Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Self-Evaluated Policies with Zero Knowledge Proofs

Scientific Students' Association Report

Author:

Martin Farkas
Balázs Ádám Toldi

Advisor:

Dr. Imre Kocsis
Bertalan Zoltán Péter

2023

Contents

Kivonat

Adatok tulajdonjogának és biztonságának növekvő jelentősége miatt a Zero Knowledge proof (tudásmentes bizonyítás, ZKP) protokollok a kriptográfiai kutatás alapkövévé váltak, és egyre szélesebb körben használatosak. Mivel a hatékony ZKP-k kialakítása bonyolult, magasabb szintű, főként imperatív nyelvekre szabott keretrendszereket hoztak létre ennek könnyítése érdekében.

Ezzel párhuzamosan növekszik az érdeklődés a deklaratív programozási nyelvek iránt a hozzáférés-ellenőrzés, az authorizáció, és az általános eljárásrendek kiértékelése terén. Ezen területek szorosan kapcsolódnak az adatvédelemhez, és nagyban hasznosítanak adatvédelmet megőrző technológiákat.

ZKP-k és a deklaratív programozási nyelvek integrációja még kiforratlan terület, különösen olyan kontextusokban, ahol eljárásrendek érvényesítése szükséges. Bevezetjük az Önkiértékelésű Eljárásrendek fogalmát: az értékelt eljárásrendet, annak alanya által kiszámított eredményét, a számítás igazolását és minden egyéb nyilvános bemenetét kriptográfiailag összekapcsoljuk, így egy saját magában értelmezhető bizonyítást kapunk a kiértékelésre. Célunk a következő kulcskérdés megválaszolása: Hogyan használhatjuk ki a Prolog nagy kifejezőerejét zökkenőmentesen, miközben fenntartjuk a ZKP-k erős adatvédelmi garanciáit egy Önkiértékelésű Eljárásrend keretein belül?

Munkánkban egy Prologra szabott, magas szintű ZKP keretrendszert mutatunk be. Prolog meta-interpretáció felhasználásával és az eljárásrend kiértékelések hatékony reprezentációjával hidat képzünk a kifejező eljárásrendek és a ZKP-k adatvédelmi garanciái között. Keretrendszerünk a privát bemeneti adatok biztonságára összpontosít egy publikus eljárásrend mellett. Az architektúra képes hatékonyan ellenőrizni a Prolog kiértékelési fáit és lehetőséget biztosít a Prolog programokhoz aritmetikai áramkörök létrehozására, melyek képesek a bizonyítási fa ellenőrzésére a bemeneti adatok magánjellegének védelme mellett.

Továbbá bemutatjuk keretrendszerünk implementációját egy pénzügyi-jellegű példán keresztül, nevezetesen a lakossági energiavásárlást államilag támogató eljárásrenddel, amely privát adatként védi a felhasználó fogyasztását. A kiértékelés eredményét egy Önkiértékelésű Eljárásrendbe integrálva - a bizonyítással és a nyilvános bemenetekkel együtt - az alany igazolhatja, hogy mennyi támogatásra jogosult, anélkül, hogy felfedné a mögöttes fogyasztását.

Munkánk bemutatja a Prolog és a ZKP-k integrálásának lehetőségeit, és új megközelítést nyújt a privát adatokat védő deklaratív eljárásrendek fejlesztése terén. Keretrendszerünk kiindulásként szolgálhat a jövőbeli kutatásokhoz, potenciális alkalmazásokkal olyan területeken, mint az Önrendelkezésű Identitások, a Korlát-Logikai Programozás és egyéb területeken.

Abstract

Given the rising interest in data ownership and security, Zero Knowledge proof (ZKP) protocols have emerged as a cornerstone of cryptographic research and are becoming increasingly widely used. As performant ZKPs are convoluted to design, frameworks for higher-level, mostly imperative languages have been proposed for generating ZKPs.

At the same time, there has been a resurgence of interest in declarative programming languages in the context of access control, authorization, and general policy enforcement and evaluation. These fields closely relate to privacy and would greatly benefit from privacy-preserving techniques.

There is a gap in integrating ZKPs with declarative programming languages, especially in contexts requiring rigorous policy enforcement. We introduce the concept of Self-Evaluated Policies: the evaluated policy, its result computed by the subject, the proof of computation, and all public inputs are cryptographically bound together and can by themselves serve as proof of compliance. We aim to answer the pivotal question: How can one seamlessly harness the strengths of Prolog's expressive power while ensuring the robust privacy guarantees of ZKPs within a Self-Evaluated Policy?

In this paper, we present a high-level ZKP framework tailored to Prolog. By leveraging Prolog's meta-interpretation capabilities and by constructing an efficient representation of the evaluations, we bridge the gap between expressive policy definition and the robust privacy assurances of ZKPs. Our framework focuses on protecting private input data for a public policy. The architecture can efficiently verify Prolog proof trees and provides a mechanism to generate arithmetic circuits specific to a Prolog program, which is able to validate the proof tree while preserving the privacy of underlying data.

We detail the implementation of our framework over an example financial use case, specifically, privacy-enhanced governmental support allocation for residential energy expenses. By bundling the result of the evaluation into a Self-Evaluated Policy together with the proof and public inputs, the subject can prove how much support they are entitled to without revealing their underlying consumption.

Our approach highlights the possibilities of integrating Prolog with ZKPs and offers insights for advancements in privacy-preserving policy evaluations. Our framework serves as a reference for future research, with potential applications in domains like Self-Sovereign Identities, Constraint Logic Programming, and beyond.

Chapter 1

Introduction

In today's world, data ownership and security have developed from peripheral concerns to paramount priorities. As individuals and institutions increasingly interact in digital realms, the necessity for privacy, confidentiality, and transparency emerges at the forefront of technological discourse. In this context, people, private companies and public institutions may recognize a shared need to preserve ownership over their identity, their data, and to preserve their right to an explanation of the algorithms, rules, and policies they are subject to. There are emerging solutions to fulfil these needs, coming from all over the field of privacy and confidentiality-preserving technologies.

One general concern arises when a party is the subject of a ruleset, ranging from *access control* to *governmental support policies*. For them to be able to receive the benefits, from *access* to *financing*, they are obligated to share their data with the executor of the ruleset, who evaluates based on the arguments provided, and returns *an* answer. Some solutions arose to mitigate specific aspects of this conundrum. Self-sovereign identities to increase ownership and decision power of individuals over their own data and identity, blockchain infrastructure to provide an irrefutable, single source of truth, and most importantly to our discussion, Zero-Knowledge Proofs, which enable innovative and pioneering privacy-preserving practices.

To add to the collection of these methods, we are addressing the challenge of defining and implementing a policy framework which allows its subjects to evaluate the policies themselves, in private, on their own hardware. We call this kind of policy a *Self-Evaluated policy*. We deemed that the policy should be written using a declarative language because these kinds of policies are easier to develop, are widespread in the contexts of access control and modern service platforms, and because they possess different computational structures, such as *proof trees* in Prolog, that help us make our framework more efficient and modular. Our language of choice is Prolog, because it natively supports arithmetics, among other functions, and is well suited for meta-interpretation, thus, the generation of proof trees for evaluations is attainable.

In this paper, we present the following new results:

- An efficient methodology to transform Prolog terms into a form suitable for utilization within a zero-knowledge environment
- A novel framework for constructing zero-knowledge provers that facilitates verifying the validity of a proof tree corresponding to a Prolog program, including arithmetic

- A unique usage of Prolog meta-interpreters as the computational backbone of our approach, as part of a modular architecture
- A Proof-of-concept implementation of the toolchain

Our paper is organised as follows: In Chapter 2, we introduce our example use case for Self-evaluated Policies and enumerate its requirements. Then, we discuss paradigms and specific technologies that help us realize this use case, specifically: Self-Sovereign Identities, Zero-Knowledge Proof systems, and declarative policy languages. We build up the concepts needed to understand Verifiable Credentials, the Circom Zero-Knowledge system, and Prolog evaluation.

Then, in Chapter 3, we also discuss Circuitree [27] and Otti [8], two Zero-Knowledge tools which we think are closely related to our work. We briefly introduce their approaches and highlight the similarities between our systems.

The second part of our paper discusses our contributions: In Chapter 4, we introduce *Prolog Meta-interpretation*, the process which enables us to generate *Prolog proof-trees*

Next, in Chapter 5 showcases how the proof trees are verified. It describes the constraint system our framework generates.

In Chapter 6, we discuss and evaluate the implementation of our toolchain. We examine its correctness, soundness, and completeness, test its performance, and put it in context of the use-case and other systems. The source code ti our implementation is available on GitHub[1].

Finally, in Chapter 7, we conclude our results and write about the potential future work relating to Self-evaluated Policies and our Circom-based toolchain.

[1] https://github.com/BlackLight54/Self-Evaluating-Policies-w-ZKPs

Chapter 2

From Self-Sovereign Identities to Self-Evaluated Policies

In this chapter, we detail our chosen use-case, the ZKP system and the declarative language we use, and generally, present the background of our work.

2.1 Use-case for Self-Evaluated Policies

During the second part of 2022, the Bank of International Settlements (BIS) Innovation Hub in London, in cooperation with the Bank of England, initiated an exploratory project to create an API definition and a prototype implementation for retail Central Bank Digital Currency (CBDC) systems. As a second phase of that project, at the beginning of 2023, the BIS IH published an open call for participation to envision, define, and prototype CBDC Use-Cases over the CBDC API and prototype called Project Rosalind [12].

A joint team [1] of BME and MNB, the Central Bank of Hungary was selected for participation, defined and created a use case and gained the chance to showcase the use case to the Central Banking community as a finalist of the Phase 2 tech-sprint [11].

The use-case revolves around energy price support, and showcases a scheme where citizens do not have to prefinance energy price support; instead, they are supported on a per energy bill basis, by submitting outstanding bills, and documents proving their various in-need status personal aspects to a (governmental) support office.

Due to the online, per-bill, and per-citizen nature of the scheme, the support office is able to apply energy price support policies with fine granularity, decide on support in real time, and is able to apply policies which take into account multiple policy objectives. In the prototype, these constitute the incentivization of energy use reduction (on a twelve-month rolling window basis), and the protection of vulnerable groups of society (such as job seekers, large families, pensioners, and single parents).

The prototype utilized innovative financial primitives of the Rosalind API to introduce privacy between energy suppliers and citizens in the energy price support context: the last step of the worked-out scheme is the payment of energy bills from citizen accounts, to energy suppliers, so that the supplier is not aware whether that specific payment was supported by the support office, and if it was, then to what extent. At the same time, the protocol also ensures the atomicity of joint payment of the supported and the unsupported

[1] participants: I. Kocsis, L. Gönczy, B.Z. Péter, B.Á. Toldi, L. Ónozó, Á. Nyikes, G. Magyar

part of bills, strongly reducing the level of trust necessary between citizens and the support office in terms of the appropriate use of funds.

The experience gained with the successful prototype showed that there is further room for improvement in a number of privacy and trust aspects. Importantly, in the current scheme, although the current support policy may be published by the support office in a publicly verifiable way, in the end, it is the support office that computes and decides on the support allowance of any bill and citizen. From the privacy point of view, it is questionable whether a scheme where privacy-sensitive information has to be submitted to the support office in a fully interpretable and irreprudiable way is optimal. We believe that at least the option of enabling the self-evaluation of such policies, and the withholding of as much sensitive information as possible should be worked out. This way, although the choice is influenced certainly by a great number of non-technical (legal, policy, etc.) considerations, decision-makers and the public influencing decisions can be aware of this alternative.

Certainly, Self-evaluation of policies creates the need for proving the correctness and validity of self-computed support allowances. The key research questions of this work were motivated by this goal; we plan to integrate the results back into the already existing prototype.

To fulfil this use-case, we define the requirements of a Self-evaluated policy:

- An evaluable representation of the policy can be transferred to its subject through a trusted channel
- The subject can evaluate the policy based on the public and private facts they hold, in private, on their own hardware
- The subject can create a Zero-Knowledge Proof of the result of their evaluation, meaning they are able to prove the outcome of the evaluation without revealing any private facts they hold
- The subject can communicate this proof through a trusted channel

2.2 Self-Sovereign Identities

In recent years, **Self-Sovereign Identities (SSI)** emerged as a novel solution for trusted data distribution over the internet. According to Brian Behlendorf, GM for Blockchain, Healthcare, and Identity at the Linux Foundation, **SSI** is "the most crucial fix for today's broken Internet." [7] It is a decentralized, peer-to-peer identity architecture based on the principle that each peer has total control over the identifiers they own or are subject to. We will briefly summarize the key building blocks of **SSI**, similarly as laid out by Drummond Reed and Alex Preukschat [7], with a focus on Verifiable Credentials, which provide a solution for standardised management of trusted data to be used as a Self-evaluated Policy, and also serve as its container, meaning the policy can be communicated in a trusted and verifiable way. Leveraging **SSI**, we have a cryptographic framework for communicating and distributing facts, policies, and proofs in a peer-to-peer and privacy-preserving way over the Internet.

2.2.1 ToIP protocol stack

The protocol stack, which serves as the entry point into the introduction of **SSI**, is the Trust Over IP (ToIP) [51] protocol stack (Fig.2.1). It aims to bring together and standardize

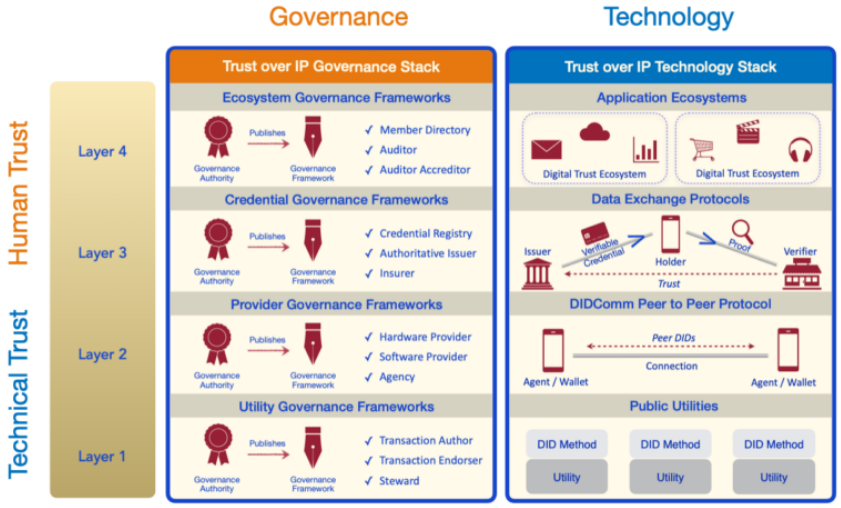


Figure 2.1: Trust Over IP Stack: `https://trustoverip.org/toip-model/`

the main ideas in SSI as a trust layer over the internet. We mainly focus on Verifiable Credentials located on the technology side of the stack in the Data Exchange Protocols Layer later in our introduction, because these are the objects our toolchain gets its trusted input from, meaning the policy, the public inputs, and the private inputs are *inside* Verifiable Credentials. To summarize the roles of the layers, the lower two levels of the stack focus on meeting the technical requirements of digital trust with blockchains and secure and private transaction protocols. In contrast, the top two layers focus on meeting human requirements with cryptographic credential protocols, which can be integrated into specific but standardised software ecosystems. Our toolchain would fit between Layer 3 and Layer 4 in this paradigm because it is a method for exchanged policy data to be interpreted and acted on.

In summary, a ToIP protocol can establish trusted, secure, and private peer-to-peer connections, issue, exchange, and verify digital credentials, and store public credential data on Verifiable Data registries, which may use decentralized or distributed record-keeping technologies.

2.2.1.1 Verifiable Credentials

"A verifiable credential is a tamper-evident credential that has authorship that can be cryptographically verified." [39]

Credentials contain claims about their subject made by their issuers, usually in JSON-LD format, cryptographically linked to both their issuer and their subject. These claims can then be verified by presenting them to a Verifier, through a verifier, meaning the claims are communicated through a secure channel, with the instructions and resources (e.g. signatures and public keys, possibly from PKI) to verify their origin and subject.

Digital credentials are held within digital wallets, which are digital structures designed to store Verifiable Credentials securely, and with the ability to present these Credentials using various cryptographic approaches.

A key type of Verifiable Credentials relevant to our use-case is *Anonymous Credentials* [37], which allow their controllers to prove a subset of claims in their Verifiable Credential (i.e. *selective disclosure*), or some statement about their attributes without revealing their

identity or even the underlying attribute. Anonymous Credentials use Zero-Knowledge Proofs that enable verifiably answering questions, such as "Are you above the legal drinking age?" without revealing any sensitive data. We detail an example of this type of Credential scheme in Section 2.3.

Through SSI and Verifiable Credentials, it is possible to distribute policies and claim facts about a subject in a privacy-preserving way. In this paper, we aim to leverage these possibilities to enable an entity to enforce a policy without getting to know the underlying facts that the policy is evaluated on.

2.3 Survey of Zero-Knowledge Proof systems

In this section, we define and introduce Zero-Knowledge Proof systems, explore a few concepts related to them, and discuss some existing frameworks, with considerations on how do they fit into our approach.

2.3.1 Definiton and examples

> **Definition 1 (Zero-Knowledge Proof).** A cryptographic scheme where a Prover is able to convince a Verifier that a statement is true, without providing any more information than that single bit (that is, that the statement is true rather than false). ▪
>
> *Computer Security Resource Center*
> *Information Technology Laboratory* [23]

The concept of Zero-Knowlege Proofs originates from Goldwasser et al. [30].

The classic example of a **Zero-Knowledge Proof Protocol** is the story of Ali Baba's cave [48]. There are two people: Peggy - the *Prover* -, and Victor - the *Verifier*. The story takes place in a ring-shaped cave, with the entrance on one side and a magic door on the other, which can only be opened and passed through by recanting a secret word. Peggy knows this secret word and wishes to prove this fact to Victor, but without sharing the word itself. There is a method for Peggy to prove knowledge of the secret to Victor, which is Zero-Knowledge, meaning she doesn't have to reveal the secret: Peggy enters the cave and chooses a path while Victor waits outside. Unseen by Victor, she chooses to walk either clockwise or counterclockwise. Victor then enters the cave and calls out the direction from which he expects Peggy to emerge. Knowing the secret word allows Peggy to always meet Victor's expectations, either by walking back the way she entered or using the secret word to pass through the magic door.

A single iteration gives Peggy a 50% chance of success even if she doesn't know the secret, as she might randomly choose the correct path. Repeating the protocol reduces the probability of a false positive, with ten iterations leaving only a $2^{-10} = 0.0009\%$ chance that Peggy is deceiving.

This story is an example of an **Interactive Zero-Knowledge Protocol**, where the two parties engage in an interactive exchange of information, through which the Verifier can ascertain that the Prover indeed possesses the knowledge they claim.

In many SSI ecosystems, Zero-Knowledge proofs, specifically set-membership [41], and range proofs [19] are used. A well-demonstrated and implemented in this regard is selective

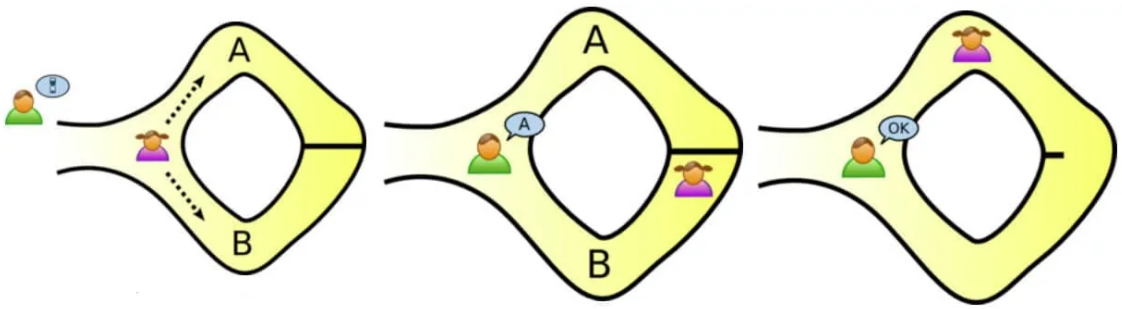


Figure 2.2: The example of Ali Baba's cave (from *What is Zero Knowledge Protocol (ZKP)?*)

disclosure of certain facts about a subject's credentials, often dubbed *Anonymous Credentials*. [37, 7] A common example of this is age verification for the purchase of controlled goods. Currently, in most countries, when one wishes to buy alcohol, they must show an identification card, which has a picture and their birthdate and may also show potentially much more information about the subject, such as identification number, their mother's maiden name, what type of vehicles they are allowed to drive; facts which are irrelevant for the purpose of age verification. Selective disclosure in this context means that one only has to prove the fact that they are above the drinking age, but not their exact birthdate or any other information.

There are established cryptographic frameworks for this use case in SSI, called *Selective Disclosure Predicates*, for example, in Hyperledger Aries [34]. However, they only offer set-membership (an element is within a set) [41] and range (a number is within a range) [19] proofs, which only allow them to construct proofs for simple use-cases such as this.

Our goal in this paper with Self-evaluated Policies is to extend this paradigm onto more complex predicates and use-cases, such as the use-case we discussed in Section 2.1.

2.3.2 Basic properties of Zero-Knowledge Proof systems

We introduce some basic but essential properties of Zero-Knowledge proofs, which according to Mohr et al. [42], all ZKP systems must satisfy

Definition 2 (Completeness). For all true statements being proven, there exists a proof that the Verifier will accept. ▪

Definition 3 (Soundness). For all false statements attempted to be proven, there does not exist any proof that the Verifier would accept. ▪

Definition 4 (Zero-knowledge). The Verifier does not learn any additional information from the proof or the protocol other than the fact that the statement is indeed true. ▪

2.3.3 Types of ZKPs

Current Zero-Knowledge Proof protocols can be categorized along a few properties[43]:

Interactive Zero-Knowledge Proofs A type of ZKP protocol that allows multiple messages between the Verifier and the Prover.

Non-Interactive Zero-Knowledge Proofs A type of ZKP protocol that only allows a single message from the Prover to the Verifier.

Proofs of knowledge A protocol not only allowing the Prover to prove that a statement is true but also that they know a *witness* to the truthfulness of the statement.

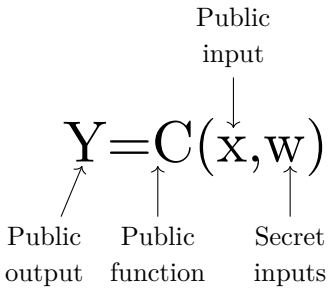
Arguments of knowledge A proof not only says that the statement is true, but the prover also knows why it is true.

Succint arguments of knowledge Communication complexity and verifier time are *polylogarithmic* in computation size.This effectively means that proofs stay relatively small (i.e. $\sim$ 1000 bytes), even for extensive programs, and they can be verified in a short amount of time (i.e. $\sim$ 1000 ms). Succinct arguments and proofs are preferred due to their efficiency. [16]

Need for a Trusted setup Some protocols require a setup "ceremony" before any sound proofs can be generated, called a *trusted setup*.

Definition 5 (Transparency). A proof system is *transparent* if it does not require a trusted setup, meaning there is no need for a trusted entity to distribute keys in a setup phase. [44] ▪

For one example, zk-SNARKs require a trusted setup phase. During this phase, a common reference string (CRS) is generated, a set of public parameters for creating and verifying proofs. However, in the process, it generates "toxic waste" as well. With this toxic waste, anyone could generate "fake" proofs. For this reason, the setup is "trusted" because the prover and the verifier must trust that the generator keeps the toxic waste secret (or removes it). Multiparty trusted setup "ceremonies" usually address this shortcoming[2]. One example of this type of these is the Powers-of-Tau to the People [46, 36].



Setup: (C,☣) $\rightarrow$ (pk,vk)

Prove: $(pk, x, w) \rightarrow$ Proof

Verify: (vk,Proof, $x, Y) \rightarrow$ Bool

Figure 2.3: zk-SNARK setup, proving, and verification activities based on a computation definition [52]

The need to perform a setup ceremony each time a proof needs to be generated can be mitigated by using an *universal setup*, a method used by PLONK [26]. This means

[2]Source: [52]

that the setup only needs to be done once, and that can be used for every program, not just one. A large number of participants can generate a reusable, and trusted *universal reference string* (powers of tau [36, 46]), that is well known through which the generation of ZKPs is made more efficient.

Zero-knowledge proofs can be constructed from NP problems. In fact, Goldreich et al. [29] showed that every language in NP has a zero-knowledge proof, given specific cryptographic assumptions. In practical terms, the two most significant types of ZKP constructions currently are zk-SNARKs and zk-STARKs.

2.3.3.1 zk-SNARKs

zk-SNARK stands for **Z**ero-**K**nowledge **S**uccinct **N**on-Interactive **Ar**gument of **K**nowledge, representing an advancement in the field of zero-Knowledge protocols. [31] Unlike interactive zero-knowledge proofs, where the prover and verifier engage in a back-and-forth dialogue, **zk-SNARKs** allow the prover to generate a single, compact proof that can be quickly verified by any party. This non-interactive nature not only enhances efficiency but also broadens the applicability of **zk-SNARKs** in various decentralized and distributed systems.

In **zk-SNARKs**, the "succinct" aspect is particularly crucial, as it ensures that the proofs are not only short in length but also require minimal computational resources to verify, irrespective of the complexity of the statement being proven. It is imperative to acknowledge that the efficiency of **zk-SNARKs** comes at the cost of requiring a trusted setup, which, if compromised through the leakage of sensitive intermediate values, the so-called "toxic waste", could potentially jeopardize the security of the entire system. Despite this, the integration of **zk-SNARKs** in various domains, such as privacy-preserving computation, attests to their potential in enhancing both security and efficiency in digital interactions.

2.3.3.2 zk-STARKs

zk-STARK stands for **z**ero-**k**nowledge **S**uccint **T**ransparent **AR**guments of **K**nowledge. [13] They utilize an Arithmetic Execution Trace, a technique that enables the conversion of program execution into a polynomial form. This transformation allows for the efficient verification of program execution's correctness, without revealing the actual data or the specific details of the computation. The transparent nature of **zk-STARK**s means they do not require a trusted setup, mitigating the risk associated with the potential exposure of sensitive information during the setup phase.

Furthermore, **zk-STARK**s demonstrate resilience against quantum attacks, ensuring their applicability and robustness in the post-quantum era. This post-quantum resilience is attributed to their reliance on post-quantum resilient cryptographic hash functions. The succinctness of **zk-STARK**s ensures that proofs are compact and can be verified efficiently, regardless of the computational complexity of the underlying statement. This property holds significant promise for enhancing efficiency and security across various domains, particularly in decentralized and distributed systems, where scalability and efficiency are paramount.

2.4 ZKP frameworks for computational models

In recent year, quite a few ZKP compilers and protocols emerged with differing characteristics, many of which we explored in the previous section. Many of these ZKP systems, among other capabilities such as set-membership and range proving, provide the ability to prove the execution of a computation. Many of these frameworks offer higher level abstractions, such as Zilch's ZeroJava [44], but many of them in the end compile to a Rank-1 Constraint System (R1CS) [54], or to Arithmetic Circuits, which in turn can be transpiled into R1CS.

For our use-case, we identified the following requirements that the underlying ZKP system should have:

Non-interactive The proof needs to be able to be verified by multiple parties, and it should be able to be used in blockchain settings, for example, as part of a Smart Contract.

Universal Requires less trust in the setup, as only one of the participants needs to be faithful in the setup process to be valid, and with 200+ participants in a *universal setup*, it is not likely that the setup is compromised.

Procedurally programmable As we are proving the evaluation of declarative policies (see Section 2.5), we don't necessarily need an OOP abstraction model, but some programming model is required because we deem that the development complexity of straight Arithmetic Circuits for our approach would be too high. Thus a procedurally programmable system is preferred.

Mature For our approach, we would prefer to use a system that is well-documented, has fewer early failures, has an active community, and is overall mature. This eases development and provides trust that this dependency of our system will be maintained well into the future.

While, as of November 2023, various systems have been published [15, 44, 22, 40, 18, 10, 17, 28, 20, 8, 35], surveying the existing research (Table 2.1), two solutions seemed especially adaptable for our use-case: Circom and LegoSNARKs.

Circom Circom [35] is a domain-specific language developed by iden3. It has a compiler that developers can use to create R1CS-based ZKP programs without needing to understand the underlying math behind it. It has an extensive standard library to help with common tasks. It also has an implementation for multiple proving systems, such as gorth16 [32], Plonk [26], and FFlonk [25], available in two libraries: SnarkJS and rapidsnark.

The representation in Figure 2.4[3] delineates the operational flow of the Circom framework. Initially, the developer formulates a Circom program using the specialized domain-specific language. This program is subsequently compiled into R1CS format. Utilizing this R1CS representation, a setup phase is executed through a universal setup mechanism, yielding both proving and verification keys. Upon code execution, tools such as SnarkJS or rapidsnark generate the corresponding witness. Leveraging this witness and the pre-established proving key, a zero-knowledge proof is produced. Furthermore, it is possible to generate an Ethereum verification smart contract written in Solidity using the verification key.

[3]Based on the figure in [52] for ZoKrates

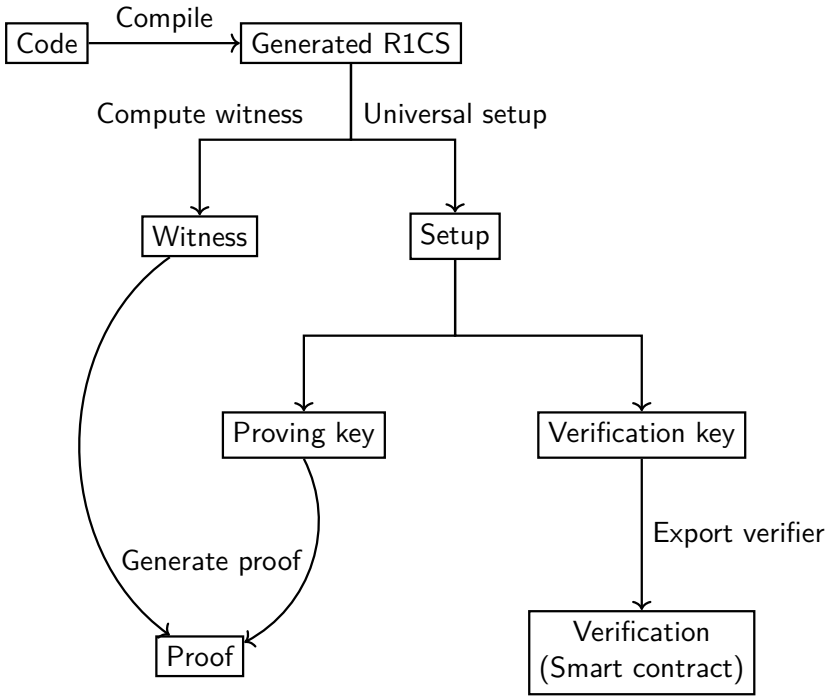


Figure 2.4: Circom workflow

LegoSNARKs LegoSNARK represents an advanced modular architecture tailored for assembling zkSNARKs by integrating specialized proof gadgets. This architecture provides a comprehensive toolkit for commit-and-prove zkSNARKs (CP-SNARKs), facilitating the development of novel CP-SNARKs from fundamental proof gadgets and extending commit-and-prove functionalities to a range of pre-existing zkSNARKs. Key benefits of employing LegoSNARK encompass its adaptability, reusability, and the potential for enhanced computational efficiency due to its modular design. Contrariwise, challenges surround the complexity of mastering the modular system and potential constraints in accommodating computations that do not align seamlessly with the inherent proof module structure.

ZKP system	**Protocol**	**Non-interactive**	**Universal**	**Ease of Use**	**Compiler Available**
TinyRAM [15]	zk-SNARK	●	○	Procedural	○
ZoKrates [22]	zk-SNARK	●	○	Procedural	●
PLONK [26]	zk-SNARK	●	●	Arithmetic Circuts	N/A
Bulletproofs [19]	zk-ShNARK	●	●	Arithmetic Circuts	N/A
Circom [35]	**zk-SNARK**	●	●	**Procedural**	●
zk-STARK [14]	zk-STARK	○	●	Assembly	N/A
Zilch [44]	zk-STARK	○	●	OOP	●

Table 2.1: Comparison of different ZKP systems [44]

2.5 Declarative policies

As the language of our Self-evaluated Policy, we use a declarative programming language, because, as stated by Godden et al. [27], they are very desirable for policy definition by virtue of their expressiveness, ease of development, completeness, and performance. To this end, we surveyed a few declarative languages as candidates for our Self-evaluated Policy language:

2.5.1 Open Policy Agent and REGO

Open Policy Agent (OPA)[2] is a rapidly emerging policy evaluation tool used on modern service platforms such as Kubernetes, Docker, and, most importantly, among existing Self-Sovereign Identity platforms [5]. **OPA** is designed to unify policy enforcement. It provides a high-level declarative language to define policies, called REGO. By leveraging JSON as its data format, **OPA** ensures a lightweight and interoperable approach to policy representation, making it an attractive option for organizations operating in diverse technological environments.

REGO is purpose-built for expressing policies over structured data. It enables users to specify policies that are both expressive and easy to reason about, ensuring that the intentions behind policies are clear and unambiguous. The language's syntax is designed to be accessible, with a focus on allowing users to express complex policies in a straightforward manner. This approach facilitates the adoption of **OPA**, as users are able to quickly become proficient in Rego and start defining policies that are tailored to their specific needs.

One obstacle to using **OPA** in a ZKP system is its complex evaluation semantic, meaning that **OPA**, opposed to Datalog and Prolog, generates an imperative evaluation plan which can be compiled to WASM, that contains imperative structures, API calls, and a changing application state. It may be possible to make **OPA** Zero-Knowledge, for example with the zkWASM[6] framework and using the compiled WASM, but this solution didn't fit our approach, and its feasibility is unclear.

Open Policy Agent stands out as a robust and versatile tool for policy evaluation. While its emergence is relatively recent, **OPA**'s potential for impact in the realm of policy evaluation is substantial, warranting close attention and consideration in this field.

2.5.2 Datalog

Datalog is a query and rule language for deductive databases. It is based on the logic programming paradigm, information is represented using facts and rules. Facts are basic statements that express explicit pieces of information, and rules define relationships between different facts, allowing for the derivation of new facts from existing ones.

The language is particularly well-suited for expressing recursive queries and is used in various domains such as program analysis, knowledge representation, database management, and access control. **Datalog** is designed to be more expressive than traditional relational database query languages like SQL, while also being more tractable and easier to optimize than full-fledged logic programming languages like Prolog.

One of the key features of **Datalog** is its simplicity and declarative nature, which allows users to focus on specifying what they want to compute, rather than how to compute it. This makes Datalog programs easier to understand, maintain, and optimize compared to imperative programming languages.

A note regarding pure **Datalog** is that by itself, it cannot handle arithmetic, although extensions exist to resolve this limitation, like DatalogZ.

Datalog provides a powerful and flexible framework for expressing complex queries and relationships in a concise and readable manner, making it a valuable tool for a wide range of applications in computer science and related fields. Still, it has its limitations.

2.6 Prolog as policy language

As the decalarative policy language for our prototype of Self-Evaluated Policies, we chose Prolog. The reason for this are the following:

- Compared to Open Policy Agent's **REGO**, Prolog's resolution semantics are much simpler, and therefore it is a much better candidate for a first prototype using our approach. Specifically it is relatively convenient to create proof of evaluation rather than proof of execution using proof trees and meta-interpretation, as we discuss in Section 4.2 and Section 4.3.
- Compared to pure **Datalog**, Prolog offers built-in functions, which when reified with a meta-interpreter (Section 4.3), are easily integrated into a proof of evaluation (Section 4.3.3), proving functions such as arithmetic, list-management, and many more, which are missing from **Datalog**. There is also a major difference between the evaluation model of the two languages. **Datalog** evaluates queries bottom-up, while Prolog uses a top-down approach. Also, there already exists a Zero-Knowledge reasoner for **Datalog**, *Circuitree* [27], which fills the niche of pure **Datalog** policy evaluation. We discuss *Circuitree* as a related work in Section 3.1.

We now introduce the syntax and semantics of Prolog statements and then present the evaluation semantics relevant to our use case, proof of policy evaluation. In a sense, we first show how Prolog programs are written and then discuss how they are evaluated.

2.6.1 Syntax

Our objective in this section is to describe Prolog's fundamental representation syntax, within the context of First-Order Predicate calculus, including terms, predicates, rules, facts, atoms, programs, and goals.

Definition 6 (First-Order Predicate calculus). Given a set $\mathcal{V}$ of variables, a set $\mathcal{F}$ of function symbols, a set $\mathcal{P}$ of predicate symbols, and the logical symbols $\neg, \wedge, \vee, \rightarrow, \forall, \exists$, the predicate calculus is a formal system that extends propositional calculus by allowing quantification over variables and the use of predicates. [38]

- **Terms**: A term is either a variable from $\mathcal{V}$, a constant (called an atom in Prolog), or a composite formed by applying a function symbol from $\mathcal{F}$ to a tuple of terms. For example, if f is a function symbol in $\mathcal{F}$ and X and Y are variables in $\mathcal{V}$, then $f(X, Y)$ is a term.
- **Predicates**: Predicates are Boolean-valued functions, with zero or more arguments. Given a predicate symbol P from $\mathcal{P}$ with arity n and terms $t_1, t_2, ..., t_n$, the expression $P(t_1, t_2, ..., t_n)$ is a predicate.

English	Predicate calculus	Prolog
and	$\wedge$	,
or	$\vee$	;
if (Horn clause)	$\leftarrow$	:-
not	$\neg$	not

Table 2.2: Connectives syntax in Prolog [38]

- **Logical Connectives**: These are symbols that are used to combine predicates into more complex expressions:
  - $\neg P$: Negation, the truth value is the opposite of P.
  - $P \wedge Q$: Conjunction, true only if both P and Q are true.
  - $P \vee Q$: Disjunction, true if at least one of P or Q is true.
  - $P \rightarrow Q$: Implication, false only if P is true and Q is false.
- **Quantifiers**: They express properties or relations over all or some members of the domain:
  - $\forall x\, P(x)$: Universal quantification, states that P is true for all instances of x in the domain.
  - $\exists x\, P(x)$: Existential quantification asserts that there exists at least one instance of x in the domain for which P is true. ▪

Predicates in predicate calculus can express the properties and relations of objects, and they can be used to construct detailed and expressive mathematical statements. The expressive power of predicate calculus goes beyond that of propositional calculus due to its capability to handle variables, quantifiers, and predicates. For example, in predicate calculus we can say "All women are intelligent." instead of "Kate is intelligent.", "Sarah is intelligent.", "Eva is intelligent.", and so on [38].

Definition 7 (Prolog Rule). A Prolog rule represents an implication in the predicate calculus and forms the basis of reasoning within a Prolog program. A rule in Prolog has the following syntax:

$$a :\text{-}\ b1, b2, b3, \ldots, bn$$

Which can be interpreted in the form:

$$A \leftarrow B_1, B_2, \ldots, B_n$$

Where:

- A is the head of the rule, representing a predicate that is concluded to be true if the body is true.
- $B_1, B_2, \ldots, B_n$ constitute the body of the rule, which are predicates that must be satisfied for the head A to hold. Each B_i for $1 \leq i \leq n$ is a predicate that the Prolog system will attempt to prove true. ▪

The interpretation of the rule is that if all predicates in the body (on the right-hand side of $\leftarrow$) are true, then the predicate in the head (on the left-hand side) is also true.

A Prolog rule is a Horn clause with one positive literal A and $n \in \mathbf{N}$ negative literals $\neg B_1, \neg B_2, \ldots, \neg B_n$.

Definition 8 (Horn Clause). A Horn clause is a clause (a disjunction of literals) with at most one positive literal. In other words, a Horn clause can be in one of three forms:

1. A single positive literal (Fact): A.
2. A disjunction of negative literals (Goal): $\neg B_1 \vee \neg B_2 \vee \cdots \vee \neg B_n$.
3. A disjunction of one positive literal and any number of negative literals (Rule): $\neg B_1 \vee \neg B_2 \vee \cdots \vee \neg B_n \vee A$. ▪

The name *Horn clause* is derived from the logician Alfred Horn, who identified this subset of clauses in propositional logic [33]. Horn clauses are significant due to their computational properties: determining satisfiability for a conjunction of Horn clauses can be done in polynomial time, whereas it is NP-complete for general clauses [33, 21, 24].

Definition 9 (Prolog Fact). A fact in Prolog is a rule with no body. It is an atomic formula that is unconditionally true in the context of the program P. It may be a predicate with arity $0 \cdots n$. It has the form:

$$a.$$

Where a is a predicate. ▪

For instance, in Prolog, $parent(alice, bob)$ can be a fact stating that Alice is a parent of Bob.

Definition 10 (Prolog Atoms). In the context of Prolog, an atom is a fundamental, indivisible entity used to represent a constant value. It is a term with no internal structure discernible within Prolog. Common examples of atoms include:

- Alphanumeric strings beginning with a lowercase letter (e.g., ‘apple‘, ‘john‘).
- Strings of special characters (e.g., ‘+-/*‘).
- Strings enclosed in single, or double quotes (e.g., ‘’Hello World’‘, ‘"Hello World"‘).

Atoms can appear as arguments to predicates, as the names of functors in compound terms, or as elements in a list. ▪

In essence, atoms in Prolog are constants, function symbols, and predicate symbols in predicate calculus, providing a way to represent specific, unchanging values.

Definition 11 (Prolog Program). A Prolog program P is a finite set consisting of Prolog rules and Prolog facts. Formally, let R be the set of all possible Prolog rules and F be the set of all possible Prolog facts, then:

$$P \subseteq R \cup F$$

The program represents a knowledge base (database) in which the rules define relationships and implications between predicates, and the facts establish foundational truths. When

queried with a goal, the Prolog interpreter attempts to derive the truth of the goal using the rules and facts in P through a process of logical inference. The entirety of P provides the context and knowledge against which such queries are evaluated. We further explain the exact semantics of this mechanism in the next subsection. ▪

Essentially, a Prolog program serves as both a repository of knowledge and a mechanism for reasoning about that knowledge. For example, consider the following program:

```
parent(anne, bob).
parent(bob, carol).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This program means that Bob is Alice's parent, Carol is Bob's parent, all parents are ancestors, and all parents of ancestors are ancestors. We can then query this program to find out if a given person is the ancestor of any other.

Definition 12 (Prolog Goal). In the context of a Prolog program P, a goal G represents a logical assertion or query that seeks confirmation of its truth value within the knowledge represented by P. Formally, given the universe of all possible predicates $\mathcal{P}$, a goal G can be defined as:

$$G \subseteq \mathcal{P}$$

Each predicate within G is a proposition that the Prolog system is tasked to verify using the rules and facts from P. The intrinsic semantics of G is a request for the system to find evidence, based on the knowledge in P, that supports the truth of the assertions within G, or to deduce their falsity in the absence of such evidence.

To denote that goal G can be satisfied over program P, we use the following notation:

$$P \vdash G$$

▪

Using Prolog's syntax, a goal can be written as:

- A single predicate: $parent(anne, bob)$.
- A conjunction of multiple predicates: $parent(anne, bob), parent(bob, carol)$.

2.6.2 Resolution Semantics

Now that we defined Prolog programs, the following questions arise:

- *How does prolog evaluate the queried goals?*
- *What assumptions does it have?*
- *How does it handle arithmetic efficiently?*

"The Prolog interpreter uses pattern-directed search to find whether these queries logically follow from the contents of the database. The interpreter processes queries, searching the database in left to right depth-first order to find out whether the query is a logical consequence of the database of specifications. Prolog is primarily an interpreted language.

Some versions of Prolog run in interpretive mode only, while others allow compilation of part or all of the set of specifications for faster execution." [38]

Following the SWI-Prolog convention, we will refer to these pre-compiled rules and facts as built-in predicates, and will treat them as a special case, because their de-compilation is not supported by the meta-predicates, and need to be proven some other way, as we discuss in Section 4.3. They offer various functionalities ranging from arithmetic operations, list manipulation, to advanced features like file I/O, meta-programming, etc. They are often implemented in a low-level language and don't have the clear logical semantics that user-defined Prolog rules have.

The program - database, knowledge base - is a model of a world made up of facts, over which we can ask questions.

```
program:
    parent(anne, bob).
    parent(bob, carol).
    ancestor(X, Y) :- parent(X, Y).
    ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
query: ?- ancestor(anne,carol).
    true.
```

To reason over this "world", we must take the following assumption:

Definition 13 (Closed world assumption). In Prolog, the *closed world assumption*, also known as *negation as failure* is the assumption that "anything is false whose opposite is not provably true." [38] ▪

The computational efficiency gained by this assumption is undercut in some context by the complexity of development and the difficulty in utilising multi-valued logics. [38] Before going over the algorithm Prolog usues for evaluation, we must define a few key concepts:

Definition 14 (Variable binding). In Prolog, a variable binding means the specific values taken up by the variables in an expression. These values must be consistent for each variable. ▪

For example, in the expression `ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).` a valid variable binding must give values to the **X**,**Y**, and **Z** variables consistently.

Definition 15 (Substitution). Substitution refers to the assignment of specific terms to the variables within an expression to render it a ground expression (i.e., an expression with no free variables).

Formally, a substitution θ is represented as a set of pairs $x_1/t_1, x_2/t_2, \ldots, x_n/t_n$, where each x_i is a distinct variable and each t_i is a term. When applied to a Prolog expression, it replaces every occurrence of variable x_i with the corresponding term t_i. ▪

For instance, consider this rule and fact in Prolog:

```
parent(X, Y) :- mother(X, Y).
mother(mary, john).
```

To provide an example, if we pose a query `?- parent(mary, Y).`, Prolog seeks to unify the goal with the rule's head. During this process, a substitution $X/mary, Y/john$ is identified and applied, rendering the goal satisfiable.

To solve a query, prolog searches the knowledge base using the *backtrack* algorithm. It goes through the following steps [38]:

1. **Unification**: The Prolog interpreter starts by searching from the top of the program to find a rule or fact that matches the first goal. If it finds a match, it will "unify" the matching rule/fact's head with the goal. Unification involves finding a substitution of variables that makes two predicates identical. If no such substitution exists, unification fails.

2. **Recursive Evaluation**: If the goal was unified with a rule (as opposed to a simple fact), then the body of that rule becomes the new set of goals that need to be satisfied. Prolog then recursively attempts to satisfy each of these sub-goals.

3. **Backtracking**: If Prolog fails to satisfy any of the sub-goals, it will backtrack, undoing any bindings of variables it made during the failed attempt and then trying alternative rules or facts. This process continues until either the goal is satisfied, or all possible rules and facts have been tried and found wanting.

There are two other attributes of prolog resolutions to note, which which will help us confirm the correctness of our approach, both for the policy and the proof generation [9].

Soundness If Prolog concludes that a goal is true, it must indeed be a logical consequence of the program. This ensures reliability. If prolog tells us that a fact is true, we can be sure it is. For example, consider a knowledge base that has the facts "All men are mortal" and "Socrates is a man." If you query if "Socrates is mortal", a sound system would rightly affirm this.

Completeness If a goal is a logical consequence of the program, Prolog will eventually conclude it's true, given infinite time and memory. This ensures exhaustiveness, meaning Prolog will return all results of a query with enough resources.

In summary, the essence of Prolog's evaluation mechanism is a depth-first search with backtracking, trying to unify the queried goal with known facts and rules in the program. It operates under specific assumptions, has some limitations but is empowered by a set of built-in predicates. When it comes to creating proofs of evaluation, special considerations have to be made, especially for built-in predicates to ensure that the proof generated is logically sound.

Chapter 3

Related Works

3.1 Circuitree: A Datalog Reasoner in Zero-Knowledge

Circuitree [27] is a Datalog reasoner in Zero-Knowledge, meaning it can reason over Datalog facts and rules without revealing the underlying facts to the verifier. It operates on a Datalog dataset, and encrypted private inputs. Its main advantages are that it operates over a high-level declarative language, which makes it a very good candidate for defining access-control policies. It is a really fast and well-optimized Zero-Knowledge System, both in terms of proof generation and proof verification time. It can be integrated into other applications and toolchains. Goodden et al. highlight many possible use cases for their system, from which we would specifically feature that **Circuitree** may serve as an alternative approach for The W3C's *linked data proofs*[1], from which Verifiable Credentials would greatly benefit. We were greatly inspired by their approach, however, there are some major differences between our systems.

3.1.1 Approach to Datalog reasoning in Zero-Knowledge

The approach of **Circuitree** can be outlined as following Datalog's bottom up resolution semantics, and then manually constructing an R1CS using Bulletproofs, by adding constraints based on the steps the resolution takes, as opposed to proving the imperative run of a foreign resolution engine. This approach allows Godden et al. to provide groundbreaking performance, while using a declarative language. Same as Datalog's resolution semantics, their solution is iterative, meaning the application of the rules happens in a loop. The main use case, which they use as an example through their paper, is a COVID-19 certificate based access control system, which is a great example for the possible uses of declarative policies. The computational complexity of their implementation is well defined, and they offer some insightful optimizations. Of course, there are some limitations, both some inherent to Datalog, and some because of the current capabilities and performance of Zero-Knowledge protocols, specifically Bulletproofs. One aim of Godden et al. is to mend the shortcomings of pure Datalog by implementing arithmetic in their system.

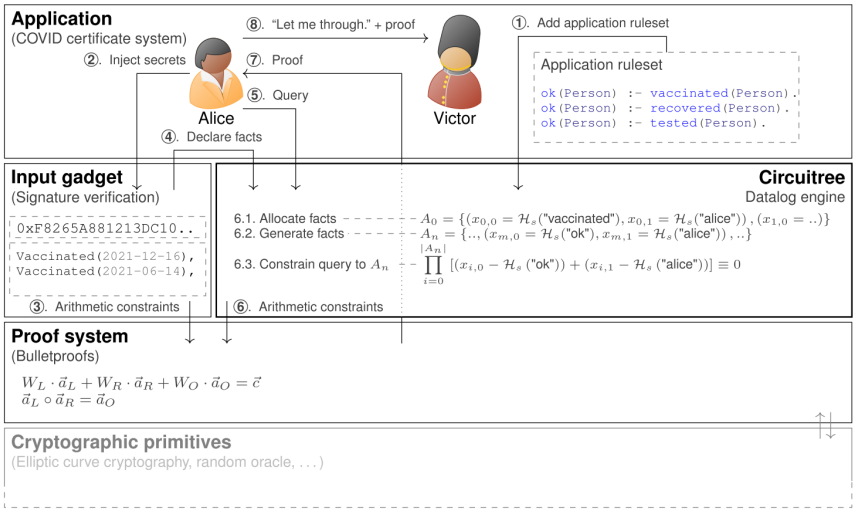


Figure 3.1: Relations between application, Circuitree, proof system and underlying cryptographic primitives. Source: Circuitree: A Datalog Reasoner in Zero-Knowledge, page 2 [27]

3.2 Efficient Representation of Numerical Optimization Problems for SNARKs

In their paper, Angel et al. propose **Otti** [8], a general-purpose zk-SNARK compiler which provides support for numerical optimization problems, such as Linear Programming, Semi-Definite Programming, and a class of Stochastic Gradient Descent instances. As input, **Otti** takes arbitrary optimization problems defined in a subset of C, and produces an R1CS optimality checker, along with a witness to this checker using a foreign solver, meaning it can use state-of-the art solvers, while still providing a Zero-Knowledge Proof. It was tested with the Spartan proof system, and provided orders of magnitude faster proof times, because "on average, proof generation for LP and SDP is 30–40× more expensive than finding the solutions themselves using existing solvers."

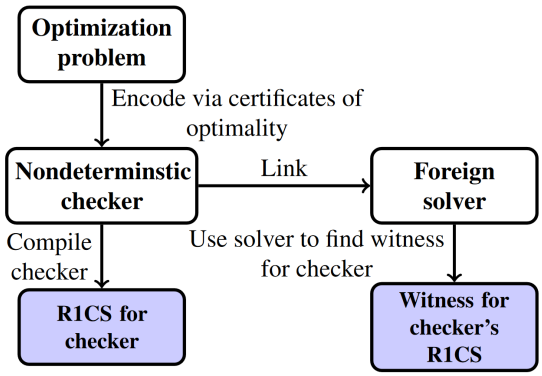


Figure 3.2: High-level workflow of Otti. Source: Efficient Representation of Numerical Optimization Problems for SNARKs page 4227

3.2.1 Brief overview of Otti

The core idea of **Otti** is to separate the evaluation of a CSP, and the proof of the optimality of the solution. To this end, they utilize *certificates of optionality*, which is a computational structure, using which it is possible to verify the optimality of CSP solutions. They also introduce *probabilistic certificates of optimality*, which is a construct proving the estimated optimality of Stochastic Gradient Descents.

Angel et al. verify, test, and evaluate **Otti** for all stated usecases in detail, and on a wide array of datasets. They note very significant performance improvement compared to existing systems.

Our approach inherits methods from both **Circiutree** and **Otti**. Circuitree defines the architecture of a ZKP system proving the evaluation of a declarative program, focusing on the policy evaluation use-case. Otti uses computational structure of certificates of optimality and foreign solvers to prove optimaltiy of a CSP. Simmilarly, we use the computational structure of a proof-tree, and use the SWI-PL foreign envirovment in our system.

Chapter 4

A ZKP-based approach for Self-evaluated Policies

Our approach to Self-evaluated Policies leverages existing techniques from previous research (Chapter 3), as well as several unique solutions to the problem of Zero-Knowledge policy evaluation. To prove that a policy written in Prolog has been evaluated with the private inputs, we use a *proof-tree*, which is a computational structure that can be efficiently manufactured and is a certificate that the policy has been evaluated correctly. [49]

After a proof tree is generated using a foreign Prolog implementation, we feed it into a "checker" arithmetic circuit (Chapter 5), which ensures its validity, i.e., it hasn't been tampered with or maliciously generated, and generates a Zero-Knowledge proof, showing the fact that the Prover has evaluated the policy, and the evaluation finished with some result. This process is visualized by Figure 4.1 and 4.2.

We hope that by enabling the use of any high-performance Prolog solver, like SWI-Prolog [3], we make our system more modular and hopefully achieve better performance than by reimplementing a solver in arithmetic circuits.

4.1 Adding provable private facts to a knowledge base

To reason over private facts in an external Prolog environment, we must be able to prove that the private facts that come from Verifiable Credentials are true. In SSI, the truth of the facts is proven by them being signed by a trusted Issuer. There are existing Zero-Knowledge systems that can prove that a message has been signed without revealing the underlying message. [45] We outline a few approaches for how private facts may be extracted from the Verifiable Credential and how their proof may be integrated into our system:

Extracting facts from a Verifiable Credential (VC)

Storing Prolog facts in VC One clean solution would be to store the Prolog facts in the VC by themselves. This obviously puts the burden on the Issuer to include them in the Verifiable Credential in the first place. This simply means the JSON-LD contains a field that contains the Prolog facts in the Prolog syntax.

Converting JSON-LD to Prolog facts Another approach is to convert JSON-LD into Prolog syntax. This poses the challenge that then the converting algorithm's execution should be provable in some way. This could be done using recursive proofs, ZKP gadgets, or by extending the existing proving logic.

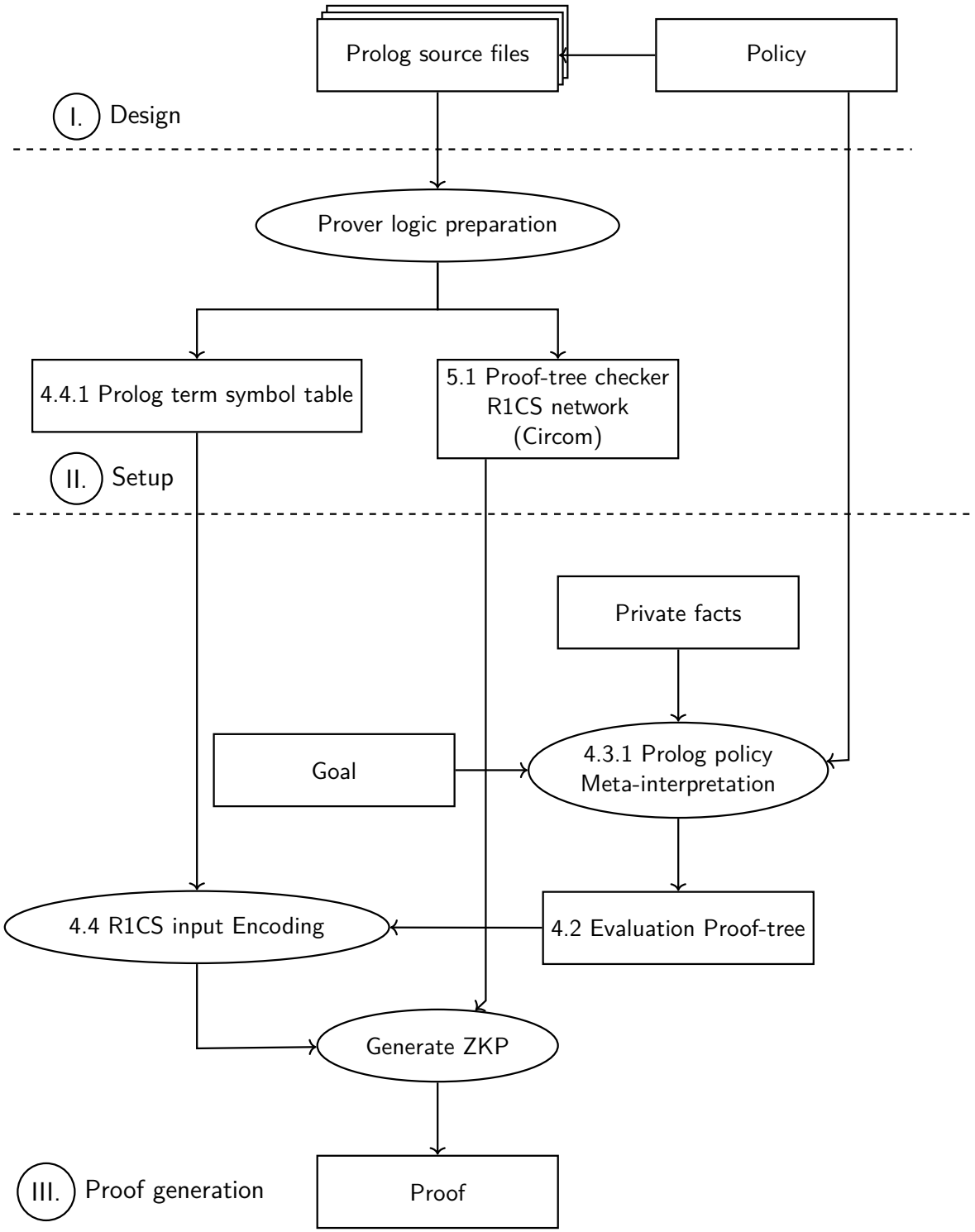


Figure 4.1: Overview of our proposed toolchain

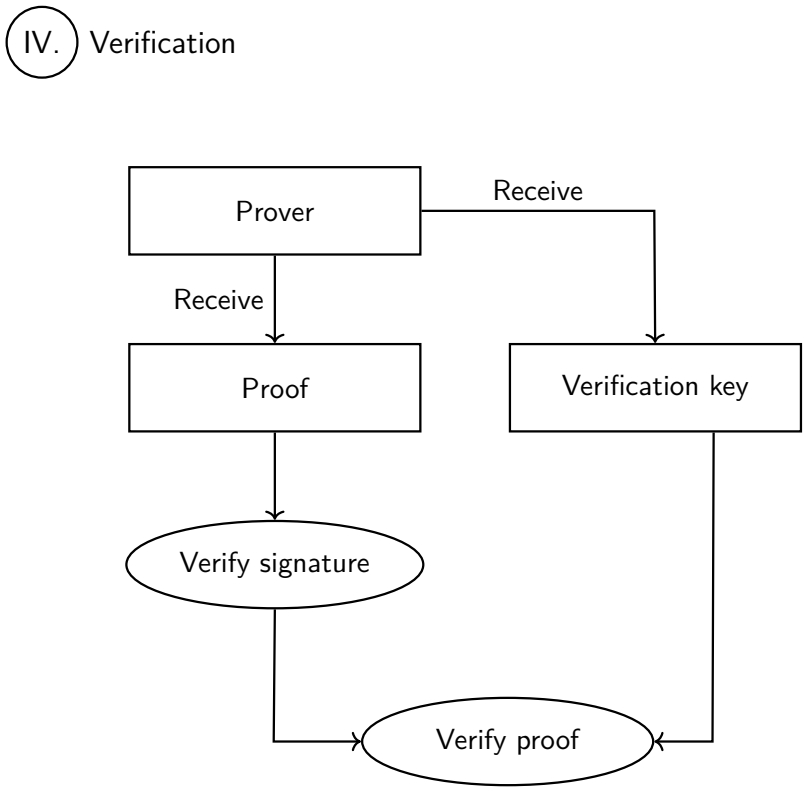


Figure 4.2: Overview of the verification side

Integrating proof of private facts into our approach

Extend Proof-tree checker 5.1 R1CS network As we are defining our Arithmetic Circuits with Circom, which takes arbitrary input, an adequate first idea would be to extend our proof-tree checking logic by verifying the prolog facts or their correct conversion. This isn't trivial, as writing arithmetic circuits, even through the abstraction of Circom, is still challenging. Nevertheless, according to our experience with Circom, this approach is certainly viable.

Linking proof-gadgets The linking of small specialized *proof gadgets* has been proven viable by **LegoSNARK** [20]. Specifically, leveraging their proof system, we would be able to integrate our circuits along with a fact verification logic.

We think that this architectural decision, as in how do we integrate JSON-LD facts into our Zero-Knowledge system, is mainly a question of how we should integrate within other systems, specifically how do we fit into an application within an SSI ecosystem. As our paper concentrates on the core functions and optimizations of Self-evaluated Policies, we defer a more extensive analysis of these approaches to future work.

If the Prover has their private and public facts (from a Verifiable Credential), they can be joined with the policy definition into a single knowledge base, over which they can reason. This means that they can evaluate whether they satisfy specific expectations of the policy or can perform calculations defined within.

4.2 Proof trees

For proving the correct evaluation of a Prolog program, we utilize proof trees, also known as resolution trees, as defined by Russel and Norvig. [49]

Definition 16 (Proof Tree). Let P be a program and G a goal. If $P \nvDash G$, then the proof tree for G is empty. Otherwise, it is a tree t recursively defined as follows:

- if there is a fact f in P and a substitution θ such that $G_\theta = f_\theta$, then G_θ is a leaf of t .
- otherwise there must be a clause $H \leftarrow B_1, \ldots, B_n \in P$ and a substitution θ' such that $H\theta' = G\theta'$ and $P \vDash B_j\theta' \forall j$, $G\theta'$ is the root of t and there is a subtree of t for each $\forall B_j\theta'$ which is a proof tree for $B_j\theta'$. ▪

Paraphrasing the definition, a proof tree is a tree whose nodes are rules defined in the program and whose leaves are facts in the knowledge base. Each node is the head of a Horn clause, and its children are its body.

A proof tree proves that goal G can be reached over program P because the verifier can trace back each unification and each fact to the knowledge base. [49]

Other approaches for proving the evaluation of a program - both declarative and imperative - often rely on explicitly retracing the steps of the execution of the program from beginning to end and adding a constraint to the constraint system with each step from a valid starting state, thus inductively proving that the program executed, and therefore, the evaluation is correct. Our approach differs in a key way, specifically that by creating a proof tree first and then proving its correctness, we decouple the execution and proof generation phases.

A valid goal may have more than one corresponding proof tree, meaning there are multiple ways to unify the predicates and facts. It is possible to produce and use all proof trees that may exist for a given goal for a given program, by, for example, using a visitor pattern.[47] For our purposes, we want to verify that the goal can be reached. To this end, in our paper, we will consider only one of the proof trees because a single proof tree is sufficient proof that a goal can be reached.

An example proof tree can be seen on figure 4.3.

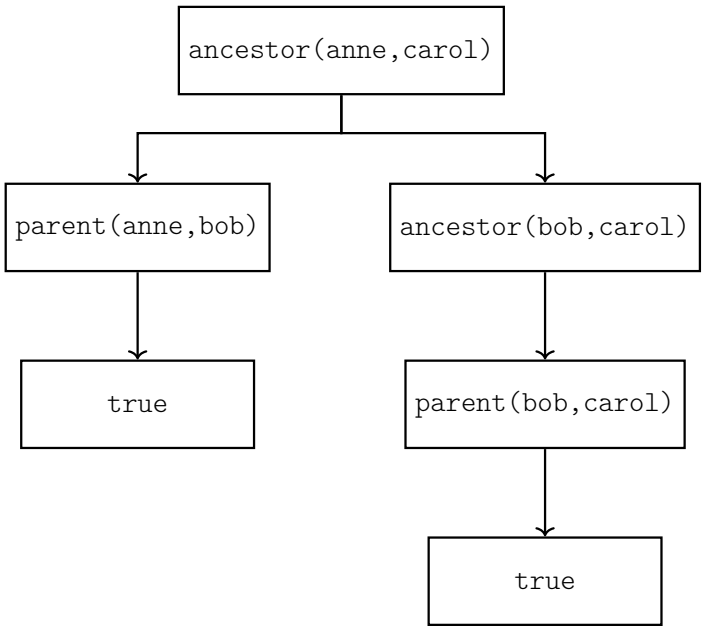


Figure 4.3: Example proof tree

4.2.1 Pruning Proof Trees

To simplify proof trees or increase their expressive power, especially in the context of ZKPs , we use a technique used by Passerini et al. [47], the pruning of proof trees.

During the creation of a proof of evaluation, built-in predicates pose challenges. As mentioned, they are pre-compiled and thus unavailable to be reasoned over using the built-in meta-predicates. They often have diverging or prolog-variant specific semantics, but we can define a subset of them, which have clear meaning regardless of the different prolog-variants, such as *=/2* for unification, *is/2* for arithmetics, *findall/3* for knowledge list creation, and so on.

One approach is to replace them with equivalent logical constructs where possible. Another approach, especially for complex built-ins, is to trust their operation and treat them as "black boxes" with a known input-output behaviour.

To this end, we define a *pruned proof tree*, where the leaves are not only facts of the knowledge base, but built-in predicates as logical statements, which evaluate to true. Essentially expanding the previous definition:

- If G is a built in predicate, there must be substitution θ'' such that $G\theta'' \vdash P$ and $G\theta'' \sim true$, then G is a leaf of t.

During the verification of a proof tree which contains built-in predicates, we must, of course, check not only if they are the subgoal of the previous goal but that they are indeed logically true.

This technique allows us to use built-in predicates in our proof trees and, thus, in our policy programs, which enables not only the use of arithmetic but also many other precompiled functions prolog has if we are able to check their truth value during verification.

This process is visualized on figure 4.4.

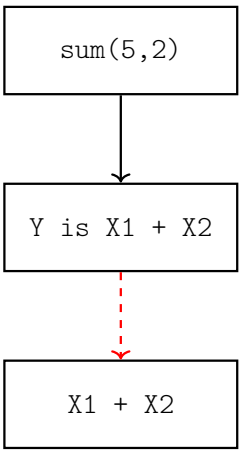


Figure 4.4: Example proof tree pruning

4.3 Generating proof-trees using meta-interpretation

Our approach relies on generating a proof tree for a Prolog goal. While there are imperative ways for creating such trees, e.g. Tau-Prolog's derivation tree [4], we chose meta-interpretation as our method of choice because Prolog is very well-suited for meta-interpretation, and specifically for proof-tree generation; and also to demonstrate how can Prolog be used inside a ZKP toolchain not just as a policy language, but as an active part of the toolchain. In this section, we define meta-interpretation, show a couple of examples in Prolog, and propose how proof trees may be generated with this technique, specifically in the context of providing proof for program evaluation. For our understanding of meta-interpretation and, in particular, Prolog meta-interpretation, we rely on the definitions and techniques of Markus Triska's The *Power of Prolog* and chapter 6 of George F. Luger's *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java.* [53, 38]

4.3.1 Meta-interpretation in prolog

Interpretation is a fundamental concept in computer science, encompassing the evaluation of programs, and it plays a vital role both theoretically and practically. One instance of interpretation is when a program evaluates other programs, as is the case with language interpreters. For instance, a program reading and adjusting its settings from a configuration file interprets the "configuration language" encoded within that file. Another example would be interpreted scripting languages, such as JavaSricpt or Python.

Meta-interpretation, on the other hand, introduces a layer of abstraction by having an interpreter for a language similar to, or even identical to, its own implementation language. In essence, meta-interpretation allows an interpreter to evaluate programs written in the same language as itself. This concept leads to unique possibilities for introspection, recursion, and self-evaluation within the interpreter.

Definitions

object-level the level of the program being interpreted

meta-level the level of the meta-interpreter

meta-circular a meta-interpreter that can interpret its own source code

absorption a meta-interpreter uses an implicit language feature

reification a meta-interpreter makes a language feature explicit, observable

meta-call when a goal is dynamically invoked: $Goal$. $call(Goal)$. This is available in Prolog out of the box.

reflection and introspection : a program's ability to examine itself, available in many other languages, e.g. Java. In Prolog, the main tool for introspection, among others, is also a meta-call: $clause(Goal, Body)$.

Prolog stands out as particularly well-suited for meta-interpretation for several reasons.

- Firstly, Prolog programs can be naturally represented as Prolog terms, making them easily inspectable and manipulable using built-in mechanisms. This intrinsic homoiconic nature, shared with languages like Lisp and machine language, simplifies the meta-interpreter's task[50].
- Secondly, Prolog's implicit computation strategy and support for all-solutions predicates enable concise specifications within interpreters. This feature is not exclusive to Prolog, as languages like Tcl and PostScript also possess it, but Prolog's seamless integration is notable.
- Thirdly, Prolog allows variables from the object-level (the program being interpreted) to be treated as variables on the meta-level (the interpreter). This feature enables the interpreter to delegate the handling of the interpreted program's binding environment to the underlying Prolog engine. In essence, Prolog's simplicity, where the primary construct is in the form of $Head \leftarrow Body$ facilitates this alignment of object-level and meta-level features, making it unique among languages.

By using established meta-interpretation techniques [38], we can re-define the semantics of Prolog, such that - in a program written in the sublanguage understood by the meta-interpreter - we can reason about any true statement. The method can be thought of as asking Prolog "how" it evaluates the current goal recursively. To satisfy the definition, we can use the $clause/2$ meta-predicate, which takes a goal term, and if it evaluates to true, gives the body of the valid unification, which are the predicates the goal unifies to.

If the body is the *true* fact, the goal is a fact and satisfies the first part of the definition.

```
program:
    parent(anne,bob).

?- clause(parent(anne,bob),B)
    B = true
```

If the body contains other predicates, then the goal is a rule and satisfies the second part of the definition. Then recursively calling $clause/2$ on the body predicates, we can produce the corresponding subtrees.

```
program:
    parent(anne, bob).
    parent(bob, carol).
    ancestor(X, Y) :- parent(X, Y).
    ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
?- clause(ancestor(anne,carol),B)
    B = (parent(anne,bob), ancestor(bob,carol))
```

At the core of a basic Prolog meta-interpreter lies the ability to prove the evaluation of a Prolog program. This is accomplished by interpreting the queried Prolog program and

extracting the states represented as nodes within the proof tree. One such foundational meta-interpreter is the "vanilla" meta-interpreter.

The vanilla meta-interpreter begins with the base case, where it asserts that any query for "true" always holds. It then proceeds to handle conjunctions, representing them as "A, B," by recursively invoking itself for both A and B. In the general case, it employs the "clause(G, Body)" mechanism to extract clauses and recursively interpret the body. [53]

```
vanilla meta-interpreter:
    mi(true).
    mi((A,B)) :-
        mi(A),
        mi(B).
    mi(Goal) :-
        Goal \= true,
        Goal \= (_,_),
        clause(Goal, Body),
        mi(Body).
```

Despite its simplicity, the vanilla meta-interpreter has limitations, including absorbing most language functions, generating false choice points, and struggling to handle built-in predicates due to their often private and precompiled nature within specific Prolog implementations. Nevertheless, this basic structure serves as a foundation for Prolog meta-interpreters.

Prolog meta-interpreters can be significantly enhanced in terms of performance, complexity, and simplicity(with as few as two clauses by modifying the representation of the object-level program). This opens up possibilities for further optimizing and refining the interpretation process, which we demonstrate in the next section by defining a meta-interpreter that is able to generate proof trees to be used in our toolchain.

4.3.2 Proof-tree generation

Utilizing prolog's meta-interpretation capabilities[53], we can generate proof trees that can stand as the basis of verification for the evaluation of a program. To produce a proof tree, a meta-interpreter needs to reify the unification of a goal, which can be done with the *clause*/2 predicate, as shown before, then store each subgoal in the correct tree structure and also reify the recursive unification of the subgoals, meaning the body of the original goal.

Using the following techniques [38], we build the proof tree recursively, with the proofs of the goals woven together into the proof tree. The proof of *true* fact is *true*, and the proof of two predicates *and*-ed together is the list of their proofs. The proof of a disjunction is either one or the other subgoal's proof. By these principles, this is what a basic meta-interpreter capable of generating proof-trees may look like:

```
mi_proof_tree(true, [true]).

mi_proof_tree((A, B), Proof) :-
    mi_proof_tree(A, ProofA),
    mi_proof_tree(B, ProofB),
    append([ProofA, ProofB], Proof).

mi_proof_tree((A; _), Proof) :-
    mi_proof_tree(A, Proof).

mi_proof_tree((_; B), Proof) :-
    mi_proof_tree(B, Proof).

mi_proof_tree(Goal, [Goal|Proof]) :-
    clause(Goal, Body),
    mi_proof_tree(Body, Proof).
```

These reifications - truth checking, conjunction, disjunction, and of course, the unification of generic goals - are viable for simple programs but are far from the complete set. For example, a meta-interpreter may reify negation, cuts, and built-in predicates such as $is/2$(arithmetic) and $findall/3$. To this end, we define a sublanguage which our meta-interpreter understands, thus defining the exact expressiveness of our proven programs. We discuss these considerations in the next subsection.

4.3.3 Sub-language definition and other considerations

A meta-interpreter capable of generating proof trees can reason over programs which only contain language features which it reifies, and we can only verify the inferences of a proof tree if the semantics of the statements are well-defined - for predicates in the program the definition is explicit, for built-in predicates, no so much.

To avoid running into undefined or under-defined behaviour using our approach, we posit that each toolchain - i.e. meta-interpreter, encoder, prover generator - is able to reason over a well-defined sub-language of Prolog (Section 4.3.3). The semantics need to be consistent in each step; a proof tree of a predicate must show unification the same way the prover generator accepts them - e.g. If the body of an indication is returned as an array from the meta-interpreter, the prover generator shouldn't try to understand them as a right-recursive tree -, the meta-interpreter must only prune the predicates the prover generator is defined to understand the semantics of - e.g. the meta interpreter prunes `is/2` and the prover generator understands that the predicate `N is 1 + 1` built-in means N equals two -, and more broadly, should operate on a *white-list* like approach, meaning only those inputs are accepted in each tool, for which behaviour is clearly and explicitly defined.

In essence, a **sub-language definition** for a toolchain is a dictionary of language constructs, with a clear definition of their semantics corresponding to each one of them, which the tools understand.

A key decision in how we choose to define our sub-language is whether we include and thus allow negation in Prolog programs. Prolog is capable of handling negation all by itself, but it breaks a crucial

One other consideration is the use of SLD trees - which are another established representation of Prolog program evaluation - , and why we avoid them. As stated by Passerini et al. [47], SLD trees are vastly more complex, more unstructured, and contain - for our purposes - unimportant information, such as information about failed paths. The goal of our approach is to prove that a prolog goal is a consequence of a policy program, and therefore because proof trees provide sufficient proof of that fact, we chose to use them over SLD trees for

their simplicity, and with the assumption that a smaller input tree corresponds to a smaller prover generation and proving time.

4.4 Representation of prolog proof trees

To utilize the Prolog proof trees derived from our meta-interpreter within a ZKP framework, it is imperative to encode the tree into a format compatible with the designated toolchain. In the subsequent section, we delineate the methodology employed for the encoding of the Prolog resolution trees.

4.4.1 Encoding prolog terms and predicates

The smallest components of the Prolog resolution trees are the terms and the predicates. Each term and predicate is systematically assigned a distinct numerical value through enumeration. A predicate is characterized by an array of integers, wherein the initial integer represents the predicate's identifier, followed by the numbers corresponding to the predicate's arguments. Additionally, specific terms such as "true" and "false" and arithmetic operators (e.g., "+" "-" "is" or "/=") are encoded analogously to predicates.

For uniformity, the encoded predicate arrays must adhere to a consistent length. When a predicate array contains fewer arguments than the standard length, zero-padding should be applied to denote vacant slots. However, incorporating list arguments directly would result in excessively large predicate arrays. To circumvent this, list arguments are extracted from predicates and stored in a designated lookup table storage structure. This lookup table retains the entirety of the list data without increasing the size of the predicates. Each extracted list argument is substituted with its corresponding index within the lookup table. It's essential to highlight that this lookup table is also maintained at a uniform size for standardization.

4.4.1.1 Encoding Example

Consider the previously mentioned Prolog program:

```
parent(anne, bob).
parent(bob, carol).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

The respective encoding for the terms and predicates are as follows:

anne	→	1
bob	→	2
carol	→	3
parent	→	4
ancestor	→	5
true	→	6
false	→	7

Table 4.1: Term and predicate symbol table for the Alice-ancestor example

Given this encoding:

→ The Prolog term parent(anne,bob)parent(anne,bob) is translated to the array: $[4, 1, 2]$.

→ The special term "true" is encoded as $[6, 0, 0]$.

This encoding provides a structured numerical representation of the Prolog terms and predicates.

4.4.2 Encoding the tree nodes

In the constructed resolution tree, every node consists of two components: the 'goal' and its corresponding 'unification' list. The 'goal' refers to a predicate, which is encoded as delineated in 4.4.1. Meanwhile, the 'unification' list component comprises an array containing arrays of encoded predicates. It's important to ensure that the number of unification lists present in each node remains consistent throughout the entire resolution tree.

4.4.2.1 Example tree node

Within the framework of our resolution tree, let's examine a specific node for better comprehension. Let's use the same Prolog program as in our previous example in section 4.4.1.1. Suppose the Goal is represented as $[5, 1, 3]$, which translates to the predicate `parent(anne,carol)`. Moving on to the Unifications, they are presented as $[[5, 2, 3], [4, 1, 2]]$. These encoded sequences correspond to the predicate `ancestor(bob,carol)` and `parent(anne,bob)`, respectively. In essence, this node's primary objective is to determine if Anne is Carol's parent, as depicted by the encoded predicate $[5, 1, 3]$. The unifications offer potential resolutions, signifying relationships such as Bob being an ancestor of Carol and Anne being Bob's parent.

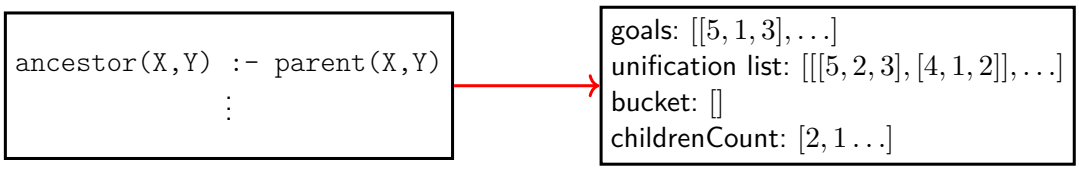


Figure 4.5: Encoding represenattion

4.4.3 Representation of the Proof Tree

In our computational framework, the proof tree is transformed into a linear array consisting of encoded data structures. Sequentially, each node in the array is immediately followed by its child nodes. This linearization process employs a Breadth-First Search traversal of the proof tree. To accommodate the variability in the number of children per node, it is essential to maintain an auxiliary array that records the child count for each respective node.

Built-in predicates are represented similarly; they can be recognized by the fact that they have no children but don't unify with "true". The checker circuit recognizes them because their semantics must programmed for them to work. When they are recognized, the checker activates a constraint equal to their semantics.

The final representation is visualized on figure 4.5

Chapter 5

Constraint system representation of proof tree checking

5.1 Algorithm for Verifying the Proof Tree

The algorithm's primary objective is to ensure that a given Prolog proof tree accurately represents valid resolutions in a given Prolog program using ZKPs. By encoding this tree as an array depicting the complete BFS path, we ensure efficient processing and verification. The core steps of the algorithm are outlined below:

5.1.1 Node Validation

To guarantee the integrity of the proof tree, each node must accurately reflect its intended Prolog goal and the associated unifications. This stage is highlighted in figure 5.1.

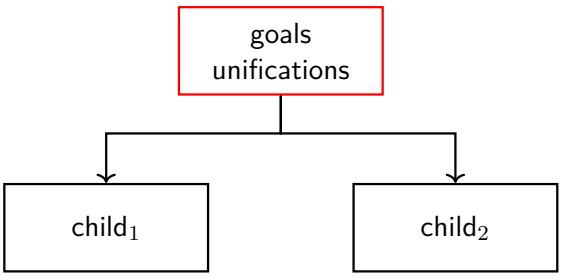


Figure 5.1: Node verification highlight

For every node in the tree, the algorithm inspects the goal (head) and its corresponding unifications (body). If the goal is represented as a rule in the Prolog program, the aim is to ensure a perfect match between the parameters of both. For instance, if a node presents a goal represented as a(x,y), and its corresponding unification is denoted by b(x,y), then the parameters (x,y) should be identical across both the goal and the unification.

If the goal is a predicate that is known to exist with some arguments in the knowledge base, the algorithm must cross-check if the goal with the correct arguments is in the knowledge base.

This is also the phase where arithmetic operations are checked from the unification list. This ensures that the values used in the proof tree are indeed correct.

This part of the verification procedure is demonstrated with pseudo-code in Algorithm 1.

Algorithm 1 The algorithm to check if a tree node was valid with the prolog program

```
1: procedure checkProofTreeNode(proof-tree node)
2:     goal ← the goal of the node
3:     unification ← the list of unifications
4:     knowledgeBase ← The names of the goals that should be checked against the knowledge base
5:     builtIns ← The names of the predicates that have their semantics defined
6:     if goal.predicateName in knowledgeBase then
7:         assert isValidInKnowledgeBase(goal)
8:     if goal.predicateName in knowledgeBase then
9:         assert isValidBuiltIn(goal, knowledgeBase)
10:    for i in 0..len(unification) do
11:        for j in j..len(unification) do
12:            if unification[i].predicateName == unification[j].predicateName then
13:                for k in 0..len(unification) do
14:                    for l in 0..len(unification) do
15:                        if unification[i][k].variableName == unification[j][l].variableName then
16:                            assert unification[i][k].value == unification[j][l].value
17:    return true
```

5.1.1.1 Example

Let's take our previous example Prolog program about the ancestors of `bob` and `carol`, and assume that our goal is to prove that `anne` is the ancestor of `carol`. In the root of the proof tree for this statement, we will see our goal and its unifications. In this node, we need to ensure that all the variables are filled in with the same parameters as in its definition. This process is demonstrated in figure 5.2. Essentially, we need to make sure that if the first parameter of the goal is `anne`, then the first parameter in the first part of the unification body is also `anne`, and so on.

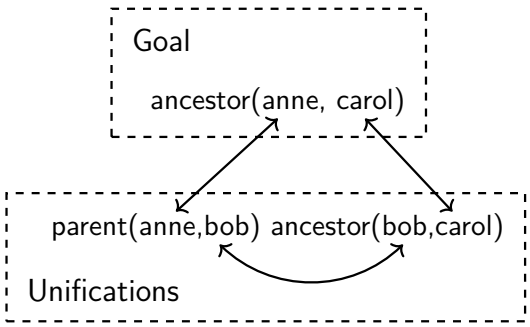


Figure 5.2: Node checking example

5.1.2 Transition Verification

To ensure the logical progression and coherence of the proof tree, the algorithm verifies the transitions between parent nodes and their respective child nodes. This part is highlighted in figure 5.3.

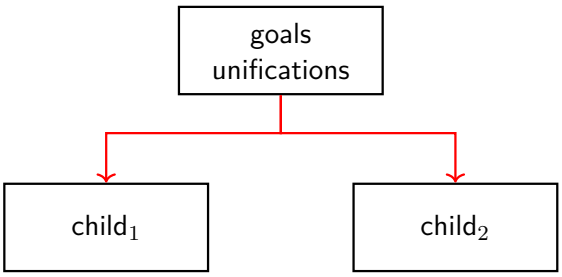


Figure 5.3: Transition verification highlight

Each unification in a parent node should correctly transition to its child nodes in the tree. For example, if a parent node has a unification defined as b(x,y), the subsequent child node should accurately present a goal of b(x,y). This check ensures the logical flow and accuracy of goal transitions within the tree.

Algorithm 2 The algorithm to check if a tree node was valid with the prolog program

```
1: procedure checkTransition(proof-tree root)
2:     unification ← root.unification
3:     children ← root.children
4:     for i in 0..len(children) do
5:         assert children[i].goal.name == unification[i].name
6:         assert children[i].goal.args.len() == unification[i].args.len()
7:         for j in 0..len(children[i].goal.args) do
8:             assert children[i].goal.args[j] == unification[i].args[j]
9:     return true
```

5.1.3 Recursive Processing

To guarantee the holistic verification of the entire proof tree, the algorithm adopts a recursive approach.

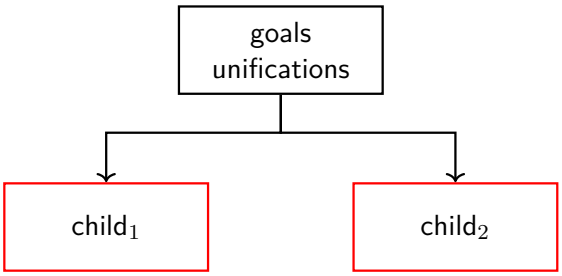


Figure 5.4: Recursive verification highlight

After verifying a node and its immediate transitions, the algorithm proceeds to its child nodes. By recursively applying the steps of node validation and transition verification, the algorithm ensures that each tree segment, from root to leaves, adheres to the intended Prolog program's logic and structure. This process part is highlighted in figure 5.4.

The procedure is summarized in algorithm 3.

Algorithm 3 Recursive processing of proof tree nodes

```
1: procedure processNode(proof-tree node)
2:     children ← the children of the current node
3:     checkProofTreeNode(node)
4:     checkTransition(node)
5:     for i in 0...len(children) do
6:         checkProofTreeNode(children[i])
7:         checkTransition(children[i])
```

5.1.4 Additional constraints

To further bolster the robustness of the validation process, several additional constraints have been posited to certify that the proof tree retains a 'well-structured' demeanour:

Constraint on the root node: It's essential to ensure that the root node does not have an empty clause as its goal. This stipulation underpins the foundational logic driving the algorithm.

Constraint Concerning Empty Nodes: Should a node be void of content, its child nodes must be similarly empty, thereby preserving the entire tree's structural integrity and logical coherence.

5.1.5 Verifying built-in predicates

Because their semantics need to be defined in advance, as stated in Section 4.2.1, the checker circuit can recognize them, and activate a constraint equal to their semantics.

For example, for the predicate `A is B + C`, the corresponding constraint is `A === B + C`.

Chapter 6

Circom-based toolchain and evaluation

We implemented a proof-of-concept toolchain for our approach. Specifically, we created a policy for the use case defined in Section 2.1, designed a meta-interpreter along the requirements laid out in Section 4.3, created a symbol table logic along the lines of 4.4.1, along with proof-tree encoding 4.4, and finally a Circom code generator, which can create policy-specific Proof-tree checker Circom programs, which in turn compile into R1CS, through which we realise the demands of Chapter 5.

6.1 Meta-interpreter

The meta-interpreter we designed is capable of generating proof trees for Prolog programs. This proof tree is structured as a recursive dictionary, where each node contains the current goal, and its children -stored as a list - are the predicates that unify with it.

The interpreted programs can contain conjunction, disjunction, built-in predicates, and, of course, general goals because the Meta-interpreter reifies these language features. The specific behaviour for each is the following:

Conjunction The proof trees of the conjoined goals are joined because both are unified with the previous goal.

Disjunction From the disjoined proof trees, the Meta-interpreter selects the unified one, which defaults to the first if both can be unified.

Built-in predicates The built-in predicates are pruned, meaning they are marked as special leaves, which have to be checked with a different logic than normal facts, specifically by their exact semantics.

General goals For general goals, the Meta-interpreter unifies the body of the goal predicate. If they are facts, they unify with *true*.

The meta-interpreter only obtains a single correct proof tree because that is sufficient proof for a program evaluation. Of course, the consideration that has to be made during policy design is that if the given goal has variables, either all evaluations result in the same variable substitutions or all variable substitutions are accepted if they are the logical consequences of the program.

The meta-interpretation returns a proof tree, which is converted to JSON format, to be processed by the encoding logic and then the Circom checker. We achieve this by using the previously mentioned meta-interpretation techniques, meta-calls, and SWI-Prolog's JSON library. Because Prolog is *sound* and *complete*, the Meta-interpreter inherits these properties, meaning that if a goal is the consequence of the interpreted program, the Meta-interpreter will find its proof tree. It will never find a proof tree for a goal that is not the consequence of the program. Also, we tried it on a few Prolog programs, like the *alice-ancestor* example, the *bob-grandpa* example, and standard predicates, such as *sum*, and of course, the policy for our use-case.

During these evaluations, the Meta-interpreter reached the correct conclusions within milliseconds.

6.2 Encoding logic

In our ZKP framework, a dedicated encoding mechanism is deployed to construct a symbol table explicitly tailored to represent Prolog terms. This procedure commences with the initial parsing of the Prolog source file. Subsequently, each term within the program is assigned a unique identifier through an enumeration-based approach. This process is initiated by identifying and cataloguing all atoms, followed by progression to predicate names, and culminates in the finalisation with variable terms. This systematic methodology ensures an adequate characterisation of Prolog terms within the ZKP context.

Utilising this technique, we guarantee each term has a unique identifier, permitting direct referencing via its assigned number. Our decision to adopt enumeration stems from its inherent simplicity, offering a straightforward process to construct a collision-free symbol table.

6.3 Circom program generator

The Circom code generator orchestrates the assembly of a program tailored for generating zero-knowledge proofs. Initially, it undertakes the task of parsing the prolog program. After this, it forms the symbol table, a process delineated in section 6.2.

To realise the whole assembly of the ZKP program, the code generator leverages a pre-established template file. At the Circom layer, the generator formulates a distinct template for each Prolog clause, verifying every rule embedded within said clause. The methodology for this is expounded upon in section 5.1.1.

For the instantiation of these templates, we've conceptualised a distinct component template. This entity is entrusted with the task of invoking the specific component requisite for the evaluation of a particular node, a responsibility executed by our code generator.

Moreover, the component incorporates the knowledge base of the Prolog program directly into the resultant code, embodied within a standalone knowledge-base checker component.

6.4 Performance evaluation

In our empirical analysis, we evaluated the proposed methodology utilising a sample policy derived from the use case delineated in Section 2.1. The Prolog representation of this policy

comprises multiple clauses, encompasses list manipulation, executes arithmetic functions, and integrates both disjunctive and conjunctive operations. Within this policy's proof tree, the observed maximal depth was 15, and the highest branching factor was 13. Additionally, an encoded predicate contained five constituent elements.

In the given experiment, the generation of a zero-knowledge proof was completed in approximately 13 minutes. The arithmetic circuit encompassed over 1.1 million constraints. This result highlights the effectiveness and robustness of our toolchain in managing tasks of significant complexity and magnitude. Furthermore, the demonstrated performance confirms its applicability for our specific use case.

During our testing, we observed that the number of constraints grows linearly based on the maximum number of nodes in the proof tree.

6.4.1 Comparison with Other Systems

To objectively evaluate the efficiency of our toolchain, we draw a comparative analysis with analogous systems, specifically Circuitree [35], detailed in section 3.1. Circuitree's depth tree encompasses 5 levels with approximately 10^5 constraints, and the proof generation time within their framework is approximately 100 seconds. Despite the variations in foundational methodologies, it is discernible that our toolchain exhibits proficiency in handling challenges of increased arithmetic circuit complexity within an acceptable temporal scope.

6.4.2 Further testing

In subsequent research, we aim to assess our methodology using an automated evaluation system designed to analyse Prolog programs of varying complexity and dimensions. Such an examination could clarify the scalability of this approach.

6.4.3 Improving performance

A consideration for policy design for our approach, because of the nature of arithmetic circuits, is maximum proof-tree size. Suppose the range of proof-tree sizes the policy program can produce varies. In that case, the generated circom code always has to assume the maximum size, increasing the size of the circuit and thus decreasing the performance, although only linearly, as stated before.

6.4.4 Integration with Other Systems

Our toolchain is designed not just for standalone utility but also for interoperability. With the capability to integrate with 'traditional' blockchains, we can employ Plonk to generate a verifier smart contract for platforms like Ethereum using Circom. Furthermore, our toolchain successfully meets the initial use case we set out to solve, emphasising its effectiveness and adaptability.

Chapter 7

Conclusion

We successfully designed a framework for creating Self-evaluated Policies written in Prolog, using the Circom compiler. The framework is modular in the sense that it is able to use any Porolog implementation and the many Zero-Kownledge Protocols supported by Circom. We verified that this approach to ZKP design is feasible, both in terms of architecture and performance. Our evaluation of the proof-of-concept toolchain showed that our encoding approach allows for linear scaling of proof times with input size, which is a very desirable attribute. The tools were tested individually, and integrated together, with a wide range of inputs. The results show that with our approach, Self-evaluated Policies are achievable, and hold many opportunities for further research and improvement. Certainly, a key takeaway from our research is that Zero-Knowledge systems are rapidly increasing in capability and possible complexity, along with their performance and ease of use.

In the future, we aim to validate our system more rigorously, with a model-based approach. We also want to improve its performance by optimizing the checker circuit. Furthermore, we plan to integrate our results into the BME-MNB government energy support use-case prototype discussed in Section 2.1

Bibliography

[1] : *BBS Cryptosuite v2023*. – URL https://w3c.github.io/vc-di-bbs/. – Zugriffsdatum: 2023-11-02

[2] : *Open Policy Agent*. – URL https://www.openpolicyagent.org/. – Zugriffsdatum: 2023-10-29

[3] : *SWI-Prolog*. – URL https://www.swi-prolog.org/. – Zugriffsdatum: 2023-11-01

[4] : *Tau Prolog: A Prolog interpreter in JavaScript*. – URL http://tau-prolog.org/. – Zugriffsdatum: 2023-10-17

[5] : *walt.id: Open Policy Agent (OPA)*. – URL https://docs.walt.id/v/ssikit/concepts/open-policy-agent. – Zugriffsdatum: 2023-11-01

[6] : *ZK-WASM*. September 2021. – URL https://delphinuslab.com/zk-wasm/. – Zugriffsdatum: 2023-11-01

[7] Alex Preukschat, Drummond R.: *Self-Sovereign Identity: Decentralized digital identity and verifiable credentials*. Manning, 2021. – ISBN 978-1-61729-659-8

[8] Angel, Sebastian ; Blumberg, Andrew J. ; Ioannidis, Eleftherios ; Woods, Jess: Efficient Representation of Numerical Optimization Problems for {SNARKs}, URL https://www.usenix.org/conference/usenixsecurity22/presentation/angel. – Zugriffsdatum: 2023-08-21, 2022, S. 4273–4290. – ISBN 978-1-939133-31-1

[9] Apt, Krzysztof R. ; Bol, Roland N. ; Klop, Jan W.: *On the safe termination of Prolog programs*. Centrum voor Wiskunde en Informatica, 1989

[10] Bangerter, E. ; Krenn, S. ; Sadeghi, A. ; Schneider, T.: YACZK: Yet Another Compiler for Zero-Knowledge (Poster Abstract), URL https://www.semanticscholar.org/paper/YACZK%3A-Yet-Another-Compiler-for-Zero-Knowledge-Bangerter-Krenn/4d46f0e6332b40bdf9a22c9a14a35ed154e341f1. – Zugriffsdatum: 2023-08-21, August 2010

[11] Bank of International Settlements Innovation Hub: Project Rosalind: building API prototypes for retail CBDC ecosystem innovation. (2023), Juni. – URL https://www.bis.org/publ/othp69.htm. – Zugriffsdatum: 2023-11-01

[12] Bank of International Settlements Innovation Hub: Project Rosalind: developing prototypes for an application programming interface to distribute retail CBDC. (2023), Juni. – URL https://www.bis.org/about/bisih/topics/cbdc/rosalind.htm. – Zugriffsdatum: 2023-11-01

[13] Ben-Sasson, Eli ; Bentov, Iddo ; Horesh, Yinon ; Riabzev, Michael: *Scalable, transparent, and post-quantum secure computational integrity*. 2018. – URL https://eprint.iacr.org/2018/046. – Zugriffsdatum: 2023-11-01. – Publication info: Preprint. MINOR revision.

[14] Ben-Sasson, Eli ; Bentov, Iddo ; Horesh, Yinon ; Riabzev, Michael: Scalable Zero Knowledge with No Trusted Setup. In: Boldyreva, Alexandra (Hrsg.) ; Micciancio, Daniele (Hrsg.): *Advances in Cryptology – CRYPTO 2019*. Cham : Springer International Publishing, 2019 (Lecture Notes in Computer Science), S. 701–732. – ISBN 978-3-030-26954-8

[15] Ben-Sasson, Eli ; Chiesa, Alessandro ; Genkin, Daniel ; Tromer, Eran ; Virza, Madars: SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In: Canetti, Ran (Hrsg.) ; Garay, Juan A. (Hrsg.): *Advances in Cryptology – CRYPTO 2013*. Berlin, Heidelberg : Springer, 2013 (Lecture Notes in Computer Science), S. 90–108. – ISBN 978-3-642-40084-1

[16] Bootle, Jonathan ; Chiesa, Alessandro ; Liu, Siqi: Zero-Knowledge IOPs with Linear-Time Prover and Polylogarithmic-Time Verifier. In: Dunkelman, Orr (Hrsg.) ; Dziembowski, Stefan (Hrsg.): *Advances in Cryptology – EUROCRYPT 2022*. Cham : Springer International Publishing, 2022 (Lecture Notes in Computer Science), S. 275–304. – ISBN 978-3-031-07085-3

[17] Bowe, Sean: *Bellman: zk-SNARKs in Rust*. April 2017. – URL https://electriccoin.co/blog/bellman-zksnarks-in-rust/. – Zugriffsdatum: 2023-08-23

[18] Briner, Thomas: *Compiler for zero-knowledge proof-of-knowledge protocols*, Swiss Federal Institute of Technology Zürich, Dissertation, März 2004

[19] Bünz, Benedikt ; Bootle, Jonathan ; Boneh, Dan ; Poelstra, Andrew ; Wuille, Pieter ; Maxwell, Greg: Bulletproofs: Short Proofs for Confidential Transactions and More. In: *2018 IEEE Symposium on Security and Privacy (SP)*, Mai 2018, S. 315–334. – ISSN: 2375-1207

[20] Campanelli, Matteo ; Fiore, Dario ; Querol, Anaïs: LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA : Association for Computing Machinery, November 2019 (CCS '19), S. 2075–2092. – URL https://doi.org/10.1145/3319535.3339820. – Zugriffsdatum: 2023-10-29. – ISBN 978-1-4503-6747-9

[21] Dowling, William F. ; Gallier, Jean H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. In: *The Journal of Logic Programming* 1 (1984), Oktober, Nr. 3, S. 267–284. – URL https://www.sciencedirect.com/science/article/pii/0743106684900141. – Zugriffsdatum: 2023-10-18. – ISSN 0743-1066

[22] Eberhardt, Jacob ; Tai, Stefan: ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, URL https://ieeexplore.ieee.org/document/8726497. – Zugriffsdatum: 2023-10-29, Juli 2018, S. 1084–1091

[23] Editor, CSRC C.: *Zero-Knowledge Proof - Glossary | CSRC*. – URL https://csrc.nist.gov/glossary/term/zero_knowledge_proof. – Zugriffsdatum: 2023-10-25

[24] Emden, Maarten ; Kowalski, Robert: The Semantics of Predicate Logic as a Programming Language. In: *J. ACM* 23 (1976), Januar, S. 733–742

[25] Gabizon, Ariel ; Williamson, Zachary J.: *fflonk: a Fast-Fourier inspired verifier efficient version of PlonK*. 2021. – URL https://eprint.iacr.org/2021/1167. – Zugriffsdatum: 2023-11-02. – Publication info: Preprint. MINOR revision.

[26] Gabizon, Ariel ; Williamson, Zachary J. ; Ciobotaru, Oana: *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. 2019. – URL https://eprint.iacr.org/2019/953. – Zugriffsdatum: 2023-10-29. – Publication info: Preprint.

[27] Godden, Tom ; Smet, Ruben D. ; Debruyne, Christophe ; Vandervelden, Thibaut ; Steenhaut, Kris ; Braeken, An: Circuitree: A Datalog Reasoner in Zero-Knowledge. In: *IEEE Access* 10 (2022), S. 21384–21396. – ISSN 2169-3536

[28] Goldberg, Lior ; Papini, Shahar ; Riabzev, Michael: *Cairo – a Turing-complete STARK-friendly CPU architecture*. 2021. – URL https://eprint.iacr.org/2021/1063. – Zugriffsdatum: 2023-10-29. – Publication info: Preprint.

[29] Goldreich, Oded ; Micali, Silvio ; Wigderson, Avi: Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. In: *Journal of the ACM* 38 (1991), Juli, Nr. 3, S. 690–728. – URL https://dl.acm.org/doi/10.1145/116825.116852. – Zugriffsdatum: 2023-10-26. – ISSN 0004-5411

[30] Goldwasser, Shafi ; Micali, Silvio ; Rackoff, Charles: The Knowledge Complexity of Interactive Proof Systems. In: *SIAM Journal on Computing* 18 (1989), Februar, Nr. 1, S. 186–208. – URL https://epubs.siam.org/doi/10.1137/0218012. – Zugriffsdatum: 2023-10-24. – Publisher: Society for Industrial and Applied Mathematics. – ISSN 0097-5397

[31] Groth, Jens: Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In: Abe, Masayuki (Hrsg.): *Advances in Cryptology - ASIACRYPT 2010*. Berlin, Heidelberg : Springer, 2010 (Lecture Notes in Computer Science), S. 321–340. – ISBN 978-3-642-17373-8

[32] Groth, Jens: *On the Size of Pairing-based Non-interactive Arguments*. 2016. – URL https://eprint.iacr.org/2016/260. – Zugriffsdatum: 2023-10-29. – Publication info: A minor revision of an IACR publication in EUROCRYPT 2016

[33] Horn, Alfred: On Sentences Which are True of Direct Unions of Algebras. In: *The Journal of Symbolic Logic* 16 (1951), Nr. 1, S. 14–21. – URL https://www.jstor.org/stable/2268661. – Zugriffsdatum: 2023-10-18. – ISSN 0022-4812

[34] Hyperledger Foundation: *Hyperledger Aries*. – URL https://wiki.hyperledger.org/display/ARIES. – Zugriffsdatum: 2023-10-26

[35] Iden3: *Circom: Circuit compiler for zero-knowledge proofs*. Oktober 2023. – URL https://github.com/iden3/circom. – Zugriffsdatum: 2023-10-24

[36] Jie, Koh W.: *Announcing the Perpetual Powers of Tau Ceremony to benefit all zk-SNARK projects.* November 2020. – URL https://medium.com/coinmonks/announcing-the-perpetual-powers-of-tau-ceremony-to-benefit-all-zk-snark-projects-c – Zugriffsdatum: 2023-11-02

[37] Lodder, DK M. ; Khovratovich, Dmitry: Anonymous credentials 2.0. (2019)

[38] Luger, George F. ; Stubblefield, William A.: *AI Algorithms, Data Structures and Idioms in Prolog, Lisp, and Java.* Springer Nature, 2009

[39] Manu Sporny, David C.: Verifiable credentials data model v2.0. . – URL https://www.w3.org/TR/vc-data-model-2.0/. – Zugriffsdatum: 2022-09-26

[40] Meiklejohn, Sarah ; Erway, C. C. ; Küpçü, Alptekin ; Hinkle, Theodora ; Lysyanskaya, Anna: *ZKPDL: A Language-Based System for Efficient Zero-Knowledge Proofs and Electronic Cash.* 2012. – URL https://eprint.iacr.org/2012/226. – Zugriffsdatum: 2023-08-21. – Publication info: Published elsewhere. full version of paper published at USENIX Security 2010

[41] Merkle, Ralph C.: *Method of providing digital signatures.* 1979. – URL https://patents.google.com/patent/US4309569. – tex.num: US4309569A

[42] Mohr, Austin: A survey of zero-knowledge proofs with applications to cryptography. In: *Southern Illinois University, Carbondale* (2007), S. 1–12

[43] Morais, Eduardo ; Koens, Tommy ; Wijk, Cees van ; Koren, Aleksei: *A Survey on Zero Knowledge Range Proofs and Applications.* Juli 2019. – URL http://arxiv.org/abs/1907.06381. – Zugriffsdatum: 2023-08-23. – arXiv:1907.06381 [cs]

[44] Mouris, Dimitris ; Tsoutsos, Nektarios G.: *Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs.* 2020. – URL https://eprint.iacr.org/2020/1155. – Zugriffsdatum: 2023-10-29. – Publication info: Published elsewhere. IEEE Transactions on Information Forensics and Security

[45] Nguyen, Khanh Q. ; Bao, Feng ; Mu, Yi ; Varadharajan, Vijay: Zero-Knowledge Proofs of Possession of Digital Signatures and Its Applications. In: Varadharajan, Vijay (Hrsg.) ; Mu, Yi (Hrsg.): *Information and Communication Security.* Berlin, Heidelberg : Springer, 1999 (Lecture Notes in Computer Science), S. 103–118. – ISBN 978-3-540-47942-0

[46] Nikolaenko, Valeria ; Ragsdale, Sam ; Bonneau, Joseph ; Boneh, Dan: *Powers-of-Tau to the People: Decentralizing Setup Ceremonies.* 2022. – URL https://eprint.iacr.org/2022/1592. – Zugriffsdatum: 2023-11-02. – Publication info: Preprint.

[47] Passerini, Andrea ; Frasconi, Paolo: Kernels on Prolog Proof Trees: Statistical Learning in the ILP Setting. In: *Journal of Machine Learning Research* (2006), Juni, S. 307–342

[48] Quisquater, Jean-Jacques ; Quisquater, Myriam ; Quisquater, Muriel ; Quisquater, Michaël ; Guillou, Louis ; Guillou, Marie A. ; Guillou, Gaïd ; Guillou, Anna ; Guillou, Gwenolé ; Guillou, Soazig: How to Explain Zero-Knowledge Protocols to Your Children. In: Brassard, Gilles (Hrsg.): *Advances in Cryptology — CRYPTO' 89 Proceedings.* New York, NY : Springer, 1990 (Lecture Notes in Computer Science), S. 628–631. – ISBN 978-0-387-34805-6

[49] Russell, Stuart J. ; Norvig, Peter: *Artificial intelligence: A modern approach (2nd edition)*. Prentice Hall, Dezember 2002. – ISBN 0-13-790395-2

[50] Sterling, Leon ; Shapiro, Ehud: *The art of Prolog: advanced programming techniques*. Cambridge, MA, USA : MIT Press, August 1986. – ISBN 978-0-262-19250-7

[51] The Trust Over IP Foundation: *Trust Over IP - Defining a complete architecture for Internet-scale digital trust*. – URL https://trustoverip.org/. – Zugriffsdatum: 2023-11-01

[52] Toldi, Balazs A.: Blockchain-based, confidentiality-preserving orchestration of collaborative workflows. Budapest, Hungary : Budapest University of Technology and Economics, 2023. – URL https://tdk.bme.hu/VIK/sw8/Kollaborativ-munkafolyamatok-titkossagmegorzo

[53] Triska, Markus: *The Power of Prolog*. – URL https://www.metalevel.at/acomip/. – Zugriffsdatum: 2023-10-17

[54] ZKProof Standards: *An R1CS Based Implementation of Bulletproofs - Cathie Yun, Interstellar*. April 2019. – URL https://www.youtube.com/watch?v=V_IphW7_1Jw. – Zugriffsdatum: 2023-10-25