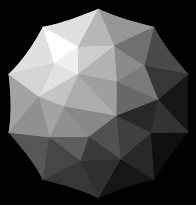
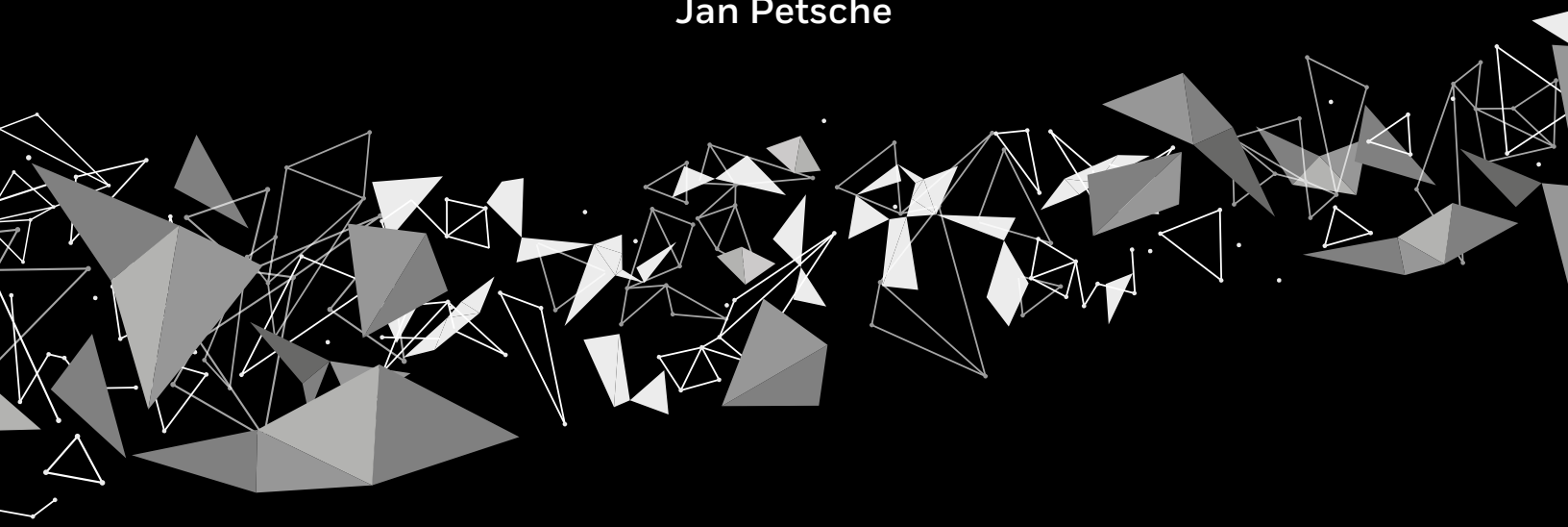


XAIN.

The FROST Language

A trusted and user-centric access control language: Enabling delegation of fine-grained policies in shared ecosystems

Kwok Cheung
Michael R. A. Huth
Laurence M. Kirk
Leif-Nissen Lundbæk
Rodolphe Marques
Jan Petsche



Yellow Paper: Version 0.8.1

Contents

1 Introduction

1.1 Motivation & Aims of FROST

It has long been recognized that we have entered an era in which the physical and perhaps legal possession of resources has become less important than the ability to access and use them. In the more recent past, data – in particular the so called *Big Data* – has become another precious resource, named by some as the "new oil" that can fuel future economies. As in the case of physical resources, the value of Big Data also resides in the ability to access it, for example for machine learning and knowledge inference and transfer techniques.

It is less clear, though, how this shift from possession to access will manifest itself in social, commercial, and cultural ways. For example, some argue that people will increasingly seek and get access to "pre-paid experiences" [37]; others may predict that people will stop owning personal cars; and so forth. Such predictions are useful for policy makers, urban planners, and others. But what interests and motivates us here is the widely shared observation that industry verticals and service sectors are increasingly forced or incentivized to create and participate in digital ecosystems that push the established understanding of organizational boundaries. And that these emerging ecosystems will have common technological needs that have to be met regardless of whether particular trend predictions – such as the cessation of personal car ownership – will be correct or not.

Let us identify some forces and developments at play in shaping such shared ecosystems:

- The digitization of resources and services, and their rich data streams, require a multitude of stakeholders to share access during the life cycles of resources and data.
- Technological innovations such as blockchain challenge conventional ICT infrastructures that rely on a central authority in their access and service provision.
- Inadequate ways of data sharing across organizational boundaries – the proverbial email attachment of cvs files due to reluctance of exposing database APIs to external parties.
- Increased user demands on personalized, fine-grained services that make users active participants rather than passive consumers.
- Changes in both data privacy regulation and privacy perception of users, who seek more control over their data and trust central authorities less.

Businesses can therefore create economic value through innovations that address the above shortcomings and also seize the above opportunities. But they can no longer do this in isolation, given that products and services will be facilitated through economic platforms that are shared by other commercial, potentially competing, players.

For example, the manufacturer of a car may wish to maintain a numerical passport on the vehicle and retain access to such data even when the car changes ownership. Similarly, a potential buyer of a used car may need to gain access to some of that passport information in order to evaluate the purchase prize. Or someone may want to lease the car from a dealer and also allow family members to use the car within some restrictions that she formulates.

It is clear that such a vision requires a supporting layer of technology through which such fine-grained access and delegation of access to resources and data can be securely articulated and enforced. This technology should also operate at a suitable level of abstraction so that it can easily be integrated with particular data fabrics and communication transport layers. Additionally, it should empower the owners and users of resources and data, rather than exposing them to the potential abuse of access and data by a central party.

This Yellow Paper therefore presents FROST, a technology that realizes the above vision by creating a low-level technology layer on which shared ecosystems can built products and services around the sharing of access to resources and data. If we compare such a shared ecosystem with a city, it is pleasing to note that the Greek word for city, *Polis*, has two meanings: one of them referring to the city itself, and another one referring to the community of its citizens.

Our aspiration is that FROST will similarly take on these meanings: offering the ability to shape ecosystems with rules and organizational structures as found in cities, but also empowering its citizens to thrive in that space.

1.2 Intended Audience

This yellow paper means to be accessible to a broad range of readers: scientists, engineers in particular verticals such as Automotive, Blockchain developers, practitioners in Information Security and Enterprise Systems and – to a lesser extent – decision makers in Business, Policy, and Governance. As a consequence, we will at times oversimplify technical presentations but also strive to provide enough technical content so that experts may judge the merits of the suggested overall approach.

1.3 Outline of Yellow Paper

Our policy language for access control is introduced in Section 2. A mathematically equivalent representation of such policies in terms of Boolean Circuits, as well as verification tools based on the synthesis of such circuits, are the subject of Section 3. In Section 4, we describe our approach to specifying and computing obligations for a request made under a policy; this completes the description of the mathematical representation of policies within our implementation. A detailed account of our approach to delegation along bounded delegation chains, including the ability to delegate the writing of policies, is discussed in Section 5. A review of potential attack vectors on our architecture and their potential mitigation is presented in Section 6. Exemplary use cases of our architecture are briefly outlined in Section 7 and we offer a concluding summary of our approach in Section 8.

2 Our Access-Control Policy Language FROST

2.1 Access Control Framework

The area of *Access Control* provides models, methods, and tools for controlling which agents – be they humans, mechanical devices or AIs – have access to what resources and under which circumstances. Many modern access-control architectures formulate the intended access control in a *policy*. One advantage of policy-based access control is that it supports the composition and mobility of controls – making it ideal for open and distributed systems.

Access control is a key security mechanism in enterprise systems, operating systems, physical buildings – to name a few prominent applications. An access-control policy may be somewhat static, e.g., saying which areas in an airport terminal are accessible to passengers who went through a security check already. However, such policies may also be more dynamic and allow for the delegation of access privileges to other agents. In our use case below, e.g., we see how the owner of a vehicle can grant a delivery person access to the trunk of the vehicle under certain conditions – say, within a given time interval. Access to the trunk then creates an *obligation*, to trigger a notification of the owner that access did indeed occur.

The conceptual architecture of FROST's access-control framework is shown in Figure 1. Device or resource owners manage policies in a policy-administration point (PAP). Policies are either stored within a Policy Store on an embedded device or sent as payload in a cybersecurity protocol, depending on constraints such as storage capacity of embedded clients. If needed, a Policy Retrieval Point (PRP) selects the appropriate policy for a submitted access request from the policy store.

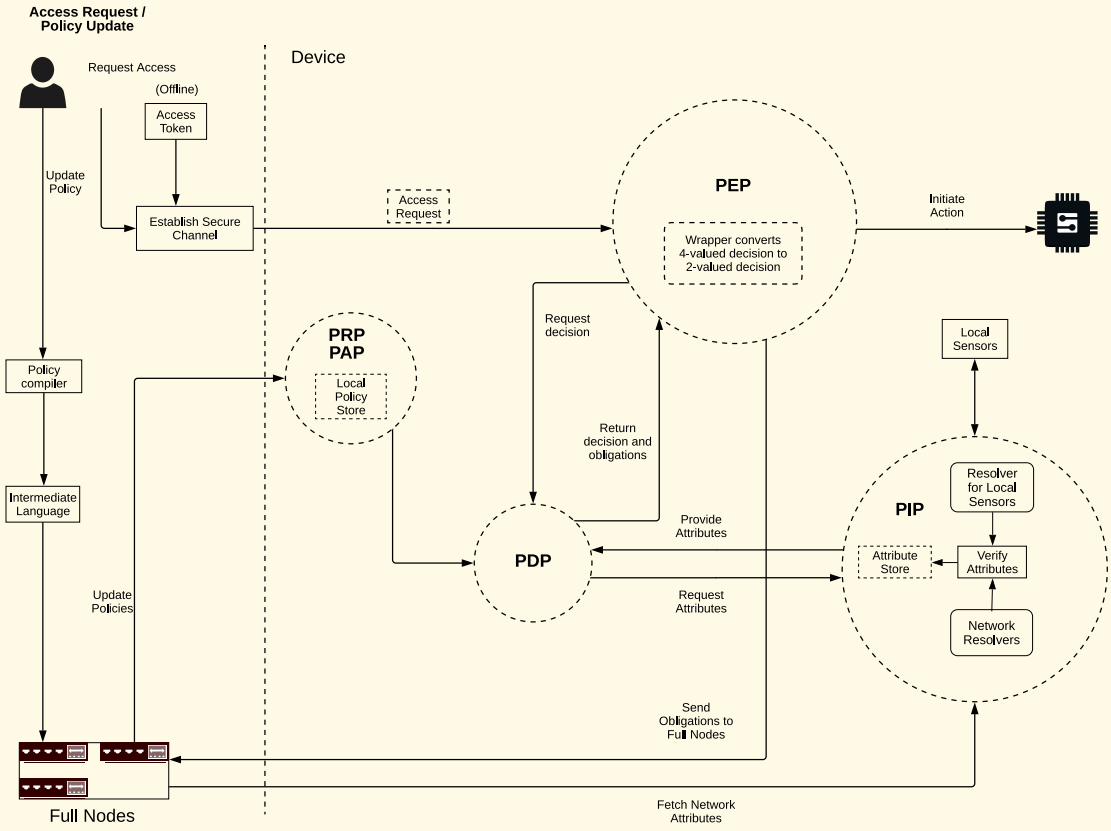


Figure 1: Architecture diagram of the FROST access-control framework

The user communicates with the PAP and the resource via a secure channel, where server authentication is separated from user authentication. The user's knowledge of the resource will be obtained through out-of-band means.

A request to access a resource can, in the abstract, be seen as a tuple

$$(user, resource, action, context) \tag{1}$$

where $user$ is the machine, human, computer method or other entity making the request – known as the *requester*; $recource$ is the object of this request; $action$ is the concrete action requested (e.g. reading data, opening a trunk or editing a policy in a policy store); and $context$ is supplied by the requester, system or environment and contains any contextual information (e.g. sensor readings) pertaining to and informing this request.

Such a request reaches the policy-decision point (PDP) that retrieves the appropriate policy for ruling on this request from the PRP in order to compute a decision and (optionally) a set of obligations; in the simplest form, a sole policy is stored in that store and said policy is evaluated

for all issued requests. In addition to $context$ provided by the requester, the PDP may consult policy-information points (PIP) to assist in the evaluation of the policy. The computed decision (notably grant or deny, but maybe logging the request, reporting a conflict or indicating a lack of a decision) and obligations are then communicated to the policy-enforcement point (PEP).

The task of the PEP is to enact that decision and to deal with operational complexities of the request, device interaction or input from the PDP. For example, the device may be unable to enact the requested action or the decision computed by the PDP may express a conflict or lack of knowledge. Our architecture focuses such operational issues and their resolution within the PEP, as these may also be device-specific. This makes the PDP more generic and reusable.

The remaining parts of this architecture, shown in Figure 1, illustrate the intended *agnostic nature* of this approach: the handling of policies in transit and at rest, as well as the administrative aspects such as logs of requests and decisions can be accommodated on a variety of data platforms – including distributed databases, blockchains, and cloud environments.

Supplied information for evaluating a policy is typically organized into *attributes*. The latter conveniently abstract subjects, resources, and contexts into those properties about them that should inform the decision of a policy made on a given request; see e.g. [20] for such a language designed for open systems. This attribute-based approach brings obvious advantages over conventional approaches such as access-control lists – to name scalability, composability, and intuitive formulation of policies.

It also aligns with privacy needs in identity management, e.g., the use of pseudonyms or support for the attribute "being of legal age" as an abstraction of an identify. An attribute may be application-specific (e.g. the type of cargo in a container) or more universally known (e.g. the age of a user or the current time in UTC). The authentication and integrity checks of attribute *values* clearly need to be supported, whenever information sources allow for this.

Therefore, we base our approach to user-centric, distributed, and resilient access control on an *extensible, attribute-based policy language* called FROST. Extensibility merely reflects that specific use cases or application domains require domain-specific attributes whose syntax and semantics will then be plugged into our FROST language without affecting its own semantics, structure, and tool chain. The name FROST is an acronym that stands for:

F = **Flexible**, e.g. the freedom in the delegation of access and policy writing, and the agnostic view on back-ends

R = **Resilient**, e.g. our use of cybersecurity protocols and other hardening mechanisms such as policy authentications

O = **Open**, e.g. that the entire architecture can host an emerging ecosystem of activity, and that we plan to publish our approach as open-source technology

S = **Service-Enabling**, e.g. for user-centric abilities to deliver goods, or as a transparent and resilient means of tracking parts in their life-cycle

T = **Trusted**, e.g. through its combination of cybersecurity technology, validation tools, and support for secure and transparent audits.

Let us now describe this FROST language and its representational layers.

2.2 Policy Language for FROST

The FROST language we propose may be seen as an instance of the language PBel and we here want to expressly acknowledge and apply the contributions made on PBel in the previously

published, scientific papers [11, 12]. A basic form of policy is captured in a *rule*. We propose two simple types of rules from which more complex policies can be formed:

$$\texttt{grant if } cond \qquad\qquad \texttt{deny if } cond \tag{2}$$

where $cond$ is a logical expression built from attributes, their values and comparisons, as well as logical operators. For example, we may specify an access-control rule

$$\begin{aligned} \texttt{grant if}\quad & (\mathbf{object} == vehicle)\ \&\&\ (\mathbf{subject} == vehicle.owner.daughter)\ \&\& \\ & (\mathbf{action} == driveVehicle)\ \&\& \\ & (owner.daughter.isInsured == true)\ \&\& \\ & (0900 \leq localTime)\ \&\&\ (localTime \leq 2000). \end{aligned} \tag{3}$$

This rule implicitly refers to the request made by the daughter of the owner of the vehicle to drive that car. This rule applies only if the requested resource is that vehicle, the action is to drive that vehicle, and the requester is the daughter of the owner of that vehicle. In those circumstances, access is granted if the daughter is insured and the local time is between 9am and 8pm. The intuition is that this rule does not apply whenever its condition $cond$ evaluates to false, including in cases in which the request is not of that type.

Policy Composition in Distributed Environments In open and distributed systems, rules may not always apply to a request. This suggests that we want support for a third outcome `undef` which states that the rule or policy has no opinion on the made request, a so called *policy gap* at which the policy is "undefined". Semantically, a rule of form `grant if` $cond$ is therefore really a shorthand for

$$\texttt{grant if } cond \texttt{ else undef} \tag{4}$$

and a similar shorthand meaning applies to rules of form `deny if` $cond$.

The open and distributed nature of a system will often generate situations in which more than one rule may apply; an overall access-control policy may thus have evidence for `grant` as well as evidence for `deny`. Such an apparent conflict should be made explicit in the language itself, so that policy compositions can reflect and then appropriate act on it. This suggests a fourth value `conflict` as policy outcome, to denote such conflict as a result of policy evaluation.

It should be stressed that decisions `undef` and `conflict` are not enforceable by a PEP. But they are important to have and compute over for at least two reasons:

- these values can facilitate policy composition, e.g. a policy that returns `undef` may be *ignored* within a composition with another policy, and
- policy analysis can discover requests for which policies may have gaps or conflicts, to aid the verification of the correctness of policies, their refinements, and compositions.

The conceptual grammar for our access-control FROST language is depicted in Figure 2. The syntactic category dec ranges over all four possible policy decisions `grant`, `deny`, `undef`, and `conflict`. A $term$ either refers to a $constant$ (the set of constants may subsume keywords such as $\mathbf{subject}$, $\mathbf{action}$, and $\mathbf{object}$), an $entity$ (a variable), a term indexed by an $attribute$ or the application of an n-ary operator op to n many terms (where $n \geq 1$). The choice of n-ary operators is in part domain-specific but in part also generic (e.g. operators for arithmetic). Note that terms are assumed to be well-typed given their intended meaning, which our implementation does reflect. The index operation allows us to write terms such as $vehicle.owner.daughter$ where our type discipline will rule out conditions such as $vehicle.owner.daughter < localTime$.

$$
\begin{aligned}
dec &::= \texttt{grant} \mid \texttt{deny} \mid \texttt{undef} \mid \texttt{conflict} \\
term &::= constant \mid entity \mid op(term, \ldots, term) \mid term.attribute \\
cond &::= (term == term) \mid (term < term) \mid (term \leq term) \mid \cdots \\
&\qquad \texttt{true} \mid \neg cond \mid (cond \;\&\&\; cond) \mid (cond \;||\; cond) \\
rule &::= \texttt{grant if } cond \mid \texttt{deny if } cond \\
guard &::= \texttt{true} \mid pol\ \texttt{eval}\ dec \mid (guard \;\&\&\; guard) \\
pol &::= dec \mid rule \mid \texttt{case}\ \{\ [guard\colon pol]^{+}\ [\texttt{true}\colon pol]\ \}
\end{aligned}
$$

Figure 2: Grammars for our attribute-based access-control FROST language

The syntactic clause $cond$ for conditions captures propositional logic with `true` denoting truth, $\neg$ negation, $\&\&$ conjunction, and $||$ disjunction; its atomic expressions are obtained by relational operators applied to terms. The set of relational operators will contain generic ones such as those for equality and inequality (say over the integers) but also more domain-specific ones – e.g. a ternary one saying that an entity is the parent of two other entities.

The category $rule$ is defined as already discussed. A policy pol is either a constant policy dec, a rule $rule$ or a `case`-statement. The latter contains at least one case, where a case is a pair of a $guard$ and a policy pol and the last case has the catch-all guard `true`. Guards are essentially conjunctions of expressions of form pol `eval` dec which specify that policy pol evaluates to decision dec. A `case`-statement evaluates to the first policy pol_i of a pair $guard_i\colon pol_i$ within that `case`-statement for which $guard_i$ is true. It is helpful to think of the policies within guards as well as the pol_i as *sub-policies* of the composition that is specified by a `case`-statement.

Further note that the grammar for terms is extensible and may contain clauses that plug in domain-specific aspects. In an advanced digital manufacturing plant, e.g., we may be interested in a ternary operation on terms denoting that two specific robots which carry vehicle chassis on them approach the same team of workers.

For a policy pol, let us assume that the PIPs are able to obtain values for all attributes occurring in pol, so that the values of all terms of the policy pol can be computed. From the values for terms, we can then compute the truth values of all atomic conditions within policy pol. And from that we may follow the semantics of rules and of `case`-statements (the latter used for policy composition) to compute the outcome of this policy – which is either `grant`, `deny`, `undef` or `conflict`.

2.3 Functional Completeness

Let us write

$$\mathbf{4} = \{\texttt{grant}, \texttt{deny}, \texttt{undef}, \texttt{conflict}\} \tag{5}$$

for this set of possible policy decisions. Ideally, we would like to have a set of policy composition operators that can express all possible functions of type $\mathbf{4}^n \to \mathbf{4}$ for arities $n \geq 0$ where composition operators for $n = 0$ merely return a decision seen as a constant policy.

Note that the clause pol `eval` dec together with conjunction allows us to express any one of the 4^n many rows in a "truth table" for a function f of type $\mathbf{4}^n \to \mathbf{4}$. This means that syntactic clause $guard$ can express any such truth table row. Thus, a `case`-statement can go through all the 4^n cases of that function f and return in each case the decision specified by f for that row as a constant policy.

This means that the FROST language is functionally complete as a policy composition language: for any $n \geq 0$ and any function $f\colon \mathbf{4}^n \to \mathbf{4}$ we can define a bespoke syntactic operator over n policies that has the meaning of f and is expressive in the policy language of Figure 2. This is not just a theoretical result. It also gives us assurance that any policy composition needs that arise in use cases or specific application domains can be expressed as "macros" that compile into the above policy language, provided the meaning of such compositions is a function of the meanings of its argument policies.

Let us illustrate this result for the operation `join`. The intuition of P `join` Q is that it combines the results of policies P and Q so that the result is an *information join*. Namely, that this join returns

- the result of one of these policies if the other one returns `undef`,
- `conflict` if one of the policies returns `conflict`; or if one of them returns `grant` and the other one returns `deny`,
- the result of policy P in all other cases (which will be the same result as that of Q).

We can express this in the FROST language as seen in Figure 3. The encoding involves only 7 cases and not $4^2 = 16$ cases as the above method for "truth tables" would construct.

```
case {
  [(P eval undef): Q]
  [(Q eval undef): P]
  [(P eval conflict): conflict]
  [(Q eval conflict): conflict]
  [((P eval deny) && (Q eval grant)): conflict]
  [((P eval grant) && (Q eval deny)): conflict]
  [true: P]
}
```

Figure 3: Example of a derived policy composition operator expressed in the FROST language, the *information join* P `join` Q of the results of two policies P and Q.

3 Intermediate Language and Policy Verification

PEPs can only enforce policies that are free of gaps and conflicts; such policies are Boolean conditions – true for exactly those requests that will be granted. We now define two compilations of policies into such Boolean conditions which can be used to both verify that policies written in the FROST language are enforceable and for generating the to be enforced conditions, which implementations may represent in different form – e.g., as Boolean Circuits [42] or Binary Decision Diagrams [14].

These compilations provide also the formal underpinnings for a whole range of policy analyses that have important applications, such as a formal verification that a Boolean circuit does indeed faithfully represent a particular access-control policy written in our policy language.

3.1 Compiling Policies into Boolean Conditions

In [11, 12], the four possible decisions `grant`, `deny`, `undef` and `conflict` were represented in the 4-valued Belnap bi-lattice depicted in Figure 4. On this bi-lattice, we can define two subsets: $GoC = \{\texttt{grant}, \texttt{conflict}\}$ and $DoC = \{\texttt{deny}, \texttt{conflict}\}$. We can uniquely identify an element of that bi-lattice by saying whether or not it is a member of GoC ("grant or conflict") and of DoC ("deny or conflict"). For example, the element `undef` is the unique one that is neither in GoC nor in DoC. And `grant` is the unique element that is in GoC but not in DoC.

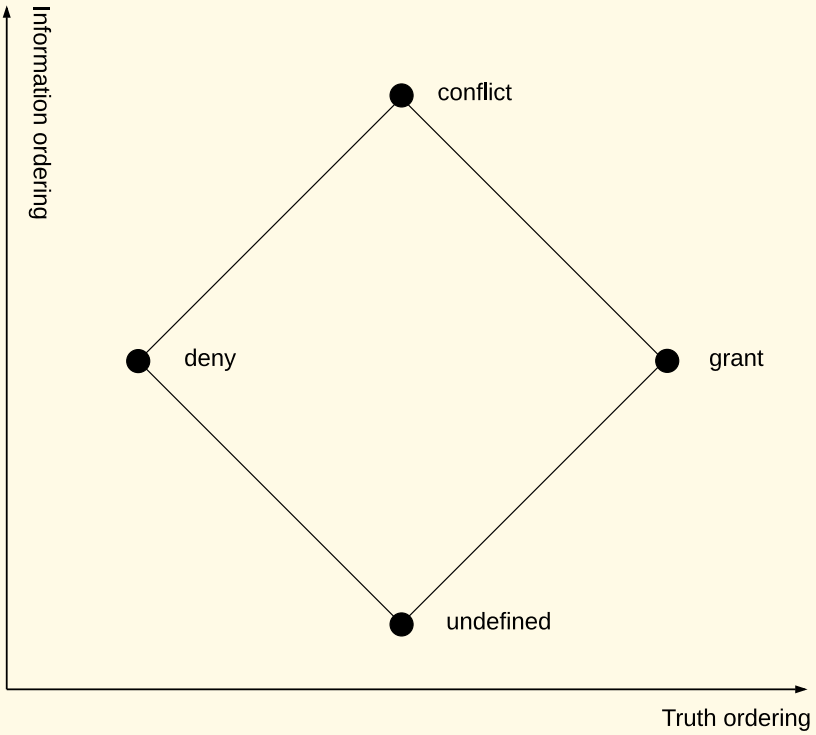


Figure 4: The 4-valued Belnap bi-lattice, as used for access control in [11, 12]. The x-axis represents the *truth* ordering whereas the y-axis represents the *information* ordering.

This simple observation holds the key to representing a policy pol in a `join` normal form as

$$pol \;\equiv\; (\texttt{grant if } \mathsf{GoC}(pol)) \texttt{ join } (\texttt{deny if } \mathsf{DoC}(pol)) \tag{6}$$

where $\mathsf{GoC}(pol)$ is a Boolean (2-valued) expression specifying the exact conditions under which pol computes to `grant` or `conflict`, and where $\mathsf{DoC}(pol)$ is a Boolean (2-valued) expression specifying the exact conditions for when pol computes to `deny` or `conflict`. The correctness/exactness of the specifications $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ is what makes the representation of pol in (6) correct.

The expressions $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ are defined inductively over the grammar of policies for the FROST language, shown in Figure 5. These definitions make use of an auxiliary predicate $\mathsf{T}(guard)$ that specifies the exact conditions for when a guard expression $guard$ is true. We point out that the definition of $\mathsf{T}(guard)$ appeals to the predicates $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ – making these three predicates mutually recursive.

The formal proof that these expressions $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ correctly specify the behavior of a policy pol is a variant of the proofs given in [11, 12] for a similar but different setting of policy language.

$$\mathsf{GoC}(\texttt{grant}) \equiv \mathsf{true} \qquad \mathsf{GoC}(\texttt{deny}) \equiv \mathsf{false}$$
$$\mathsf{GoC}(\texttt{conflict}) \equiv \mathsf{true} \qquad \mathsf{GoC}(\texttt{undef}) \equiv \mathsf{false}$$
$$\mathsf{GoC}(\texttt{grant if } cond) \equiv cond \qquad \mathsf{GoC}(\texttt{deny if } cond) \equiv \mathsf{false}$$

$$\begin{aligned}\mathsf{GoC}(\texttt{case}\,\{\,[g_1 \colon p_1]\,\ldots\,[g_{n-1} \colon p_{n-1}]\,[\mathsf{true} \colon p_n]\,\}) \equiv \\ (\mathsf{T}(g_1)\ \&\&\ \mathsf{GoC}(p_1))\ ||\ (\neg\mathsf{T}(g_1)\ \&\&\ \mathsf{T}(g_2)\ \&\&\ \mathsf{GoC}(p_2))\ ||\ \cdots \\ \cdots\ ||\ (\neg\mathsf{T}(g_1)\ \&\&\ \ldots\ \&\&\ \neg\mathsf{T}(g_{n-1})\ \&\&\ \mathsf{T}(\mathsf{true})\ \&\&\ \mathsf{GoC}(p_n))\end{aligned}$$

$$\mathsf{DoC}(\texttt{deny}) \equiv \mathsf{true} \qquad \mathsf{DoC}(\texttt{grant}) \equiv \mathsf{false}$$
$$\mathsf{DoC}(\texttt{conflict}) \equiv \mathsf{true} \qquad \mathsf{DoC}(\texttt{undef}) \equiv \mathsf{false}$$
$$\mathsf{DoC}(\texttt{deny if } cond) \equiv cond \qquad \mathsf{DoC}(\texttt{grant if } cond) \equiv \mathsf{false}$$

$$\begin{aligned}\mathsf{DoC}(\texttt{case}\,\{\,[g_1 \colon p_1]\,\ldots\,[g_{n-1} \colon p_{n-1}]\,[\mathsf{true} \colon p_n]\,\}) \equiv \\ (\mathsf{T}(g_1)\ \&\&\ \mathsf{DoC}(p_1))\ ||\ (\neg\mathsf{T}(g_1)\ \&\&\ \mathsf{T}(g_2)\ \&\&\ \mathsf{DoC}(p_2))\ ||\ \cdots \\ \cdots\ ||\ (\neg\mathsf{T}(g_1)\ \&\&\ \ldots\ \&\&\ \neg\mathsf{T}(g_{n-1})\ \&\&\ \mathsf{T}(\mathsf{true})\ \&\&\ \mathsf{DoC}(p_n))\end{aligned}$$

Figure 5: Compilation $\mathsf{GoC}(pol)$ generates a condition that is true iff pol returns `grant` or `conflict`. Similarly, $\mathsf{DoC}(pol)$ is true iff pol returns `deny` or `conflict`. Both depend on function $\mathsf{T}(guard)$ that specifies the condition for when $guard$ evaluates to true, defined in Figure 6.

$$\mathsf{T}(\mathsf{true}) \equiv \mathsf{true}$$
$$\mathsf{T}(g_1\ \&\&\ g_2) \equiv \mathsf{T}(g_1)\ \&\&\ \mathsf{T}(g_2)$$
$$\mathsf{T}(pol\ \texttt{eval}\ dec) \equiv \begin{cases} \mathsf{GoC}(pol)\ \&\&\ \ \mathsf{DoC}(pol) & \text{if } dec \text{ equals } \texttt{conflict} \\ \neg\mathsf{GoC}(pol)\ \&\&\ \ \mathsf{DoC}(pol) & \text{if } dec \text{ equals } \texttt{deny} \\ \mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol) & \text{if } dec \text{ equals } \texttt{grant} \\ \neg\mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol) & \text{if } dec \text{ equals } \texttt{undef} \end{cases}$$

Figure 6: Compilation of a guard into an equivalent condition from syntactic category $cond$. Its compilation of expressions pol `eval` $cond$ uses $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ of Figure 5.

It is easy to show that expressions of form $\mathsf{T}(guard)$, $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ are such that they are generated by the grammar for syntactic category $cond$ in Figure 2. It is in that sense that we may think of $cond$ as an intermediate language for policies – keeping in mind that the outputs of two such expressions may have to be combined in a 4-valued join as specified in (6).

3.2 Policy Verification

With these predicates at hand, it is now easy to express important policy analyses as satisfiability problems over the logic and theories that interpret atomic conditions, and the attributes that they contain. For example, a policy pol from the FROST language is gap free iff the predicate $\neg\mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol)$ is unsatisfiable. Similarly, a policy pol from the FROST language is

conflict free iff $\mathsf{GoC}(pol)$ $\&\&$ $\mathsf{DoC}(pol)$ is unsatisfiable. In particular, if a policy pol is gap free and conflict free, then we may use $\mathsf{GoC}(pol)$ as an intermediate representation of this policy. This is a Boolean "circuit" that evaluates to true if the policy grants an access request, and to false if the policy denies an access request.

This technology can also be used to verify the integrity of a representation of a policy pol from our FROST language.

Example 1 (Verification)**.** *Suppose, e.g., that φ is a Boolean Circuit that is claimed to faithfully represent pol in that truth of the circuit means* `grant` *and falsity means* `deny`*. Anyone can then verify this claim by*

1. *generating the conditions $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ as specified in Figure 5,*
2. *using a formal verification tool to show that*
   - *(a) $\neg\mathsf{GoC}(pol)$ $\&\&$ $\neg\mathsf{DoC}(pol)$ is unsatisfiable, and so pol is indeed gap free,*
   - *(b) $\mathsf{GoC}(pol)$ $\&\&$ $\mathsf{DoC}(pol)$ is unsatisfiable, and so pol is indeed conflict-free,*
3. *verifying that $\mathsf{GoC}(pol)$ and φ are logically equivalent.*

If any of the verification tasks in items 2 or 3 fail, this means that the integrity of φ has been disproved; otherwise, we have proof that φ indeed faithfully represents the policy pol.

A similar set of tasks can show that φ_{GoC} faithfully represents $\mathsf{GoC}(pol)$ and φ_{DoC} faithfully represents $\mathsf{DoC}(pol)$ for a policy pol that may neither be free of gaps nor free of conflicts. This would then verify the integrity of these two Boolean Circuits, which essentially capture the join normal form of that policy pol.

We may also verify other aspects of policy administration. In change management, e.g. we may need assurance that an updated policy pol' is not more permissive than some currently used policy pol. For example, a proof that

$$\mathsf{T}\big(pol' \texttt{ eval grant}\big) \;\&\&\; (\mathsf{T}(pol \texttt{ eval undef}) \;||\; \mathsf{T}(pol \texttt{ eval deny}))$$

is unsatisfiable would show that policy pol' can never grant whenever policy pol either denies or has a gap.

The verification tools one would use for these and other validation tasks may vary. Rewrite rules may capture equational theories of operators for transforming policies into equivalent ones and tools could perform such rewrite logic. This can also be helpful for simplifying policies in a manner that is meaningful to users.

Deeper semantic analyses may be obtained by the use of solvers for SMT ("Satisfiability Modulo Theories" [21]) such as the OpenSMT solver [13], or through the use of theorem provers such as Isabelle [5, 34]. And these powerful and mature tools from formal methods may also be used to compress Boolean conditions into provably equivalent ones. Such compression of policies will be particularly beneficial if policies need to be stored and executed on resource-constrained devices.

A PDP may need to process a policy that contains gaps or conflicts. For example, a policy may be submitted off-line through a cybersecurity protocol and so the PDP cannot make any assumptions about the semantic behavior of that policy. A PDP may therefore need to *wrap* this policy, at the top-level, into an idiom that makes the composed policy enforceable for a PEP. A

policy wrapper that forces all conflicts and gaps of a policy pol to be denials would then be

```
case {
  [pol eval undef: deny]
  [pol eval conflict: deny]
  [true: pol]
}
```

This *deny-by-default* composition pattern can not only be expressed in our policy language but may also be hard-coded in a PDP. For example, let a PDP execute two Boolean circuits φ_{GoC} and φ_{DoC}, where these circuits capture the meaning of $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$, respectively. The PDP may then combine the outputs of these circuits as seen in the truth table in Figure 7.

$\mathsf{GoC}(pol)$	$\mathsf{DoC}(pol)$	output of PDP
true	true	deny
true	false	grant
false	true	deny
false	false	deny

Figure 7: Combining outputs of circuits $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ so that the combination honors all grants and denials of pol but overrides all gaps and conflicts of pol into denial.

3.3 Domain-Specific Policy Languages

We envision the creation of more abstract, declarative languages in which access-control policies may be codified and that compile into our FROST language. App developers can then articulate access controls via these more abstract languages.

At this more abstract level, one may also guide programmers by offering programming idioms that already come with desired guarantees. For example, in [11, 12] fragments of a policy language were devised that can be interpreted as type systems for policy composition where well-typed policies are free of gaps, conflicts or both (depending on the used type system). Use of such idioms will bring value to app development for access control. Also, specific domains such as the manufacturing floors of SMEs may have common policy patterns that could be documented, shared, reused, and adapted across organizations. Thus we have potentially a whole host of different domain-specific languages (DSLs), each tailored to the particular domain but all built upon our core FROST language.

Broadly speaking, there are two main approaches to building a DSL. We may consider it a "standalone" language, whereby it becomes necessary to build the various stages of a standard compiler pipeline—lexer / parser, code generator, and so forth. DSLs are often small in comparison to general-purpose programming languages, e.g. they are typically non-Turing complete. Therefore, a common and often more convenient approach is to *embed* such a DSL within a more general-purpose host language. More often that not, this eliminates the need to build a separate lexer and parser, for example. Clearly, the more powerful the host language, the more the DSL can leverage from the features provided by its host language.

In the above, we have described policies defined in our FROST language with a declarative, compositional structure. That is, policies can be combined together in different ways to form larger composite policies. This suggests that a declarative language—in particular, a typed functional language—makes for a natural host for our FROST language. And indeed,

embedded DSLs are in general a well-suited use case for functional programming [25]. Implementation of an embedded DSL in this way then essentially reduces to writing a library of combinators, i.e. functions that create, use or combine in various ways a data type of policies. Importantly, these combinators are not necessarily user-facing; rather, it is the syntactic sugar written in terms of these combinations that would comprise the "primitives" of the DSL that a policy author would interact with.

For instance, our FROST language and its intermediate form are agnostic to, but support, the use of patterns that may facilitate policy writing within DSLs. In the following example, a DSL may want to attach a *target* T to a policy pol saying that the policy only applies whenever T is true, otherwise it returns `undef`. We may write this operator in the form pol `if` T, which generalizes the syntax for rules but is expressible in our FROST language. In particular, one merely has to specify how such a construct compiles into our policy language, without having to modify other parts of the tool chain such as the functions $\mathsf{GoC}(P)$ and $\mathsf{DoC}(P)$ used for policy analysis. Figure 8 shows one way in which this operator may be defined in our FROST language.

```
case {
  [((grant if T) eval grant): pol]
  [true: undef]
}
```

Figure 8: Encoding of a policy operator pol `if` T in our FROST language: this restricts policy pol to a target condition T. The verbose and indirect use of testing whether T is true, seen in expression (`grant if` T) `eval grant`, is an artifact of the syntactic structure of guards.

4 Accountability Through Obligations

It is useful to be able to articulate that a certain policy decision, if enacted, creates certain obligations which the ambient system expects to be fulfilled, e.g. within a certain period of time. One design choice is to make such obligations into decisions themselves, e.g. as in a rule of form $create\_log$ `if` $cond$. But this introduces many different types of decisions, and thus complicates the functionality of the PDP and PEP as well as policy analysis.

Our approach is, instead, to extend our FROST language so that obligations are optional annotations of policy terms. A simple approach is to annotate occurrences of rules so that the grammar for $rule$ becomes

$$rule \ ::= \ \texttt{grant}\,\{obl^*\}\ \texttt{if}\ cond \ \mid \ \texttt{deny}\,\{obl^*\}\ \texttt{if}\ cond$$

where $\{obl^*\}$ refers to a finite (possibly empty) set of obligations associated with a decision. In a rule, we interpret $dec\{\,\}$ as dec, a decision that contains no obligations. The static policy analyses discussed above – e.g. whether a policy is free of gaps – can then ignore such annotations.

The language for expressing obligations is not proscribed by our FROST language but needs to be interpretable by PEPs, e.g. to fulfill an obligation of creating a log entry or of notifying a certain user that an action took place. In fact, a PEP may need to deny an access request if the obligation associated to a `grant` cannot be fulfilled. Moreover, there needs to be an understanding of which obligations stated in rules within a policy would be "triggered" when a policy computes to `grant` or `deny`.

We note that policies owned by different entities may be composed, and so an approach to obligations need to be mindful of this. Consider for example

$$\begin{array}{l}\texttt{case}\ \{\\ \quad [(P\ \texttt{eval grant})\ \&\&\ (Q\ \texttt{eval deny})\texttt{: deny}]\\ \quad [\texttt{true}\texttt{: } P]\\ \}\end{array} \tag{7}$$

which allows policy Q to override a grant of policy P into a deny but where all other requests are dealt with as in P. This models when the owner of P allows the owner of Q to be less permissive whenever P would grant.

If the first case above applies, we expect that the obligations are those that stem from policy Q's denying. Moreover, it would be counter-intuitive to also add obligations for P's denying (since the outcome is a grant) and also not desirable to add obligations for P's grant (since policy P would not grant on the made request).

4.1 Obligations Aggregated from Policy Composition

We now describe how the semantics of our policy language allows us to compute the set of obligations that corresponds to the decision computed by that policy. This semantics is consistent with how obligations and their combination are dealt with in XACML (see e.g. [32]).

Let ρ be a model for our condition language $cond$ so that we may define a Tarskian truth semantics $\rho \models cond$, saying that $cond$ is true in model ρ. For obligations, we are only interested in the decisions `grant` and `deny`, requiring computations for each of these: $\mathsf{oblg}(\texttt{grant}, pol, \rho)$ computes the obligations of granting for policy pol under ρ, whereas $\mathsf{oblg}(\texttt{deny}, pol, \rho)$ computes the obligations of denying for policy pol under ρ. The set of obligations for a policy pol with outcome dec, given a model ρ, is defined in Figure 9.

These definitions specify that constant policies do not trigger any obligations, and that rules trigger their specified obligations whenever the rules don't return `undef`. For `case`-statements, the obligation set for `grant`, by way of example, is computed as the union of the obligation set for `grant` of the "continuation" policy p_i and the obligation set for `grant` of the corresponding guard g_i. Note that this ignores obligations of guards and policies whose cases do not apply, and it ignores guards and policies whose case is not the first one that applies. (Alternative definitions would be possible here.)

The computation of obligation sets for guards triggers no obligations for the default guard `true` and interprets conjunctions of guards as unions of obligation sets. For guards of form pol `eval` dec' it either computes no obligations (when dec' is not the decision for which obligations are collected) or computes the obligation set for policy pol and decision dec should the latter equal dec'.

Let us illustrate this semantics by considering a model ρ for which the first case of the statement in (7) applies. Let us write W for the policy defined by that `case`-statement. Then:

$$\begin{aligned}\mathsf{oblg}(\texttt{deny}, W, \rho) &= \mathsf{oblg}'(\texttt{deny}, (P\ \texttt{eval grant})\ \&\&\ (Q\ \texttt{eval deny}), \rho) \cup \mathsf{oblg}(\texttt{deny}, \texttt{deny}, \rho)\\ &= \mathsf{oblg}'(\texttt{deny}, P\ \texttt{eval grant}, \rho) \cup \mathsf{oblg}'(\texttt{deny}, Q\ \texttt{eval deny}, \rho) \cup \{\,\}\\ &= \{\,\} \cup \mathsf{oblg}(\texttt{deny}, Q, \rho)\\ &= \mathsf{oblg}(\texttt{deny}, Q, \rho)\end{aligned} \tag{8}$$

Of course, other definitions of obligations for composed policies may apply. For example, one may think that pol `eval conflict` should inherit obligations for `grant`, respectively `deny`.

$$\mathsf{oblg}(dec,\, dec',\, \rho) \;\equiv\; \{\,\}$$

$$\mathsf{oblg}(dec,\, \texttt{grant}\,\{obl^*\}\,\texttt{if}\; cond,\, \rho) \;\equiv\; \begin{cases} \{obl^*\} & \text{if } dec = \texttt{grant} \text{ and } \rho \models cond \\ \{\,\} & \text{otherwise} \end{cases}$$

$$\mathsf{oblg}(dec,\, \texttt{deny}\,\{obl^*\}\,\texttt{if}\; cond,\, \rho) \;\equiv\; \begin{cases} \{obl^*\} & \text{if } dec = \texttt{deny} \text{ and } \rho \models cond \\ \{\,\} & \text{otherwise} \end{cases}$$

$$\mathsf{oblg}(dec,\, \texttt{case}\,\{\,[g_1 : p_1]\;\ldots\;[g_{n-1} : p_{n-1}]\,[\mathsf{true} : p_n]\,\},\, \rho) \;\equiv\; \mathsf{oblg}'(dec,\, g_i,\, \rho) \cup \mathsf{oblg}(dec,\, p_i,\, \rho)$$
$$\text{where } \rho \models \mathsf{T}(g_i) \text{ and } \rho \not\models \mathsf{T}(g_j) \text{ for all } j < i$$

$$\mathsf{oblg}'(dec,\, \mathsf{true},\, \rho) \;\equiv\; \{\,\}$$

$$\mathsf{oblg}'(dec,\, pol\;\texttt{eval}\;dec',\, \rho) \;\equiv\; \begin{cases} \mathsf{oblg}(dec,\, pol,\, \rho) & \text{if } dec = dec' \\ \{\,\} & \text{otherwise} \end{cases}$$

$$\mathsf{oblg}'(dec,\, g_1\;\&\&\;g_2,\, \rho) \;\equiv\; \mathsf{oblg}'(dec,\, g_1,\, \rho) \cup \mathsf{oblg}'(dec,\, g_2,\, \rho)$$

Figure 9: A compositional, deterministic computation of sets of obligations for policy pol from our FROST language under model ρ where dec is in $\{\texttt{deny}, \texttt{grant}\}$. Expression $\mathsf{oblg}(dec, pol, \rho)$ computes the set of obligations that arise when policy pol has outcome dec under model ρ. The expression $\mathsf{oblg}'(dec, guard, \rho)$ similarly computes the set of obligations that arise for guard $guard$ pertaining to outcome dec. These functions are mutually recursive. Only the guard g_i that gives rise to a "continuation" policy contributes any obligations.

4.2 Obligation Circuits

The compilation of obligations occurring within a policy into circuits is more involved than the compilations $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ of policies. This is so since rules may either contribute obligations (if the rule applies and has stated obligations) or not (if the rule does not apply or no obligations are stated within it). This means that we need to represent such conditionals within the circuit logic to ensure that the correct set of obligations gets synthesized. The inductive definitions for $\mathsf{oblg}(dec, pol, \rho)$ and $\mathsf{oblg}'(dec, guard, \rho)$ shown in Figure 9 also illustrate this need for conditionals (explicitly for rules and implicitly for $case$-statements).

These specifications also inductively define such circuits that, for a model ρ, evaluate to a possibly empty set of obligations: circuit $\mathsf{GObl}(pol)$ specifies the exact conditions that compute a set of obligations for policy pol whenever the latter evaluates to decision `grant` under an environment ρ; similarly, circuit $\mathsf{DObl}(pol)$ specifies the computation of the set of obligations for the policy pol whenever the latter evaluates to decision `deny` under an environment ρ.

As in the case of Boolean circuits, application of formal methods may simplify the circuits $\mathsf{GObl}(pol)$ and $\mathsf{DObl}(pol)$ into provably equivalent ones that are still meaningful for users and storable on resource-constrained devices. For instance, consider the compilation in (8) where we already simplified the resulting circuit $\mathsf{DObl}(W)$, since the first step would actually be

$$\mathsf{oblg}(\texttt{deny},\, W,\, \rho) = \begin{cases} \mathsf{oblg}'(\texttt{deny},\, G,\, \rho) \cup \mathsf{oblg}(\texttt{deny},\, \texttt{deny},\, \rho) & \text{if } \mathsf{T}(G) \\ \mathsf{oblg}'(\texttt{deny},\, \mathsf{true},\, \rho) \cup \mathsf{oblg}(\texttt{deny},\, P,\, \rho) & \text{if } \neg\mathsf{T}(G)\;\&\&\;\mathsf{T}(\mathsf{true}) \end{cases}$$

where G equals $(P\;\texttt{eval}\;\texttt{grant})\;\&\&\;(Q\;\texttt{eval}\;\texttt{deny})$. An optimization would replace the second case of the conditional clause with the empty set if no obligations would be computed for P for

decision `deny`. Note that the values at leaves are now sets, not Boolean values. Algebraic decision diagrams [4] or variants thereof may therefore be a suitable data structure for representing and optimizing obligation circuits.

4.3 Mathematical Representations of FROST Policies

The semantics that computes the access-control decision of a FROST policy pol is straightforward, given a model ρ that evaluates all atomic conditions that occur in pol. Intuitively, ρ tells us the truth value for any condition $cond$ in pol – using the familiar semantics of propositional logic. The meaning of sub-policies is then given by the stated interpretations of rules, guards, and $case$-statements. The semantics for computing the set of obligations that arise when the computed decision is either `grant` or `deny` has been defined in Figure 9.

One may then capture both of these computations in a structural operational semantics [36], which may be executed on a Landin-style abstract machine (see e.g. [33]) – which would serve as the specification of a run-time engine for the PDP. This would thus interpret a FROST policy directly to compute a decision and obligation set, which would be passed to the PEP.

But there are other options for processing such policies. For example, we may represent pol in equivalent form by two pairs of circuits:

- a pair of *Boolean Circuits* that represent $(\mathsf{GoC}(pol), \mathsf{DoC}(pol))$ and so indirectly represent the decision behavior of pol, and
- a pair of *Obligation Circuits* $(\mathsf{GObl}(pol), \mathsf{DObl}(pol))$ where $\mathsf{GObl}(pol)$ precisely captures the obligations that arise from grants of policy pol, and $\mathsf{DObl}(pol)$ precisely computes the obligations that arise from denials of policy pol.

Which approach is best may be dictated by deployment details such as resource constraints. For example, circuits may be optimized before stored on devices. In any event, once policies are executed by the PDP they are not user-facing and so do not have to be easy to understand, as long as we have a trusted means of verifying that what is running on the PDP is an equivalent representation of the user-facing policy.

5 Flexibility Through Delegation

Over the life cycle of a resource, its owner may change or other parties may not merely want to request access to that resource but also be able to formulate and enact policies that exercise partial control over that resource. Consider the example of a car as a resource. Its original manufacturer (OEM) may want to create and maintain a *numerical passport* of that car; a car dealer may want to lease the car to clients; and such clients in turn may expect to be able to access the car as proscribed in a leasing arrangement. Here, the OEM needs to delegate policy writing and enforcement to the dealer and the dealer needs to further delegate such abilities to the leaseholder – and where this delegation chain honors composition constraints that were agreed to by these parties.

The policy written and controlled by the OEM may for example include aspects that concern product recalls, the enabling or disabling of technical features in the vehicle or the ability to collect certain types of maintenance/performance data to build data models for the type of vehicle. The policy written by the car dealer may for example capture contractual aspects of a lease that the client enters into, including specific usage restrictions that would prevent the car from being driven in certain territories. The policy written by the client should allow for basic services such as the ability to drive the car within the confines of the lease, to allow others to access the car in specific ways. Examples of the latter are to let delivery agents open the trunk on demand or to let the daughter drive the car on specific days and times of the week.

This simple example suggests a delegation tree, in this case from manufacturer to dealer to client, and from the client to the delivery person (one delegation path) and to her child (another delegation path). We refer to the source of a delegation (e.g. the OEM) as the delegator and the target (here the dealer) as the delegatee. Note that the same entity can be both a delegator and a delegatee (e.g. the dealer).

There are established approaches to delegation in distributed systems in the literature, including the ability to bound and control the depth of delegation chains. Let us point out the seminal work by Abadi et al. in [1] in this regard. That approach makes delegation an explicit language construct in the language for access control itself. While this has benefits and is elegant, our setting will make the FROST language a first level on which delegation mechanisms are provided on a second level through cryptographic protocols.

5.1 Principles for our Approach to Delegation

Before we discuss technical details of how to realize such delegation, let us first formulate some principles that should inform technical solutions to delegation:

DR1 The Delegator should be able to override decisions made by policies that were authored by Delegatees.

DR2 Delegatees should be able to author and administer their own policies so that they can realize desired interactions for service creation.

DR3 Delegators can demand specific bounds on how long delegation paths that originate from them can be, and these bounds need to be enforced.

DR4 Policies along a delegation chain should be composable in varying manners, to reflect a diversity of service compositions and needs of DR1 and DR2 above.

DR5 Such compositions should be verifiable so that each entity on a delegation chain can determine that her local policy interacts with policies along that chain in the desired manner.

DR6 Every access request should have a unique path on the delegation tree whose policies, in their composition, decide that request.

Requirement DR1 ensures that resource owners, and generally those who delegate, have the ability to stay in control of access decisions. Note that this control can only be absolute for the resource owner, it is a relative one for other delegators as DR1 will also apply to all entities ahead on that chain.

Requirement DR2 means that an entity can write, update, and delete policies that it controls. Of course, such policies will be subject to composition within the delegation chain, and so there is tension between DR1 and DR2.

Requirement DR3 allows the resource owner, e.g. to say that any delegation chain can have length no larger than three delegation links. Integrity checks will need to assure that such bounds are respected. Consider the bound of three delegation links and the chain $\texttt{OEM} \rightarrow \texttt{dealer} \rightarrow \texttt{client} \rightarrow \texttt{DHL}$. This already has the maximal number of three links. It may be unreasonable to say that a link from DHL to one of its employees would break this maximum of delegation: the DHL policy would merely articulate which employees would be able to open the trunk (e.g. those whose shift aligns with the car's location).

Requirement DR4 says that we do not want to proscribe the semantics of composition for policies along a delegation chain. Certainly, we mean to offer useful idioms for this but our approach and its cybersecurity protocols are not wedded to a particular idiom. In an online-access example, we may have a delegation chain of policies $p_0 \rightarrow \cdots \rightarrow p_n$ where p_0 is the

policy of the resource owner who controls the entire composition chain conservatively as

$$p_0 \gg (p_1 \gg (\cdots \gg p_n) \dots) \tag{9}$$

where the binary policy composition operator $P \gg Q$ may be encoded as

```
case {
  [P eval conflict: deny]
  [P eval undef: Q]
  [true: P]
}
```
(10)

This denies whenever P has a conflict and defers to Q whenever P has no opinion on the request. Both definite decisions (`grant` and `deny`) of P are preserved in this composition. Therefore, the composition pattern in (10) serves as an example of an idiom for the composition of policies on a delegation path. But as already pointed out, our approach allows for use of any idiom that is expressible in our FROST language.

Requirement DR5 is important for validating the interactions of policies from different stakeholders. The above delegation chain, e.g. may be replaced with a delegation tree that has the `dealer`, as the legal owner of the car, at its root. In this case, the tree may contain the path `dealer` $\rightarrow$ `client` where there may be links from `client` to `OEM`, `DHL`, and `daughter`. For example, a request to open the trunk may stem from the client, the daughter or DHL. The request needs to make that clear to identify the unique delegation path, a requirement captured in DR6.

It is interesting to note that in this use case, the temporal order in which entities create policies may not reflect the structure in the delegation tree. The OEM creates the first policy and transfers overall control to the dealer as new owner, who then adds her own policy. The idiom used for the policy composition, though, needs to make certain interaction guarantees, e.g. that the OEM is able to maintain the numerical passport of the vehicle.

5.2 Our approach to delegation

Delegating a mere access token (say for opening a car trunk once) may typically require less scrutiny than delegating the right to writing and enforcing policies. For the latter, a low and minimal requirement seems to be that the delegator knows a digital identity of the delegatee and assigns such a right to that identity. Delegations of policies are typically assumed to take place in a social space, a shared ecosystem in which parties have already established a level of trust or entered legal agreements. The scenario of the OEM, car dealer, and leaseholder serves as a good example thereof.

A prudent delegator will of course want to insist on the use of idioms for policy composition that protect her own policy and its enforcement from the malicious or unintentional manipulation by policies written by direct or indirect delegatees. The verification tools we developed in Section 3 can be put to use to validate that a chosen idiom and policy will offer her desired behavior – regardless of which policies others on the delegation chain may have chosen.

Let us now articulate some aspects of our approach to delegation and how it is user-centric and flexible. Since policies are to be evaluated and enforced on devices within the resources themselves, our composition idioms for delegation chains always give the resource owner highest priority and its policies are the origin of each policy delegation.

Each delegator, beginning with the resource owner, specifies one or more delegatees who in turn can write policies and specify further delegatees themselves – up to a certain delegation depth d set by the resource owner. Delegation ends as soon as a delegatee acts as the final

delegatee (the leaf of the delegation path) and submits the chain of policies on the ordered *policy chain* of that path to the resource.

Since a delegator (including the resource owner) can assign more than one delegatee, the resource stores a *policy tree*. Our approach is also consistent with the use of a *policy forest*, should there be a need for maintaining several policy trees. However, this needs to ensure that the complete mediation of the access-control system cannot be corrupted by an adversary by letting a request be processed by the "wrong" policy tree.

Our approach to delegation only requires a minimal number of parties to be connected at the same time. A connection between two agents can e.g. be established online via the internet or offline via near field communication. For an online connection, means for creating consensus may be employed, e.g. a public-key registry on a blockchain may provide simpler authentication or a hash list on a blockchain may identify and authenticate policies in a cloud in order to reduce message payloads and memory requirements on devices with constrained computational resources.

The delegation chain for policies needs to be tamper resistant, especially during its creation. We achieve this through a combination of methods. The communication between a delegator and a delegatee takes place within a secure communication channel set up via a protocol similar to SSH or TLS. This provides authentication of the participants as well as confidentiality and integrity of transferred messages. The delegation actions themselves happen on top of the secure communication channel via the exchange of messages containing policies, public keys and digital signatures.

After the completion of these steps for creating a policy chain, said chain is considered to be valid, if the resource successfully verifies the associated public-key chain and digital-signature chain and if it can also verify that the constraints imposed by the resource owner are met.

For the latter, note that the resource owner controls the choice of parameters such as the maximal delegation depth d – which defines the number of delegations that are allowed to happen. But she also controls other parameters and options such as the choice of idiom for composing policies of the policy delegation chain, choice of policy encryption options, and policy wrappers that make the composition enforceable. We will describe these elements in some detail further subsequently. Below, we work with a cryptographically secure hash function H. We assume that each agents i has and is identified by a pair (pk_i, sk_i) of public and private keys for signing messages via a digital-signature algorithm and that agent i also has a policy p_i she wishes to enforce on the resource.

5.3 Initialization of a Delegation Chain

At first, the resource owner, with keys (pk_0, sk_0), needs to specify parameters par for the creation of the delegation chain. These parameters may include the delegation depth d, the type of composition P, a nonce n to prove freshness of the delegation chain, and the choice of composition idiom, e.g. the one in (9). Then she acts as the initial delegator and sends a message m_0 and its signature s_0 to the initial delegatee who has key pair (pk_1, sk_1). Message m_0 contains the parameters, the resource owner's public key and policy as well as the initial delegatee's public key. Then the hash of the message gets signed under the resource owners secret key:

$$\begin{aligned} m_0 &= par \mathbin{+\!\!+} pk_0 \mathbin{+\!\!+} p_0 \mathbin{+\!\!+} pk_1 \\ s_0 &= \mathsf{sign}(sk_0,\, H(m_0)) \end{aligned} \tag{11}$$

The initial delegatee should verify the signature and check the depth constraint to see if she could further delegate, cf. sequence diagram in Figure 10.

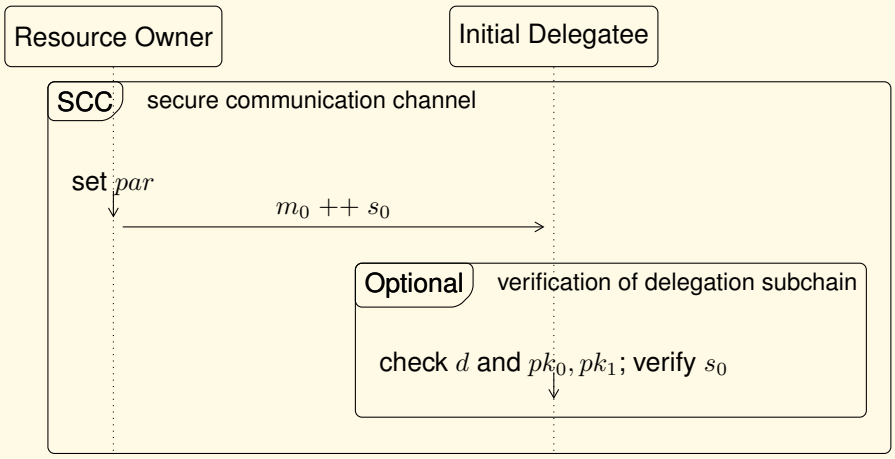


Figure 10: Initialization of the delegation chain, where the message sent via a secure channel is specified in (11) and the verification of this message is optional – in that we cannot guarantee that the initial delegatee as recipient will perform these checks.

5.4 Extending a Delegation Chain

The i-th delegator, with keys (pk_i, sk_i) is the $(i-1)$-th delegatee who inductively received a signed message already. This agent i now constructs and sends a message m_i and its signature s_i to the i-th delegatee who has key pair (pk_{i+1}, sk_{i+1}). The message m_i contains all policies and signatures received by previous delegatees on the the delegation chain, including the information about parameters par. Then agent i appends to this the i-delegator's public key and policy p_i, as well as the i-th delegatee's public key. Then the hash of the resulting message is signed with the i-th delegator's secret key:

$$\begin{aligned} m_i &= par +\!\!+ pk_0 +\!\!+ p_0 +\!\!+ pk_1 +\!\!+ s_0 +\!\!+ \cdots \\ &\qquad \cdots +\!\!+ pk_{i-1} +\!\!+ p_{i-1} +\!\!+ pk_i +\!\!+ s_{i-1} +\!\!+ pk_i +\!\!+ p_i +\!\!+ pk_{i+1} \\ s_i &= \mathsf{sign}(sk_i,\, H(m_i)) \end{aligned} \tag{12}$$

The reason that m_i contains all policies and signatures of previous messages is that each agent i can then verify the validity of the delegation sub-chain for which it is the leaf. Additionally, agent i should inspect the specified (and signed) delegation depth d in order to determine whether agent i could further delegate, cf. sequence diagram in Figure 11.

5.5 Completion of a Delegation Chain

The delegation chain is completed whenever a delegatee decides to delegate next to the resource itself. Suppose that agent n with key pair (pk_n, sk_n) wishes to delegate to the resource, which has key pair (pk_{rs}, sk_{rs}).

The creation and signature of the message that agent n sends to resource rs is conceptually the same as in (12) but for different identities of the last delegator/delegatee link:

$$\begin{aligned} m_n &= par +\!\!+ pk_0 +\!\!+ p_0 +\!\!+ pk_1 +\!\!+ s_0 +\!\!+ \cdots \\ &\qquad \cdots +\!\!+ pk_{n-1} +\!\!+ p_{n-1} +\!\!+ pk_n +\!\!+ s_{n-1} +\!\!+ pk_n +\!\!+ p_n +\!\!+ pk_{rs} \\ s_n &= \mathsf{sign}(sk_n,\, H(m_n)) \end{aligned} \tag{13}$$

The communication of such a message to the resource ends the delegation chain. The resource, as a trusted entity, will commence with the verification of that chain, cf. the sequence

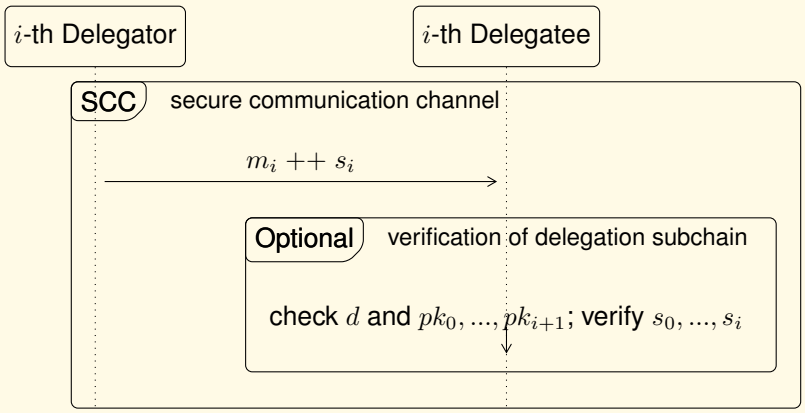


Figure 11: Extending the delegation chain that ranges from resource owner 0 to a delegatee $i-1$ to a new delegatee i, for which $i-1$ is the delegator. Message m_i and its signature s_i are specified in (12). Integrity checks are optional in that we cannot assume that they will be performed by delegatees.

diagram depicted in Figure 12. This includes checks such as that $n \leq d$ is true – failure of that will make the resource reject this chain since it would violate the delegation depth mandated by the resource owner.

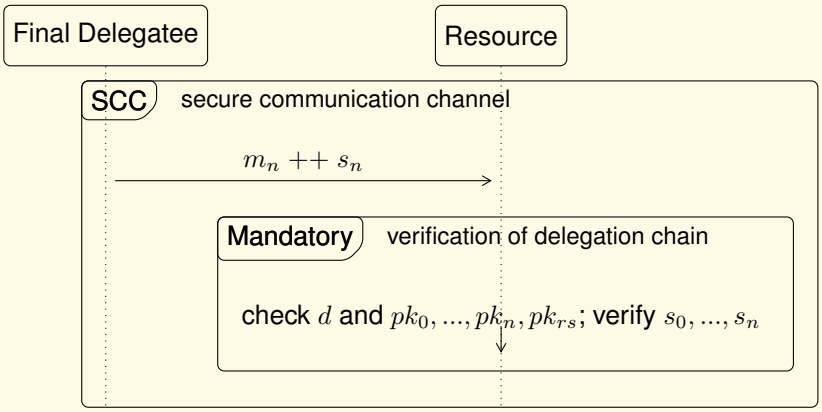


Figure 12: The last step in the delegation chain is similar, if not identical, to the step in Figure 11, except that the targeted delegatee is the resource and that the verification step is not optional – in that the resource is part of the trusted computing base and so will perform this check. Message m_i and its signature s_i are specified in (13).

Note that we explicitly allow for $n < d$, meaning that any delegatee in the delegation chain may act as the final delegatee by submitting a message and signature to the resource itself. This ensures that no delegatee further down a possible delegation chain can block the submission of a delegation subchain of delegators further up the delegation chain. However, this may allow an agent i to "deny a service" by preventing other agents from partaking in a delegation chain. The latter can be mitigated against or prevented by our approach, since we may also allow for the resource owner to demand a minimal delegation depth and she may also constrain the set of key pairs allowed within such chains. The resource would then capture these constraints in its validation logic for the mandatory verification of such chains.

5.6 Verification of Delegation Chain

We now give details of the mandatory verification that the resource performs as depicted in Figure 12. After a successful check that $n \leq d$ holds it may perform additional checks on parameters, e.g. the freshness of the delegation chain itself. If all these checks succeed, it verifies each of the signatures $s_0, \ldots, s_n$ against the calculated hashes $H(m_0), \ldots, H(m_n)$. If any of these checks fail, the resource will reject the validity of that policy chain. Otherwise, the resource will deem that delegation chain to be valid and stores it in a policy tree, as seen in the sequence diagram in Figure 13.

The delegation chain interleaves a public-key chain, a policy chain and a signature chain. The public-key chain lists the order of delegation, starting from the resource owner to intermediate delegatees to the final delegatee. Each signature proves that the respective delegator indeed intends to grant the right of policy creation to the stated delegatee, hence said delegatee is a legitimate next delegator. In particular, this proves that delegation originated from the resource owner and that delegation ends up with the resource. The signature chain, i.e. the signatures of signatures, therefore prove the intended order of delegation. The policy chain reflects the order of delegation which, together with the policy composition type and idiom specified, determines the policy that the resource will enforce for this delegation chain.

The underlying protocol for secure communication provides authentication of the recipient in each communication step. It may also provide authentication of the sender, but this is not required for the establishment of a secure communication channel. This is so since the resource is mandated to verify the entire delegation chain prior to its storage or enforcement, and this verification indirectly authenticates all agents on the delegation chain for the intents and purposes of this protocol.

Therefore, each delegatee and the resource are authenticated through the secure communication channel protocol. Moreover, the resource is able to authenticate the resource owner. To summarize, we obtain an authentication chain given by

$$pk_{i+1} \rightsquigarrow pk_{i+2} \rightsquigarrow \cdots \rightsquigarrow pk_n \rightsquigarrow pk_{rs} \rightsquigarrow pk_0 \rightsquigarrow \cdots \rightsquigarrow pk_{i-1} \rightsquigarrow pk_i.$$

In this authentication chain, senders and recipients are identified by their public keys, where the resource and the resource owner hold the keys pk_{rs} and pk_0, respectively. Thus, the i-th delegatee with public key pk_{i+1} indirectly authenticates the i-th delegator with public key pk_i. Furthermore, since the i-th delegatee is already authenticated by the i-th delegator through the secure communication protocol, we obtain the authentication *cycle* given by

$$pk_{i+1} \rightsquigarrow pk_{i+2} \rightsquigarrow \cdots \rightsquigarrow pk_n \rightsquigarrow pk_{rs} \rightsquigarrow pk_0 \rightsquigarrow \cdots \rightsquigarrow pk_{i-1} \rightsquigarrow pk_i \rightsquigarrow pk_{i+1}.$$

5.7 Data Structure for Policy Delegation Tree

The requirements for this data structure are:

DT1 It must provide support for flexibly policy composition.

DT2 Its tree structure must not offer additional attack surface.

DT3 It should accommodate change management for policies.

DT4 It should be able to protect the privacy of policies along delegation chains.

DT5 Verification complexity of any delegation chain must be at most linear in the chain length.

DT6 Meta-information, e.g. delegation parameters, must be stored at the root node.

DT7 Nodes may contain policies or hash references to policies stored securely elsewhere.

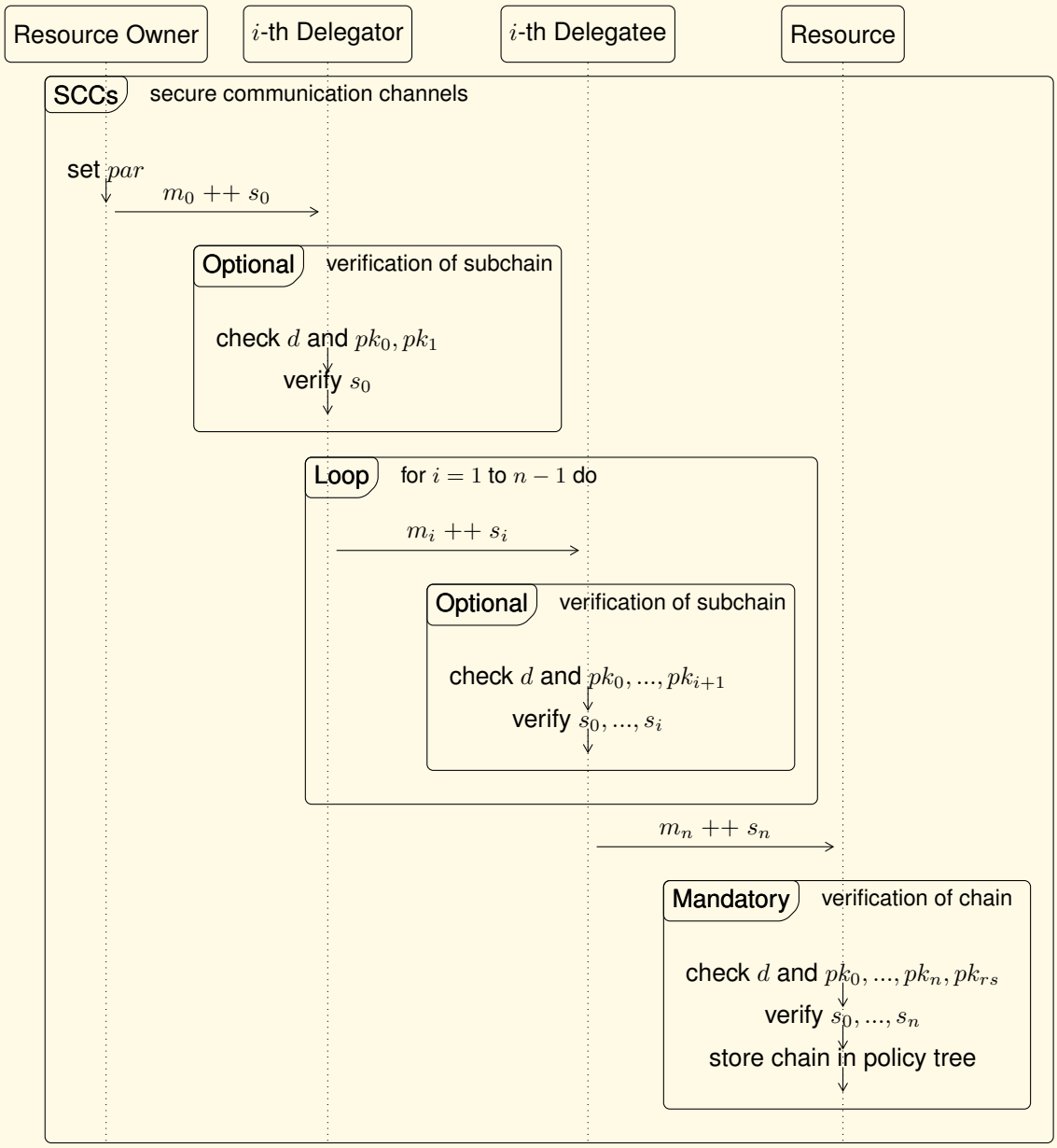


Figure 13: Summary of all optional and mandatory verification activities in the creation and deployment of the delegation chain. Deployment only happens if verification by the resource itself succeeds.

DT8 The policy tree must not have duplicate paths, be they maximal or not.

DT9 Every access request is securely mapped onto a unique delegation path for the PDP.

DT10 Resource constraints of embedded devices must be consistent with this data structure.

This suggests that each node in the tree has fields for the public key, the policy, and the

signature of each previous delegator on its path. The policy field either stores the the policy or a hash thereof, the the latter if the policy is securely accessible for integrity checks – e.g. the policy may be in the device memory or in a cloud policy store. The root node in the tree must additionally have fields for parameters such as the maximal (or even minimal) delegation depth, composition type, and composition idiom, as well as a field that specifies employed crypto-primitives such as the used hash function.

We may create a unique identifier for the policy tree, e.g. by hashing its root node or the entire tree. Our root node has no signature field, since the initial delegator doesn't have a previous delegator. All leaf nodes in the tree consist only of the signature that terminates the delegation chain, as described above. That signature or its hash may serve as a unique identifier for this maximal policy chain in the policy tree. The requirements on privacy preservation and change management may require additional parameter fields within the root node.

Policy trees also need to be unique with respect to a delegation chain. Since an author may issue many policies and the same policy can be issued by several authors, a delegation chain cannot be uniquely represented by either the public key chain or the policy chain on their own. This means that only the (public key, policy)-chain pairs are unique and a public-key chain together with its associated policy chain therefore cannot exist on two different paths.

If the resource owner decides to have more than one policy, perhaps also with different parameters, we accommodate this such that each policy of the resource owner is the root of a distinct policy tree. Therefore, our data structure is really one for a policy *forest* but where the above requirements still apply.

If the resource encounters a new delegation chain that happens to coincide with a (not necessarily maximal) subchain up to a delegator pk_i, the new delegation chain will be merged in the tree with the common initial chain and the remaining path from the new chain will become a new path in the tree from the node for pk_{i+1} onwards. This approach to adding new delegation chains to the policy forest therefore maintains the uniqueness of delegation paths, see Figure 14 for an illustration.

Given the cryptographic properties of digital signatures, checking the equivalence of (public key, policy)-chain pairs of two delegation sub-chains up to pk_{i+1} and pk'_{i+1} amounts to checking the equality of the last signature s_i and s'_i of the delegation subchain, i.e.

$$(pk_0, p_0) \rightsquigarrow \cdots \rightsquigarrow (pk_i, p_i) \rightsquigarrow pk_{i+1} \equiv (pk'_0, p'_0) \rightsquigarrow \cdots \rightsquigarrow (pk'_i, p'_i) \rightsquigarrow pk'_{i+1}$$
$$\iff \qquad s_i \text{ equals } s'_i.$$

This greatly simplifies the determination of equivalence of sub-chains. Note that the equivalence of delegation chains or sub-chains does not imply their validity in general. However, a delegation chain that is equivalent to one that has been verified already is also valid.

Note that signatures are implicitly dependent on the preceding delegation subchain. For example, the signatures named $s_{1,b}$ in the left tree of Figure 14 are all different, since one is referencing pk_2 whereas the other one refers to pk_3. However, the signatures named $s_{0,b}$ in the right tree of Figure 14 are identical, since they both reference pk_1, whereas the respective nodes of the tree may differ in their policies $p_{1,a}$ and $p_{1,b}$. Similarly, the signatures named $s_{0,a}$ and $s_{0,b}$ cannot be the same, because they reference from nodes with distinct policies $p_{0,a}$ and $p_{0,b}$, even though they originate from the same pk_0.

5.8 Policy Composition for a Delegation Chain

Policies p_i within delegation chains may of course be composed from any sub-policies in ways expressible in the FROST language. We organize the composition of policies p_i along a maximal delegation chain through two staged composition mechanisms:

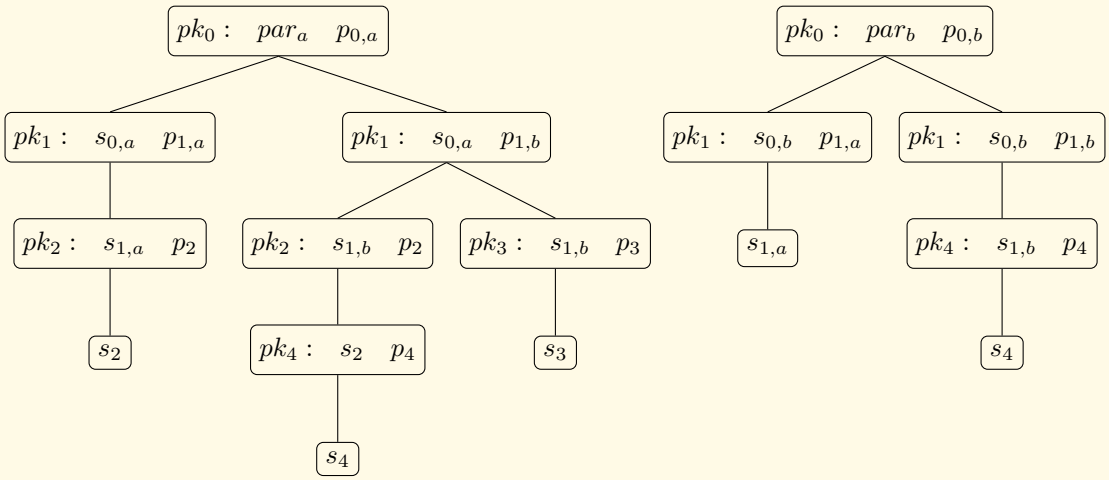


Figure 14: Illustration of our data structure for policy forests with two policy trees controlled by a resource owner. The tree on the left is consistent with $d = 3$, whereas the one on the right is consistent with $d = 2$. Different trees may contain different parameter values within their roots.

1. a *composition type*, which specifies how each policy p_i is composed (if at all) with policies preceding it on the chain, resulting in a policy P_i, and
2. a *composition idiom*, which specifies how the policies P_i computed in the first item are to be composed into the policy that will be run by the PDP.

The choice of type and idiom is ultimately in the control of the resource owner but may be the result of an external and social consensus process, e.g. it may realize conditions stated in an agreement signed by a consortium.

The most simple composition type is the one that sets P_i to be p_i. Another example of composition type is the information join, where we would set

$$P_i = p_0 \texttt{ join } p_1 \texttt{ join } \ldots \texttt{ join } p_i.$$

A variant of the latter type may be that each p_i first applies its own policy wrapper $\downarrow_i$ before forming the join with preceding policies, similarly wrapped as in

$$P_i := (p_0 \texttt{ join } (p_1 \texttt{ join } (\ldots \texttt{ join } (p_{i-1} \texttt{ join } p_i \downarrow_i) \downarrow_{i-1}) \ldots) \downarrow_1) \downarrow_0 .$$

The composition idiom then specifies how exactly the policies $P_0, \ldots, P_i$ are to be composed. One example for such an idiom is the composition of all these policies as follows:

$$P_n \texttt{ op } P_{n-1} \texttt{ op } \ldots \texttt{ op } P_1 \texttt{ op } P_0 \texttt{ op deny}, \tag{14}$$

where `op` is any binary composition operator expressive in our FROST language, e.g. the operator $\gg$ defined in (9) already. Note that this is a composition *context* as it also employs constant policy `deny` as a top-level wrapper.

Other sensible choices of `op` for the idiom patterns given in (14) are the following interpretations of P `op` Q:

- `deny` if P yields `conflict`, evaluate Q if P yields `undef`, evaluate P otherwise,

- evaluate Q if P yields `conflict`, `deny` if P yields `undef`, evaluate P otherwise,
- `deny` if P yields `conflict` or `undef`, evaluate P otherwise.

Thus the resource owner has full and fine-grained control over the composition of the policy chain and may use the verification tools presented above to explore and validate such control choices. In practice, these composition types and operators would either be chosen from a set of predefined idioms with known and validated behavior, or they bespoke solutions could be written and verified by the resource owner.

5.9 Change Management on a Delegation Chain

In practice, authors of a policy p_i may need to change that policy to some p_i' at some point in the future. While we certainly want to support such change management, we do not want to impose to all subsequent delegatee j with $i > j$ to re-sign the altered policy and to re-perform the residual delegation process. Rather, we may accommodate this by letting the i-th delegator send a message m_i' and its signature s_i' directly to the resource. Message m_i' contains all original messages and signatures from the original delegation subchain preceding i, and appends the i-th delegator's public key and altered policy as well as the public key of the resource. Then the hash of the message gets signed under the i-th delegator's key:

$$\begin{aligned} m_i' &= par \mathbin{+\!\!+} pk_0 \mathbin{+\!\!+} p_0 \mathbin{+\!\!+} pk_1 \mathbin{+\!\!+} s_0 \mathbin{+\!\!+} \cdots \\ &\qquad \cdots \mathbin{+\!\!+} pk_{i-1} \mathbin{+\!\!+} p_{i-1} \mathbin{+\!\!+} pk_i \mathbin{+\!\!+} s_{i-1} \mathbin{+\!\!+} pk_i \mathbin{+\!\!+} p_i' \mathbin{+\!\!+} pk_{rs} \\ s_i' &= \mathsf{sign}(sk_i,\, H(m_i')) \end{aligned} \tag{15}$$

The resource then checks all signatures $s_0, \ldots, s_{i-1}, s_i'$ against the respective computed hashes $H(m_0), \ldots, H(m_{i-1}), H(m_i')$. The validity of all these signatures then also identifies the respective policy sub-chain in the policy tree stored on the resource itself. In case the resource owner requests a policy change, the resource must also check the integrity of the new parameters par' and adjust the length of the delegation chain according to the new, possibly shorter, delegation depth d'.

A simple and resource-friendly approach to manifesting such changes is to replace the policies and signatures within the nodes in the policy forest where the change needs to occur, and to securely log each change event outside of the policy forest. This puts less strain on the resource and it offers an audit trail of such change management for accountability and dynamic or post-attack forensics.

Let us discuss some typical uses for this change management. Agent i may have discovered a "bug" in policy p_i (be it a security flaw, an error in business logic or some such) and policy p_i' provides a "patch" for this bug. Or agent i may upgrade a product or service which requires a change in policy to reflect this. Another example is when agent i wishes to no longer exercise control; depending on the composition type and idiom, this may be achieved by changing p_i to `undef` as p_i'. A more aggressive such change may be to choose `deny` as p_i'. The latter example also suggests that the resource owner needs to choose composition types and idioms that ensure that changes don't offer malicious or unintended corruptions of the intent in an overall policy composition on the resource.

Such changes may at times require notification of other agents on the chain that a change took place. Whether or not the new policy would also be communicated to these agents depends on whether policies are treated as being confidential, a subject to turn to next.

Notifications on a Delegation Chain We already suggested the use of an audit trail for change management. It may also be beneficial for agents to agree on a notification regime for such changes. For example, the ecosystem in which delegation takes place may want

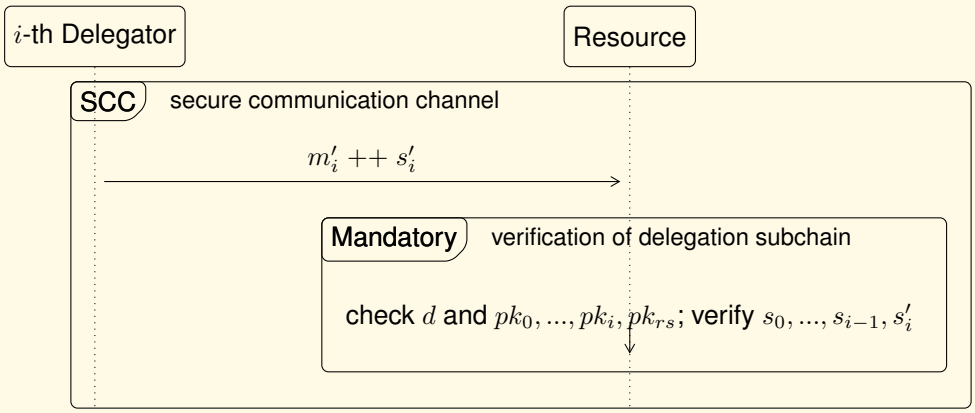


Figure 15: Agent i wants to change her policy p_i on an existing delegation chain to some policy p'_i. This is managed by directly notifying the resource of that change request, and the resource must verify the legitimacy of that request before making the change.

- that subsequent delegatees are notified about a policy substitution by a delegator,
- that previous delegators or the resource owner are notified about a policy change submitted to the resource,
- that delegatees are notified about a change of the delegation depth,
- that delegatees are notified about the deletion of an entire delegation chain by the resource owner.

Such agreements can improve the overall usability of the access-control framework and its delegation processes, but it needs to be balanced with other needs such as potential privacy concerns.

5.10 Policy Privacy on a Delegation Chain

The author of a policy p_i might want to keep its content private from other agents. A partial solution to this works as follows. The resource owner can specify additional parameters for the choice of a symmetric encryption scheme with corresponding MAC scheme, as well as an asymmetric encryption scheme. The i-th delegator then generates a random secret-key pair (k_i^{enc}, k_i^{mac}) and sends a cipher c_i and its signature s_i to the i-th delegatee. The cipher c_i consists of the symmetric encryption of the message m_i under key k_i^{enc}, the MAC of the symmetric encryption under key k_i^{mac} and the asymmetric encryption of the secret key under a public encryption key of the resource. This means that only the resource must have an additional public-private key pair $(pk_{rs}^{enc}, sk_{rs}^{enc})$ for asymmetric encryption. The message itself contains the $(i-1)$-th delegator's cipher, the i-th delegator's public key and policy as well as the i-th delegatee's public key. The hash of the cipher gets signed under the i-th delegator's secret signing key:

$$
\begin{aligned}
m_i &= c_{i-1} \mathbin{+\!\!+} s_{i-1} \mathbin{+\!\!+} pk_i \mathbin{+\!\!+} p_i \mathbin{+\!\!+} pk_{i+1} \\
c_i &= \mathsf{encrypt}(k_i^{enc},\, m_i) \mathbin{+\!\!+} \mathsf{MAC}(k_i^{mac},\, \mathsf{encrypt}(k_i^{enc},\, m_i)) \mathbin{+\!\!+} \mathsf{encrypt}\big(pk_{d+1}^{enc},\, k_i^{enc} \mathbin{+\!\!+} k_i^{mac}\big) \\
s_i &= \mathsf{sign}(sk_i,\, H(c_i))
\end{aligned}
$$

In this approach, no succeeding delegatee learns any information about the preceding delegators, their identifies, the parameters or policies – except for the identity of the immediate

predecessor. Note that the resource will know about the whole delegation chain and its policies and so its owner may also learn about it.

Once the delegation chain is submitted to the resource, the resource must start to verify the signature, decrypt the asymmetric cipher, verify the MAC, decrypt the symmetric cipher – all the way from the final part to the initial part. This checks the validity of the public-key chain and of the signature chain, but additional integrity checks would be performed as before, e.g. on the delegation depth.

A less restrictive approach might be to encrypt the policies with a symmetric key only, and to leave the parameters and public keys unconcealed. A more restrictive approach for privacy-preserving PDP evaluation may be the use of Yao's Garbled Boolean Circuits (see e.g. [16] for compilation techniques) or other such techniques from Secure Multi-Party Computations [19].

5.11 Delegation and Cryptographic Access Tokens

The resource owner may issue an access token to a requester so that the PDP on the resource can verify that request according to the policy represented by the token. We support a similar capability to each delegatee i in a delegation chain: agent i can issue an access token to a requester directly, without having to mediate this through the resource owner. This is consistent with the fact that agent i has the right to write her own policies.

This functionality can be achieved in the following way:

1. The delegatee generates a secret master key. A cryptographically secure key-derivation function is also specified by the resource owner as a parameter. A seed and a derivation pad are used to derive from the master key a fresh key in a non-reversible but verifiable manner.

2. The delegatee includes a cryptographic condition in her policy that takes a derived key and its derivation pad as input. This cryptographic condition must `grant` if and only if the derived key is indeed a result of the master key and the derivation pad under the key derivation function and it must `deny` otherwise.

3. The delegatee submits, or resubmits, the policy to the resource.

4. The delegatee sends an access token to a requester that includes a properly derived key and its derivation pad.

5. The requester presents the access token to the resource as a part of her formal access request.

6. The resource evaluates the policy composition with regard to the request.

For this approach, we note that the delegatee's policy would reach the resource as already discussed, and no further communication between the delegatee and the resource would be required for this token-based access request. In particular, the delegatee may issue any number of access tokens without having to interact directly with anyone other than the requester.

However, it needs to be kept in mind that the PDP will still evaluate the overall composed policy – regardless of whether p_i is evaluated through a token or not.

5.12 Self-Delegation and Cyclic Delegation on Delegation Chains

Our approach to delegation can create arbitrary public-key chains provided they conform to delegation-depth constraints. In particular, this allows for self-delegation and cyclic-delegation

on such chains. By self-delegation we refer to the i-th delegator specifying herself as the i-th delegatee, which yields a public-key subchain of form

$$\cdots \rightsquigarrow pk_i \rightsquigarrow pk_i \rightsquigarrow \cdots$$

Cyclic delegation refers to the more general pattern in which the i-th delegator is specified as the j-th delegatee for some $j > i$, which results in public-key subchains of form

$$\cdots \rightsquigarrow pk_i \rightsquigarrow pk_{i+1} \rightsquigarrow \cdots \rightsquigarrow pk_{j-1} \rightsquigarrow pk_i \rightsquigarrow \cdots$$

Should such behavior not be desired by the resource owner or the body that seeks such agreement, an additional parameter can be introduced and set by the resource owner at the start of any delegation chain. Then the resource can check for uniqueness of public keys along the path, in order to prevent cyclic delegation in general. Alternatively, the resource may check for pairwise uniqueness of subsequent public keys on the delegation chain in order to prevent self-delegation. Other variants of such checks may also have practical relevance, e.g. in order to rule out cylic delegations with cycles of length greater/less than or equal to some parameter c.

6 Mitigating Potential Attack Vectors

6.1 Choice of Cryptographic Primitives

We will stay clear of "rolling our own cryptography" and only rely on tried and tested cryptographic primitives. In particular, we don't allow the use of primitives that are still found in libraries but are no longer considered to be secure. At the same time, we have an incentive to make use of libraries such as those for specific embedded systems. Such usage will decrease the adoption and integration hurdles of our technology in its wider ecosystem.

At the same time, we have to acknowledge that parts of our architecture involve devices and micro-controllers whose constraints on resources may be severe. We can partly mitigate against this by making implementations mindful of given constraints. For example, if storage is limited, the policy store may be held externally or policies may be sent to the PRP directly via our delegation protocols. Similarly, some devices may have limited support for digital signatures, hashes or symmetric encryption – requiring adjustments to the trusted computing base.

On the other hand, there is the prospect that we may design novel hardware that is tailored to the needs of our core architecture, which would help with mitigating if not preventing security threats discussed further below. Such designs could also benefit from advances in Physically Uncloneable Functions, as a cheap and secure means of authenticating devices.

6.2 Risk-Aware Access Control

There are policy languages or approaches in access control that take into account risk assessments, see e.g. [2, 10, 15]. The deployment contexts of our framework clearly benefit from the ability to monitor and manage risks. FROST may use different approaches to risk, that, in their combination can strengthen the management of risks:

- anomaly detection of the network within which the framework is deployed generates information that can be expressed via attributes,
- the PEP may consult PIPs that convey dynamic information about risks and threats,
- the policy itself may contain attributes that specify how perceived or measured risks inform a policy decision.

In the latter case, such risk attributes may themselves be the result of "policies", e.g. as those defined and studied in [29, 30] for the aggregation of trust evidence.

6.3 Incomplete Information Due to Faults or Adversarial Manipulation

The evaluation of a FROST policy pol by the PDP depends on the values for attributes that occur within pol. Such values may be verifiable, e.g. a legal-age status derived from a digital identity, whereas others may require trust into the oracles that supply them, e.g. sensors that feed into PIPs.

Approaches to risk management, such as the ones discussed above, may inform how policies reflect on the degree of trust or contextualized trust one may place into obtained attribute values. Yet, another issue is how we would deal with information that is not obtainable for some unknown reason. For example, the PDP may not be able to determine whether the requester is of legal age or a sensor reading for current speed may be missing.

A conservative approach for the PDP is to deny any request if some attribute needed for evaluating the respective policy has an unknown value. A potential issue with this is that an attacker may turn this into a Denial of Service attack, by continuously disabling the information source of some needed attribute. There are techniques from AI and static analysis based on *super-valuational meaning* that may help with partially evaluating policies under such incomplete information – see e.g. [26] for applications of that in verification with temporal logic.

For example, we may evaluate the Boolean Circuits for pol with Kleene's strong 3-valued propositional logic – as familiar from hardware verification where in addition to 0 and 1 there is a third value $1/2$ meaning "don't know" or "don't care" depending on the intent of the computation. Attributes whose values are unknown then give rise to such $1/2$ inputs in the circuit. If the circuit still evaluates to 0 or 1, it means that this uncertainty did not matter and so the PDP may accept the result of the circuit regardless of the fact that not all attributes have known values.

6.4 Trusted Policy Life Cycle

A security principle for our framework is that we aim to realize trustworthy life cycles of policies. The life cycle of policies spans their conception, writing, administration, change management, tool support such as compilations, and so forth – all the way down to the embedded clients at which compilations of such policies execute. The trustworthiness of such life cycles will give policy writers, administrators, and system users assurance that the intent behind the conceived policies won't be compromised in their ultimate enactment on embedded clients.

Our FROST language and its cryptographic protocols are agnostic to the ambient system environment (blockchains, clouds, and so forth) and used transport layer. Of course, choices of such transport mechanisms and storage environments do influence how trustworthiness of a policy life cycle is established.

For example if a blockchain is used as a back-end, we may record a hash of a policy on the blockchain and a node can therefore verify that the policy has not been tampered with. Otherwise, in a could setting the policy could be signed and delivered to the device for evaluation.

In both of these examples, it seems important that the manner in which a policy is converted into enforceable representations be either *deterministic*, e.g. for a Nakamoto-style consensus, or *semantically determined* as for example described in Example 1. That way, anyone may verify the integrity of the policy as well as of its used representations. And there will be consensus about this integrity test, either deterministically or in a manner that experts can agree upon.

6.5 Policy Malware

The resource owner or its direct or indirect delegatees may unintentionally or maliciously submit a policy pol_{mw} to the PRP or Policy Store whose behavior is meaning to damage the access-control system, be it by forcing requests to be wrongfully denied, by making requests that

should be denied to be granted or by interfering with the intended composition of policies on a delegation chain. We have several means at disposal for dealing with this, let us mention:

- Policies may be hashed on a blockchain which essentially makes them immutable and so agents can verify whether a policy p_i is as it should be.
- Tools from Formal Methods can be used to verify that intended good behavior can be realized and bad behavior is prevented, even for unknown policies of delegatees.
- Cybersecurity protocols authenticate any policy change requests, and a secure audit trail will log any such changes to create accountability and forensic support.
- In more high-assurance settings, the PDP may recompile all policies, and verify the correctness of these compilations against internal or securely stored external information.

6.6 Exploiting Gaps Between Abstraction Layers

Security vulnerabilities are most severe and most often found within the layers of two particular layers of abstraction, and use of such layers is a fundamental principal for building digital systems [31]. One example here would be the design of more abstract DSLs for policy abstraction. We then would like to verify the compiler that converts programs written in that DSL into FROST policies. Additionally, we have to verify that each application of this compiler has indeed been correctly performed, i.e. we need proof that the generated FROST policy has been obtained from the input program of the DSL by that compiler.

Alternatively, especially if verifying the entire compiler is challenging, we may be able to prove – automatically or with some human assistance – that the compiled FROST code has the same behavior as the DSL program in terms of computed decisions and obligations.

To summarize, the development and provisioning of mature verification tools plays a key role in understanding and minimizing any gaps in abstraction layers of languages for policy writing. If such tools also provide transparent and interpretable evidence, they can also help considerably with validating that written policies meet their desired intent – which is a pragmatic and non-technical concern of the human who writes or describes such policies.

6.7 Mathematical Models and Security Analysis

It would be of interest to develop mathematical models that can abstractly capture the dynamics of access requests, policy administration, and change management. Such models may be in the form of discrete transition systems, perhaps also with probabilistic transitions. And these models could be used to understand whether an attacker may have strategies that give them a meaningful advantage for corrupting the underlying access-control service of that architecture. Additionally, it would be of value to formally model our security protocols and to prove classical secrecy and authentication properties – e.g. against different types of capabilities of adversaries as developed in [6].

7 Exemplary Use Cases

We now exhibit potential use cases for the technology we have described in this Yellow Paper. Some of these are not just conceptual, but presently undergo pilot implementations at XAIN AG with its commercial partners. The overall aim of such pilots is to make this technology ready for production integration in several verticals within the foreseeable future.

7.1 Big-Data and Knowledge-Transfer Platform

The FROST architecture can sit on top of a big-data platform in order to control access to data and associated resources. Alongside FROST, there may also sit an AI framework that runs distributed machine learning over the big-data platform, e.g. to facilitate learning across organizational boundaries in a privacy-preserving manner. FROST then can articulate and enforce the controls for data access and knowledge sharing across the big-data and AI platforms, while also generating trustworthy and immutable audit trails hosted on a bespoke blockchain.

AI engines may e.g. learn work-flow actions that are optimized to ensure compliance in the regulated financial sector, and where such insights can be shared across different banks through controls provided by FROST. This can thus establish sufficient trust in such a shared and conventionally highly distrustful ecosystem while also creating value for all its participants.

7.2 Life Cycle of Parts, Machines, and Devices

Current solutions to tracking the provenance of parts, machines and other forms of devices typically rely on a sole technology vendor. This not only limits the use cases in that domain by what services the particular vendor does offer, but it also creates interoperability issues should such services be deployed in shared ecosystems in which organizations may use different vendors for such solutions.

The FROST architecture and technology layer offers a viable and value-creating alternative in that regard: it is agnostic to the used transport layer, data storage architecture, and other back-end structures. Different organizations only need to use interfaces for the open-source FROST technology in order to create coherent such services in shared ecosystems. The extensibility of the FROST language also gives organizations freedom in creating novel services.

Consider the example of an OEM for high-speed trains, who needs to track parts, their provenance in the supply chain, their maintenance, and other life cycle characteristics. This is not just an activity confined to that OEM, but also may involve the OEM of the part itself, suppliers, and technical regulators. These parties all have different needs for accessing data and devices, and FROST provides the technology layer in which these needs can be consistently realized – with appropriate audits and controls.

In this use case, the utility of FROST extends beyond the tracking of parts. For example, FROST may also be used to control access to other parts of the train such as the driver's area, devices through which trains can be operated, doors, and so forth.

7.3 Mobility & IoT

The *Mobility Open Blockchain Initiative* is a good example of the high momentum that digitization has in building up shared ecosystems for intelligent transportation systems. The FROST technology and its policy language and tools can influence the shaping of this space, for example with use cases similar to those described in the last two subsections.

The story for intelligent transportation R&D is nuanced, however. For example, viable solutions may be specific to large cities and their local, historically grown infrastructure; and autonomous vehicles and Uber-style car access may only be cost-effective in urban areas.

The technology that FROST offers is not wedded to any such predictions and will provide a fitting layer for control to resources and data for a wide range of socio-technical developments – be they fleet management, traditional car ownership, short-term car rental, Uber-style services or other emerging service innovations.

7.4 Health and Patient Care

With an aging population and a rise of life expectancy, there is a growing need for a wide range of medical and assisted-living care. One concern with the provisioning of such care is that patients as well as carers ought to be in control of their personal and medical data, and that the delivery of medical and care services should be fair, transparent, and accountable.

To consider a real example, a carer is employed by a service company to visit patients and prepare their weekly medication supply. But she has only 15 minutes allocated per visit and a commercial tracking system does not give her any flexibility in proving that any delays are warranted or that a "delay" is actually a system flaw and she is actually with a patient at that time. These socio-technical or socio-economic issues change the behavior of the carer in ways that benefit neither the carer, nor the patient, nor the care company. The FROST technology can help with creating more fair, transparent, yet still accountable such tracking systems

7.5 Porsche Pilot

Together with Porsche, XAIN ran a pilot that brought blockchain technology and service-enabling, user-centric access control into a modern car – a Porsche Panamera. The outcomes of this pilot may be seen in this short video:

`https://www.youtube.com/watch?v=KvyF78RTj18`

This Yellow Paper describes the next and very significant step in the development of such technology and capabilities, with applications certainly within Automotive & IoT but also well beyond those, as the above use cases are meant to illustrate.

8 Conclusions

The digitization of verticals, services, and economic platforms is taking place at great pace and stands to have far reach. This creates limitless opportunities for the creation of novel services, products, and user engagement. One challenge in harnessing such opportunities is that this future will be one in which shared ecosystems are likely to become a predominant paradigm, e.g. the one of stakeholders that shape the intelligent transportation system of a large city.

For such ecosystems to thrive, we need underpinning technology that allows actors in that space to put them into control of their data and resources without relying on a central authority for the coordination, articulation, and operationalization of such controls.

This motivated work reported in this Yellow Paper, which introduced the FROST language and its accompanying technology architecture as a foundation that allows participants in such ecosystems to retain control of their data and resources, and to share access to these in a well-defined and auditable manner.

The FROST language is a *policy-based* approach to access control. We described its design, expressiveness, and extensibility. Then we discussed how FROST policies can be compiled into Boolean Circuits as faithful and optimizable representations that can then be put onto devices. These circuits were also shown to be the building blocks for analyses whose insights can support the validation and verification of policies and their intended behavior.

Next, we extending the FROST language with *obligations* that a system would need to fulfill on made access-control decisions. We showed that these specifications of obligations can also be captured in circuit form for faithful representation and operationalization on devices.

A core topic we then developed is that of *delegation*: the ability to transfer access rights to other actors, where these rights are not only pertaining to concrete actions such as opening a door but may also be about the ability to write and enact policies or to delegate such rights

further. These are key abilities for supporting a shared ecosystem. We outlined a security protocol that can securely realize said capabilities so that a shared policy can be represented and enacted on a device. This puts the resource owner into ultimate control and supports both online and offline modes, as well as secure and transparent change management of policies.

Then we had a look at some of the pertinent security threats to this approach to access control and its architecture, and we discussed means by which we may mitigate or eliminate such threats. Finally, we illustrated the value proposition of FROST through a couple of brief use cases chosen from a variety of verticals and service sectors.

References

[1] ABADI, M., BURROWS, M., LAMPSON, B. W., AND PLOTKIN, G. D. A Calculus for Access Control in Distributed Systems. *ACM Trans. Program. Lang. Syst. 15*, 4 (1993), 706–734.

[2] ALMUTAIRI, A., SARFRAZ, M. I., AND GHAFOOR, A. Risk-Aware Management of Virtual Resources in Access Controlled Service-Oriented Cloud Datacenters. *IEEE Trans. Cloud Computing 6*, 1 (2018), 168–181.

[3] AUGOT, D., BATINA, L., BERNSTEIN, D. J., BOS, J., BUCHMANN, J., CASTRYCK, W., DUNKELMAN, O., GÜNEYSU, T., GUERON, S., HÜLSING, A., LANGE, T., MOHAMED, M. S. E., RECHBERGER, C., SCHWABE, P., SENDRIER, N., VERCAUTEREN, F., AND YANG, B.-Y. Post-Quantum Cryptography for Long-Term Security: Initial recommendations of long-term secure post-quantum systems. *PQCRYPTO, Horizon 2020 ICT-645622* (2015), 1–10.

[4] BAHAR, R. I., FROHM, E. A., GAONA, C. M., HACHTEL, G. D., MACII, E., PARDO, A., AND SOMENZI, F. Algebraic Decision Diagrams and Their Applications. *Formal Methods in System Design 10*, 2/3 (1997), 171–206.

[5] BALLARIN, C., HOMANN, K., AND CALMET, J. Theorems and Algorithms: An Interface between Isabelle and Maple. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation, ISSAC '95, Montreal, Canada, July 10-12, 1995* (1995), pp. 150–157.

[6] BASIN, D. A., AND CREMERS, C. Know Your Enemy: Compromising Adversaries in Protocol Analysis. *ACM Trans. Inf. Syst. Secur. 17*, 2 (2014), 7:1–7:31.

[7] BERNSTEIN, D. J. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete? *Department of Computer Science (MC 152) The University of Illinois at Chicago* (2009), 1–12.

[8] BERNSTEIN, D. J. Grover vs. McEliece. *Department of Computer Science (MC 152) The University of Illinois at Chicago* (2009), 1–9.

[9] BERNSTEIN, D. J. Introduction to post-quantum cryptography. *Springer* (2009), 1–14.

[10] BIJON, K. Z., KRISHNAN, R., AND SANDHU, R. S. A framework for risk-aware role based access control. In *IEEE Conference on Communications and Network Security, CNS 2013, National Harbor, MD, USA, October 14-16, 2013* (2013), pp. 462–469.

[11] BRUNS, G., AND HUTH, M. Access-Control Policies via Belnap Logic: Effective and Efficient Composition and Analysis. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008* (2008), pp. 163–176.

[12] BRUNS, G., AND HUTH, M. Access control via belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inf. Syst. Secur. 14*, 1 (2011), 9:1–9:27.

[13] BRUTTOMESSO, R., PEK, E., SHARYGINA, N., AND TSITOVICH, A. The OpenSMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (2010), pp. 150–153.

[14] BRYANT, R. E. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv. 24*, 3 (1992), 293–318.

[15] BURNETT, C., CHEN, L., EDWARDS, P., AND NORMAN, T. J. TRAAC: Trust and risk aware access control. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014* (2014), pp. 371–378.

[16] BÜSCHER, N., FRANZ, M., HOLZER, A., VEITH, H., AND KATZENBEISSER, S. On compiling Boolean circuits optimized for secure multi-party computation. *Formal Methods in System Design 51*, 2 (2017), 308–331.

[17] CAMPAGNA, M., CHEN, L., DAGDELEN, Ö., DING, J., FERNICK, J. K., GISIN, N., HAYFORD, D., JENNEWEIN, T., LÜTKENHAUS, N., MOSCA, M., NEILL, B., PECEN, M., PERLNER, R., RIBORDY, G., SCHANCK, J. M., STEBILA, D., WALENTA, N., WHYTE, W., AND ZHANG, Z. Quantum Safe Cryptography and Security: An introduction, benefits, enablers and challenges. *European Telecommunications Standards Institute* (2015), 1–64.

[18] CHEN, L., JORDAN, S., LIU, Y.-K., MOODY, D., PERALTA, R., PERLNER, R., AND SMITH-TONE, D. Report on Post-Quantum Cryptography. *National Institute of Standards and Technology* (2016), 1–15.

[19] CRAMER, R., DAMGÅRD, I., AND NIELSEN, J. B. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[20] CRAMPTON, J., AND MORISSET, C. PTaCL: A Language for Attribute-Based Access Control in Open Systems. In *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings* (2012), pp. 390–409.

[21] DE MOURA, L. M., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (2008), pp. 337–340.

[22] DING, J., AND SCHMIDT, D. Rainbow, a New Multivariable Polynomial Signature Scheme. *Springer-Verlag Berlin Heidelberg* (2005), 164–175.

[23] FEO, L. D. Mathematics of Isogeny Based Cryptography. *École mathématique africaine May 10 âĂŞ 23, 2017, Thiès, Senegal* (2017), 1–44.

[24] FEO, L. D., JAO, D., AND PLÛT, J. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. *Springer-Verlag Berlin Heidelberg* (2011), 19–34.

[25] GIBBONS, J. Functional Programming for Domain-Specific Languages. In *Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers* (2013), pp. 1–28.

[26] GODEFROID, P., AND HUTH, M. Model Checking Vs. Generalized Model Checking: Semantic Minimizations for Temporal Logics. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings* (2005), pp. 158–167.

[27] GROVER, L. K. A fast quantum mechanical algorithm for database search. *Proceedings, STOC 1996, Philadelphia PA USA* (1996), 212–219.

[28] HUELSING, A., BUTIN, D., GAZDAG, S., RIJNEVELD, J., AND MOHAISEN, A. XMSS: eXtended Merkle Signature Scheme. *Internet Research Task Force (IRTF), RFC 8391* (2018), 1–74.

[29] HUTH, M., AND KUO, J. H. On Designing Usable Policy Languages for Declarative Trust Aggregation. In *Human Aspects of Information Security, Privacy, and Trust - Second International Conference, HAS 2014, Held as Part of HCI International 2014, Heraklion, Crete, Greece, June 22-27, 2014. Proceedings* (2014), pp. 45–56.

[30] HUTH, M., AND KUO, J. H. PEALT: An Automated Reasoning Tool for Numerical Aggregation of Trust Evidence. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings* (2014), pp. 109–123.

[31] KRAMER, J. Is abstraction the key to computing? *Commun. ACM 50*, 4 (2007), 36–42.

[32] LI, N., WANG, Q., QARDAJI, W. H., BERTINO, E., RAO, P., LOBO, J., AND LIN, D. Access control policy combining: theory meets practice. In *14th ACM Symposium on Access Control Models and Technologies, SACMAT 2009, Stresa, Italy, June 3-5, 2009, Proceedings* (2009), pp. 135–144.

[33] NIELSON, H. R., AND NIELSON, F. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.

[34] PAULSON, L. C. Isabelle: The Next Seven Hundred Theorem Provers. In *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings* (1988), pp. 772–773.

[35] PERLNER, R. A., AND COOPER, D. A. Quantum Resistant Public Key Cryptography: A Survey. *National Institute of Standards and Technology* (2009), 85–93.

[36] PLOTKIN, G. D. The origins of structural operational semantics. *J. Log. Algebr. Program. 60-61* (2004), 3–15.

[37] RIFKIN, J. *The Age of Access*, first edition ed. Penguin, 31 August 2000.

[38] ROETTELER, M., NAEHRIG, M., SVORE, K. M., AND LAUTER, K. Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms. *Microsoft Research, USA* (2017), 1–24.

[39] SHOR, P. W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, Nov. 20âĂŞ22, 1994, IEEE Computer Society Press* (1995), 124–134.

[40] STEBILA, D., AND MOSCA, M. Post-Quantum Key Exchange for the Internet and the Open Quantum Safe Project. *Based on the Stafford Tavares Invited Lecture at Selected Areas in Cryptography (SAC) 2016 by D. Stebila* (2017), 1–22.

[41] STEWART, I., ILIE, D., ZAMYATIN, A., WERNER, S., TORSHIZI, M. F., AND KNOTTENBELT, W. J. Committing to quantum resistance: a slow defence for Bitcoin against a fast quantum computing attack. *Royal Society Open Science* (2018), 1–12.

[42] VOLLMER, H. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag, Berlin, Heidelberg, 1999.

A Post-quantum Cryptography Security Review

Every cryptographic schemes that is believed to secure relies on computational problems that are assumed to be hard to solve. Many of the underlying mathematical problems forming the foundation for these computational problems are conjectured to be hard and some are even proven to be NP-hard [9].

The security of cryptographic schemes is measured with respect to the number of bits of input or output values and measures the units of time it takes to break the scheme and additionally the amount of memory it takes to break the scheme in the specified time. But computational hardness is relative to the computer system used for classical computers with bits compared to quantum computers with qubits. This means that the security levels, also called bit strength, of some cryptographic schemes are substantially lower when under attack by a quantum algorithm versus a classical algorithm.

A.1 Pre-quantum Security Levels

In the following we provide a short overview over the security levels of cryptographic schemes under a classical pre-quantum attack algorithm. Symmetric ciphers with key length of n bits may have a security level of up to n bits. Todays common ciphers like AES offer almost the claimed levels and the best known attacks are only about 4 times faster than brute force attacks, i.e. they reach a security level of about $n-2$ bits. Hash functions with output length of n bits may have a security level for collision resistance of up to $n/2$ bits and a security level for preimage resistance of up to n bits. Current hash functions like SHA2 also offer these security levels and the only known attacks work for non-full rounds of the underlying algorithms.

The security level of asymmetric ciphers with key length of n bits strongly depends on the conjectured hardness of their underlying mathematical problems, where the most commonly employed ones are the following:

- **Integer factorization**: The RSA algorithm is based on the assumed computational hardness of integer factorization of large semiprimes. There exist sub-exponential algorithms like the general number field sieve with complexity

$$\mathcal{O}\left(\exp\left(\sqrt[3]{\frac{64}{9}n(\log n)^2}\right)\right)$$

  which lower its security level to approximately $1.75 \cdot n^{1/3}(\log_2 n)^{2/3}$ bits for key length of n bits (e.g. 3072 bit RSA key for 128 bit security level and 15360 bit RSA key for 265 bit security level needed).

- **Discrete logarithm**: The Diffie-Hellman key exchange, ElGamal ciphers and DSA signatures are all based on the assumed computational hardness of the discrete logarithm

over finite fields. There exist sub-exponential algorithms that are closely related to the algorithms for integer factorization and the security levels asymmetric ciphers based on the discrete logarithm are therefore about the same as for RSA.

- **Elliptic curve discrete logarithm**: The elliptic curve Diffie-Hellman key exchange, ciphers and signatures are all based on the conjectured computational hardness of the discrete logarithm in elliptic curves over finite fields. There are known algorithms of complexity $\mathcal{O}(2^{n/2})$ like Pollard's rho and the baby-step giant-step which reduce their security level to approximately $n/2$ bits for key length of n bits (e.g. 256 bit ECC key for 128 bit security level and 512 bit ECC key for 256 bit security level needed).

A.2 Quantum algorithms

Current research might lead to large scale quantum computers which would enable the use of quantum algorithms exploiting qubits and their entaglement in quantum computer designs to perform exponentially more computations in the same time compared to classical computers with the same number of bits as qubits. Especially two major quantum algorithms, namely Grover's [27] and Shor's [39] algorithms, are briefly summarized in the following:

- **Grover's algorithm**: It provides a quantum algorithm for pre-image computation of an input N to a black box function for a given output (or it can be equivalently seen as a search in a database). Instead of a complexity of $\mathcal{O}(N)$ on classical computers, Grover's algorithm has a complexity of $\mathcal{O}(N^{1/2})$ on quantum computers, which is proven to be asymptotically optimal (for fixed size of N a even further reduced complexity of $\mathcal{O}(N^{1/3})$ is possible). In particular, this is not an exponential speedup, i.e. there are currently no known polynomial time quantum algorithms for NP-complete problems like pre-image computation. In terms of input length of n bits Grover's algorithm has a complexity of $\mathcal{O}(2^{n/2})$ on quantum computers compared to $\mathcal{O}(2^n)$ on classical computers.

- **Shor's algorithm**: It provides a quantum algorithm for integer factorization of large (odd and non-prime power) integers N. Instead of a complexity of described above for integer factorization on classical computers, Shor's algorithm has a complexity of $\mathcal{O}((\log N)^3)$ on quantum computers. Notably, this is an exponential speedup resulting in a polynomial time quantum algorithm and in terms of integers of length of n bits Shor's algorithm has a complexity of $\mathcal{O}(n^3)$ on quantum computers compared to roughly $\mathcal{O}(2^{n^{1/3}(\log n)^{2/3}})$ on classical computers.

Currently these quantum algorithms cannot be applied to realistic scenarios, because they require a sufficiently strong quantum computer, i.e. for instance $9\,n + 2\,\lceil \log_2(n) \rceil + 10$ qubits and a quantum circuit of $448\,n^3\,log_2(n) + 4090\,n^3$ gates for an application of Shor's algorithm to compute the elliptic curve discrete logarithm over n bit-sized prime fields [38]. But once employable their implications range from having to double key sizes to the need for designing and using completely new algorithms. Therefore, different standardization bodies like NIST [18], ETSI [17] and PQCRYPTO [3] have started a review and standards process on post-quantum secure cryptographic schemes with different focus on e.g. long-term security, efficiency and implementational details.

A.3 Post-quantum Security Levels

Symmetric ciphers and hash functions are susceptible to Grover's algorithm, but not to Shor's algorithm. Hence, it is sufficient to double (or triple) the number of bits for the key length of symmetric ciphers [3] and the output length of hash functions [7], respectively, to counter the impact of Grover's algorithm and to maintain a security level of n bits.

In contrast to that all classical asymmetrical ciphers based on integer factorization and discrete logarithm are susceptible to Shor's algorithm. Due to the polynomial time attacks, these ciphers are not post-quantum secure and would require exponential growth in key size to maintain a security level of n bits, which is practically unfeasible for secure key length. Post-quantum secure asymmetric cryptography therefore needs to employ algorithms on more complex algebraic structures, which are believed to be unsusceptible to Shor's algorithm and which, whenever possible, have provable reductions to hard problems. The following algorithms under review by the standardization bodies differ in the necessary key size for same security levels, the computational efficiency for same security levels, the applicability to and the constructability of encryption or signature schemes, as well as the straight-forwardness and ease of replacing current pre-quantum cryptographic schemes [18, 35, 40]:

- **Latice-based cryptography**: Algebraic structures (like polynomial rings) on lattices over finite fields are employed to construct key exchange, encryption and signature schemes. They are often provably quantum secure depending on the conjectured hardness of the underlying shortest-vector-problem (SVP) and are also proven to be NP-hard under some reductions to SVP like the learning-with-errors (LWE) and ring-LWE problems [40]. Some of these problems can also be shown to be hard *on average*, which is attractive for cryptography that works with randomness.

- **Multivariate cryptography**: Multivariate polynomials over finite fields are employed to build key exchange, encryption and signatures schemes, e.g. the Rainbow signature scheme [22]. These are based on the proven NP-hardness of the underlying problem of solving systems of multivariate polynomials.

- **Hash-based cryptography**: The NP-hard problem of pre-image computation for (hypothetical) one-way functions is carried over to cryptographically secure hash functions for which pre-image computation is believed to be computationally hard. These offer applications to signature schemes, e.g. the eXtended Merkle Signature Scheme (XMSS) [28] based on one-time signatures and hash trees.

- **Code-based cryptography**: Error correction codes are employed to construct key exchange, encryption and signatures schemes, e.g. McEliece encryption schemes based on Goppa error correction codes [8]. The underlying problem of decoding a general linear code is proven to be NP-hard.

- **Supersingular elliptic curve isogeny cryptography**: Isogenies between supersingular elliptic curves over finite fields are used to build key exchange and encryption schemes, e.g. Diffie-Hellman key exchange with perfect forward secrecy [24]. These are based on the hard problem of computing the isogeny from the image of a supersingular elliptic curve's torsion subgroup under that isogeny [23], which has a proven complexity of $\mathcal{O}(2^{n/4})$ for primes of length of n bits on classical computers, and a conjectured complexity of $\mathcal{O}(2^{n/6})$ on quantum computers.

The transition to post-quantum secure schemes should be done once they are considered standardized and sufficiently trusted by the cryptographic community. Some standardization bodies, e.g. like NIST, insist on a slow review and standards process here, since the pace and nature of evolution for post-quantum computations is unclear. All types of applications which use ephemeral keys should switch when these issues are settled, but all types of applications which use persistent (at least for some longer time) keys need extra considerations, see e.g. the case of transitioning bitcoin public keys from post-quantum insecure signature schemes to post-quantum secure signature schemes in [41].

Symmetric encryption schemes should be used with double key length and authentication for symmetric encryption schemes (MACs) could be used as they are, or should be used with double key length if one wants to be very conservative. Also hash functions should be used with double output length. Asymmetric encryption and signature schemes could switch to post-quantum secure schemes that are known and studied for long time, like McEliece code based encryption schemes, or otherwise should be updated to post-quantum secure schemes as soon as they are standardized, like lattice-based or isogeny encryption schemes. Here, differentiation between short and long term usage is important: schemes employed for encrypting and signing messages with a short life span might keep on using post-quantum insecure schemes like RSA or ECDSA for some time, but schemes employed for encrypting and signing long term messages should transition to post-quantum secure schemes sooner than later.

Obviously, schemes that are conjectured to be post-quantum secure provide protection against quantum computers. Additionally they also provide better protection against classical computers if any mathematical breakthrough regarding the underlying problems of current and future cryptographic schemes is achieved, since provable NP-hard problems are more likely to be secure than problems that are just conjectured to be hard. Due to their higher complexity post-quantum secure schemes need more computational power and memory to be performed, which poses a problem for hardware constrained devices and time dependent services. But next to presumable hardware advancements the research on post-quantum secure cryptography will most likely improve the efficiency of current and new cryptographic algorithms. Also, switching to post-quantum secure schemes that are somewhat similar to todays widely employed schemes will foster their acceptance and make transition smoother, e.g. supersingular elliptic curve isogeny cryptography provides in principle similar Diffie-Hellman key exchange mechanisms like classical elliptic curve Diffie-Hellman key exchange protocols.