

# Extending Expressive Access Policies with Privacy Features\*

Stefan More

stefan.more@iaik.tugraz.at  
Graz University of Technology  
Graz, Austria

Lukas Alber

lukas.alber@iaik.tugraz.at  
Graz University of Technology  
Graz, Austria

Sebastian Ramacher

sebastian.ramacher@ait.ac.at  
AIT Austrian Institute of Technology  
Vienna, Austria

Marco Herzl

herzl@student.tugraz.at  
Graz University of Technology  
Graz, Austria

## ABSTRACT

Authentication, authorization, and trust verification are central parts of an access control system. The conditions for granting access in such a system are collected in access policies. Since access conditions are often complex, dedicated languages – policy languages – for defining policies are in use.

However, current policy languages are unable to express such conditions having privacy of users in mind. With privacy-preserving technologies, users are enabled to prove information to the access system without revealing it.

In this work, we present a generic design for supporting privacy-preserving technologies in policy languages. Our design prevents unnecessary disclosure of sensitive information while still allowing the formulation of expressive rules for access control. For that we make use of zero-knowledge proofs (NIZKs). We demonstrate our design by applying it to the TPL policy language, while using SNARKs. Also, we evaluate the resulting ZK-TPL language and its associated toolchain. Our evaluation shows that for regular-sized credentials communication and verification overhead is negligible.

## 1 INTRODUCTION

Trust verification, authentication, and authorization are core concepts of access management. Conditions out of all three concepts specify whether access can be granted. They can involve the user, resources, the requested operation, and facts from the environment. The conditions together form a set of rules called *policy*. While a policy may be simple and only define a single bit that grants access to a resource, often its conditions are complex and require an elaborate implementation of multiple intertwined checks into access control systems. Furthermore, with many clients and services interacting, it can become quite tedious to implement different variants of access logic.

**Access policy languages** [7, 16, 25, 36, 49, 51] enable the codification and re-use of access policies while decoupling them from the deployed access control systems. Furthermore, policy languages offer a higher level of abstraction that facilitates the design of policies without requiring concrete insights into the implementation of the underlying access control system. An example identity management model is Self-Sovereign Identity (SSI) [6]. Since this model

often involves complex access policies, previous research has already addressed the integration of policy languages into the SSI model [4, 11], but neglects the topic of privacy.

To better highlight the privacy issues we aim to tackle, we first summarize the *policy-based and SSI-based access control* model: First, a policy is defined by a domain expert, codifying rules for authentication, authorization, and trust verification. This policy is then stored at a *service provider (SP)*. As soon as a *user* wants to access a resource or consume a service at this SP, they need to show that they fulfill the access policy. For that the user relies on self-sovereign identity attributes, which they receive from an issuer or identity provider in the form of *digital credentials*. The user then stores their credentials in a *digital identity wallet* [40]. To authenticate, the user bundles the needed credentials to its service request and sends it to the SP. After receiving the request, the SP uses a *policy interpreter* to verify the incoming user request by applying the access policy.

However, this approach suffers from **privacy issues**. Users are often in possession of credentials that certify numerous attributes. When showing a credential to an SP, users reveal all attributes to the verifier, which is often neither desirable nor necessary to fulfill an authentication request. By integrating privacy-preserving technologies into the access control process, users are enabled to only reveal a subset of the attributes or even prove a statement without revealing any attribute at all. For example, by introducing Camenisch-Lysyanskaya (CL) signatures [21] into W3C's verifiable presentations [44], support for privacy-preserving showings is achieved. Those features are well-understood in the field of attributed-based credentials [23, 24] and have been studied for SSI systems [2, 3, 39].

The proliferation of privacy-preserving in combination with policy languages also enable new use cases. For example, in data marketplaces [33, 38] they allow data owners to define access policies based on the data sellers credential. Thereby, privacy beyond the privacy of the shared data in such a marketplace may be achieved.

### 1.1 Challenges

While the above-mentioned research and standards are helpful, integrating privacy-preserving technologies into policy-based access control systems is not straightforward. Several challenges need to be addressed: How should sensitive attributes be marked hidden in a policy language? Which statements on the hidden attributes need to be revealed? How is the user informed on the statements they

\*This is the full version of a paper which appears in 21th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2022), December 9-11, 2022, Wuhan, China, IEEE. [http://www.ieee-hust-ncc.org/2022/TrustCom/accepted\\_paper.html](http://www.ieee-hust-ncc.org/2022/TrustCom/accepted_paper.html)

need to prove? How can a user-side wallet implementation support all the possible proof types different SPs may ask for? Which privacy-preserving technologies can help to overcome these challenges while being flexible enough to preserve the expressiveness of the policy languages?

## 1.2 Our contribution

In this work, we close the gap by extending policy language systems with privacy features using zero-knowledge proofs. Our contribution is as follows:

**Privacy-preserving Policy System:** We introduce a generalized approach for extending existing policy language-based access control systems relying on SSI. For convenience, we will refer to the latter as *policy systems*. The policy’s author codifies which statements the user should prove and which information needs to be revealed to receive access. The author then uses our *policy compiler* to derive a presentation request they provide to users. The presentation request informs the user about the attributes they need to reveal and statements they need to prove. That enables users to only provide the required attributes and statements, and hide the rest of the credential data. While we focus our implementation and evaluation on the SSI model, the design itself is generic. It can be applied to various policy systems and identity management models, enriching them with privacy features.

**Implementation:** To show the feasibility of our design, we provide a concrete instantiation of our policy system extension. First, we adapt the TPL trust policy system [36] to enable privacy-preserving access control. Second, we extend the syntax of TPL policies to allow denoting whether attributes should be revealed, as well as which statements should be proven without revealing additional information. Third, we automatically compile the policies into the corresponding circuits for SNARK-based zero-knowledge proofs [15]. Users later execute these circuits to generate proofs of attributes without revealing their value.

**Evaluation:** Finally, we evaluate our implementation and discuss the overhead introduced by the additional computations. We perform the evaluation for different commitments (SHA256, Poseidon) and two elliptic curves (ALT-BN128, BLS12-381). We observe that the duration of the one-time setup and the computation of a proof depend on the choice of both commitment and curve. However, only the verification is time-sensitive, since it needs to happen in real-time during the user’s request for access. We found that the verification introduces a negligible performance overhead. Since SNARKs are non-interactive, also the access verification is non-interactive. Thus, a privacy-preserving showing requires no additional communication rounds between the user and the SP.

## 1.3 Related Work

Decentralized services like IPFS<sup>1</sup> store user data redundantly on one or more nodes. However, if the data is sensitive, it should not be accessible arbitrarily. Therefore, Prünster et al. [41] show an approach on how to outsource data protected by an access policy without needing to involve the data owner and thus ensure decentralization. Roughly speaking, the sensitive data’s encryption key is split into several parts stored on different, selected nodes.

<sup>1</sup><https://ipfs.io/>, accessed on 2022-07-20

On access, these nodes evaluate the accessing party’s attributes using a policy. If all agree on granting access, the encryption key can be recovered from the key shares, and the data turns accessible. All in all, their fully decentralized ABAC implementation focuses on decentralized user data access (compared to service access) and uses the eXtensible Access Control Markup Language (XACML<sup>2</sup>) standard as policy language.

Belchior et al. [11] propose a Self-Sovereign Identity based access control (SSIBAC) for service providers. It leverages conventional attribute-based access control using the attributes in SSI credentials, profiting from its decentralized nature. SSIBAC uses the XACML standard<sup>3</sup> for policy specification. Their implementation is based on W3C Verifiable Credentials [44], Hyperledger Indy<sup>4</sup> and Aries<sup>5</sup> for communication and distributed storage. While they mention the possibility of introducing privacy-enhancing technologies, they do not discuss the integration into an access policy language.

The ABC4Trust project focused on privacy-enhancing attribute-based credentials (ABC) and can be seen as preliminary work for the W3C VC standard [44] and modern SSI [13, 14, 43]. In their approach, the verifier sends a so-called presentation policy to the user. This policy states the conditions the user has to fulfill in order to access the service. On the user-side, the ABC engine then matches the needed attributes, and finally, a presentation token is created consisting of cryptographic evidence that the user satisfies the policy. This proof can later be verified for access control purposes. Since the whole policy must be proven through the presentation token, it can not contain any conditions that are difficult or impossible to translate to a cryptographic proof (e.g., online lookup for trust verification). Additionally, the project supports a limited list of functions for use in predicates on private attributes.<sup>6</sup> Thus, policies in ABC4Trust enable some privacy features but are limited in their flexibility.

## 2 BACKGROUND

### 2.1 Non-Interactive Zero-Knowledge Proof Systems

Non-interactive zero-knowledge (NIZK) proof systems represent powerful tools that enable a prover to convince a verifier of the validity of a statement without revealing any other information. For an NP-language  $L \subset X$  and a statement  $x \in L$ , a prover can present a proof to the verifier that  $x \in L$ , e.g., there exists a witness  $w$  such that  $x \in L$ . No other information about the witness  $w$  is leaked to the verifier. Formally, let  $R$  be the associated witness relation such that  $L = \{x \in X \mid \exists w: R(x, w) = 1\}$ . A non-interactive proof system  $\Pi$  consists of algorithms  $\text{Setup}(1^\kappa)$  producing a common reference string  $\text{crs}$ ,  $\text{Proof}(\text{crs}, x, w)$  taking a statement  $x \in X$  and a witness  $w$ , and outputting a proof  $\pi$ , and  $\text{Verify}(\text{crs}, x, \pi)$  taking a statement  $x$  and a proof  $\pi$ , and outputting the verification status. We require such a proof system to be *complete* (i.e., all proofs for statements in the language verify), *sound* (i.e., a proof for a statement outside

<sup>2</sup><http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html>

<sup>3</sup><http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html>

<sup>4</sup><https://github.com/hyperledger/indy-sdk>, accessed on 2022-07-03

<sup>5</sup><https://github.com/hyperledger/aries>, accessed on 2022-07-03

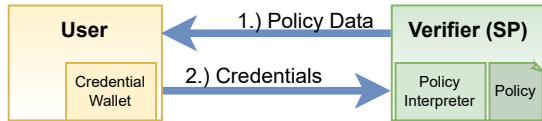
<sup>6</sup>[https://abc4trust.eu/download/Deliverable\\_D2.2.pdf](https://abc4trust.eu/download/Deliverable_D2.2.pdf), Section 4.4.3

the language verifies only with negligible probability), and *zero-knowledge* (i.e., a proof reveals no information on the witness).

Succinct Non-Interactive Arguments of Knowledge (SNARKs) are one of such systems and of particular interest to us, as they come with small proofs that are independent of the witness size and allow for fast verification. With the seminal work of Groth [30], SNARKs have been improved in various directions. To reduce the trust assumptions necessary for the generation of the common reference string (CRS), subversion-resistant and updatable versions have been investigated [1, 31]. SNARKs have been extended with stronger security notions such as strong simulation-sound extractability. Toolsets including ZoKrates [26], arkworks<sup>7</sup> or xjs-nark [34] provide compilers to turn arbitrary programs into circuits suitable for SNARKs or implement building blocks to help with the design of suitable circuits.

In this paper, we build our system on **ZoKrates**. It offers a high-level language syntax akin to Python with static typing. It allows implementing functions and programs that represent statements to be proven with a NIZK. Internally, the program will be represented as rank-1 constraint systems to be consumed by bellman<sup>8</sup> which implements Groth’s SNARK [30].

## 2.2 TPL Policy System



**Figure 1: High-level architecture of an access process using a policy system.**

A policy language enables the SP to specify precise conditions based on which access to a resource can be granted. We build upon the TPL language introduced by Mödersheim et al. [36]. Together with its toolchain, we call it a policy system. This system enables an expressive definition of access policies. Subsequently, it allows for automated decisions on whether incoming access requests can be accepted. The expressiveness and the flexibility of the system allow it to be used in different a variety of SPs. The system has its origin in the LIGHTest project [19] It supports DNS-based trust scheme verification [42] to derive trust relations and establish trust in credentials. Later, Alber et al. [4] added SSI concepts as an additional trust anchor option. The authoring of policies can happen natively in TPL, which has a syntax similar to Prolog. Additionally, non-technical domain experts are provided with an authoring tool with a graphical user interface [37, 47, 48].

A TPL policy (see Listing 1 for an example) is a set of several rules in the form Horn clauses:  $p(t) : -q_1(u_1), \dots, q_n(u_n)$ . A rule evaluates to true if all queries  $q_i(u_i)$  return true. A predicate  $p$  can be defined by multiple rules (of the same name) and evaluates to true if any of the rules are satisfied. To check whether a specific policy is met, the TPL interpreter evaluates a solution for query  $p(s)$ . If it is fulfilled for the provided input data (e.g., identity data),

the interpreter returns true. The interpreter evaluates the query by searching for suitable rules for substitution. If a query  $p(s)$  and predicate  $p(t)$  share an unifiable  $s$  and  $t$ , a unifier is calculated and applied to the subqueries of the predicate’s rule. Subsequently, all subqueries are evaluated in the same way. In case of a subquery returning false, another rule needs to be found for substitution. If all substitutable rules are exhausted, and no solution can be found, false is returned.

Subqueries are evaluated recursively until a ground truth is found. Such truth can be a relational operation or a built-in predicate. The TPL backend system handles all the built-in predicates. They help, for example, with server lookups for trust information discovery in eIDAS [45] and SSI [4].

The entry point of a policy is the *accept* predicate. To check if an incoming presentation is acceptable, the *accept* predicate is initially called through a corresponding query. The presentation token itself is the only input argument.

```

accept(Presentation) :-
    set_format(Presentation, w3cVP),
    extract(Presentation, verifiableCredential, Cred),
    set_format(Cred, w3cVC),
    extract(Cred, issuerDID, DIDissuer),
    checkQualified(DIDissuer),
    checkSig(Cred, DIDissuer),

    extract(Cred, credentialSubject, Subject),
    extract(Subject, date_of_birth, Birthdate),
    calculateAge(Birthdate, Age), Age >= 18,

    extract(Subject, username, Username),
    print(Username).
    
```

**Listing 1: Example TPL policy for W3C Verifiable Credentials, with the trust-check omitted for clarity.**

## 3 CONCEPT

In this section, we describe the design of our access policy system with privacy-preserving features. In Section 4 we discuss a concrete instantiation of this design.

Before describing the different components, actors and the process, we will present the high-level idea of our concept.

### 3.1 High-Level Idea

**Preliminary: Commit-sign-proof Credentials** One common approach (cf. [21, 28]) to design attribute credentials is to first commit to the attributes. This commitment is then signed by the issuer (i.e., identity provider).<sup>9</sup> For privacy-preserving showings, the user later proves consistency of any revealed attribute with respect to the commitment. The latter proof is combined with a proof of knowledge of a signature on the commitment, or by directly providing the signature to the verifier.

**Compiling Access Policies into NIZK Proofs** As in other systems with access control, rules that have to be satisfied must be represented as program logic, forming a policy. Hence, we extend

<sup>7</sup><https://arkworks.rs>, accessed on 2022-07-07

<sup>8</sup><https://github.com/zkrypto/bellman>, accessed on 2022-07-01

<sup>9</sup>The signature and commitment may coincide, but for giving an intuition, we consider them distinct.

the concept of commit-sign-proof credentials with the possibility for a policy designer to codify such access rules. For any committed-to attribute, we enable the policy designer to decide whether the user needs to reveal an attribute to the service provider or whether it is sufficient to convince the verifier that an attribute satisfies a policy rule without revealing it. Our system automatically informs the user about the policy and compiles it into the corresponding NIZK proofs. That is, the user proves the consistency of revealed attributes with the public commitment. Similarly, for all rules involving hidden attributes, the user also generates proofs of knowledge of these attributes and that they satisfy the specified rules.

In our system, we allow the policy to express rules with respect to any credential format: attributes that are encoded in some form of data structure that has some public reference value. The latter may consist of a signed commitment or a signature directly over the attributes. The statement for the proof system is then built accordingly.

### 3.2 Components

Our system consists of the following actors/components:

**User (Prover):** The actor that wants to access a resource or consume a service at the SP, and needs to authenticate to do so. Their identity attributes are stored in form of **digital credentials** in a **digital identity wallet** [40]. Part of the wallet, the user's system is also a **policy client**, which prepares a **presentation token** that satisfies a **presentation request**.

**Service Provider (Verifier):** The SP is the actor (or their system) which provides access-protected services or resources to users. To control who can access a resource and how users are authenticated. The administrator of the SP creates a **policy** that encodes access control and trust rules. The SP uses a **policy compiler** to generate the presentation request, which they provide to the user. After receiving a service request, the SP uses a (**extended**) **policy interpreter** to verify the request.

### 3.3 Phases

We now discuss the entire process of our concept. We split the process into the following phases: (1) The setup of the cryptographic system, authoring of a policy, and publishing of the presentation request, (2) computing of a presentation token, and finally (3) the verification of the presentation token.

**(1) Setup System and Policy:** If the employed proof system requires a specific setup, performing it is the first step. For example, when using SNARKs, a trusted third party generates a CRS. The so-obtained common material is published and retrieved by all system participants.

During the setup phase, a *policy* is authored by the service provider (SP). This policy encodes the rules a user of the SP needs to fulfill to use the service or access a resource. Depending on the nature of an SP, there can be different policies for different services or resources.

Authors of this policy are either technical personnel or domain experts of the SP. Although an author without technical knowledge but with domain expertise can use a graphical policy authoring tool to create the policy [47, 48], a policy is, in the end, always encoded in machine-readable form. As part of the policy, the author specifies

which attributes a user has to provide and what credential types the SP accepts. Additionally, the policy author defines two types of rules the user attributes have to satisfy, which are differentiated by whether they operate on private attributes or revealed attributes. Our system later transforms the first set of rules into statements for the NIZK proof system. Thus, the user can prove that their credentials fulfill these rules without revealing the values of the attributes. The second set of rules operates on attributes the user must reveal to the SP. These rules are used when the SP requires the attributes for further processing or trust management. Depending on the NIZK proof system, the SP at this stage also compiles parts of the policy into an intermediate representation for the policy client.

The encoded policy, together with metadata about the service, forms the *presentation request* for a specific service. Before initiating a service request, users need to know what data they are required to provide. Therefore, the presentation request is published by the SP.

**(2) Authentication at SP:** When user want to access a service or consume a resource, they have to authenticate with the SP. To do so, they first retrieve the corresponding presentation request from the SP's website or another form of a service catalog.

The user then extracts the policy from the presentation request and executes it using the policy client. While doing so, the client retrieves the respective credentials from the user's identity wallet [22]. For the rules on public attributes, the client extracts the attributes from the credential, thereby revealing their values. It then computes a NIZK statement that proves that the value was indeed extracted from the credential, i.e., to prove the consistency of the values with the commitment. This statement proves that the revealed attributes are linked to their credential. As a special case, if all attributes of a credential are specified as revealed in a policy, the client adds the full credential to the response instead of a proof. For the policy rules on private attributes, the client computes a NIZK statement which proves that the attributes fulfill the rules. Again, the client adds a statement to the proof that the attributes were indeed extracted from the credential.

For the computation of proofs, the client uses the common material retrieved in the setup phase and the NIZK statements. The client also appends the metadata (e.g., issuer information) of all involved credentials to the response.

After executing the policy, the client encodes all proofs, revealed credentials, and credential metadata into a *presentation token*.

Then, the user adds service-specific data and sends it alongside the token to the SP.

**(3) Verification of Presentation Tokens:** On receiving a request, the SP loads the policy for the respective service. The SP's policy interpreter then uses a *NIZK verifier* and a *policy verifier* to check the presentation token.

The SP extracts the NIZK proofs from the presentation token and verifies them using the NIZK verifier. As inputs for the NIZK verifier, the policy and its proof system-specific representation, respectively, need to be provided. Additionally, all public reference values, i.e., the commitments and all revealed attributes, need to be known to the NIZK verifier. Hence, the SP extracts these values from the presentation token and provides them to the NIZK verifier. After this step, the verifier is convinced that the proven statements match the requested statements.

As next step, the SP initializes the policy verifier and executes the remaining rules of the policy. All rules that work only on revealed or public data are validated by the policy verifier.

**Ensuring Trustworthiness of Tokens:** Besides evaluating the rules on the revealed attributes, this phase verifies the token’s trustworthiness. To do so, the policy verifier uses the trust rules encoded in the policy to check the issuer of the commitments. That forms a trust chain from the issuer along the signature to the commitment, which is in turn linked with the proof and consequently the attributes.

The trust rules specify which issuers are trustworthy for which type of credentials. That can, for instance, be done by providing a list of trusted issuers. A more flexible method is to define a trusted trust scheme: One example of a possible trust scheme is Europe’s eIDAS trust framework. Another example are SSI trust schemes established using distributed ledgers. Depending on the defined trust scheme, the policy verifier automatically retrieves trust status information about the credential issuers from online registries. This process ensures that public and private attributes can be trusted. Therefore, all NIZK statements on these attributes are trustworthy.

After the NIZK verifier and the policy verifier conclude that the presentation token is trustworthy and fulfills the user’s policy, the SP grants the user access to the service.

## 4 IMPLEMENTATION

While our concept is described on a generic level, the concrete choice of policy system and proof system is important to assess the feasibility and to evaluate the performance and security. Thus, we provide a concrete instantiation of our concept.

Our implementation builds on the policy system *TPL* (cf. Section 2), which we extend with privacy features. Since *TPL* uses a logic-based syntax, we need to compile the rules encoded in form of *TPL* predicates to suitable statements for NIZK proofs. As NIZK proof system we use SNARKs, since it enables small proofs. We instantiate our SNARKs with the Groth16 proving scheme, which we execute with the help of the Bellman library. Furthermore, we integrate the ZoKrates zero-knowledge toolbox [26] as an intermediate layer in the transformation process. Thus, we compile *TPL* policies into ZoKrates programs, which are then mapped to circuits for bellman. An advantage of this intermediate step is that the SP can already compile the policy into a ZoKrates program and directly share that with the user as part of the presentation request. The user only needs to provide their credentials to the ZoKrates prover and execute that program. Then, they send the resulting proof alongside the selected revealed credentials to the SP as part of the presentation token. Finally, the SP uses the ZoKrates verifier to ensure the proof is valid. In the next step, they forward the NIZK verification result and the rest of the presentation token to the *TPL* verifier. In addition to executing the policy on the revealed credentials, the *TPL* verifier also checks whether all data is trustworthy. An overview of this process is shown in Figure 2.

In our implementation we enable the privacy-preserving showing of attributes originating from credentials as well as private statements on these attributes driven by *TPL* policies. In the following sections, we discuss the integration of our concept (cf. Section 3) into the *TPL* system in more detail. Specifically, Section 4.1 presents

the extensions of *TPL* to ZK-*TPL* from the point of view of the policy author. And Section 4.2 covers aspects of compiling ZK-*TPL* into NIZK statements using ZoKrates. Finally, Section 4.3 presents the evaluation of our implementation.

### 4.1 Extending *TPL* with Zero-Knowledge Rules

We now focus on the concrete changes to the *TPL* syntax to express ZK rules. A policy author defines in a policy which attributes a user needs to reveal. There are multiple options to denote this in a *TPL* policy: We now discuss a set of different options to extend the syntax *TPL* to do so.

**Option 1: Naming Convention:** In *TPL*, the type of terms such as atoms and variables is defined by their name. Any term starting with an uppercase letter followed by letters, numbers or underscores represents a variable. Whereas terms starting with lowercase letters refer to atoms (constants). Hence, in the same way we could refer to attributes that are not revealed via a naming convention. However, as adding new conventions to the *TPL* specification would lead to ambiguities, we consider this approach to be error-prone and unintuitive.

**Option 2: Privacy Predicate:** Another approach is to represent the hidden and revealed nature of attributes explicitly via a special predicate. As with other domain-specific predicates that are available for *TPL*, a new predicate can be defined that only evaluates to true if the corresponding attribute is hidden. With this approach, all predicates related to this attribute need then to cope with a potentially hidden attribute value.

**Option 3: zkaccept-Predicate:** Finally, the third (and chosen) option is to add a new *zkaccept* predicate in addition to the *accept* entrypointy-predicate for *TPL* programs. A policy is satisfied if and only if both *accept* and *zkaccept* evaluate to true. Consequently, *accept* and *zkaccept* are the two main predicates that represent a *TPL* rule set. With this approach, the meaning of the *accept* predicate is untouched and interpreted as before. In the *zkaccept* predicates, all statements are interpreted with respect to hidden attributes and cause the creation and verification of the corresponding NIZK proofs. Our approach is exemplified by the *TPL* policy in Listing 2. It requires the owner of the credential to be of 18 years or older, whereas neither the calculated age nor the *date\_of\_birth* attribute are revealed to the verifier. The example policy also contains a *semester* attribute, which is revealed to the verifier since it is only used in the *accept* predicate, but not in the *zkaccept* predicate.

We opted to implement the third approach since it provides a clear differentiation between predicates applied to public and hidden data points. As such, we consider it easier for the policy designer to design and reason about the policy. From a technical perspective, we expect all three approaches to be implementable with reasonable effort.

**Consistency Checks:** When evaluating the policy on the prover or verifier side, one needs to take care of multiple issues. First, an attribute can only appear as either hidden or revealed attribute. If a revealed attribute is also used in the *zkaccept* predicate, this is likely a mistake of the policy designer. Thus, the compiler needs to check if this invariant is satisfied and yield an alert if not. Secondly, when two or more distinct attributes of the same data structure,

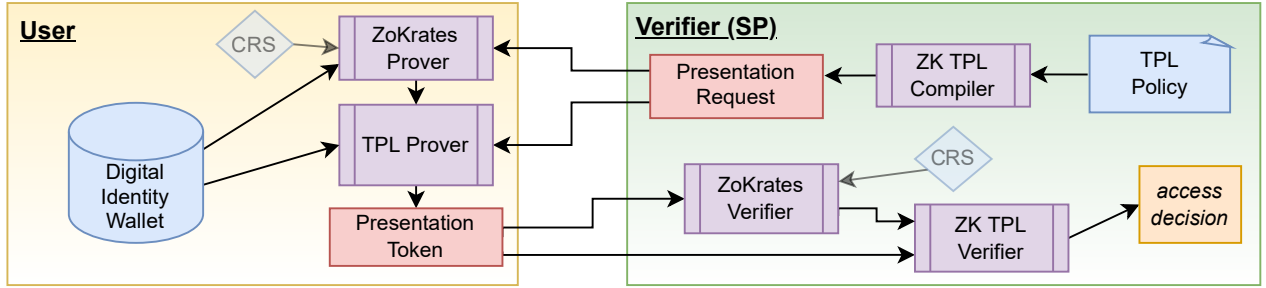


Figure 2: Architectural overview of our implementation including the actors and process flow.

```

zkaccept(Presentation) :-
  set_format(Presentation, w3cVP),
  extract(Presentation, verifiableCredential, Cred),
  set_format(Cred, w3cVC_student_id),
  extract(Cred, credentialSubject, Subject),

  % range proof on private attribute
  extract(Subject, date_of_birth, Birthdate),
  calculateAge(Birthdate, Age), Age >= 18.

accept(Presentation) :-
  set_format(Presentation, w3cVP),
  extract(Presentation, verifiableCredential, Cred),
  set_format(Cred, w3cVC_student_id),
  extract(Cred, issuerDID, DIDIssuer),
  checkQualified(DIDIssuer),
  checkSig(Cred, DIDIssuer),

  extract(Cred, credentialSubject, Subject),
  % revealed attribute
  extract(Subject, semester, Semester),
  print(Semester).

```

Listing 2: TPL policy from Listing 1 extended with privacy-preserving features: from the full credential, only the statement about age (derived from Birthdate and Semester attribute) are revealed.

e.g., a credential, are used in `accept` and `zkaccept`, consistency needs to be ensured. Hence, the zero-knowledge proof needs to be extended with statements ensuring consistency of all publicly revealed attributes.

## 4.2 Compiling TPL policies to NIZK circuits

**ZK-TPL Compiler:** As ZoKrates provides a domain-specific yet high-level language that closely resembles the syntax of Python, the TPL rules need to be compiled to this language. ZoKrates itself then compiles the corresponding code to suitable circuits for the underlying NIZK library, i.e., bellman. Hence, we provide the ZK-TPL compiler to transform policies from TPL syntax into ZoKrates’ proof program syntax, as shown in Figure 2. The ZoKrates standard library already provides several cryptographic primitives such as the compression function of SHA256, and SHA256 for a fixed number of calls to the compression functions, i.e., SHA256 for fixed input lengths, or Pedersen commitments. Therefore, parts of the functionality we require are covered by the standard library. Comparison operators for primitive types are also provided by ZoKrates.

When compiling TPL policies to ZoKrates programs, we consider the following challenges:

**1.) Constant-length Attributes:** When generating the ZoKrates proof program, a challenge is to map attributes to either private or public variables, and how to encode their lengths. As lengths can already be sensitive information for various data points, they are encoded as fixed-length string with  $\emptyset$  to pad to the maximal length. Thus, there is a compromise between runtime costs for the additional padding, security, and functionality. Length restrictions may be problematic for field types with international variations such as the use of first and last names.

**2.) Arithmetic:** While integer types are available in various forms (8 bit to 64 bit), ZoKrates also provides a native field type representing  $\mathbb{Z}_p$  where the prime  $p$  depends on the choice of elliptic curve used by ZoKrates. In general,  $p$  will be large ( $\geq 256$  bit) and all the arithmetic of the smaller types is implemented as  $\mathbb{Z}_p$ -arithmetic. Hence, when compiling arithmetic involving hidden attributes, arithmetic is best represented using the field type unless specific features of the fixed bit-width types are needed.

**3.) Representing Strings as Numbers:** Third, parsing arbitrary strings as integers is a complex and expensive task when performed inside ZoKrates. Conversion of an array of `u8s` into a `field` involves arithmetic and potentially additional checks of well-formedness, e.g., that the individual bytes are ASCII digits, or that the full string is valid UTF-8. Hence, we perform the parsing outside the ZK component as much as possible. To ensure the integrity of the proof, this pre-processing step uses the same encoding of attributes than the issuer of the credential. Thus, we require that the hash of the parsed data matches the hash used in the credential as commitment.

Note that our ZK-TPL compiler together with ZoKrates define the encoding of data and its representation in the rank-1 constraint system of the underlying SNARK. Therefore, any change our compiler or in ZoKrates may render old proofs unverifiable. For short-lived or interactive scenarios, we thus require compatible encodings for both prover and verifier.

**Example ZoKrates Program:** Listing 3 presents an example ZK program, generated by our ZK-TPL compiler. For the inputs to the hash function, we opted to directly use `u32` arrays as expected by the provided implementation of SHA256. When using different hash function designs with ZK-friendly permutations such as GMiMC [5] or Poseidon [29], the inputs and outputs can be represented as `field` elements. Thereby, we would be able to significantly improve

```

import "hashes/sha256/sha256" as sha256

def compare(u32[8] h1, u32[8] h2) -> bool:
    return h1[0] == h2[0] && h1[1] == h2[1] &&
           h1[2] == h2[2] && h1[3] == h2[3] &&
           h1[4] == h2[4] && h1[5] == h2[5] &&
           h1[6] == h2[6] && h1[7] == h2[7]

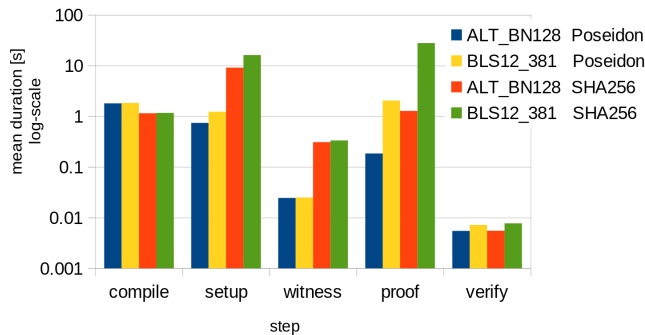
def main(u32[8] pub_hash, u32 currentYear,
         private u32 birthYear, u32 semester) -> bool:

    // Encode full credential, append SHA256 padding:
    u32[1][16] enc_cred = [[birthYear, semester,
                           2147483648, 0, 0, 0, 0, 0,
                           0, 0, 0, 0, 0, 0, 0, 64]]

    // Calculate age using private attribute:
    u32 age = currentYear - birthYear
    // Proof that attributes fulfill the age check
    // and the consistency of data used for the proof:
    return age >= 18
           && compare(pub_hash, sha256(enc_cred))
    
```

**Listing 3: ZoKrates program generated by compiling the TPL policy from Listing 2. Contains private `birthYear` attribute, revealed `semester` attribute, and (simplified) age check. It also proves the consistency of the attributes w.r.t. `pub_hash` commitment of the credential. The magic-numbers for the encoded credential are SHA256 padding-constants.**

the performance of the ZK program. With the goal in mind to be as widely usable as possible, we consider support for common hashes such as SHA256 essential.



**Figure 3: Overhead evaluation results of the example policy in Listing 2 for different commitments and curves.**

### 4.3 Evaluation

To evaluate our proof-of-concept implementation perform several benchmarks and compare them with the evaluation of the TPL system without any privacy features. Existing TPL benchmarks focus on the verification phase, which takes one to ten seconds for realistic policies and includes the retrieval of trust information from online registries [38]. Standard TPL needs no setup phase, and the authentication phase is a trivial process for the user.

**Setup:** To measure the impact of the privacy extension to the existing TPL toolchain in Java, we run benchmarks on a 2022 business

laptop. Our prover and verifier tools use ZoKrates, which we configured to use the bellmann backend with Groth16 [30]. We evaluate the performance on the ALT\_BN128 [10] and BLS12\_381 [17] curves; while the former provides 100 bit of security, the latter is slower but provides  $\geq 117$  bit of security [9, 50]. Additionally, we compare the performance of SHA256 with ZK-friendly Poseidon [29] (cf. Section 4.2).

**Results:** We divide our three phases (cf. Section 3) in two categories: (1) The setup phase (compilation of policy, CRS setup) is a one-time phase and only performed once for each policy. (2) The authentication phase (computation of witness, generation of proof) and verification phase (verification of proof and execution of policy) are repeated phases and executed for each authentication process.

Figure 3 visualizes the results of our evaluation. We observe that the verification duration and the size of the proof transmitted to the verifier are independent of the complexity of the policy. This is because the size of SNARK proofs amounts to only 3 elliptic curve points and is independent of the witness size. Also, the size of the transmitted proof naturally depends on the size of the revealed attributes. The performance of the setup and authentication phases depend on the number of attributes that are part of the credential. This is because all attributes are part of the credential’s signature and thus need to be part of the commitment in the proof (cf. Listing 3). The performance of calculating the commitment in the proof program is linear in the size of the attributes.

## 5 DISCUSSION

**NIZK Setup:** From a deployment point of view, policy-dependent setup phases may hamper the adoption of such a system. As this dependency is mainly influenced by the underlying proof system, an efficient proof system with a universal CRS is of paramount importance for more flexible applications. Also, the need for a trusted third party for the CRS generation is not ideal in some use cases. However, it has already been shown that the TTP can be replaced using secure multi-party computation for the setup algorithm [18]. Alternatively, it is also possible to employ transparent, subversion-resistant, or updatable proof systems [1, 12, 20, 27, 31], where knowledge of secret trapdoors no longer poses a threat.

**Constant-length Attributes:** During the implementation of the ZK-TPL interpreter and the design of example ZK-TPL policies, we observe multiple restrictions inherent to the use of attributes with arbitrary types. Specifically, when dealing with string attributes, all the strings need to be encoded with a constant length. Otherwise, the length of the strings could reduce the anonymity set and the mere knowledge of the string lengths leaks sensitive information. This also extends to primitives that consume these strings, e.g., hash functions, as the number of compression function evaluations depends on the size of the input.

**Future Work on NIZK Toolchains:** While NIZKs and SNARKs are known for languages in all of NP (cf. [32] and others), for practical purposes the situation is significantly different. Yet, as implementations of SNARKs gain better toolchains with support for higher-level abstractions, SNARKs can be applied to solve more interesting challenges. These toolchains need to abstract technical details such as rank-1 constraint systems and other arithmetization

techniques to be useful for a wider audience. With ongoing scientific and engineering effort, these abstractions are rendered more efficient, less costly, and more expressive.

**Future Work on Policy Authoring Tools:** Since the capabilities of the policy language got extended, we need to update the GUI-based authoring tools [37, 47] in future work. Attributes should be hidden by default and only be revealed when indicated. Non-technical policy authors should be able to use zero-knowledge features in a graphical manner without being familiar any underlying details.

**Communicating Privacy Implications to Users:** While we extend the capabilities available to a policy designer, the consequences of revealing certain attributes also need to be explained to the user. Some works have developed interfaces that highlight the revealed attributes and data flows to the user. Examples in various directions include Angulo et al.’s approach [8], which provides visualizations of policies, and Mikkelsen et al. [35] presenting a user interface to disclose attributes of a credential selectively. Alternatively, privacy metrics [46] offer tools to attach scores based on the processed data and the type of performed computations. Using these techniques may help visualize a user’s potential privacy risks based on the policy in question.

## ACKNOWLEDGMENTS

This work was supported by the European Union’s Horizon 2020 research and innovation programme under grant agreement № 871473 (KRAKEN). We thank our colleague Jakob Heher for proofreading our manuscript.

## REFERENCES

- [1] Behzad Abdolmaleki, Sebastian Ramacher, and Daniel Slamanig. 2020. Lift-and-Shift: Obtaining Simulation Extractable Subversion and Updatable SNARKs Generically. In *CCS*. ACM, 1987–2005.
- [2] Andreas Abraham, Felix Hörandner, Olamide Omolola, and Sebastian Ramacher. 2019. Privacy-Preserving eID Derivation for Self-Sovereign Identity Systems. In *ICICS (LNCS, Vol. 11999)*. Springer, 307–323.
- [3] Andreas Abraham, Karl Koch, Stefan More, Sebastian Ramacher, and Miha Stopar. 2021. Privacy-Preserving eID Derivation to Self-Sovereign Identity Systems with Offline Revocation. In *TrustCom*. IEEE, 506–513.
- [4] Lukas Alber, Stefan More, Sebastian Mödersheim, and Anders Schlichtkrull. 2021. Adapting the TPL Trust Policy Language for a Self-Sovereign Identity World. In *Open Identity Summit (LNI, Vol. P-312)*. Gesellschaft für Informatik e.V., 107–118.
- [5] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger. 2019. Feistel Structures for MPC, and More. In *ESORICS (2) (LNCS, Vol. 11736)*. Springer, 151–171.
- [6] Christopher Allen. 2016. The Path to Self-Sovereign-Identity. <http://www.lifewithalacrity.com/2016/04/the-path-to-self-sovereign-identity.html> online, accessed on 2022-07-13.
- [7] Christopher Alm, Ruben Wolf, and Joachim Posegga. 2009. The OPL Access Control Policy Language. In *TrustBus (LNCS, Vol. 5695)*. Springer, 138–148.
- [8] Julio Angulo, Simone Fischer-Hübner, Tobias Pulls, and Ulrich König. 2011. HCI for Policy Display and Administration. In *Privacy and Identity Management for Life*. Springer, 261–277.
- [9] Razvan Barbulescu and Sylvain Duesne. 2019. Updating Key Size Estimations for Pairings. *J. Cryptol.* 32, 4 (2019), 1298–1336.
- [10] Paulo S. L. M. Barreto and Michael Naehrig. 2005. Pairing-Friendly Elliptic Curves of Prime Order. In *SAC (LNCS, Vol. 3897)*. Springer, 319–331.
- [11] Rafael Belchior, Benedikt Putz, Günther Pernul, Miguel Correia, André Vasconcelos, and Sérgio Guerreiro. 2020. SSIBAC: Self-Sovereign Identity Based Access Control. In *TrustCom*. IEEE, 1935–1943.
- [12] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable Zero Knowledge with No Trusted Setup. In *CRYPTO (3) (LNCS, Vol. 11694)*. Springer, 701–732.
- [13] Patrik Bichsel, Jan Camenisch, Maria Dubovitskaya, Robert R. Enderlein, Stephan Krenn, Ioannis Krontiris, Anja Lehmann, Gregory Neven, Christian Paquin, Franz-Stefan Preiss, Kai Rannenberg, and Ahmad Sabouri. 2015. An Architecture for Privacy-ABCs. In *Attribute-based Credentials for Trust*. Springer, 11–78.
- [14] Patrik Bichsel, Jan Camenisch, Maria Dubovitskaya, Robert R. Enderlein, Stephan Krenn, Anja Lehmann, Gregory Neven, and Franz-Stefan Preiss. 2015. Cryptographic Protocols Underlying Privacy-ABCs. In *Attribute-based Credentials for Trust*. Springer, 79–108.
- [15] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*. ACM, 326–349.
- [16] Matt Blaze, Joan Feigenbaum, and Jack Lacy. 1996. Decentralized Trust Management. In *IEEE S&P*. IEEE, 164–173.
- [17] Sean Bowe. 2017. BLS12-381: New zk-SNARK elliptic curve construction. <https://electriccoin.co/blog/new-zk-snark-curve/>
- [18] Sean Bowe, Ariel Gabizon, and Matthew D. Green. 2018. A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK. In *Financial Cryptography Workshops (LNCS, Vol. 10958)*. Springer, 64–77.
- [19] Bud P. Bruegger and Peter Lipp. 2016. LIGHT<sup>est</sup> - A Lightweight Infrastructure for Global Heterogeneous Trust Management. In *Open Identity Summit (LNI, Vol. P-264)*. GI, 15–26.
- [20] Benedikt Bünz, Ben Fisch, and Alan Szeplieniec. 2020. Transparent SNARKs from DARK Compilers. In *EUROCRYPT (1) (LNCS, Vol. 12105)*. Springer, 677–706.
- [21] Jan Camenisch and Anna Lysyanskaya. 2002. A Signature Scheme with Efficient Protocols. In *SCN (LNCS, Vol. 2576)*. Springer, 268–289.
- [22] Jan Camenisch, Sebastian Mödersheim, Gregory Neven, Franz-Stefan Preiss, and Alfredo Rial. 2015. A Prolog Program for Matching Attribute-Based Credentials to Access Control Policies. Research Report. IBM.
- [23] David Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM* 24, 2 (1981), 84–88.
- [24] David Chaum. 1985. Security Without Identification: Transaction Systems to Make Big Brother Obsolete. *Commun. ACM* 28, 10 (1985), 1030–1044.
- [25] Juri Luca De Coi and Daniel Olmedilla. 2008. A Review of Trust Management, Security and Privacy Policy Languages. In *SECURITY*. INSTICC Press, 483–490.
- [26] Jacob Eberhardt and Stefan Tai. 2018. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. In *iThings/GreenCom/CPSCoM/SmartData*. IEEE, 1084–1091.
- [27] Georg Fuchsbauer. 2018. Subversion-Zero-Knowledge SNARKs. In *PKC (1) (LNCS, Vol. 10769)*. Springer, 315–347.
- [28] Georg Fuchsbauer, Christian Hanser, and Daniel Slamanig. 2019. Structure-Preserving Signatures on Equivalence Classes and Constant-Size Anonymous Credentials. *J. Cryptol.* 32, 2 (2019), 498–546.
- [29] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX*. USENIX Association, 519–535.
- [30] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT (2) (LNCS, Vol. 9666)*. Springer, 305–326.
- [31] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. 2018. Updatable and Universal Common Reference Strings with Applications to zk-SNARKs. In *CRYPTO (3) (LNCS, Vol. 10993)*. Springer, 698–728.
- [32] Jens Groth, Rafail Ostrovsky, and Amit Sahai. 2006. Perfect Non-interactive Zero Knowledge for NP. In *EUROCRYPT (LNCS, Vol. 4004)*. Springer, 339–358.
- [33] Karl Koch, Stephan Krenn, Donato Pellegrino, and Sebastian Ramacher. 2020. Privacy-Preserving Analytics for Data Markets Using MPC. In *Privacy and Identity Management (IFIP Advances in Information and Communication Technology, Vol. 619)*. Springer, 226–246.
- [34] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. 2018. xJSnark: A Framework for Efficient Verifiable Computation. In *IEEE S&P*. IEEE, 944–961.
- [35] Gert Læssøe Mikkelsen, Kasper Damgård, Hans Guldager, Jonas Lindstrøm Jensen, Jesus Luna Garcia, Janus Dam Nielsen, Pascal Paillier, Giancarlo Pellegrino, Michael Bladt Stausholm, Neeraj Suri, and Heng Zhang. 2015. Technical Implementation and Feasibility. In *Attribute-based Credentials for Trust*. Springer, 255–317.
- [36] Sebastian Mödersheim, Anders Schlichtkrull, Georg Wagner, Stefan More, and Lukas Alber. 2019. TPL: A Trust Policy Language. In *IFIPTM (IFIP Advances in Information and Communication Technology, Vol. 563)*. Springer, 209–223.
- [37] Sebastian Alexander Mödersheim and Bihang Ni. 2019. GTPL: A Graphical Trust Policy Language. In *Open Identity Summit (LNI, Vol. P-293)*. GI, 107–118.
- [38] Stefan More and Lukas Alber. 2022. YOU SHALL NOT COMPUTE on my Data: Access Policies for Privacy-Preserving Data Marketplaces and an Implementation for a Distributed Market using MPC. In *ARES*. ACM, 137:1–137:8.
- [39] Alexander Mühlh, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. 2018. A survey on essential components of a self-sovereign identity. *Comput. Sci. Rev.* 30 (2018), 80–86.
- [40] Blaz Podgorelec, Lukas Alber, and Thomas Zefferer. 2022. What is a (Digital) Identity Wallet? A Systematic Literature Review. In *The 46th IEEE Computer Society Signature Conference on Computers, Software, and Applications (COMPSAC)*



- 2022).
- [41] Bernd Prünster, Dominik Ziegler, and Gerald Palfinger. 2020. Multiply, Divide, and Conquer - Making Fully Decentralised Access Control a Reality. In *NSS (LNCS, Vol. 12570)*. Springer, 311–326.
  - [42] Heiko Roßnagel. 2017. A Mechanism for Discovery and Verification of Trust Scheme Memberships: The Lightest Reference Architecture. In *Open Identity Summit (LNI, Vol. P-277)*. Gesellschaft für Informatik, Bonn, 81–92.
  - [43] Ahmad Sabouri, Ioannis Krontiris, and Kai Rannenberg. 2012. Attribute-Based Credentials for Trust (ABC4Trust). In *TrustBus (LNCS, Vol. 7449)*. Springer, 218–219.
  - [44] Manu Sporny, Dave Noble, Grant Longley, David Chadwick, et al. 2022. *Verifiable Credentials Data Model 1.1*. W3C Recommendation. W3C. <https://www.w3.org/TR/2022/REC-vc-data-model-20220303/>
  - [45] Georg Wagner, Sven Wagner, Stefan More, and Martin Hoffmann. 2019. DNS-based Trust Scheme Publication and Discovery. In *Open Identity Summit (LNI, Vol. P-293)*. GI, 49–58.
  - [46] Isabel Wagner and David Eckhoff. 2018. Technical Privacy Metrics: A Systematic Survey. *ACM Comput. Surv.* 51, 3 (2018), 57:1–57:38.
  - [47] Stephanie Weinhardt and Olamide Omolola. 2019. Usability of Policy Authoring Tools: A Layered Approach. In *ICISSP*. SciTePress, 301–308.
  - [48] Stephanie Weinhardt and Doreen St. Pierre. 2019. Lessons learned - Conducting a User Experience evaluation of a Trust Policy Authoring Tool. In *Open Identity Summit (LNI, Vol. P-293)*. GI, 185–190.
  - [49] Walt Yao. 2003. Fidelis: A Policy-Driven Trust Management Framework. In *iTrust (LNCS, Vol. 2692)*. Springer, 301–317.
  - [50] Shoko Yonezawa, Tetsutaro Kobayashi, and Tsunekazu Saito. 2019. *Pairing-Friendly Curves*. Internet-Draft draft-yonezawa-pairing-friendly-curves-02. IETF Secretariat. <http://www.ietf.org/internet-drafts/draft-yonezawa-pairing-friendly-curves-02.txt>
  - [51] Eric Yuan and Jin Tong. 2005. Attributed Based Access Control (ABAC) for Web Services. In *ICWS*. IEEE, 561–569.