* `EPAL` *[17]. These languages define* dynamic *programs, in the sense that the access policy that they produce can change over time, based on the activation of particular conditions. As an example the* SecPal *policy in Figure 18 applied a boolean condition on a dynamic expression, e.g.:* `time()`. Hapi, *in turn, is based on a* static *semantics, for policies, once defined, are immutable.*

SecPAL

```
STS says Alice is a Researcher
FileServer says Alice can read /project
Alice says Cluster can read /project/data
 if time() < 07/09/2006
```

EPAL

```
ALLOW [Data User] TO PERFORM [Action] ON
[Data Type] FOR [Purpose] IF [Condition]
AND CARRY OUT [Obligation].
```

Figure 18: Access rights specified in two different DSLs: Epal and SecPal.

[5]For examples of policy specifications used in Microsoft's products, see `https://docs.microsoft.com/en-us/rest/api/media/operations/accesspolicy`. See `https://cloud.google.com/iam/docs/policies` for specifications used in Google's products, and `https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html` for Amazon's.

7.2. Semantics

Languages conceived to specify data-access policies differ on the mutability of specifications. Immutable specifications yield fixed access policies–they are *static*. In other words, the set of tuples determined by the access policy is fixed on time. Several of such languages describe immutable access policies. Such is the case of Hapi, Legalease, and several textual formats used by companies like Amazon (e.g., IAM JSON) or Microsoft (e.g., Azure Key Vault).

Many languages, in turn, allow users to specify access policies that change over time. These policies are *dynamic*. Mutability can be accomplished through a number of ways. A common kind of dynamic policy consists in associating time constraints with access rights. The Epal specification in Example 17 performs such association. Dynamic policies add *behavior* to data specifications. DSLs that fit into the dynamic category can be Turing Complete, for instance. Although we recognize the merits of mutability, we opted to maintain Hapi static for two reasons:

Simplicity: keeping specification and enforcement separate leads to a simpler syntax and less complex semantics. Hapi requires no logic or arithmetic expressions nor loop statements, for instance.

Flexibility: Hapi can be used in any system that supports storage of textual information. Enforcement can be implemented in different programming languages. Example 4 shows how Hapi policies can be enforced by a Python script.

As pointed out by Morel and Pardo [24], most machine-readable privacy policy languages have been designed with little impact to the industry. The lack of adoption, they argue, is related to the lack of a practical and scalable implementation. We believe this limitation is in part due to mutability requirements. Mutable policy specification languages are hard to integrate into existing systems. The benefits gained from applying these languages are often outweighed by the difficulties engineers encounter when trying to implement them. Hapi,

being a static language, can be deployed as a self-contained component into any workflow that uses JSON or YAML formats, for instance.

The semantics of any Hapi program is, ultimately, a set of tuples that specify an access policy, as described in Definition 4. This paucity of capabilities is in contrast to other security languages, such as Binder [33] and D1LP [34]. These languages usually have a data-specification core augmented with extra syntax to support services like communication and storage. Some of these languages, like Binder, support computations to the extent that they let programmers write predicates and inference rules. Example 18 compares an access specification in Binder and Hapi.

Example 18. *Figure 19 (a) shows a program taken from the original description of* Binder *[33]. Figures 19 (b,c) show the same specification written in* Hapi. Binder *is a subset of* Datalog*, which, in turn, borrows the bulk of its syntax and semantics from* Prolog*. In contrast to* Hapi, Binder *supports the association of contracts with cryptographic keys, and the communication of such contracts across a distributed environment. Contracts can be imported through the "*`says`*" predicate.*

Binder

```
can(X, read, resource_r) :-
  rsa:3:c1ebab5d says
  employee(X, bigco).

rsa:3:c1ebab5d says
employee(john_smith, bigco).
```

(a)

```
1  data Employee = BigCO(john_smith);
2
3  data Action = read;
4
5  data Resource = resource_r;
```

(b)

```
6  main = ALLOW {
7    Employee: BigCO
8    Action: read
9    Resources: resource_r
10 };
```

(c)

Figure 19: (a) Specification of access rights taken from Binder's original description. (b-c) Equivalent Hapi specification.

7.3. Hapi *and* Legalease

Hapi, as already mentioned in Section 1, materializes our understanding of Legalease. The core differences between these two domain-specific languages have already been explained in Section 1. Nevertheless, we emphasize that Hapi relies on a completely different interpreter to compute an access policy. These languages use different algebraic abstractions to organize policy rights:

whereas LEGALEASE is built around the notion of a lattice, HAPI uses a simpler structure: a partially ordered set. Thus, while any access policy that can be specified in LEGALEASE can also be described in HAPI, the opposite is not true, as Example 19 shows.

Example 19. *The* `Actors` *declaration seen in Figure 15 cannot be described in* LEGALEASE. *The partial order that characterizes the* `Actors` *poset does not form a lattice. To see why, notice that there is no single least upper bound between* `Daniel` *and* `Chris`*, for instance. Similarly, there is no single greatest lower bound between* `Intern` *and* `Suspicious`*.*

Example 19 indicates that the problem of converting a HAPI specification to LEGALEASE is not well-defined. A given poset might have multiple different *completions* [35]. If S is a poset, then a completion L of S is a lattice whose partially ordered set subsumes the ordering relations in S. Although it is possible to find a completion for any poset, this task might require the addition of new elements to the latter. Even the Dedekind–MacNeille completion, i.e., the smallest completion, might require the addition of new elements to the poset [36].

The disadvantage of HAPI's extra generality is speed: computing an access policy in LEGALEASE is faster—linear on the number of atoms. The same task in HAPI requires set addition and subtraction; hence, it is quadratic on the number of atoms. Nevertheless, conceiving an experiment to compare this difference in running time is not trivial. Access specifications, in practice, tend to be short; hence, the time to verify if a given tuple belongs or not into some access policy is negligible. These specifications are still meant to be read by human beings—long policies would go against this conception.

8. Conclusion

This paper has presented HAPI, a domain-specific language for the specification of access policies. HAPI is strongly based on the design of LEGALEASE, a DSL from Microsoft, that serves similar purposes: the declaration of access

rights. However, Hapi is different—and new—in many ways. Some of these differences are of fundamental importance, like the choice of the underlying algebraic representation of policies: Hapi uses partially ordered sets, whereas Legalease requires lattices. In hindsight, we believe that the option for posets was correct. The original definition of Hapi was, indeed, based on lattices; however, the need to meet lattice properties would put too much burden on developers. Seemingly intuitive declarations, like the `actors` definition in Figure 15 would be prevented by Legalease, while being allowed in Hapi. We believe that the price to pay for this extra flexibility is low: verifying if an element is allowed by a policy requires intersecting ordered sets in Hapi, whereas it amounts to computing lowest-bounds in Legalease.

Future Work. Hapi is still under development, and we hope, with time, to add more features to its ecosystem. An example of feature that we would like to incorporate into Hapi is the ability to associate a policy with a public cryptographic key, similar to what Binder [33] does. Incremental (Just-in-time) translation of Hapi specifications into YAML files is also something in our wish list. Currently, users might find that the need for frequent recompilation of large policies might take a non-negligible toll on running time.

Software. Hapi is free software. A GUI environment for Hapi is available at `http://cuda.dcc.ufmg.br/hapi/`. An online book about this DSL is available at `http://cuda.dcc.ufmg.br/hapidoc/`. The language's implementation is available at `https://github.com/lac-dcc/hapi`, under the GPL-3.0 license.

Acknowledgments

The first two authors of this paper have been supported by scholarships donated by `Cyral Inc` (`https://cyral.com/`). Fernando Pereira has been supported by CNPq (Grant 406377/2018-9); FAPEMIG (Grant PPM-00333-18) and CAPES (Edital CAPES PrInt).

References

[1] A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts, 8th Edition, Wiley Publishing, USA, 2008.

[2] J. Varia, S. Mathew, Overview of amazon web services (2014).

[3] R. J. Dudley, N. Duchene, Microsoft Azure: Enterprise Application Development, Packt Publishing, USA, 2010.

[4] B. Stonehem, Google Android Firebase: Learning the Basics, First Rank, USA, 2016.

[5] R. Agrawal, J. Kiernan, R. Srikant, Y. Xu, XPref: A preference language for P3P, Comput. Netw. 48 (5) (2005) 809–827.

[6] A. H. Anderson, A comparison of two privacy policy languages: EPAL and XACML, in: SWS, Association for Computing Machinery, New York, NY, USA, 2006, p. 53–60. doi:10.1145/1180367.1180378.
URL https://doi.org/10.1145/1180367.1180378

[7] P. Ashley, S. Hada, G. Karjoth, M. Schunter, E-P3P privacy policies and privacy authorization, in: WPES, Association for Computing Machinery, New York, NY, USA, 2002, p. 103–109. doi:10.1145/644527.644538.
URL https://doi.org/10.1145/644527.644538

[8] A. Barth, A. Datta, J. C. Mitchell, H. Nissenbaum, Privacy and contextual integrity: Framework and applications, in: Security and Privacy, IEEE Computer Society, USA, 2006, p. 184–198. doi:10.1109/SP.2006.32.
URL https://doi.org/10.1109/SP.2006.32

[9] M. Y. Becker, A. Malkis, L. Bussard, A practical generic privacy language, in: S. Jha, A. Mathuria (Eds.), ICISS, Vol. 6503 of Lecture Notes in Computer Science, Springer, 2010, pp. 125–139. doi:10.1007/978-3-642-17714-9_10.

[10] M. Y. Becker, C. Fournet, A. D. Gordon, SecPAL: Design and semantics of a decentralized authorization language, J. Comput. Secur. 18 (4) (2010) 619–665.

[11] K. Bohrer, B. Holland, Customer profile exchange (cpexchange) specification (2000).
URL http://www.ctiforum.com/standard/standard/www.cpexchange.org/cpexchangev1_0F.pdf

[12] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, J. Reagle, The platform for privacy preferences 1.0 (p3p1. 0) specification (2002).
URL https://elearn.inf.tu-dresden.de/hades/teleseminare/wise0405/Act.%208%20Models%20Languages%20Pierangela/Materials/P3P.pdf

[13] J. Iyilade, J. Vassileva, P2U: A privacy policy specification language for secondary data sharing and usage, in: SPW, IEEE Computer Society, 2014, pp. 18–22. doi:10.1109/SPW.2014.12.
URL https://doi.org/10.1109/SPW.2014.12

[14] L. Kagal, T. W. Finin, A. Joshi, A policy language for a pervasive computing environment, in: POLICY, IEEE Computer Society, 2003, p. 63. doi:10.1109/POLICY.2003.1206958.
URL https://doi.org/10.1109/POLICY.2003.1206958

[15] M. Lorch, S. Proctor, R. Lepro, D. Kafura, S. Shah, First experiences using XACML for access control in distributed systems, in: XMLSEC, Association for Computing Machinery, New York, NY, USA, 2003, p. 25–37. doi:10.1145/968559.968563.

[16] D. Metayer, A formal privacy management framework, in: FAST, Springer-Verlag, Berlin, Heidelberg, 2009, p. 162–176.
URL https://doi.org/10.1007/978-3-642-01465-9_11

[17] W. H. Stufflebeam, A. I. Antón, Q. He, N. Jain, Specifying privacy policies with P3P and EPAL: Lessons learned, in: WPES, Association for Computing Machinery, New York, NY, USA, 2004, p. 35. doi:10.1145/1029179.1029190.

[18] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, J. M. Wing, Bootstrapping privacy compliance in big data systems, in: S&P, IEEE Computer Society, USA, 2014, p. 327–342. doi:10.1109/SP.2014.28.

[19] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, Role-based access control models, Computer 29 (2) (1996) 38–47. doi:10.1109/2.485845.
URL https://doi.org/10.1109/2.485845

[20] V. Julião, A. Holmquist, F. Lúcio, C. Simões, F. Pereira, Hapi: A domain-specific language for the declaration of access policies, in: SBLP, Association for Computing Machinery, New York, NY, USA, 2021, p. 9–16. doi:10.1145/3475061.3475084.

[21] V. Juliao, A. Holmquist, C. Simoes, F. M. Q. Pereira, Hapi, the definitive guide, http://cuda.dcc.ufmg.br/hapidoc/ (2021).

[22] A. Erickson, Comparative analysis of the EU's GDPR and brazil's LGPD: Enforcement challenges with the LGPD, Brooklyn Journal of International Law 2 (9) (2019) 859–.

[23] A. M. Geden, T. K. Bensghir, Reflections from GDPR to turkish data protection act in the context of privacy principles, in: IMISC, Ankara Yildrim Beyazit University, Ankara, Turkey, 2018, pp. 118–122.

[24] V. Morel, R. Pardo, Sok: Three facets of privacy policies, in: Workshop on Privacy in the Electronic Society, Association for Computing Machinery, New York, NY, USA, 2020, p. 41–56.
URL https://doi.org/10.1145/3411497.3420216

[25] B. W. Lampson, Protection, SIGOPS Oper. Syst. Rev. 8 (1) (1974) 18–24. doi:10.1145/775265.775268.

[26] A. Gerl, Modelling of a privacy language and efficient policy-based de-identification, Ph.D. thesis, U. Passau and INSA Lyon (2019).

[27] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[28] T. J. Parr, R. W. Quong, ANTLR: A predicated-LL(k) parser generator, Softw. Pract. Exper. 25 (7) (1995) 789–810. doi:10.1002/spe.4380250705.

[29] S. Kasem-Madani, M. Meier, Security and privacy policy languages: A survey, categorization and gap identification, CoRR abs/1512.00201 (2015) 1–18. arXiv:1512.00201.
URL http://arxiv.org/abs/1512.00201

[30] S. Saklikar, S. Saha, Next steps for security assertion markup language (saml), in: SWS, ACM, New York, NY, USA, 2007, p. 52–65. doi:10.1145/1314418.1314427.

[31] A. Gerl, N. Bennani, H. Kosch, L. Brunie, LPL, towards a GDPR-compliant privacy language: Formal definition and usage, Trans. Large Scale Data Knowl. Centered Syst. 37 (2018) 41–80. doi:10.1007/978-3-662-57932-9_2.
URL https://doi.org/10.1007/978-3-662-57932-9_2

[32] H. X. Son, N. M. Hoang, A novel attribute-based access control system for fine-grained privacy protection, in: ICCSP, Association for Computing Machinery, New York, NY, USA, 2019, p. 76–80. doi:10.1145/3309074.3309091.

[33] J. DeTreville, Binder, a logic-based security language, in: Proceedings of the 2002 IEEE Symposium on Security and Privacy, SP '02, IEEE Computer Society, USA, 2002, p. 105.

[34] N. Li, B. N. Grosof, J. Feigenbaum, Delegation logic: A logic-based approach to distributed authorization, ACM Trans. Inf. Syst. Secur. 6 (1) (2003) 128–171. doi:10.1145/605434.605438.
URL https://doi.org/10.1145/605434.605438

[35] B. Schroder, Ordered Sets, Birkhauser, Germany, 2016.

[36] B. Davey, H. Priestley, Introduction to Lattices and Order, Cambridge University Press, Cambridge, UK, 2002.