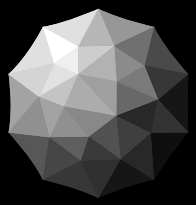
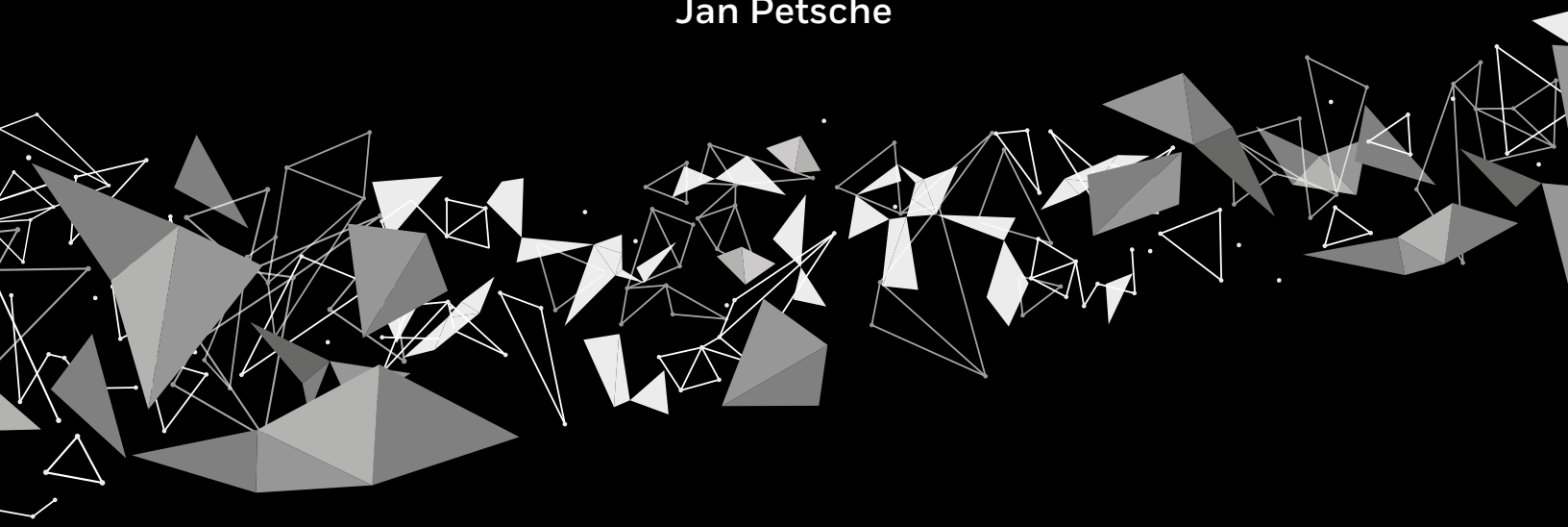


XAIN.

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 0.9

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 [49]. 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" [43]; 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. We highlight this central requirement here:

> **Flexible But Contained Delegation:** *We need the ability for owners and users of data and resources to delegate basic access rights and to delegate (a) the ability to delegate such rights and (b) the ability to formulate rich access controls – and to do all of that with appropriate controls for such powerful delegation mechanisms.*

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.

This paper is expressly written to be agnostic of any implementation of FROST technology, such implementations will be the subject of future papers.

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.

In Appendix A, we offer a review of post-quantum cryptography as something that informs implementation choices for the FROST technology. Appendix B discusses a deep embedding of the FROST language within an existing functional programming language.

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.

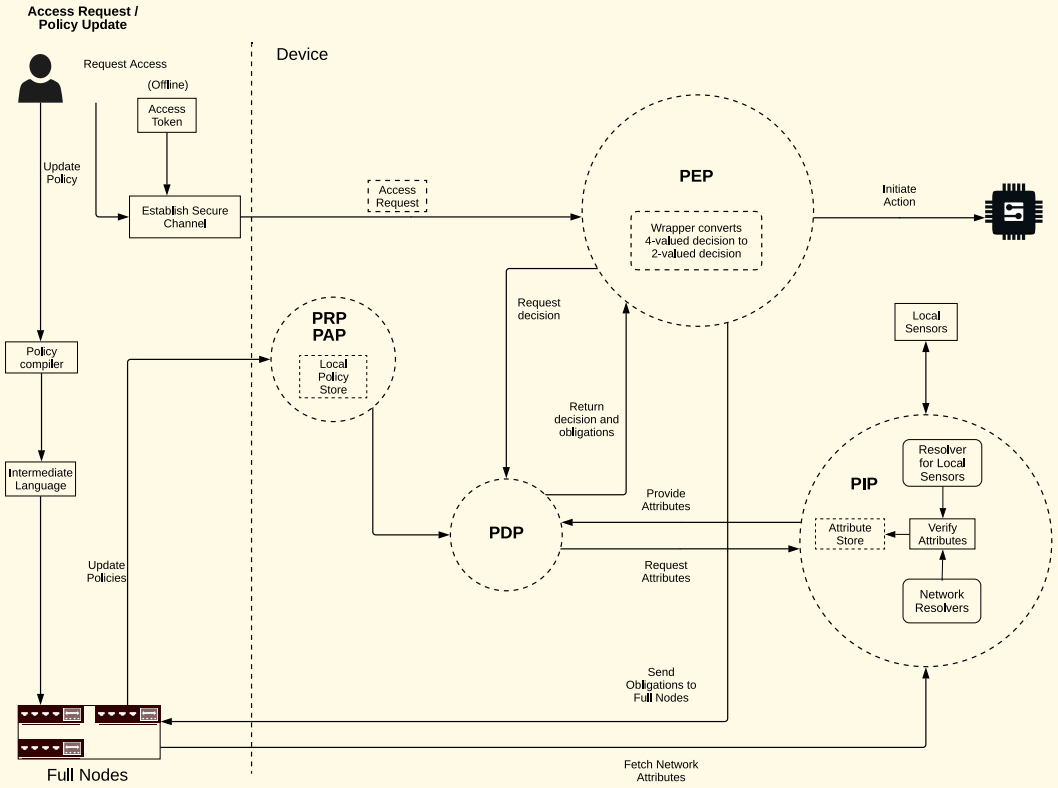


Figure 1: Architecture diagram of the FROST access-control framework

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

$$(user, resource, action, context)$$

where $user$ is the machine, human, computer method or other entity making the request – known as the *requester*; $resource$ 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 a concrete 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 delegation 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, and peer-reviewed 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:

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

where $cond$ is a logical expression built from attributes, their values and comparisons, as well as logical operators.

Example 1. *To illustrate, we may specify an access-control rule*

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

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

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

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 appropriately 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 as such 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 $n \geqslant 1$). 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 $vehicle.owner.daughter < localTime$.

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

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`-policy.

The latter contains at least two cases, 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`-policy evaluates to the first policy pol_i of a pair $guard_i\colon pol_i$ within that `case`-policy 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`-policy. In the `case`-policy of Example 2 below, these sub-policies are P, Q, and `conflict`.

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`-policies (the latter used for policy composition) to compute the outcome of this policy – which is either `grant`, `deny`, `undef` or `conflict`. In Section 6.3, we provide an approach that can compute such meaning also when not all attribute values are known, without giving an attacker a means of exploiting such incomplete information.

2.3 Functional Completeness

Let us write

$$\mathbf{4} = \{\texttt{grant}, \texttt{deny}, \texttt{undef}, \texttt{conflict}\} \tag{1}$$

for the 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 $n \geqslant 0$ where composition operators for $n = 0$ merely return a decision seen as a constant policy. This will guarantee that any policy behavior of this type can be written in the FROST language.

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`-policy 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 $n \geqslant 0$ 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).

Example 2. *We can express this specification of policy behavior 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.*

$$
\begin{array}{l}
\texttt{case}\ \{ \\
\quad [(P\ \texttt{eval undef})\colon Q] \\
\quad [(Q\ \texttt{eval undef})\colon P] \\
\quad [(P\ \texttt{eval conflict})\colon \texttt{conflict}] \\
\quad [(Q\ \texttt{eval conflict})\colon \texttt{conflict}] \\
\quad [((P\ \texttt{eval deny})\ \&\&\ (Q\ \texttt{eval grant}))\colon \texttt{conflict}] \\
\quad [((P\ \texttt{eval grant})\ \&\&\ (Q\ \texttt{eval deny}))\colon \texttt{conflict}] \\
\quad [\texttt{true}\colon P] \\
\}
\end{array}
$$

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

Policy Decision Points (PDPs) need to evaluate a policy pol in order to compute its decision for a given access request and its context. An implementation therefore needs to provide a run-time system for such evaluations. For example, policies may be embedded into the Scala programming language and executed on the trusted Java virtual machine for FROST deployments on enterprise systems.

However, we foremost want to realize FROST close to resources, where computational constraints typically prevent the use of such powerful run-time systems. Therefore, our approach is to represent a policy pol via two Boolean Circuits [50] whose propositional variables represent atomic conditions within pol. The intuition for this is that the four possible decisions of pol can

be encoded as a pair of Boolean decisions, one from each circuit, since $2 \cdot 2 = 4$. We may then optimize these Boolean Circuits by interpreting them as Binary Decision Diagrams [14].

These compilations of pol into two Boolean Circuits 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 bilattice depicted in Figure 4. On this bilattice, we can define two subsets: $GoC = \{\texttt{grant}, \texttt{conflict}\}$ and $DoC = \{\texttt{deny}, \texttt{conflict}\}$. We can uniquely identify an element of that bilattice 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 from the set 4 in (1) that is neither in GoC nor in DoC. And `grant` is the unique such element that is in GoC but not in DoC.

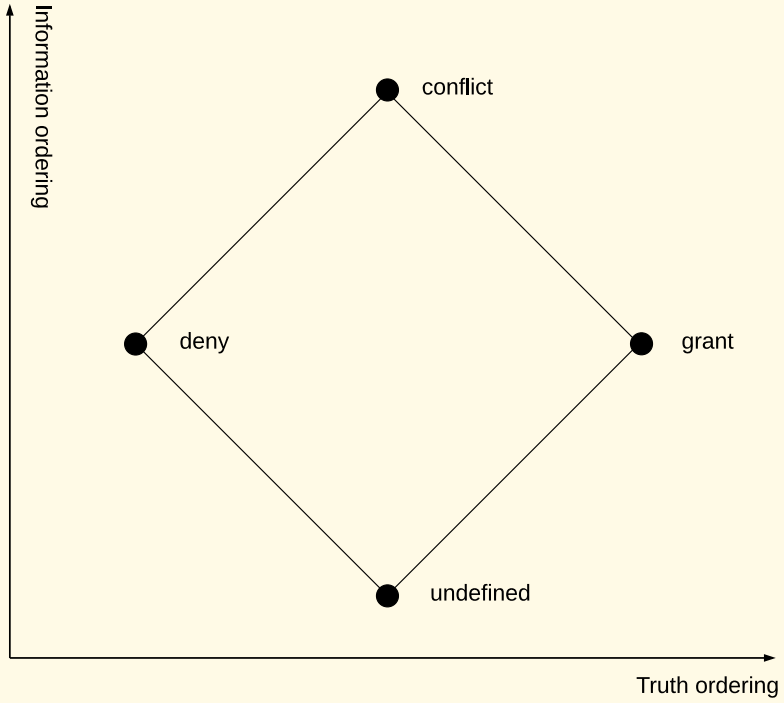


Figure 4: The 4-valued Belnap bilattice, as used for access control in [11, 12]. The x-axis represents the *truth* ordering whereas the y-axis represents the *information* ordering.

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

$$pol \;\equiv\; (\texttt{grant if } \mathsf{GoC}(pol)) \;\texttt{join}\; (\texttt{deny if } \mathsf{DoC}(pol)) \tag{2}$$

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 (2) 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.

These predicates make use of a convenience predicate $\mathsf{R}(g_i)$, defined in Figure 7, that specifies the exact logical condition for executing case $g_i : p_i$ in a given `case`-policy.

$$\begin{aligned}
\mathsf{GoC}(\texttt{grant}) &\equiv \mathsf{true} & \mathsf{GoC}(\texttt{deny}) &\equiv \mathsf{false}\\
\mathsf{GoC}(\texttt{conflict}) &\equiv \mathsf{true} & \mathsf{GoC}(\texttt{undef}) &\equiv \mathsf{false}\\
\mathsf{GoC}(\texttt{grant if } cond) &\equiv cond & \mathsf{GoC}(\texttt{deny if } cond) &\equiv \mathsf{false}
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{DoC}(\texttt{deny}) &\equiv \mathsf{true} & \mathsf{DoC}(\texttt{grant}) &\equiv \mathsf{false}\\
\mathsf{DoC}(\texttt{conflict}) &\equiv \mathsf{true} & \mathsf{DoC}(\texttt{undef}) &\equiv \mathsf{false}\\
\mathsf{DoC}(\texttt{deny if } cond) &\equiv cond & \mathsf{DoC}(\texttt{grant if } cond) &\equiv \mathsf{false}
\end{aligned}$$

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

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 $\mathsf{R}(guard)$ that specifies the conditions needed to reach the execution of the respective continuation policy, defined in Figure 7.

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

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.

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.

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 that syntactic category $cond$ 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 (2).

$$
\begin{aligned}
\mathsf{R}(g_1) &\equiv \quad \mathsf{T}(g_1) \\
\mathsf{R}(g_i) &\equiv \neg\mathsf{T}(g_1)\ \&\&\ \ldots\ \&\&\ \neg\mathsf{T}(g_{i-1})\ \&\&\ \mathsf{T}(g_i)\,, \quad 1 < i < n \\
\mathsf{R}(\mathsf{true}) &\equiv \neg\mathsf{T}(g_1)\ \&\&\ \ldots\ \&\&\ \neg\mathsf{T}(g_{n-1})
\end{aligned}
$$

Figure 7: For the `case`-policy of Figure 5, predicate $\mathsf{R}(g_i)$ spells out the exact conditions on the input needed to reach the execution of the continuation policy p_i, for each $1 \leqslant i < n$. The predicate $\mathsf{R}(\mathsf{true})$ spells out the exact conditions to reach the execution of the default policy p_n. These predicates depend on predicate $\mathsf{T}(guard)$ of Figure 6.

3.2 Boolean Circuits as Binary Decision Diagrams

The Boolean Circuits $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ are Boolean functions over Boolean variables if we interpret the atomic conditions occurring in pol as Boolean variables. This interpretation is useful for circuit optimization, as we will now discuss.

A *Reduced Ordered Binary Decision Diagram* (ROBDD) [14] B_f represents a Boolean function f with n Boolean variables x_1 to x_n as a directed acyclic graph (DAG) with the following properties:

- the DAG B_f has a root note,
- every non-terminal node in B_f denotes some propositional variable x_i and has two children labeled with 0 (variable x_i is false) and 1 (variable x_i is true), respectively,
- the DAG B_f has exactly two leaves (terminal nodes) 0 and 1, the possible results of function f,
- the DAG B_f is reduced in that it does not contain redundancies such as isomorphic sub-DAGs,
- the DAG B_f is ordered in that there is a linear order on variables such that non-leaf child nodes are lower in that variable ordering than their parent node.

We refer to ROBDDs as BDDs subsequently. To evaluate a BDD B_f, we start at its root node and follow the path determined by the truth values of all its variables x_i (go to the 0-child of node x_i if x_i if false, otherwise go to its 1-child), till we reach either the 0 or 1 leaf as result. Figure 8 depicts an example of a BDD.

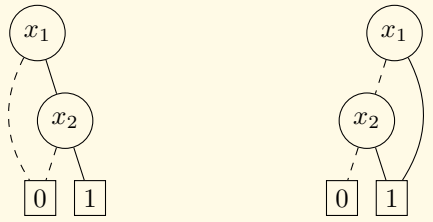


Figure 8: BDDs for x_1 && x_2 (left) and x_1 || x_2 (right) with variable ordering $x_1 < x_2$. Dashed edges represent low edges to 0-children, solid edges represent high edges to 1-children.

ROBDDs have *canonical form:* for every Boolean function f and linear ordering of its variables there is only one ROBDD that represents f (up to graph isomorphism). This is very useful, e.g. we can check the equality of two BDDs by equality of the hashes of their (canonical) form. And it also guarantees that circuits that have trivial meaning (always output 0 or always output 1) will be reduced to these constant Boolean functions, for *all* chosen variable orderings.

The ability of BDDs to eliminate redundancies in representations is also appealing because it can minimize the size and complexity of Boolean Circuits that execute on resource-constrained PDPs. And this is especially so since, for a given linear ordering of variables, an algorithm can synthesize the BDD for function f efficiently in the size of the BDDs it generates for sub-functions of function f.

The latter is an important point since the size of BDDs typically depends on the chosen variable ordering, where heuristics often determine good choices. In fact, the more important an efficient representation of a circuit will be, the more variable orderings we may try out. For deployment, we may then choose the ordering whose BDDs (and circuits) are most compact, by whatever desired measure of efficiency.

We use BDDs in the generation of $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ as follows:

1. We assign each atomic condition in pol, e.g. $\mathbf{subject} = owner$, to a unique atomic proposition x_i.
2. We re-interpret the definitions in Figures 5-7 over BDDs where each atomic condition is interpreted as its corresponding propositional atom x_i.
3. This re-interpretation generates BDDs $B_{\mathsf{GoC}(pol)}$ and $B_{\mathsf{DoC}(pol)}$.
4. These BDDs are then re-compiled into the Boolean circuit data format such that
   - the logical structure of both BDDs is being preserved, and
   - each x_i in these BDDs is modified to the atomic condition it denotes.

Let us illustrate the second step above on some examples, where the linear ordering of variables is given and implicit. The definition $\mathsf{GoC}(\texttt{grant if } cond) = cond$ does not compute $cond$ but computes the BDD that is synthesized from the Boolean function $cond$. Similarly, $\mathsf{GoC}(\texttt{deny if } cond) = \mathsf{false}$ is interpreted as generating the unique BDD that contains only the leaf 0, which is also its root.

Other equational definitions contain logical operators on the righthand side. For example, $\mathsf{T}(g_1 \mathbin{\&\&} g_2) \equiv \mathsf{T}(g_1) \mathbin{\&\&} \mathsf{T}(g_2)$ is interpreted as:

1. synthesize the BDD B_1 for $\mathsf{T}(g_1)$,
2. synthesize the BDD B_2 for $\mathsf{T}(g_2)$,
3. synthesize the BDD that is the logical conjunction of B_1 and B_2.

The last step, and in fact all such synthesis work, relies on the abstract data type for BDDs, in which all logical operations are supported as operations on the corresponding BDDs. This abstract datatype has mature implementations in packages for a variety of programming languages, including Java and C; this is another benefit of using BDDs for circuit optimization of FROST policies.

Let us briefly illustrate on a FROST policy how BDDs can help with circuit optimization.

Example 3 (Boolean Circuit as BDD)**.** *Consider the deny-by default* `case`*-policy of Example 6 denoted as Q below and its compilation according to Figure 5. Even for simple policies P like the grant rule of Example 1 the circuits are growing fast, resulting into the* $\mathsf{GoC}(Q)$ *circuit*

$$
\begin{array}{l}
(\neg\mathsf{GoC}(P) \mathbin{\&\&} \neg\mathsf{DoC}(P) \mathbin{\&\&} \mathsf{false}) \mid\mid \\
(\neg(\neg\mathsf{GoC}(P) \mathbin{\&\&} \neg\mathsf{DoC}(P)) \mathbin{\&\&} \mathsf{GoC}(P) \mathbin{\&\&} \mathsf{DoC}(P) \mathbin{\&\&} \mathsf{false}) \mid\mid \\
(\neg(\neg\mathsf{GoC}(P) \mathbin{\&\&} \neg\mathsf{DoC}(P)) \mathbin{\&\&} \neg(\mathsf{GoC}(P) \mathbin{\&\&} \mathsf{DoC}(P)) \mathbin{\&\&} \mathsf{GoC}(P))
\end{array}
$$

and the $\mathsf{DoC}(Q)$ *circuit*

$$
\begin{array}{l}
(\neg\mathsf{GoC}(P) \mathbin{\&\&} \neg\mathsf{DoC}(P) \mathbin{\&\&} \mathsf{true}) \mid\mid \\
(\neg(\neg\mathsf{GoC}(P) \mathbin{\&\&} \neg\mathsf{DoC}(P)) \mathbin{\&\&} \mathsf{GoC}(P) \mathbin{\&\&} \mathsf{DoC}(P) \mathbin{\&\&} \mathsf{true}) \mid\mid \\
(\neg(\neg\mathsf{GoC}(P) \mathbin{\&\&} \neg\mathsf{DoC}(P)) \mathbin{\&\&} \neg(\mathsf{GoC}(P) \mathbin{\&\&} \mathsf{DoC}(P)) \mathbin{\&\&} \mathsf{DoC}(P))
\end{array}
$$

Since P is of form `grant if` $cond$ *where $cond$ is the conjunction of six atomic conditions, we require corresponding Boolean variables x_1 to x_6. By abuse of notation, we may then write*

$$\mathsf{GoC}(P) = x_1 \mathbin{\&\&} x_2 \mathbin{\&\&} x_3 \mathbin{\&\&} x_4 \mathbin{\&\&} x_5 \mathbin{\&\&} x_6 \qquad \mathsf{DoC}(P) = \mathsf{false}$$

If we enumerate all variables x_i in order of appearance in condition $cond$, i.e. $x_1 < \ldots < x_6$, then the corresponding BDDs significantly simplify the circuits for the entire case statement.

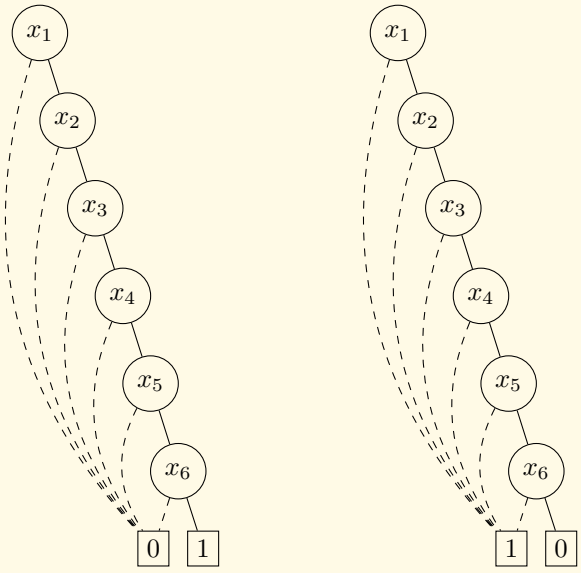


Figure 9: BDDs of Example 3 for circuit $\mathsf{GoC}(Q)$ (left) and circuit $\mathsf{DoC}(Q)$ (right) where Q is the `case`-policy of Example 6 with P being the policy of Example 1.

These BDDs would then be encoded in the data format for conditions, in this case as a chain of if-statements for example.

3.3 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 $\neg\mathsf{GoC}(pol) \mathbin{\&\&} \neg\mathsf{DoC}(pol)$ is unsatisfiable. Similarly, a policy pol from the FROST language is conflict free iff $\mathsf{GoC}(pol) \mathbin{\&\&} \mathsf{DoC}(pol)$ 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.

Since these policies make use of typed attributes, the satisfiability of conditions used in verification depends on the combination of theories for these typed expressions. For example, an attribute $agent.reputation$ may have an axiom

$$\forall\, agents\colon 0 \leqslant agent.reputation \leqslant 1 \tag{3}$$

saying that the reputation of any agent is a real number between 0 and 1. A theory will have a number of such axioms, and we typically deal with a combination of theories, e.g. we may want to combine the axiom in (3) with the usual theory of real-valued arithmetic, and with theories that capture domain-specific knowledge.

Definition 1. *Throughout this Yellow Paper, we write $\mathcal{T}$ to denote a finite set of axioms that capture a combination of theories of interest.*

We then say that a formula ϕ, for example one generated by syntactic category $cond$, is satisfiable modulo $\mathcal{T}$*, iff the formula*

$$\phi \wedge \bigwedge_{\psi \in \mathcal{T}} \psi$$

is satisfiable. We write $\mathsf{SAT}_{\mathcal{T}}(\phi)$ *to denote this test for satisfiability.*

In other words, a formula ϕ is satisfiable modulo $\mathcal{T}$ if we can make ϕ true in a setting that also makes all formulas in $\mathcal{T}$ true. For example, $0 < agent.reputation < 1$ is satisfiable whereas $(0 < agent.reputation < 1)\ \&\&\ (agent.reputation \leqslant (agent.reputation)^2)$ is unsatisfiable modulo any theory $\mathcal{T}$ that contains the axiom in (3).

For example, when we state in the next example below that $\neg\mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol)$ is unsatisfiable the formal meaning of this is that $\mathsf{SAT}_{\mathcal{T}}(\neg\mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol))$ does not hold. Subsequently, we will often use this informal wording with its intended formal meaning.

This technology, which performs policy analysis by reducing it to satisfiability modulo theories, can also be used to verify the integrity of a representation of a policy pol from our FROST language.

Example 4 (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.

Example 5 (Change Management)**.** *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*

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

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" [22]) – which can be used to compute answers to $\mathsf{SAT}_{\mathcal{T}}(\phi)$ – such as the

OpenSMT solver [13], or through the use of theorem provers such as Isabelle [5, 40]. 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.

Example 6 (Deny-by-default `case`-policy)**.** *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 10.*

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

Figure 10: 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.

3.4 Dead-Code Analysis

We now describe a source-to-source transformation of FROST policies to that *all* intended execution paths of the transformed policy are realizable by some input. These transformations can be seen both as code optimization but also as a means of flagging up potential oversights or misunderstandings of programmers. An execution path that is not realizable is caused by some control structure not allowing for the execution of all its branches. In the FROST language, we have two such control structures: rules and `case`-policy. We refer to branches that can never be executed as *dead code*, the familiar term for this phenomenon in program analysis (see e.g. [23]). Dead code is hardly ever intended to be written by any programmer, therefore there is great benefit in detecting and removing it before a FROST policy is compiled into Boolean circuits for execution.

Execution of FROST Policies Cannot Get Stuck Let us first note that a policy written in FROST, and where the atomic predicates are well typed, can never get “stuck” in its execution

upon a complete input – a vector of values for the attributes occurring in that policy. This is so since constant policies immediately return a decision, and rules can evaluate their condition (due to well typedness of the condition) and then immediately return `undef` or the decision value of the rule (`grant` or `deny`). For `case`-policies, execution cannot stop either: either one of the first $n-1$ guards g_i evaluates to `true` or the default case applies. The evaluation of guards g_i cannot get stuck, by an argument that uses structural induction, since its policies do not get stuck (again, by structural induction) and the equality test of policy decisions and evaluation of logical conjunctions cannot get stuck either due to well typedness. Therefore, a continuation policy p_i of that `case`-policy, with $1 \leqslant i \leqslant n$, gets executed. By structural induction, that policy p_i also does not get stuck.

Detecting and Eliminating Dead Code The code transformations that we propose on FROST policies are source to source: a FROST policy p is transformed to a semantically equivalent FROST policy p' such that each of the execution paths of p' does actually occur for some suitable input to policy p'. This is true whenever the satisfiability checker we employ for expressions of syntactic clause $cond$ are complete, meaning that all calls to it with conditions $cond$ return either "satisfiable" or "unsatisfiable". Should some calls to this SAT checker return "don't know" instead (the underlying theory may be undecidable in general or the input may be computationally too demanding), we will interpret this answer soundly so that we only ever remove dead code; this is then an *under-approximation* as it may leave some dead code within a policy. However, assuming that no SAT call returns "don't know", then we are guaranteed that all dead code will have been removed by our algorithm below from the policy p.

The algorithm performs a recursive descent over the abstract syntax tree of policy p, processing `case`-policy higher up in that tree first, before processing `case`-policies further below in that tree. The reason for that is that this processing may remove cases and therefore some continuation policies p_i, which may themselves be `case`-policies and so subject to this analysis in principal. But the algorithm thus has no need of processing such p_i as they would be removed in any event in the `case`-policy in which its case gets removed. Note that the same policy p_i, as syntactic object, may occur elsewhere within p where it may not be removed, but then this is a different subtree of the abstract syntax tree of p.

Before we present that algorithm, let us understand the computational issues at hand for each of the syntactic categories of FROST policies.

- **Constant policies:** there are no execution paths here, a constant policy denotes the end of an execution path and so no dead code can originate from constant policies. Therefore, the algorithm will return them unchanged.
- **Rules:** a rule has two branches, depending on the truth value of its condition. In programming terms, a rule is akin to an if-statement. We want to ensure that both branches of a rule can be executed by some inputs. The algorithm therefore needs to be able to decide whether that is possible. We will see below that any failure to reach a branch will make the algorithm transform the rule into a semantically equivalent constant policy.
- `case`**-policies:** a general `case`-policy has $n > 1$ branches, the first $n-1$ guarded by guards g_i with continuation policies p_i, and the n-th one with the default guard `true` and continuation policy p_n. We want to make sure that each of these n cases can indeed be reached by some input to the policy. Otherwise, cases can be removed and may need to be flagged up to the programmer as a potential coding error.

We won't discuss how and when to flag up such issues to programmers here, as this is an orthogonal usability issue of the policy analysis and its implementation. We may address the above semantic considerations with predicates, the first one we already defined above and

recall here for sake of convenience:

$$\mathsf{SAT}_{\mathcal{T}}\,(cond) = \text{formula } cond \text{ is satisfiable}$$
$$\mathsf{NonTr}\,(cond) = \text{formula } cond \text{ is neither logically valid nor unsatisfiable}$$

We note that $\mathsf{NonTr}\,(cond)$ – standing for "Non-Trivial" – can be derived from $\mathsf{SAT}_{\mathcal{T}}\,(\cdot)$ as

$$\mathsf{NonTr}\,(cond) \quad \text{iff} \quad \mathsf{SAT}_{\mathcal{T}}\,(\neg cond)\ \&\&\ \mathsf{SAT}_{\mathcal{T}}\,(cond) \tag{4}$$

It is easy to see that we can detect any dead code in rules with calls to the predicate $\mathsf{NonTr}\,(\cdot)$. For example, consider a rule `grant if` $cond$. This rule does not have dead code if both of its branches can be executed, in other words, if both `grant` and `undef` can be returned by this rule for appropriate inputs. We now show that the latter is equivalent to $\mathsf{NonTr}\,(cond)$ being true.

First, let $\mathsf{NonTr}\,(cond)$ hold, then:

- a witness for the satisfiability of $cond$ exists by (4) as $cond$ is satisfiable, and this witness represents an input on which this rule returns `grant`,
- a witness to the satisfiability of $\neg cond$ exists by (4) since $cond$ is not valid, and this witness represents an input on which this rule returns `undef`.

Second and conversely, two different inputs to that rule that make the rule return `grant` and `undef`, respectively, function as witnesses to the satisfiability of $cond$ and $\neg cond$, respectively; and this implies that $\mathsf{NonTr}\,(cond)$ holds by (4).

From the above, we see that if $\mathsf{NonTr}\,(cond)$ holds for a rule with condition $cond$, the dead code transformation leaves that rule unchanged. What if $\mathsf{NonTr}\,(cond)$ does not hold? We then need to make a distinction. By definition of $\mathsf{NonTr}\,(cond)$ and given that it is false in this instance, we have two cases by appeal to (4):

- $cond$ is unsatisfiable; then the rule always returns `undef` and so the dead-code transformation turns this rule into the constant policy `undef`.
- $\neg cond$ is unsatisfiable; then the rule always returns `grant` and so the dead-code transformation turns this rule into the constant policy `grant` – if the rule is of form `deny if` $cond$, the transformation returns the constant policy `deny` in this case for a similar reason.

The remaining clause for policies is the `case`-policy with $n > 1$ cases, $n-1$ guards g_i with continuation policies p_i, and a default n-th case with continuation policy p_n. The predicate $\mathsf{T}(g)$ is defined as in the Figure 6, it spells out the logical condition for guard g to be true, and $\mathsf{T}(g)$ is an expression of syntactic clause $cond$. As defined in Figure 7, the predicate $\mathsf{R}(g_i)$ spells out the exact conditions on the input needed to execute continuation policy p_i – for each $1 \leqslant i < n$. Similarly, $\mathsf{R}(\mathtt{true})$ spells out the exact conditions for reaching the execution of the default continuation policy p_n.

We should note that these formulas $\mathsf{R}(g_i)$ and $\mathsf{R}(\mathtt{true})$ are dependent on the concrete `case`-policy. For example, for $n = 8$ the removal of the 4-th case as dead code would require us to recompute $\mathsf{R}(g_i)$ for $5 \leqslant i \leqslant 8$ – by removing the conjunct $\neg\mathsf{T}(g_4)$ from these formulas.

It turns out that the treatment of `case`-policies is relatively simple in that we can process its guards in order as specified in the algorithm given in Figure 11. We point out that the first and last if-statement of that algorithm capture the aforementioned under-approximation: whenever a SAT call returns "don't know", that if-statement won't execute its sole branch and so the considered case will not be removed from the `case`-policy. The explanation of the algorithm is given in the caption of Figure 11.

Let us consider some examples of dead-code analysis. The first one illustrates how the instantiation of a composition pattern gives rise to code optimization opportunities that our dead code analysis can pick up.

```
REM = ∅;
for (i = 1 to n − 1) {
  if (SAT_T (R(g_i)) returns "unsatisfiable") {
    remove case g_i : p_i from case-policy;
    recompute R(g_j) for all i < j ⩽ n;
    REM = REM ∪ {i};
  }
  if (REM = {1, ..., n − 1}) {
    return p_n;
  }
  // R(g_i) is "satisfiable" for all i ∈ {1, ..., n − 1} \ REM
  if (SAT_T (R(true)) returns "unsatisfiable") {
    remove case true : p_n;
    m = max({1, ..., n − 1} \ REM);
    if (|{1, ..., n − 1} \ REM| == 1) {
      return p_m;
    } else {
      g_m = true;
      return computed case-policy;
    }
  }
}
```

Figure 11: Pseudo-code for detecting and removing dead code in a case-policy. For case-policies, dead code consists of entire cases $g_i : p_i$ or $\texttt{true} : p_n$. The algorithm iterates through all non-default cases in their declared order. If $\mathsf{R}(g_i)$ is unsatisfiable, the case $g_i : p_i$ gets removed from the case-policy and all predicates $\mathsf{R}(g_j)$ with $i < j \leqslant n$ get recomputed to reflect that removal. Set REM records the indices of those case that get removed.
If this iterative process happens to remove all non-default cases, then the remaining case-policy really consists only of the default case $\texttt{true} : p_n$ and so the entire case-policy is being replaced with policy p_n. Otherwise, a non-empty list of non-default cases $g_i : p_i$ with $i \in \{1, \ldots, n-1\} \setminus REM$ from the input case-policy remains.
For the reachability predicate $\mathsf{R}(\texttt{true})$ of this remaining case-policy, we check whether that formula is unsatisfiable. If so, the previous case can function as default case and we remove default case $\texttt{true} : p_n$. Then we need to make a case distinction: if there is only one case remaining, which will be $g_m : p_m$, then policy p_m is returned and so the case-policy is replaced with p_m by our dead-code analysis; otherwise more than one case remains and we set g_m to be true making $\texttt{true} : p_m$ the new default case and return that case-policy.
This achieves that, whenever a case-policy is being returned, all remaining $\mathsf{R}(g_i)$ are satisfiable, including the one for the default case. Since these predicates capture reachability of these clauses, this demonstrates that the returned case-policy is free of dead cases. Of course, the same dead code analysis needs to be performed next on each remaining continuation policy p_i, including in the case in which only p_n or p_m is returned by the above algorithm.

Example 7. *Re-consider the* `case`*-policy in Figure 3 which encodes the meaning of the* `join` *composition operator. Let us assume that we now instantiate that* `case`*-policy with a policy* P *that can only evaluate to* `grant` *or* `undef`*, and a policy* Q *that cannot evaluate to* `undef` *but to all other three values.*

Then our algorithm will remove the second, third, and fifth case due to resulting unsatisfiabilities from the possible outputs of P *and* Q*. It will therefore compute the optimized* `case`*-policy*

$$
\begin{array}{l}
\texttt{case}\ \{ \\
\quad [(P\ \texttt{eval undef}) \colon Q] \\
\quad [(Q\ \texttt{eval conflict}) \colon \texttt{conflict}] \\
\quad [((P\ \texttt{eval grant})\ \&\&\ (Q\ \texttt{eval deny})) \colon \texttt{conflict}] \\
\quad [\mathsf{true} \colon P] \\
\}
\end{array}
$$

Then we have that

$$
\begin{aligned}
\mathsf{R}(\mathsf{true}) = \neg(P\ \texttt{eval undef})\ \&\&\ \neg(Q\ \texttt{eval conflict})\ \&\& \\
\neg((P\ \texttt{eval grant})\ \&\&\ (Q\ \texttt{eval deny}))
\end{aligned}
$$

This formula is satisfiable when P *and* Q *evaluate to* `grant`*. Therefore, the default case remains as is (the last if-statement in Figure 11 is not executed), and the above* `case`*-policy is the output of our algorithm.*

Let us know illustrate how our dead-code analysis can also flag up potential programmer issues or errors that developers should be alerted to:

Example 8. *Consider the policy*

$$P = \texttt{grant if}\ ((user.reputation > 1.5)\ \&\&\ user.insured)$$

and some other policy Q *within a* `case`*-policy*

$$
\begin{array}{l}
\texttt{case}\ \{ \\
\quad [P\ \texttt{eval undef} \colon \texttt{deny}] \\
\quad [P\ \texttt{eval grant} \colon \texttt{grant}] \\
\quad [\mathsf{true} \colon Q] \\
\}
\end{array}
$$

Let the theory $\mathcal{T}$ *contain the axiom in* (3)*. Then*

$$\mathsf{SAT}_{\mathcal{T}}\ (\neg((user.reputation > 1.5)\ \&\&\ user.insured))$$

is satisfiable, e.g. by making $user.insured$ *false and setting* $user.reputation$ *to* 0.2*. Therefore,* P `eval undef` *is also satisfiable. However,* $\mathsf{SAT}_{\mathcal{T}}\ ((user.reputation > 1.5)\ \&\&\ user.insured)$ *is unsatisfiable, since no reputation score can be above* 1.0*. The* 1.5 *in policy* P *is presumably a typo made by the programmer. Therefore, the dead-code analysis will remove only the second case from this* `case`*-policy and turn this into*

$$
\begin{array}{l}
\texttt{case}\ \{ \\
\quad [P\ \texttt{eval undef} \colon \texttt{deny}] \\
\quad [\mathsf{true} \colon Q] \\
\}
\end{array}
$$

The programmer might then reflect on this removal and its subsequent analysis would help with identifying the programming error.

3.5 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 a natural host for our FROST language. Indeed, embedded DSLs are in general a well-suited use case for functional programming [27]. 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 combinators 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 – a concept familiar from XACML [44] – 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 12 shows one way in which this operator may be defined in our FROST language.

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

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

Figure 12: 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.

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

```
case {
  [(P eval grant) && (Q eval deny): deny]
  [true: P]
}
```
(5)

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. [35]).

For obligations, we are only interested in the decisions `grant` and `deny`, requiring computations for each of these: $\mathrm{oblg}(\texttt{grant}, pol, \rho)$ computes the obligations of granting for policy pol under ρ, whereas $\mathrm{oblg}(\texttt{deny}, pol, \rho)$ 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 13.

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

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

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

$$\mathsf{oblg}(dec,\, \texttt{case}\,\{\,[g_1 : p_1]\;\ldots\;[g_{n-1} : p_{n-1}]\,[\mathsf{true} : p_n]\,\},\, \rho) \;\equiv\; \mathsf{oblg}'(dec,\, g_i,\, \rho) \cup \mathsf{oblg}(dec,\, p_i,\, \rho)$$
$$\text{where } \rho \models \mathsf{R}(g_i)$$

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

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

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

Figure 13: A compositional, deterministic computation of sets of obligations for policy pol from our FROST language under model ρ where dec is in $\{\texttt{deny}, \texttt{grant}\}$. Models ρ make all formulas in $\mathcal{T}$ true and define a Tarskian truth semantics $\rho \models cond$. 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.

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 scenario in which the first case of the statement in (5) applies. Let us write W for the policy defined by that `case`-statement. Then:

$$\begin{aligned} \mathsf{oblg}(\texttt{deny},\, W,\, \rho) &= \mathsf{oblg}'(\texttt{deny},\, (P\;\texttt{eval}\;\texttt{grant})\;\&\&\;(Q\;\texttt{eval}\;\texttt{deny}),\, \rho) \cup \mathsf{oblg}(\texttt{deny},\, \texttt{deny},\, \rho) \\ &= \mathsf{oblg}'(\texttt{deny},\, P\;\texttt{eval}\;\texttt{grant},\, \rho) \cup \mathsf{oblg}'(\texttt{deny},\, Q\;\texttt{eval}\;\texttt{deny},\, \rho) \cup \{\,\} \\ &= \{\,\} \cup \mathsf{oblg}(\texttt{deny},\, Q,\, \rho) \\ &= \mathsf{oblg}(\texttt{deny},\, Q,\, \rho) \end{aligned} \tag{6}$$

The parameter ρ is a model that captures a Tarskian truth semantics for conditions and also satisfies all formulas in $\mathcal{T}$. 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`.

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 13 also illustrate this need for conditionals (explicitly for rules and implicitly for `case`-policies).

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 (6) where we already simplified the resulting circuit $\mathsf{DObl}(W)$, since the first step would actually be

$$\mathsf{oblg}(\texttt{deny},\, W,\, \rho) = \begin{cases} \mathsf{oblg}'(\texttt{deny},\, G,\, \rho) \cup \mathsf{oblg}(\texttt{deny},\, \texttt{deny},\, \rho) & \text{if } \mathsf{R}(G) \\ \mathsf{oblg}'(\texttt{deny},\, \texttt{true},\, \rho) \cup \mathsf{oblg}(\texttt{deny},\, P,\, \rho) & \text{if } \mathsf{R}(\texttt{true}) \end{cases}$$

where G equals $(P\ \texttt{eval grant})\ \&\&\ (Q\ \texttt{eval deny})$. 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`-policies. The semantics for computing the set of obligations that arise when the computed decision is either `grant` or `deny` has been defined in Figure 13.

One may then capture both of these computations in a structural operational semantics [42], which may be executed on a Landin-style abstract machine (see e.g. [39]) – 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.

Alternatively, one may think of obligations as computational effects and this is illustrated in Appendix B. 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 a specific vehicle model. 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 seeks simpler engineering solutions and will thus 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 `OEM` $\rightarrow$ `dealer` $\rightarrow$ `client` $\rightarrow$ `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 delegates to p_1 ... delegates to p_n" where p_0 is the policy of the resource owner who controls the entire composition chain conservatively through the composition idiom

$$p_0 \gg (p_1 \gg (\cdots \gg p_n) \dots) \tag{7}$$

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

$$\begin{array}{l} \texttt{case}\,\{ \\ \quad [P\ \texttt{eval conflict: deny}] \\ \quad [P\ \texttt{eval undef:}\ Q] \\ \quad [\texttt{true:}\ P] \\ \} \end{array} \tag{8}$$

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 (8) 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, e.g., 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 (7). 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:

$$\begin{aligned} m_0 &= par +\!\!+ pk_0 +\!\!+ p_0 +\!\!+ pk_1 \\ s_0 &= \mathsf{sign}(sk_0,\, H(m_0)) \end{aligned} \tag{9}$$

The initial delegatee should verify the signature and check the depth constraint to see if she could further delegate, cf. sequence diagram in Figure 14.

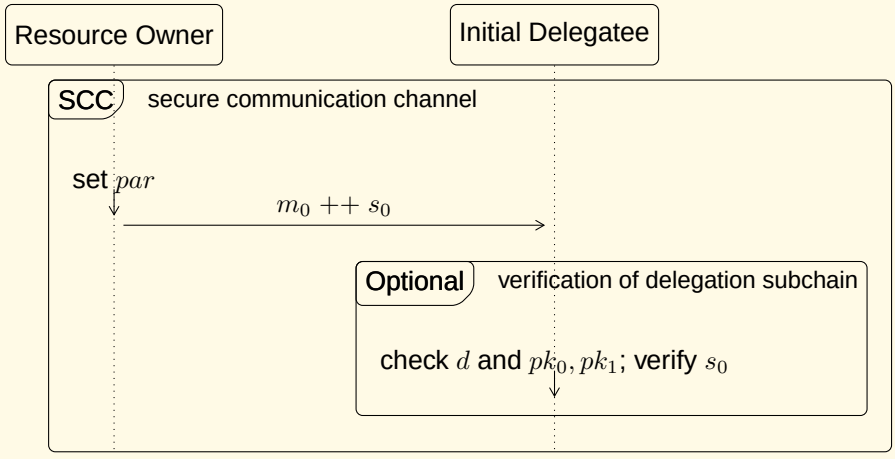


Figure 14: Initialization of the delegation chain, where the message sent via a secure channel is specified in (9) and the verification of this message is optional – in that we cannot guarantee that the initial delegatee as recipient will perform these checks.

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:

$$\begin{aligned} m_i &= par \mathbin{+\!\!+} pk_0 \mathbin{+\!\!+} p_0 \mathbin{+\!\!+} pk_1 \mathbin{+\!\!+} s_0 \mathbin{+\!\!+} \cdots \\ &\quad \cdots \mathbin{+\!\!+} pk_{i-1} \mathbin{+\!\!+} p_{i-1} \mathbin{+\!\!+} pk_i \mathbin{+\!\!+} s_{i-1} \mathbin{+\!\!+} pk_i \mathbin{+\!\!+} p_i \mathbin{+\!\!+} pk_{i+1} \\ s_i &= \mathsf{sign}(sk_i,\, H(m_i)) \end{aligned} \tag{10}$$

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 15.

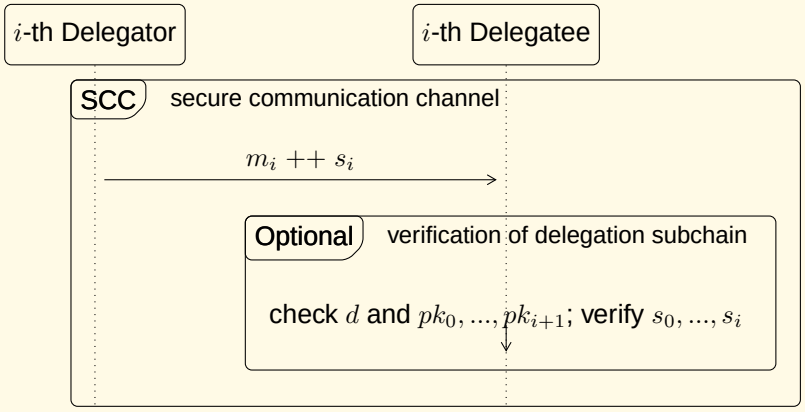


Figure 15: Extending the delegation chain that ranges from resource owner 0 to a delegatee $i-1$ to a new delegatee i, for which $i-1$ is the delegator. Message m_i and its signature s_i are specified in (10). Integrity checks are optional in that we cannot assume that they will be performed by delegatees.

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 (10) but for different identities of the last delegator/delegatee link:

$$\begin{aligned} m_n &= par \mathbin{+\!\!+} pk_0 \mathbin{+\!\!+} p_0 \mathbin{+\!\!+} pk_1 \mathbin{+\!\!+} s_0 \mathbin{+\!\!+} \cdots \\ &\quad \cdots \mathbin{+\!\!+} pk_{n-1} \mathbin{+\!\!+} p_{n-1} \mathbin{+\!\!+} pk_n \mathbin{+\!\!+} s_{n-1} \mathbin{+\!\!+} pk_n \mathbin{+\!\!+} p_n \mathbin{+\!\!+} pk_{rs} \\ s_n &= \mathsf{sign}(sk_n,\, H(m_n)) \end{aligned} \tag{11}$$

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 diagram depicted in Figure 16. This includes checks such as that $n \leqslant d$ is true – failure of that will make the resource reject this chain since it would violate the delegation depth mandated by the resource owner.

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

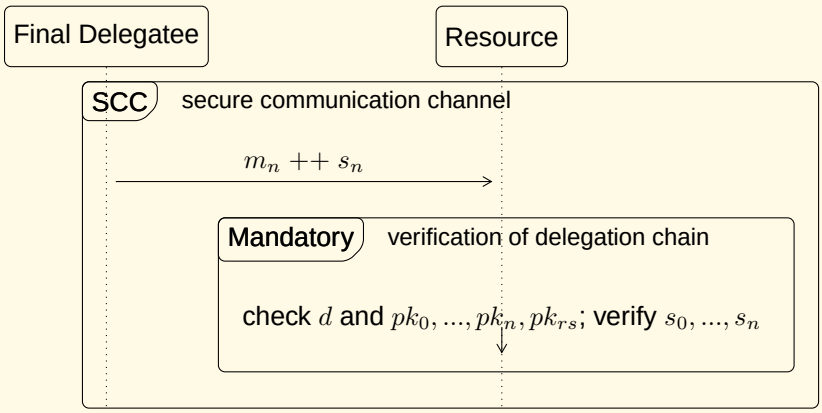


Figure 16: The last step in the delegation chain is similar, if not identical, to the step in Figure 15, except that the targeted delegatee is the resource and that the verification step is not optional – in that the resource is part of the trusted computing base and so will perform this check. Message m_i and its signature s_i are specified in (11).

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 a Delegation Chain

We now give details of the mandatory verification that the resource performs as depicted in Figure 16. After a successful check that $n \leqslant d$ 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 17.

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 commu-

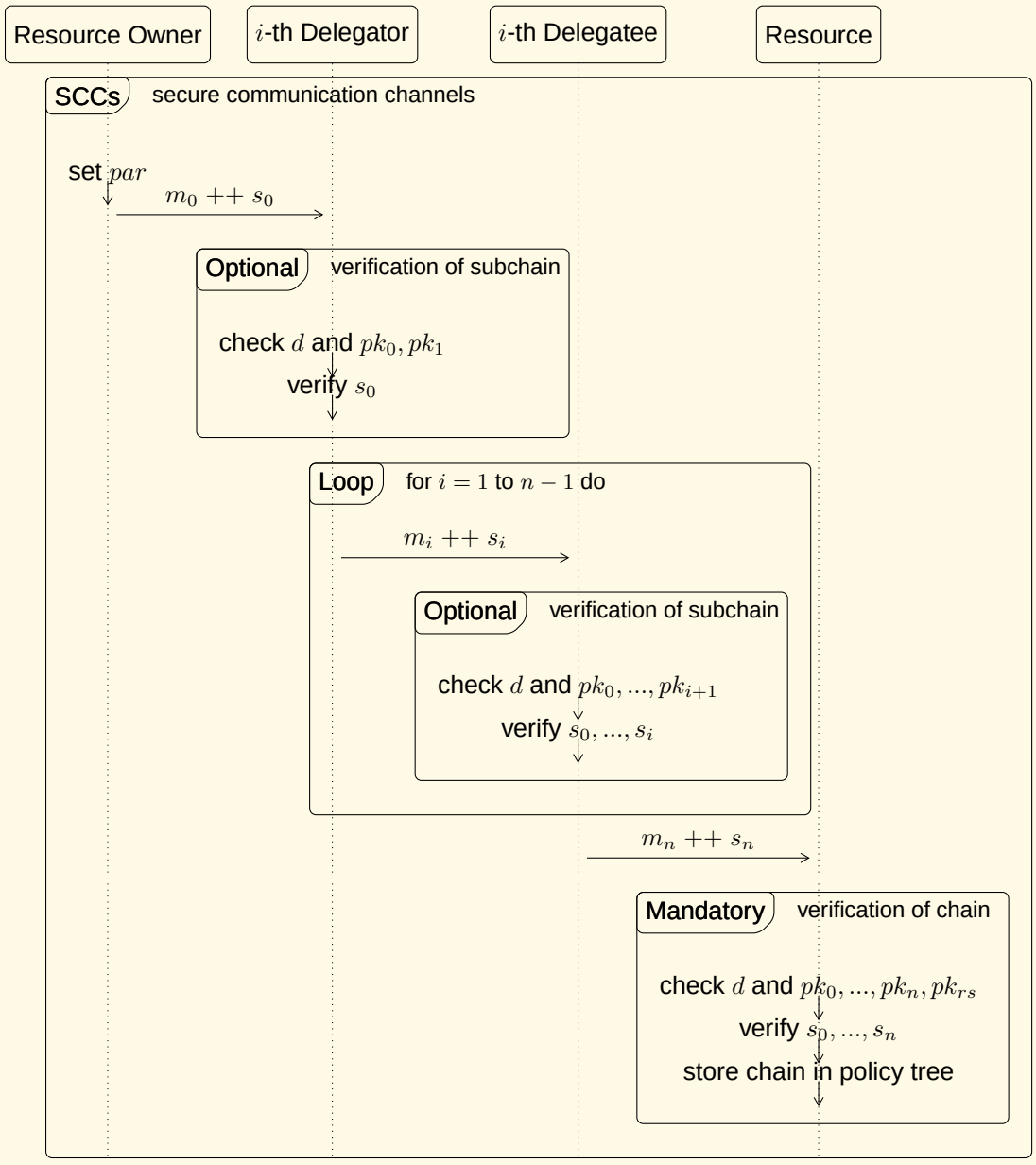


Figure 17: Summary of all optional and mandatory verification activities in the creation and deployment of the delegation chain. Deployment only happens if verification by the resource itself succeeds.

nication 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 Trees

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.
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 This data structure must be consistent with resource constraints of embedded devices.

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 policy or a hash thereof, 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 18 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.

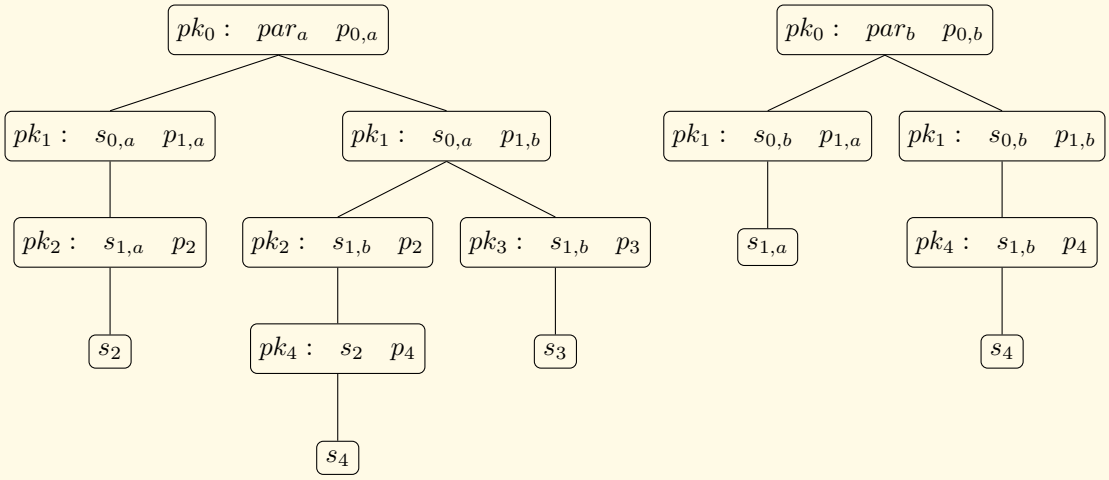


Figure 18: Illustration of our data structure for policy forests with two policy trees controlled by a resource owner. The tree on the left is consistent with $d = 3$, whereas the one on the right is consistent with $d = 2$. Different trees may contain different parameter values within their roots.

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 18 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 18 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:

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:

$$P_n \texttt{ op } P_{n-1} \texttt{ op } \ldots \texttt{ op } P_1 \texttt{ op } P_0 \texttt{ op deny}, \tag{12}$$

where `op` is any binary composition operator expressive in our FROST language, e.g. the operator $\gg$ defined in (7) 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 (12) 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 delegatees j with $j > i$ 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:

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

The resource 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.

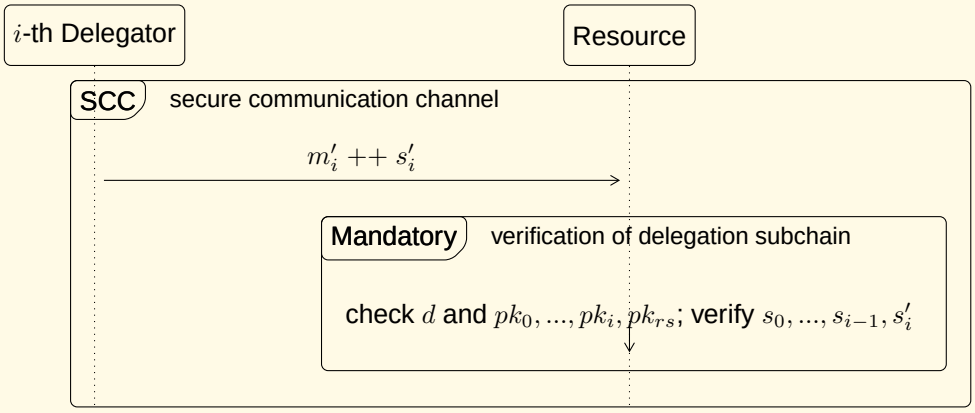


Figure 19: Agent i wants to change her policy p_i on an existing delegation chain to some policy p'_i. This is managed by directly notifying the resource of that change request, and the resource must verify the legitimacy of that request before making the change.

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. Policy verification can achieve this.

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 that

- subsequent delegatees are notified about a policy substitution by a delegator,
- previous delegators or the resource owner are notified about a policy change submitted to the resource,
- delegatees are notified about a change of the delegation depth, and
- 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, and they may want to reflect 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 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, 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 within its 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 [32, 33] 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, a sensor reading for current speed may be missing or a dynamic type error may make the truth value of an atomic condition unknown. A conservative approach for the PDP is to deny any request if some attribute needed for evaluating the respective policy will have 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.

We are instead using techniques from AI and static analysis based on *super-valuational meaning* to evaluate policies under such incomplete information – see e.g. [28] for applications of that in verification with temporal logic. Let us write $\perp$ for both unknown truth value and unknown attribute value. We then can evaluate terms such that $\perp \cdot 0$, e.g., computes to 0 whereas $\perp + 0$ computes to $\perp$. Similarly, $\perp < x$ would result in truth value $\perp$.

Then we use Kleene's strong 3-valued propositional logic shown in Figure 20 – as familiar from hardware verification where in addition to true and false there is a third value $\perp$ meaning "don't know" or "don't care" depending on the intent of the computation. Specifically, the Boolean circuits $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ have now input values $\perp$, false, and true and are evaluated under Kleene's semantics given in that figure.

x	$\neg x$
false	true
$\perp$	$\perp$
true	false

x &&& y		y = false	y = $\perp$	y = true
x	false	false	false	false
	$\perp$	false	$\perp$	$\perp$
	true	false	$\perp$	true

$x \mid\mid y$		y = false	y = $\perp$	y = true
x	false	false	$\perp$	true
	$\perp$	$\perp$	$\perp$	true
	true	true	true	true

Figure 20: Kleene's strong 3-valued propositional logic

Recall that any policy pol is equivalent to its `join` normal form

$$(\texttt{grant if } \mathsf{GoC}(pol)) \texttt{ join } (\texttt{deny if } \mathsf{DoC}(pol))$$

Intuitively, we mean to say that if the condition for granting has truth value $\bot$, the value of that rule should be `undef`. However, whenever the condition for denying has truth value $\bot$, we want that rule to `deny`. This ensures that an attacker who withholds information can only decrease the policy decision in the truth ordering of the Belnap bilattice. We formalize this:

Definition 2 (Partial evaluation of Boolean circuits)**.** *Let ρ be a partial model, which maps attributes either to legitimate values or to $\bot$. We may evaluate pol through this normal form and Kleene's strong 3-valued semantics as follows:*

1. *Use ρ to evaluate all atomic conditions in pol to a truth value* true, false *or $\bot$.*
2. *Evaluate* $\mathsf{GoC}(pol)$ *with Kleene's semantics to a truth value v_{GoC}, which is* true, false *or $\bot$.*
3. *Evaluate* $\mathsf{DoC}(pol)$ *with Kleene's semantics to a truth value v_{DoC}, which is* true, false *or $\bot$.*
4. *If v_{GoC} equals $\bot$, then we change the value of v_{GoC} to* false*.*
5. *If v_{DoC} equals $\bot$, then we change the value of v_{DoC} to* true*.*
6. *With these possibly overwritten values of v_{GoC} and v_{DoC}, we compute the policy result as follows:*
   - *if both v_{GoC} and v_{DoC} are* true*, return* `conflict`*,*
   - *if both v_{GoC} and v_{DoC} are* false*, return* `undef`*,*
   - *if v_{GoC} is* true *and v_{DoC} is* false*, return* `grant`*,*
   - *otherwise, return* `deny`*.*

Example 9. *Consider "truth-ordering" negation policies Q, Q' applied to rules P, P':*

$$Q = \texttt{case}\ \{\ [P\ \texttt{eval grant: deny}]\ [P\ \texttt{eval deny: grant}]\ [\texttt{true: } P]\ \}$$

$$Q' = \texttt{case}\ \{\ [P'\ \texttt{eval grant: deny}]\ [P'\ \texttt{eval deny: grant}]\ [\texttt{true: } P']\ \}$$

$$P = \texttt{grant if } cond \qquad P' = \texttt{deny if } cond$$

wherefore it holds $\mathsf{GoC}(Q)$ *is logically equivalent to* false *and* $\mathsf{DoC}(Q)$ *logically equivalent to $cond$. Moreover, we have the equivalences* $\mathsf{GoC}(Q') \equiv cond$ *and* $\mathsf{DoC}(Q') \equiv$ false*, which in turn means that Q is equivalent to P' and Q' to P.*

- *If $\rho(cond)$ equals $\bot$, then for Q' value v_{GoC} equals $\bot$ and gets overwritten into* false*. The value v_{DoC} is* false *and so the overall result is* `undef` *as intended.*
- *Similarly for Q with $\rho(cond) = \bot$, value v_{GoC} is* false *and v_{DoC} is $\bot$ and gets overwritten to* true*. Therefore, the computed result is* `deny` *as intended.*

Let ρ be the partial model that is being evaluated. Then we want re-assurance that the evaluation under ρ' with $\rho' \trianglelefteq \rho$ does not allow for an undesired change of the result, e.g. from a `deny` to a `grant`. Here $\rho' \trianglelefteq \rho$ means that $\rho'(t) \neq \bot$ implies $\rho'(t) = \rho(t)$ for all attributes t.

Similarly, we may want a certain kind of monotonicity, meaning that the result computed for any ρ'' with $\rho \trianglelefteq \rho''$ is monotonically above the result computed for ρ. For example, for monotonicity with respect to the truth ordering of the Belnap bilattice we would say that if the result is `grant` under ρ, then we may expect that the result is also `grant` for all ρ'' with $\rho \trianglelefteq \rho''$.

For the above method of Definition 2 that appeals to the `join` normal form we can indeed prove such a theorem. Consider the table in Figure 21. This shows that the above method computes a decision that is the least (i.e. the order-theoretic meet) in the truth ordering of the

GoC(pol)	DoC(pol)	computed decision	possible decisions
$\bot$	$\bot$	deny	deny, conflict, undef, grant
$\bot$	true	deny	deny, conflict
$\bot$	false	undef	undef, grant
true	$\bot$	conflict	conflict, grant
false	$\bot$	deny	deny, undef

Figure 21: Outputs computed as in Definition 2 of the two Boolean circuits where at least one output is $\bot$ (third column); what decisions are possible over complete models that independently refine the $\bot$ outputs into true or false (fourth column); in all rows, computed decisions are the meet in the truth ordering of the Belnap bilattice over all possible decisions.

Belnap bilattice amongst those decisions that could in principle be computed if any $\bot$ outputs of these two Boolean circuits were to be refined into true or false.

This confirms that this method is conservative in regard to refining incomplete models. Note that this also integrates well with having a top-level wrapper that converts any conflict or undef into deny – since the computed decision is never grant in situations in which at least one Boolean circuit outputs $\bot$ – as seen in Figure 21.

Let us now turn to the other issue, that an adversary may withhold information in order to influence the policy evaluation to his advantage. Consider that ρ is the partial model under which pol would be evaluated if the adversary did not have an opportunity to withhold information. Note that ρ may be a maximal model with respect to $\trianglelefteq$ or may also have $\bot$ values, e.g. due to non-adversarial system faults. An adversary who then withholds information that is present in ρ constructs a partial model ρ' with $\rho' \trianglelefteq \rho$ such that pol gets evaluated under ρ' and not under ρ. We mean to show that our method ensures that this can only decrease the decision in the truth ordering of the Belnap bilattice.

Let us write $\trianglelefteq$ for the information ordering over the 3-valued logic as well, where $\bot \trianglelefteq$ true and $\bot \trianglelefteq$ false, and true and false are maximal elements in that partial order. It is easy to see that $\rho' \trianglelefteq \rho$ implies that $\rho'(cond) \trianglelefteq \rho(cond)$ for all formulas of propositional logic $cond$. In particular, if $\rho'(\mathsf{GoC}(pol))$ and $\rho'(\mathsf{DoC}(pol))$ are different from $\bot$, then $\rho(\mathsf{GoC}(pol))$ equals $\rho'(\mathsf{GoC}(pol))$ and $\rho(\mathsf{DoC}(pol))$ equals $\rho'(\mathsf{DoC}(pol))$, thus ρ' and ρ compute the same decision of policy pol.

Otherwise, at least one of $\rho'(\mathsf{GoC}(pol))$ and $\rho'(\mathsf{DoC}(pol))$ equals $\bot$ and then the computed decision for pol under ρ' is as specified in Figure 21. In particular, if ρ' then wants to realize a decision other than deny, it can only be one of the following:

- undef in the third row: in that case ρ either computes the same results for the circuits (and so no attack occurs), ρ refines the sole $\bot$ to true (so the adversary turns a grant into an undef) or refines it to false (so the adversary cannot change the decision of undef at all).
- conflict in the fourth row: in that case ρ either computes the same results for the circuits (so no attack occurs), ρ refines $\bot$ to true (so the adversary cannot change the decision of conflict) or refines it to false (so the adversary would change a grant into a conflict).

This suggests that, apart from being able to realize a request denial, an adversary has no ability of manipulating policy outcomes to his advantage by withholding attribute information: even though a change from grant to conflict might be a concern within a composition context, e.g., policies are evaluated as composites in the PDP and so this seems to be a non-issue. Note that a deny-by-default wrapper for the table in Figure 21 would deny whenever one of the circuits outputs $\bot$. The next example shows our approach has indeed benefits:

Example 10. *Consider two atomic conditions c_1 and c_2 and the policy P defined as the join of*

two granting rules: $(\texttt{grant if } c_1)$ $\texttt{join}$ $(\texttt{grant if } c_2)$. *Then* $\mathsf{GoC}(P)$ *is logically equivalent to* $c_1 \mid\mid c_2$. *Let* ρ *be a partial model with* $\rho(c_1) = \bot$ *and* $\rho(c_2) = \mathsf{true}$. *An approach that denies on partial models would here return* `deny`. *In contrast, our approach does return* `grant`*:*

- *The evaluation of* $\mathsf{GoC}(P)$ *on* ρ *renders* $\bot \mid\mid \mathsf{true} = \mathsf{true}$.
- *Since* $\mathsf{DoC}(P)$ *is logically equivalent to* false*, its evaluation returns* false *on* ρ.

The combined decision is therefore `grant` *in our approach.*

Attributes whose values are unknown then give rise to such $\bot$ inputs in the circuit. If the circuit still evaluates to true or false, this uncertainty did then not matter and so the PDP may accept the result of the circuit regardless of the fact that not all attributes have known values.

We follow the above approach also in the computation of Obligation circuits, with similar resiliency to adversarial manipulation that may withhold information. In particular, the computation of obligations follows the specification in Figure 13, but the conditionals for rules are extended to reflect the 3-valued truth semantics, that $\rho(cond)$ be an element in $\{\mathsf{true}, \mathsf{false}, \bot\}$. Also, the truth value of guards pol `eval` dec is determined by computing the decision for pol as under Definition 2, and then checking for equality of that result with dec in the usual 2-valued meaning of equations.

In the example below, let us write $cond{\downarrow}$ and $cond{\uparrow}$ to denote the conversion of truth value $\bot$ of a condition $cond$ to false and true, respectively, where other truth values are unchanged.

Example 11. *Consider again* Q' *with* $\rho(cond) = \bot$. *Then the obligations for granting and denying of* Q' *compute to*

$$\mathsf{GObl}(Q') = \begin{cases} \mathsf{GObl}(P') & \textit{if } \mathsf{false} \\ \{\,\} & \textit{if } cond{\uparrow} \\ \mathsf{GObl}(P') & \textit{if } \neg(cond{\uparrow}) \end{cases} \qquad \mathsf{DObl}(Q') = \begin{cases} \{\,\} & \textit{if } \mathsf{false} \\ \mathsf{DObl}(P') & \textit{if } cond{\uparrow} \\ \mathsf{DObl}(P') & \textit{if } \neg(cond{\uparrow}) \end{cases}$$

and $\mathsf{GObl}(P') = \{\,\}$ *which reflects that the only source of obligations is the* `deny`*-rule* P'.

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 4. 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 [34]. 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].

Security Analysis for the FROST Delegation Protocol All considerations below assume that the private keys of the respective parties have not been exposed, otherwise complete impersonation is possible. Also it is assumed that communication between two parties happens over a secure communication channel providing confidentiality and integrity of the messages sent therein, which in turn includes the assumptions and requirements for the chosen secure communication channel protocol.

Attack Model The delegatee's public keys are known to the respective delegators and each delegator knows the resource's public key. Communication of participants in the delegation process happens inside a secure communication channel providing confidentiality and integrity of the communication established by an appropriate protocol. An adversary may have significant computational resources and is able to capture, delete and replay messages sent over the secure communication channel. But she cannot undetectably create or modify messages therein and cannot learn anything about the content of the messages except for their length.

Authentication During each delegation step authentication is performed between the current pair of delegator and delegatee. The delegator, who may be authenticated, acts as the client and the delegatee, who must be authenticated, acts as the server in authentication protocol terms, which prevents man-in-the-middle attacks. Successful authentication establishes a secure communication channel which provides confidentiality and integrity of the messages sent therein.

This prevents adversaries to create or modify messages without detection, and it prevents replay attacks or the ability to learn anything about the messages except for their length. Deletion of messages may lead to denial-of-service attacks.

Delegatees might attempt to authenticate their delegators if they want to assure that they are communicating with a trusted entity. Whether or not the delegator is authenticated by the delegatee, if the delegatee knows about the public keys of the previous delegators, then the delegatee has the cryptographic guarantee via signatures that each delegator indeed authenticated their delegatees before.

Once a delegation chain is submitted to the resource, the resource authenticates the resource owner as the initial delegator. That turns the authentication chain into an authentication cycle and implicitly establishes authentication of each delegator by its respective delegatee.

Delegation Subchains Each delegator knows the public key of the resource and is able to authenticate the resource. Hence, each delegator can act as the final delegatee by submitting her delegation subchain to the resource. This prevents denial-of-service attacks by delegatees amid the delegation chain, e.g. caused by inactive delegatees or by delegatees being under denial-of-services attacks themselves.

Every delegator can change their submitted policy via direct communication with the resource. That allows fixing the intended behavior of policies in case of discovered vulnerabilities as well as completely altering the behavior of policies in case of new requirements. Also, each delegator can revoke given delegation rights via direct communication with the resource, which allows to contain malicious behavior from subsequent delegatees. In particular, there are no dependencies on previous delegators or subsequent delegatees for policy changes and therefore no additional denial-of-service attack vectors for this.

The encrypt-then-mac approach for policy privacy in delegation subchains provides authenticated encryption against all subsequent delegatees. Since the resource decrypts the delegation chain, the resource owner can learn about the encrypted content as well.

Verification Due to the verification of all signatures of the delegation chain by the resource and the corresponding authentication chain, only authorized policy chains get stored in the policy trees of the resource. Tampering will be detected by the cryptographic means provided by the secure communication channel.

The verification of subchains by delegatees is optional, since delegatees can only further extend the delegation chain, but they are not able to execute policies themselves. It assures

delegatees of their given delegation rights, but it doesn't enable delegatees to alter these in any non-detectable way.

The inclusion of sender and recipient in the, possibly encrypted, signed delegation messages prevents any false attribution of delegators and delegatees.

Policy Composition Policies are only as reliable as the author who is writing them. Delegation rights should only be given to delegatees who are trustworthy and who are trusted to only further delegate to trustworthy delegatees. Any impact of malicious policies from delegatees should be hampered or prevented by conservatively written delegator's policies and delegation rights may be revoked.

These threats can also be mitigated against by choosing a policy idiom for composition of policies from delegatees such that static controls are assured: e.g. that a denial of a resource owner not be turned into an overall grant.

Any unintended but formally correct behavior of composed policies, which may e.g. be caused by the liberal choice of delegation parameters or misconceived interpretations of the FROST language, should be addressed by proper documentation and policy-analysis tools.

Missing-information attacks to gain advantages by withholding information are prevented by monotonicity properties of the FROST language on handling missing information. Privilege-escalation attacks to obtain unexpected access rights are substantially dependent on delegator's policies as well as the choice of composition types and idioms, hence they should be written conservatively and checked by policy analysis tools.

7 Use Cases

To illustrate the value proposition of the FROST language, including its usage within a network its possible support of a token, we present three use cases and the services and products that they can enable.

7.1 Automotive

Together with Porsche, XAIN AG 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 found in Section 6 of [36] and in this short video:

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

This Yellow Paper describes the next and significant step in the development of such technology and capabilities, with applications within the Automotive sector and well beyond. Car digitalization will create economic value for car manufacturers, insurers, passengers and drivers, infrastructure providers, and other third parties. R&D in this space has huge momentum, meaning that commercial use cases will be integrated into production within the next 3-7 years. This momentum has huge commercial potential, considering that revenue increases in Automotive Cybersecurity alone are predicted to be dramatic [21].

7.1.1 The Car as a Delivery Box

The utilization of a car's trunk as a package delivery point was demonstrated in the PoC built for Porsche by XAIN. The vehicle's owner creates a policy stating that the delivery service company has one-time access to the vehicle's trunk within a predefined time frame. The policy is broadcast to the network and subsequently stored on the related vehicle. The necessary data

can be distributed to the responsible parcel carrier, who approaches the vehicle and requests access to the car.

If the embedded client within the vehicle proves the policy as valid, the function gets executed, the trunk opens, the transaction is logged, and the policy gets updated to prevent the parcel carrier from accessing the car a second time.

This access feature offers significant advantages to the vehicle owner since he no longer has to rely on vague, time-consuming delivery times that may not even be fulfilled. This can potentially result in a substantial increase in delivery efficiency for the logistics company that provides this service. It can feed its route-planning optimization tools with constraints stemming from FROST policies of customers, to integrate this feature in its existing cost and time-saving tools.

The FROST technology can support payments for the delivery service and enables new, more customer-friendly business models. One possible scenario is the transaction of the postage fee upon arrival of the package in the vehicle's trunk.

Furthermore, this functionality does not only allow for package delivery but also for more sophisticated use cases such as using the vehicle as a cloth exchange point with a local launderette.

7.1.2 Flexible Car Sharing Networks

FROST and its embedded client within vehicles allows for the seamless on and off boarding of an individual's vehicle to a car-sharing network. The vehicle owner creates a policy stating the time, cost, and area of usage as well as other attributes such as the required minimum rating of the requester. This would reduce the demand for parking spots in urban spaces as privately owned cars would be used more frequently, while simultaneously lowering the cost of vehicle ownership. Other parties can use this on/off-boarding feature, e.g., repair shops which pick up and drop off cars for seamless maintenance works as an extra service.

7.1.3 Telematics Insurance

Currently, insurance companies have only crude instruments at their hands to determine actual usage time of a vehicle as well as the driving style of an individual. This situation often results in unfair pricing for safety-conscious drivers and an unfair advantage for more precarious drivers, since the risk has to be covered by median pricing distributed over all insurance takers. The idea of telematics insurance aims to change that fundamentally. With an embedded client sitting within a vehicle, the transmission of real and unaltered data becomes a reality. This data can then be used to detect actual movement of the vehicle and create driving profiles connected to a user. With FROST, the owner of the vehicle stays in control of such data and would only release access to it or its statistical properties if incentivized to do so, e.g., to get a fairer insurance price.

7.2 Access to IoT-Devices

The economic and social value of IoT will in part be realized through paid services that provide access to data, data-driven insights or physical resources – in FROST, the resource owners are in ultimate control of their resources and the data they generate.

These value-creation activities benefit from the ability of FROST to control device access and functionalities, as well as the unprecedented ability to tie payments with the FROST token stand to be the technical enabler for a new ecosystem. We illustrate this with two brief use cases.

7.2.1 Shared Parcel Boxes

At the moment, many delivery companies host and maintain their network of distributed parcel boxes across a city. Since every delivery service is building their infrastructure, this results in additional costs as well as volatile availability and distance to pick-up points for end consumers. These issues could be addressed effectively provided that current players were to open up their empty delivery boxes for competitors. Using the FROST language, network, and token, companies may ensure that all the service-required data fields are shared with an infrastructure provider while also being compliant with privacy regulations such as GDPR. The FROST token can be used as the native currency by logistics companies to buy access rights for available parcel boxes in real-time. This sharing service will also decrease delivery times of packages, as it allows for a much denser distribution of parcel boxes amongst urban areas.

7.2.2 Smart Door Locks

FROST may also be applied to modern households or commercial property. Anyone who ever rented or let a room, apartment or house (e.g. by using Airbnb) had to deal with the key transfer, which is a hassle at best and misery at worst. Even pin-protected locks that store keys are subject to frustration, abuse and compromise.

Combining the FROST language, token, and network with a Smart Door lock will make such challenges a thing of the past, while still allowing for personal interactions of owners and users of properties if desired. The owner could create a new or choose an existing policy that details the tenant agreement (including the predefined time frame, payment schedule, minimum required reputation of a tenant). Using the FROST language, the owner can gain sufficient transparency of property usage while respecting individual privacy; e.g. the policy may limit the number of guests that can receive initial access to the property within the period of the rental agreement. Such controls cannot be exercised with physical keys.

7.3 Big-Data Marketplaces

Over the past decades, it has become more and more evident that enterprises have a general interest in sharing or selling their data to other parties but refrain from doing so due to insufficient trust. This lack of trust stems in part from the fact that any data-sharing platform would either be developed and maintained by one of the participating parties or by a third party – which then would have to be trusted.

In both scenarios, enterprises would have to trust in the integrity of another entity. In contrast, the FROST language, network, and the token will sit on top of a decentralized big-data platform to control access to data and associated resources, governed by the XAIN Foundation. FROST is consistent with having other technology run alongside it. For example, an AI framework may sit next to it and run distributed machine learning over the big-data platform, e.g. to facilitate learning across organizational boundaries in a privacy-preserving manner. FROST can articulate and enforce the controls for data access, data viewing rights as well as knowledge sharing across the big-data and AI platforms, while also generating trustworthy and immutable audit trails hosted on a bespoke blockchain.

This general use case of a big-data marketplace is currently being developed and implemented by XAIN AG for a variety of concrete business cases and within different industry verticals. Whilst the underlying business models differ in detail, they share the idea of giving enterprises the opportunity to leverage the value of gathered data. Such data is either connected to an individual or a resource of a company, but the technology for regulating access to data is the same for both types of agents.

Many enterprises have an interest in selling parts of their data to the highest bidder or another company of their choice, and they may want to buy data from other enterprises for

many different reasons. The use of the FROST language, token, and blockchain network in conjunction with the Big-Data Full Node Architecture of XAIN AG can instill the missing trust that prevent this exchange. With our approach, all data stays with its owner, who can create policies that regulate and monetize read access to specific data sets and data fields. The currency of choice for this value transfer is the FROST token.

7.3.1 Predictive Maintenance

At present, substantial amounts of data are being gathered from IoT devices end-points. The value residing in these data sets seem significant, yet can only be leveraged by having seamless interoperability between data generation, aggregation, and analysis engines. While the entire IoT market is expected to grow to a multi-trillion dollar market by 2025, experts predict that enabling interoperability is the key to unlock up to 40% of the entire market [37].

Together with one of the largest industrial manufacturing company in Europe and one of the largest railway operator and infrastructure owner in Europe, XAIN AG is currently building a Big-Data Marketplace that provides such interoperability to enable use cases for seamless predictive maintenance. The first use case will be built around point machines, devices that operate railway turnouts. These devices are responsible for the mechanical execution of a switch change and are operated out of centralized command centers sometimes hundreds of kilometers away from the switches.

While the railway owner is in charge of device operation, device production and maintenance is controlled by an industrial manufacturer. At present, the railway company has to manually extract and share specific data fields about the use of these devices with third parties. This exercise is costly, error-prone and is also subject to potential unintentional or malicious data theft or manipulation. However, connecting third parties directly to such data fields with conventional database APIs poses particular security risks.

The FROST language, token, and network provides fine-grained data access control that addresses such risks, increases automation and so cuts cost, and integrates well with the Big-Data architecture of XAIN AG. Therefore, using FROST reduces risks and delay-dependent costs of unexpected hardware failures, human error, and adversarial manipulation and has the potential to increase overall customer satisfaction.

7.3.2 Milestone-Informed Payment Streaming

For large manufacturing contracts in the aviation and railway industry, the time frame of a single order may span three to ten years, leaving payments released in multiple installments. When certain quality gates and advancements are fulfilled by the manufacturing company, and when an auditor representing the client authenticates approves their milestones, payments are released. The entire payment is usually split into 3 to 4 tranches over the entire life-span of the project. This results in enormous, sometimes business-threatening prepayments on the manufacturer's side, and in significant capital commitment costs on the customer's side, given that these contracts usually have multi-billion dollar volumes.

To overcome these payment issues in large and complex projects, the FROST language can be used to grant the client real-time access to selected and unaltered data sets – enabling a revolution in payment solutions for the manufacturing sector. Payment installments no longer have to cover the progress of multiple years but rather can turn into payment streaming coupled to the actual production progress, thus considerably de-risking payment schedules for both clients and manufacturers. XAIN AG is presently developing such a solution with a mobility manufacturing company that operates globally.

7.3.3 Training Access for Federated Machine Learning

FROST will also enable use cases in the area of distributed machine learning and makes thus a huge step towards the eXpandable AI Network, the strategic vision behind XAIN AG – as shown in Figure 22. *Federated Learning* is a recent development which enables training of advanced AI models without actually being able to read the data. An example of this is companies who want to train an AI model to predict if an invoice they received is fraudulent or not. A model, which was trained on just the invoice data from a sole company, is obviously inferior to a model which was trained on the invoice data of many companies.

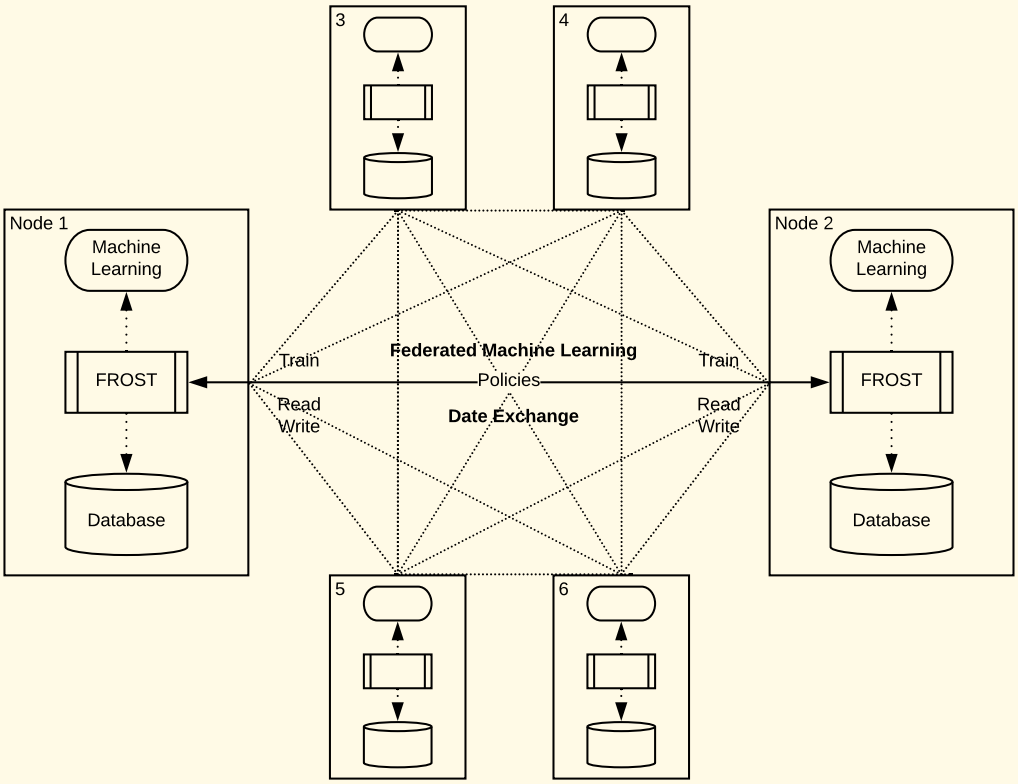


Figure 22: Architecture overview of Federated Machine Learning approach using FROST

Understandably, no company would grant read-access to all their invoices to another company. On the surface, training a model that contains insights from all invoices thus seems impossible. Fortunately, we may use FROST and Federated Learning to train a model without requiring read-access to the underlying datasets. Moreover, FROST can enable this access: a new access modality “train access” would refine the familiar “read” access modality. Its intuitive meaning would be that the data owner *“would not let anyone read this data, but that training on this data would be OK as long as the data owner gets benefits from such training”*.

This overall approach can lead to previously unthinkable business cases. For example, a company might sell training access without having to disclose any data. Or several companies might collaborate and train a model, with clear benefits to all of them. Overall, these possibilities resonate strongly with the mission of XAIN AG, to enable meaningful collaboration via an eXpandable AI Network.

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. In fact, these actors will benefit from technology that helps them with the creation of their own bespoke ecosystem for exercising and managing such controls.

These challenges, needs, and opportunities 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 first shape such ecosystems and to then 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 extended 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, and in fact creating, 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] CULVER, M. Automotive Cybersecurity Market to Reach $759 Million in Revenue in 2023, IHS Markit Reports. online, 26 September 2016. URL: `https://news.ihsmarkit.com/press-release/automotive-cybersecurity-market-reach-759-million-revenue-2023-ihs-markit-reports`.

[22] 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.

[23] DEBRAY, S. K., EVANS, W. S., MUTH, R., AND SUTTER, B. D. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst. 22*, 2 (2000), 378–415.

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

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

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

[27] 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.

[28] 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.

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

[30] HUDAK, P., PEYTON JONES, S., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J., GUZMÁN, M. M., HAMMOND, K., HUGHES, J., JOHNSSON, T., ET AL. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices 27*, 5 (1992), 1–164.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] LUNDBÆK, L.-N., JANES BEUTEL, D., HUTH, M., JACKSON, S., KIRK, L., AND STEINER, R. Proof of Kernel Work: a democratic low-energy consensus for distributed access-control protocols. *Royal Society Open Science 5*, 8 (2018).

[37] MANYIKA, J., CHUI, M., BISSON, P., WOETZEL, J., DOBBS, R., BUGHIN, J., AND AHARON, D. Unlocking the potential of the Internet of Things, McKinsey Report, published online, June 2015. URI: `https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world`.

[38] MOGGI, E. Notions of computation and monads. *Information and computation 93*, 1 (1991), 55–92.

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

[40] 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.

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

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

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

[44] RISSANEN, E. eXtensible Access Control Markup Language (XACML). OASIS Standard, 23 January 2013. URI: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.doc.

[45] 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.

[46] 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. 20-22, 1994, IEEE Computer Society Press* (1995), 124–134.

[47] 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.

[48] 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.

[49] SWINFEN-GREEN, J. Data is the new oil. Online article published in the Business Reporter (distributed with The Daily Telegraph), 27 February 2017. URL: https://www.business-reporter.co.uk/2018/02/27/data-is-the-new-oil/#gsc.tab=0.

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

[51] WADLER, P. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (1990), ACM, pp. 61–78.

A Post-quantum Cryptography Security Review

Every cryptographic scheme that is believed (and not known) to be 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] with respect to *average-case* complexity – a notion of complexity suitable for assessing cryptographic strength.

The security of cryptographic schemes is measured with respect to the number of bits of input or output values and quantifies 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 such computational complexity is relative to the computer system used, e.g. 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 time 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 problem (DLP) over finite fields. There exist sub-exponential algorithms that are closely related to the algorithms for integer factorization. Therefore, the security levels asymmetric ciphers based on the discrete logarithm are 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 problem 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 [29] and Shor's [46] 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, the time complexity can be reduced to $\mathcal{O}(N^{1/3})$). 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. This is an *exponential* speedup resulting in a polynomial time quantum algorithm and in terms of n-bit integers 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 [45]. But once these algorithims are fully deployable, 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 the same security levels, the computational efficiency for the 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, 41, 47]:

- **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 [47]. 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 [24]. 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) [31] 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 [26]. 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 [25], 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. 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 [48].

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, such as McEliece code based encryption schemes, or otherwise should be updated to post-quantum secure schemes as soon as they are standardized, such as 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 such as 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 such as classical elliptic curve Diffie-Hellman key exchange protocols.

B FROST Language as a Deep Embedding

As noted in Section 3.5, the compositional structure of FROST policies makes a strong case for its embedding within a typed functional programming language, which has benefits such as function composition and strong static typing guarantees as core features of the paradigm. We sketch some of the details of such an embedding here. While the conceptual ideas are what we wish to emphasise, we nevertheless adopt the syntax of Haskell [30] from hereon as our host language, if only to keep the discussion of a FROST embedding concrete.

B.1 Staged Compilation

A domain-specific language typically comes in one of two forms – either as a completely standalone language, or embedded within another often more general-purpose language. There is a further dichotomy for embedded DSLs (or EDSLs), for they may be deemed shallow or deep. That is, terms of the EDSL may be interpreted directly as values in a semantic domain (shallow), or simply as abstract syntax trees for presumed further stages of interpretation. Thus a deep EDSL is a natural fit for FROST, since it offers the possibility of *staged compilation*. Before giving such a compilation into an intermediate language, let us define the language terms of the deep embedding, which makes judicious use of *algebraic data types* (ADTs).

B.1.1 Abstract Syntax of Conditions and Policies

The grammar for terms and conditions in FROST, as depicted in Figure 2, can be modelled as the following ADTs defined with the **data** keyword. The abstract syntax of a FROST term is constructed as a value of type $Term$, which is either a named entity, a named attributed term, or a constant keyword (a value of the $Const$ type).

$$\textbf{data}\ Const = Subj \mid Obj \mid Act$$
$$\textbf{data}\ Term = Entity\ String \mid Attr\ Term\ String \mid Keyword\ Const$$

To keep the presentation simple, we exclude additional constants and composite terms (given by operations on subterms) here, and we also accept the possibility of constructing terms that are not well-typed – though both can be addressed with our approach.

A condition, expressed as a value of type $Cond$, is a propositional logic where $BinRel$ conditions – representing relations over terms – serve as the atomic expressions. For sake of simplicity of presentation, we only consider binary relations as such atoms. The predicate symbols in such relations are defined by $BinPred$.

$$\begin{aligned}&\textbf{data}\ BinPred = Equ \mid Lt \mid Lte \mid ...\\ &\textbf{data}\ Cond = BinRel\ BinPred\ Term\ Term \mid\\ &\qquad\qquad T \mid Not\ Cond \mid And\ Cond\ Cond \mid Or\ Cond\ Cond\end{aligned}$$

Of course, from this core propositional logic over terms, one can further derive other propositional conditions easily in the host language, e.g. the simplest of which we could define as

$$false = Not\ T$$

As we will see later, conditions form an intermediate language in our staged compilation process, but they are also part of the syntax of rules. Such rules are just one form of FROST policy, which we model with the type Pol as follows.

$$\begin{aligned}&\textbf{data}\ Dec = Grant \mid Deny \mid Gap \mid Conflict\\ &\textbf{data}\ Rule = GrantIf\ Cond \mid DenyIf\ Cond\\ &\textbf{data}\ Guard = Truth \mid Eval\ Pol\ Dec \mid Conj\ Guard\ Guard\\ &\textbf{data}\ Pol = Konst\ Dec \mid\\ &\qquad\qquad Filter\ Rule \mid\\ &\qquad\qquad Case\ [(Guard, Pol)]\ Pol\end{aligned}$$

As well as $Filter$ policies that comprise a $Rule$, policies can be categorised as "constant" $Konst$. These are simply wrappers around the values of 4-valued Belnap logic captured in Dec. The third form of policy is a $Case$, defined in terms of guards modelled by the type $Guard$ and themselves taking one of three forms. A $Case$ policy thus comprises two parts; a (possibly empty)

list $[\ldots]$ of $(Guard, Pol)$ pairs, each representing an arm of the policy, followed by another Pol representing the last *default* arm. To explain why the last arm is just this and not another $(Guard, Pol)$ pair, note that the guard $Truth$ is there merely as a matter of convenience, representing the concrete guard `true` of the last arm. Since it is always this value, it is omitted in our abstract syntax. Again to keep the presentation simple, we ignore policy obligations until later.

B.1.2 Compilation to Circuits

In the above, we defined a deep embedding of FROST, an important design choice that offers the flexibility of multi-staged compilation of policies. This is of particular significance in accommodating a compilation of policies into the intermediate language of circuit-pairs (Section 3), facilitating further stages of optimisation and verification. In the following, we give details of policy interpreters targeting this intermediate language.

Any FROST policy can be represented equivalently in join normal form as in (2), consisting of a grant-or-conflict (goc) circuit and deny-or-conflict (doc) circuit specified in our language of conditions $Cond$. The goc-doc circuits of $Konst$ and $Filter$ policies are given as follows, mirroring the definition specified in Figure 5.

$$
\begin{array}{ll}
goc :: Pol \to Cond & \\
goc\ (Konst\ Grant) & = T \\
goc\ (Konst\ Conflict) & = T \\
goc\ (Konst\ \_) & = false \\
goc\ (Filter\ (GrantIf\ cond)) & = cond \\
goc\ (Filter\ (DenyIf\ \_)) & = false
\end{array}
\qquad
\begin{array}{ll}
doc :: Pol \to Cond & \\
doc\ (Konst\ Deny) & = T \\
doc\ (Konst\ Conflict) & = T \\
doc\ (Konst\ \_) & = false \\
doc\ (Filter\ (GrantIf\ \_)) & = false \\
doc\ (Filter\ (DenyIf\ cond)) & = cond
\end{array}
$$

For $Case$ policies, we single out the "arm-less" special case – its circuit follows from the equational law $\texttt{case}\,\{\,[\texttt{true}\colon p]\,\} = p$. Otherwise, circuit generation is given by a helper function $compCase$, discussed in some detail shortly.

$$
\begin{array}{ll}
goc\ (Case\ [\,]\ defPol) & = goc\ defPol \\
goc\ (Case\ arms\ defPol) & = compCase\ True\ arms\ defPol \\
doc\ (Case\ [\,]\ defPol) & = doc\ defPol \\
doc\ (Case\ arms\ defPol) & = compCase\ False\ arms\ defPol
\end{array}
$$

We also define a function t specifying the condition for when a guard holds. It is equivalent to the map $\mathsf{T}(guard)$ in Figure 6.

$$
\begin{array}{ll}
t :: Guard \to Cond & \\
t\ Truth & = T \\
t\ (Eval\ pol\ Conflict) & = goc\ pol\ `And`\ doc\ pol \\
t\ (Eval\ pol\ Gap) & = Not\ (goc\ pol)\ `And`\ Not\ (doc\ pol) \\
t\ (Eval\ pol\ Grant) & = goc\ pol\ `And`\ Not\ (doc\ pol) \\
t\ (Eval\ pol\ Deny) & = Not\ (goc\ pol)\ `And`\ doc\ pol \\
t\ (Conj\ g_1\ g_2) & = t\ g_1\ `And`\ t\ g_2
\end{array}
$$

In defining the $compCase$ helper, we will find it helpful to use a library function $inits$ where $inits\ xs$ returns all initial segments of the list xs, shortest first[1]. To compute a circuit for $Case\ arms\ defPol$, it helps to operate on the initial segments of $arms$, which is $armInits$ in the following (note the use of $tail$, which removes the first element of a list and thus removes the useless empty segment $[\,]$).

[1] e.g. $inits\ [1,2,3]$ evaluates to $[[\,],[1],[1,2],[1,2,3]]$. In Haskell, use of the $inits$ function requires an appropriate declaration: **import** $Data.List\ (inits)$

$$
\begin{array}{l}
compCase :: Bool \to [(Guard, Pol)] \to Pol \to Cond \\
compCase\ isGoc\ arms\ defPol = \\
\quad foldr\ (Or \circ disjunct\ isGoc)\ (lastDisjunct\ isGoc\ guards\ defPol)\ armInits \\
\quad \textbf{where} \\
\quad\quad armInits = tail\ (inits\ arms) \\
\quad\quad guards\ \ \ = map\ fst\ arms
\end{array}
$$

Each segment contains enough data to determine the corresponding $disjunct$ in the resulting Or-disjunction. In particular, the $lastDisjunct$ is a special case requiring all the guards of the maximal segment (i.e. $arms$ itself), obtained by a map of the fst projection over each arm in turn. All disjuncts are combined with a $foldr$[2], where the combining operation is a composition of functions: conversion of a segment into a $disjunct$ (the flag $isGoc$ just indicates whether the circuit to be produced is the goc or doc variety), followed by Or.

A $disjunct$ is itself an And-conjunction, as in the following

$$
\begin{array}{l}
disjunct :: Bool \to [(Guard, Pol)] \to Cond \\
disjunct\ b\ arms = foldr\ (And \circ Not \circ t \circ fst)\ (t\ trueGuard)\ pairs \\
\quad `And`\ compPol\ pol \\
\quad \textbf{where} \\
\quad\quad (pairs, [(trueGuard, pol)]) = splitAt\ (length\ arms - 1)\ arms \\
\quad\quad compPol = \textbf{if}\ b\ \textbf{then}\ goc\ \textbf{else}\ doc
\end{array}
$$

Key to this is the isolation of the last element-pair of the $arms$ segment, with the help of a function $splitAt$[3]. By "splitting" $arms$ in this way at length $arms-1$, we obtain simultaneously the $trueGuard$, its corresponding policy pol and all pairs before it (which contain negative guards). Notice that such a splitting always exists, since $arms$ is non-empty – this follows from the fact that the empty case $Case\ [\,]\ p$ is dealt with already either by goc or doc.

The conjunction is again constructed with a $foldr$ where the combining function is a composition that fst-projects guards out of the pairs, translates them into conditions with t, negates with Not and combines with And. The last two conjuncts are the (positive) $trueGuard$ and the circuit for its corresponding policy pol.

The $lastDisjunct$ follows a similar pattern:

$$
\begin{array}{l}
lastDisjunct :: Bool \to [Guard] \to Pol \to Cond \\
lastDisjunct\ b\ gs\ pol = foldr\ (And \circ Not \circ t)\ T\ gs \\
\quad `And`\ compPol\ pol \\
\quad \textbf{where} \\
\quad\quad compPol = \textbf{if}\ b\ \textbf{then}\ goc\ \textbf{else}\ doc
\end{array}
$$

except that the penultimate conjunct is simply T, and the last conjunct is the circuit for the given default policy pol.

B.2 Obligations as Effects of Policy Evaluation

Much of the power of the deep EDSL approach lies in its flexibility. To illustrate, we sketch some details of another functional interpreter able to evaluate a policy directly to a decision and its obligations. Such an interpreter would be useful, e.g., as a verification backend. Depending on the use case, the FROST language may then itself be seen as a kind of intermediate language.

[2] Recall that, given a binary operation $\star$ and a starting value e, $foldr\ (\star)\ e$ reduces a list using $\star$ from right to left i.e. it realises the map $[x_1, x_2, \ldots, x_n] \longmapsto x_1 \star (x_2 \star \ldots (x_n \star e) \ldots)$

[3] where $splitAt\ n\ xs$ returns a pair of lists (ys, zs) with ys the initial segment of xs of length n, and zs the rest.

First, let us extend the language of Section B.1 with obligations. To keep the presentation simple, let us assume a data type of obligations

$$\textbf{data}\ Oblg = ...$$

Then $Filter$ policies are augmented with a list of obligations:

$$\begin{aligned}\textbf{data}\ Pol = {} & Konst\ Dec \mid \\ & Filter\ Rule\ [Oblg] \mid \\ & Case\ [(Guard, Pol)]\ Pol\end{aligned}$$

Note that we have used the list data type $[Oblg]$ as a convenient representation for finite sets, though the distinction is not in any way crucial. Indeed, it may even be useful to retain information about the ordering or repetition of obligations.

Let us assume a type of environments that represent models ρ of our condition language,

$$\textbf{data}\ Env = ...$$

and comes equipped with an API function for looking up (integer) values of terms

$$lookup :: Term \to Env \to Integer$$

In practice of course, terms may have value of type other than $Integer$, but we make this assumption here to simplify our presentation and refrain from developing a more general typing of terms here – which is not the focus of the present discussion.

A semantics of conditions can now be encoded as the following function.

$$\begin{aligned}&evalC :: Cond \to Env \to Bool \\ &evalC\ T\ \_ &&= True \\ &evalC\ (Not\ c)\ \rho &&= \neg\ (evalC\ c\ \rho) \\ &evalC\ (And\ c_1\ c_2)\ \rho &&= evalC\ c_1\ \rho \wedge evalC\ c_2\ \rho \\ &evalC\ (Or\ c_1\ c_2)\ \rho &&= evalC\ c_1\ \rho \vee evalC\ c_2\ \rho \\ &evalC\ (BinRel\ Equ\ t_1\ t_2)\ \rho &&= lookup\ t_1\ \rho \equiv lookup\ t_2\ \rho \\ &evalC\ (BinRel\ Lt\ t_1\ t_2)\ \rho &&= lookup\ t_1\ \rho < lookup\ t_2\ \rho \\ &evalC\ (BinRel\ Lte\ t_1\ t_2)\ \rho &&= lookup\ t_1\ \rho \leqslant lookup\ t_2\ \rho\end{aligned}$$

With this, the computation of policy obligations can then be given rather straightforwardly as a translation of Figure 13 as a function $oblg :: Dec \to Pol \to Env \to [Oblg]$. For example,

$$\begin{aligned}&oblg\ Grant\ (Filter\ (GrantIf\ cond)\ obls)\ \rho \\ &\quad \mid evalC\ cond\ \rho = obls \\ &\quad \mid otherwise \quad = [\,]\end{aligned}$$

That is, $Grant$-obligations of a $GrantIf\ cond$ rule policy are those $obls$ associated with the policy if the condition $cond$ holds (by evaluating it with $evalC$) under the environment ρ, otherwise there are none. $Deny$-obligations of $DenyIf$ rules policies are defined in a similar way. In all other cases of $Filter$ policy, as with $Konst$ policies, no obligations are returned.

Computing obligations for a $Case$ policy is more interesting however, since it necessarily involves a traversal over each arm of the policy until one with a satisfying guard is found. The obligations are then those of the corresponding policy, together with those of the "subpolicies" in the guard. But to evaluate a guard $Eval\ p\ d$ in turn requires the ability to evaluate *policies* p into decisions. As such, computation of obligations is inherently intertwined with policy evaluation. Indeed, they can be considered *computational side-effects* of evaluation.

B.2.1 Writer Monad: a Short Interlude

Computational effects are well-studied in programming languages, with (computational) *monads* in particular being an important discovery as an abstraction pattern for such effects [38, 51]. In our host language, monads are sufficiently prominent to form a standard type class:

$$
\begin{array}{l}
\textbf{class}\ Monad\ m\ \textbf{where} \\
\quad return :: m\ a \\
\quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

Indeed, such is the pervasiveness of monadic code, our host language supports a convenient **do**-notation as syntactic sugar for $\gg\!=$. Specifically, the following two code blocks are equivalent:

$$
\begin{array}{ll}
\textbf{do}\ x \leftarrow xm & xm \gg\!= (\lambda x \to \\
\quad y \leftarrow ym & \quad ym \gg\!= (\lambda y \to \\
\quad f\ x\ y & \quad\quad f\ x\ y))
\end{array}
$$

By convention, all instances of the $Monad$ class are required to satisfy three equational laws, omitted here for sake of brevity[4]. One instance of particular interest here is the $Writer$ monad, which can be thought of as an arbitrary data type a together with a type o of accumulating "output".

$$\textbf{data}\ Writer\ o\ a = W\ (a, o)$$

The important point to note here is that $Writer\ o\ a$ is isomorphic to the product type (a, o) of pairs, which is clear if one ignores the W wrapper. While a is arbitrary, for $Writer\ o$ to form a monad, the type o is required to be an instance of the $Monoid$ class, consisting of an associative "append" operation $\oplus$ with unit $\emptyset$.

$$
\begin{array}{ll}
\textbf{class}\ Monoid\ o\ \textbf{where} & \textbf{instance}\ Monoid\ [a]\ \textbf{where} \\
\quad \emptyset\ \ :: o & \quad \emptyset\ \ = [\,] \\
\quad (\oplus) :: o \to o \to o & \quad (\oplus) = (+\!\!+)
\end{array}
$$

Shown beside the class declaration is the canonical instance of $Monoid$ – the type of lists, together with the empty list and append.

The $Monad$ instance definition of $Writer\ o$ is thus as follows:

$$
\begin{array}{l}
\textbf{instance}\ Monoid\ o \Rightarrow Monad\ (Writer\ o)\ \textbf{where} \\
\quad return\ x \quad\quad = W\ (x, \emptyset) \\
\quad W\ (x, v) \gg\!= f = \textbf{let}\ W\ (y, v') = f\ x\ \textbf{in}\ W\ (y, v \oplus v')
\end{array}
$$

The $return$ operation yields a minimal pair of type (a, o) with the given data x and an "empty output" $\emptyset$. The $\gg\!=$ operation applies f to the data x to produce a new data value y and a new output v'. Hence the overall resulting pair consists of both y and the extended output $v \oplus v'$.

B.2.2 Effectful Policy Evaluation

From the aforementioned vantage point, policy evaluation can be fittingly captured as an effectful function, whereby obligations are aggregated alongside the evaluation process via the monad $Writer\ o\ a$. In particular, we instantiate the data type a to our type Dec of decisions, and the output type o to obligation lists $[Oblg]$. Evaluating $Konst$ and $Filter$ policies in this way follows closely the definition of $oblg$ from earlier:

[4] The laws reflect those of the mathematical notion of monads from category theory.

$$
\begin{aligned}
&evalP :: Pol \to Env \to Writer\ [Oblg]\ Dec\\
&evalP\ (Konst\ dec)\ \_ = return\ dec\\
&evalP\ (Filter\ (GrantIf\ cond)\ obls)\ \rho\\
&\quad \mid evalC\ cond\ \rho \quad = W\ (Grant, obls)\\
&\quad \mid otherwise \quad = return\ Gap\\
&evalP\ (Filter\ (DenyIf\ cond)\ obls)\ \rho\\
&\quad \mid evalC\ cond\ \rho \quad = W\ (Deny, obls)\\
&\quad \mid otherwise \quad = return\ Gap
\end{aligned}
$$

For example, a $GrantIf\ cond$ rule with obligations $obls$ evaluates to the pair $(Grant, obls)$ whenever $cond$ holds, otherwise it is $return\ Gap$, that is with empty obligations $[]$. Similarly, $Konst\ dec$ policies always evaluate to dec without obligations.

Again the interesting case is the $Case$ policy. Recall that this entails traversing over each arm (g, p) of the policy until one is found with a positive guard g. But when such an arm is found, we do not wish to have accumulated any obligations as an effect of evaluating prior (negative) guards. This suggests a need for zeroing out an accumulating obligations list when necessary, which we can define in the following way.

$$
\begin{aligned}
&clearIf :: MonadWriter\ o\ m \Rightarrow m\ a \to (a \to Bool) \to m\ a\\
&clearIf\ xm\ pred = pass\ (\textbf{do}\\
&\quad x \leftarrow xm\\
&\quad return\ (x, \textbf{if}\ pred\ x\ \textbf{then}\ const\ \emptyset\ \textbf{else}\ id))
\end{aligned}
$$

While the specific details are not so crucial here[5], roughly speaking, $clearIf\ xm\ pred$ executes the writer computation xm but clears the output (by setting it to $\emptyset$) immediately after if the data x obtained from this execution satisfies the predicate $pred$. With this and an evaluator for guards $evalG$ (to be defined shortly), we complete the definition of $evalP$ for $Case$ policies.

$$
\begin{aligned}
&evalP\ (Case\ []\ defPol)\ \rho = evalP\ defPol\ \rho\\
&evalP\ (Case\ ((g, p) : as)\ defPol)\ \rho = \textbf{do}\\
&\quad b \leftarrow evalG\ g\ \rho\ `clearIf`\ not\\
&\quad \textbf{if}\ b\ \textbf{then}\ evalP\ p\ \rho\\
&\qquad \textbf{else}\ evalP\ (Case\ as\ defPol)\ \rho
\end{aligned}
$$

In the non-empty case, in which we pattern match the arms as $(g, p) : as$, the guard g is evaluated. If it holds, the corresponding policy p is evaluated, accumulating obligations as normal. If g does not hold however, the obligations list is cleared and evaluation continues to other arms.

It remains to define $evalG$:

$$
\begin{aligned}
&evalG :: Guard \to Env \to Writer\ [Oblg]\ Bool\\
&evalG\ Truth\ \_ = return\ True\\
&evalG\ (Eval\ pol\ dec)\ \rho = \textbf{do}\\
&\quad d \leftarrow clearIf\ (evalP\ pol\ \rho)\ (\lambda d \to d \not\equiv dec \vee d \in [Gap, Conflict])\\
&\quad return\ (d \equiv dec)\\
&evalG\ (Conj\ g_1\ g_2)\ \rho = \textbf{do}\\
&\quad b_1 \leftarrow evalG\ g_1\ \rho\\
&\quad b_2 \leftarrow evalG\ g_2\ \rho\\
&\quad return\ (b_1 \wedge b_2)
\end{aligned}
$$

[5] Any instance m of the $MonadWriter\ o$ type class (of which our $Writer\ o$ is one) supports an operation $pass :: m\ (a, o \to o) \to m\ a$ such that $pass\ xm$ is an action that executes xm, which returns a value and a function, and returns the value while applying the function to the output.

To evaluate a guard of the form $Eval\ \ pol\ \ dec$, we evaluate the policy pol and compare the resulting decision with dec. If they do not match or if the decision is not enforceable (i.e. Gap or $Conflict$), the obligations list is cleared. Evaluation of a conjunction of guards amounts to that of the sub-guards, combined with logical conjunction. Note that obligations accrued from evaluating g_1 are carried over to evaluation of g_2 and so on[6].

Having now formulated our evaluation functions, it is worth reflecting a little on the monadic approach taken here. As we have seen, obligations can be seen as a side-effect of evaluation, tidily abstracted away as $Writer$ computations. This brings about advantages both in clarity and readability – if instead we had implemented our evaluation maps to directly return a product $(Dec, [Oblg])$, we would quickly discover much "boilerplate" obscuring the overall intent of the evaluation code. Another benefit of this approach is *extensibility*. As the FROST language evolves, we may find the need to add other kinds of effects to evaluation. For example, XACML has a notion of an "indeterminate" value which we do not have in our Dec type, but could easily support with an additional effect to form a larger monad. In this manner, effects of evaluation are contained while the evaluation functions themselves require little change, if indeed at all.

[6] As all $Monad$ instances are necessarily *applicative functors*, the last equation for $evalG$ can also be expressed more succinctly with applicative-style combinators e.g. $(\wedge)$ <$> $evalG\ g_1\ \rho \circledast evalG\ g_2\ \rho$.