Received January 11, 2022, accepted February 17, 2022, date of publication February 22, 2022, date of current version March 2, 2022.

*Digital Object Identifier 10.1109/ACCESS.2022.3153366*

# Circuitree: A Datalog Reasoner in Zero-Knowledge

**TOM GODDEN**[1], **RUBEN DE SMET**[2], **(Student Member, IEEE), CHRISTOPHE DEBRUYNE**[3], **THIBAUT VANDERVELDEN**[1], **KRIS STEENHAUT**[1,2], **(Member, IEEE), AND AN BRAEKEN**[1]

[1]INDI Department, Vrije Universiteit Brussel, 1050 Brussels, Belgium
[2]ETRO Department, Vrije Universiteit Brussel, 1050 Brussels, Belgium
[3]Institut Montefiore, Université de Liège, 4000 Liège, Belgium

Corresponding authors: Tom Godden (tom.godden@vub.be) and Ruben De Smet (rubedesm@vub.be)

This work was supported in part by the Project Fund for Technology and Transfer (TETRA) of Flanders Innovation & Entrepreneurship (VLAIO) under Award HBC.2020.2073-Velcro.

**ABSTRACT** Driven by the increased consciousness in data ownership and privacy, zero-knowledge proofs (ZKPs) have become a popular tool to convince a third party of the truthfulness of a statement without disclosing any further information. As ZKPs are rather complex to design, frameworks that transform high-level languages into ZKPs have been proposed. We propose Circuitree, a Datalog reasoner in zero-knowledge. Datalog is a high-level declarative logic language that is generally used for querying. Furthermore, as a logic language, it can also be used to solve logic problems. An application using Circuitree can efficiently generate ZKPs, based on Datalog rules and encrypted data, to prove that a certain conclusion follows from a Datalog ruleset and encrypted input data. Compared to existing frameworks, which generally use their own limited imperative languages, Circuitree uses an existing high-level declarative language. We point out several applications for Circuitree, including EU Digital COVID Certificates and privacy-preserving access control for peer-to-peer (p2p) networks. Circuitree's performance is evaluated for access control in a p2p network. First results show that our approach allows for fast proofs and proof verification for this application.

**INDEX TERMS** Access control, bulletproofs, datalog, privacy, zero-knowledge proof, security, identity management, verifiable computation, blockchain, privacy-enhancing technologies.

## I. INTRODUCTION

Witnessing the continuous stream of news reports about personal data leaks, data trade as a new business model, or data misuse by government agencies for various political purposes [1]–[3], it should be evident that the lack of online privacy is a big issue. While initiatives such as the General Data Protection Regulation (GDPR) covers the legal aspect, privacy-enhancing technologies (PETs) try to avoid infringements on privacy from a technological perspective.

A current paradigm in PETs is the adoption of ZKPs. A ZKP allows proving a statement that relates to some secret knowledge without revealing anything beyond the truth of this statement. ZKPs have applications in many domains, although they are primarily used for authentication and authorization. A recent example is the Signal private group system [4], wherein participants prove that they have the necessary access rights to alter an existing group definition without disclosing their identity or even their access level.

ZKPs have also been applied in blockchain technology. In cryptocurrencies, for instance, ZKPs are used to conceal transaction amounts, senders, and receivers [5], [6]. Later, proposals appeared which use ZKPs to manage anonymous credentials [7] for decentralized identities on blockchains [8].

Many different systems for proving nearly arbitrary statements in zero-knowledge have been devised in the last decade. Because designing ZKPs is complex and error-prone, various high-level programming languages and corresponding compilers for ZKP systems have been brought forward [9]–[15]. Like regular high-level languages and compilers, these tools are designed to make development easier. They accomplish this by providing abstractions that increase the efficiency of the development and the correctness of the resulting proof.

This article proposes to use Datalog as a high-level language for ZKPs, as an alternative to existing systems, which exclusively use imperative languages. Datalog is a declarative

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Imran Tariq.

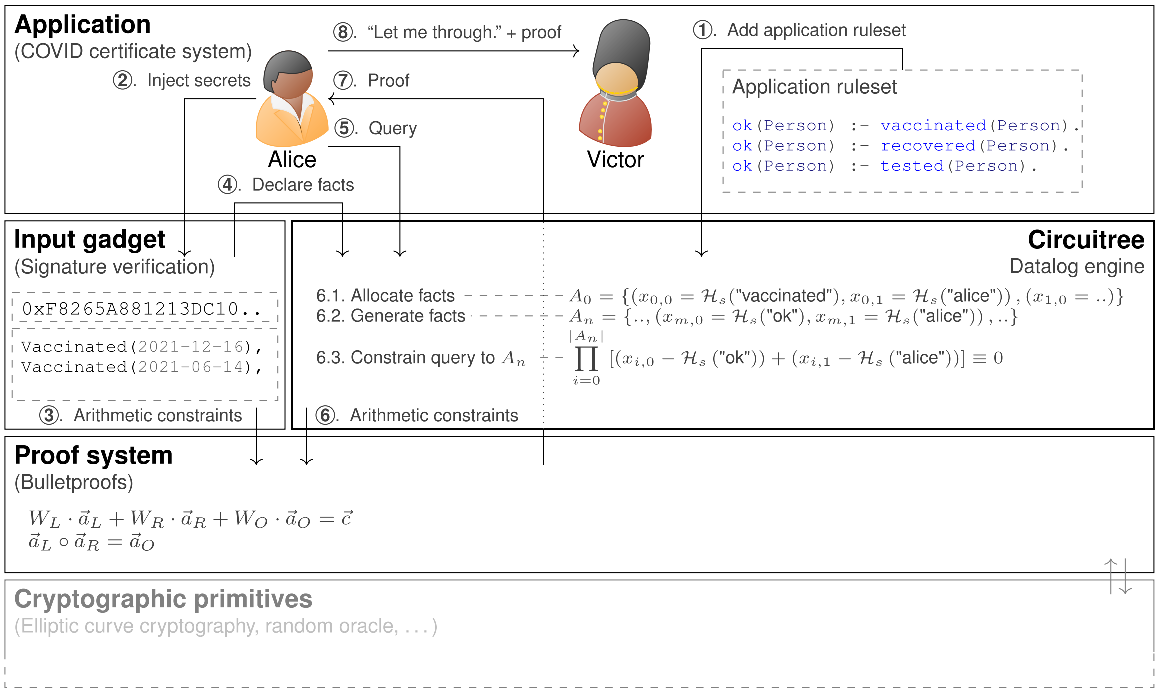

Application
(COVID certificate system)
②. Inject secrets
⑧. "Let me through." + proof
⑦. Proof
⑤. Query
Alice
④. Declare facts
Victor
①. Add application ruleset
Application ruleset
ok(Person) :- vaccinated(Person).
ok(Person) :- recovered(Person).
ok(Person) :- tested(Person).
Input gadget
(Signature verification)
0xF8265A881213DC10..
Vaccinated(2021-12-16),
Vaccinated(2021-06-14),
③. Arithmetic constraints
Circuitree
Datalog engine
6.1. Allocate facts
6.2. Generate facts
6.3. Constrain query to $A_n$
$A_0 = \{(x_{0,0} = \mathcal{H}_s(\text{"vaccinated"}), x_{0,1} = \mathcal{H}_s(\text{"alice"})), (x_{1,0} = ..)\}$
$A_n = \{.., (x_{m,0} = \mathcal{H}_s(\text{"ok"}), x_{m,1} = \mathcal{H}_s(\text{"alice"})), ..\}$
$\prod_{i=0}^{|A_n|} [(x_{i,0} - \mathcal{H}_s(\text{"ok"})) + (x_{i,1} - \mathcal{H}_s(\text{"alice"}))] \equiv 0$
⑥. Arithmetic constraints
Proof system
(Bulletproofs)
$W_L \cdot \vec{a}_L + W_R \cdot \vec{a}_R + W_O \cdot \vec{a}_O = \vec{c}$
$\vec{a}_L \circ \vec{a}_R = \vec{a}_O$
Cryptographic primitives
(Elliptic curve cryptography, random oracle, ...)

**FIGURE 1.** **Relations between application, Circuitree, proof system and underlying cryptographic primitives. In this scenario, Alice wants to go to a bar, and Victor has to verify her COVID Certificate, according to the application-specified policy. First, Circuitree is initialized with the application-specific ruleset (step 1). The secret input data are injected into the proof system through an input gadget; in the case of the EU Digital COVID certificate, a signature is verified in zero-knowledge (step 2 and 3). The corresponding facts are declared to Circuitree (step 4), in an application-specific way. When Circuitree is supplied with a query (step 5), it will allocate the facts, run the Datalog reasoner and generate the constraints for the supplied query (step 6). Finally, the proof is generated (step 7) and sent to Victor, the verifier (step 8).**

programming language. Datalog is not Turing-complete, which is a desirable property for data retrieval and querying. For instance, it is the basis of graph query languages such as SPARQL [16]. We will compile Datalog, as an existing high-level programming language, to zero-knowledge. This makes the development for declarative zero-knowledge applications significantly easier and less error-prone, compared to a direct implementation in zero-knowledge. Furthermore, this compilation introduces a layer of abstraction, which allows us to potentially exploit a large variety of zero-knowledge systems.

We describe an efficient implementation strategy in the rank-1 constraint system (R1CS) setting which we call "Circuitree". We demonstrate its performance with a Datalog reasoner using the zero-knowledge system Bulletproofs [17]. With Circuitree, we aim to provide a framework that benefits from Datalog's computational characteristics to generate efficient proofs of entailment with low latency.

Figure 1 depicts the Circuitree application stack. It illustrates where Circuitree is situated in an application, and how it relates to its underlying proof system. As a concrete example, the figure focuses on a privacy-preserving implementation of the EU Digital COVID Certificate system. The context of this example is explained in Section III-E.

A typical usage of Circuitree in an application can be roughly divided in 8 steps, also depicted in Fig. 1. First, the application ruleset is injected in Circuitree (step 1). A more complete ruleset, specific to the depicted example can be found in Fig. 1. Next (step 2), the prover injects their secret facts. These typically come from an earlier commitment, or from encrypted or signed data. An input gadget generates the necessary constraints corresponding to decryption or signature verification, and inserts them into the proof system (step 3). The prover then declares the facts retrieved from the input gadget to Circuitree (step 4). Now, Circuitree is initialized and can be queried (step 5). Circuitree generates the constraints that relate the query to the input data and ruleset (step 6). The bulk of this article (Sections V and VI) is concerned with generating these constraints in an efficient manner. Finally, the proof can be extracted from the underlying proof system (step 7), which can then be presented to a verifier (step 8).

The name *Circuitree* is derived from the tree-like structure of our generated *arithmetic circuits*, a term which is commonly (but incorrectly) used as a synonym for R1CS.

In what follows, we describe the related work and the context in which Circuitree is situated (Section II), and we illustrate the usefulness of Circuitree with multiple applications (Section III). The three sections thereafter are concerned with the technical details. We start with an overview of Circuitree's design in Section IV, and we continue with the details in Sections V and VI where we describe a naive and an improved design respectively. In Section VII we describe our implementation, in Section VIII we give a quantitative and qualitative evaluation, and we conclude in Section IX. Finally, in Section X we give some pointers to future work.

## II. CONTEXT AND RELATED WORK

Since around 1987, it has been known that all languages in the nondeterministic polynomial time (NP) complexity class have zero-knowledge proof systems [18]–[20], i.e., for every statement that can be verified in polynomial time, the truth of that statement is guaranteed without disclosing any additional information. After its theoretical inception, zero-knowledge research found an application in blockchain technology. ZKPs allow keeping the contents of a public blockchain private while maintaining the integrity and consistency constraints for which the blockchain is used [5], [6], [21].

Driven by increased interest, many different proof systems have been developed in recent years. Each of these proof systems has its strengths and weaknesses, most prominently related to performance and cryptographic assumptions. For example, Bulletproofs [17] does not require a *trusted setup*. A system with a trusted setup requires a setup phase with an honest party, before any proofs are generated. If this party were dishonest, the "toxic waste" that is produced during the setup ritual could be exploited to falsify proofs. Bulletproofs instead has *transparent setup*, but it is significantly more costly in operation compared to many succinct arguments of knowledge (SNARKs), for instance the ZK-STARK protocol [22].

### A. CONSTRAINT SYSTEMS AND NATIVE LANGUAGES

Proof systems also differ in the constraint system they are designed with, often referred to as their "native language". Bulletproofs and most SNARKs take a rank-1 constraint system (R1CS), which is a system of multiplication constraints and linear constraints taken over a prime field $\mathbb{F}_p$. Usually, the system's complexity depends on the number of multiplication gates $n$. For example, Bulletproofs produces proofs of size $\mathcal{O}(\log n)$.

Some newer proof systems devise their bespoke constraint system. For example, ZK-STARK works with an arithmetic execution trace (AET), a set of low-degree polynomials that are repetitively applied to the input data. Another example is Hoffmann, Klooß, and Rupp [23], which allows quadratic multiplication gates, instead of linear multiplication gates in standard R1CS. Since R1CS is the most widely known and used constraint system, this article will further assume R1CS. For our implementation, we will build on top of Bulletproofs.

### B. ZERO-KNOWLEDGE ABSTRACTIONS

For several reasons, high-level programming languages and abstractions have been designed on top of these zero-knowledge algorithms. Firstly, a manual implementation tightly couples an application to a particular ZKP system. Moving from e.g. Bulletproofs to ZK-STARK requires a reimplementation of the whole system. Secondly, directly designing the zero-knowledge equations is a challenging and error-prone process because of the limitations of the constraint system and the low-level aspect of the zero-knowledge tools. Thirdly, debugging these applications is challenging because of their cryptographic nature. Because of these reasons, high-level languages have been introduced that can be compiled to constraints directly, or to instruction sets that are interpreted at runtime to these constraints.

The most generic among these implement a Turing-complete programming system. TinyRAM [10], [11], for example, is a Harvard-architecture reduced instruction set computer (RISC) for which programmers design their ZKP using a subset of the C programming language. TinyRAM works by compiling the programmer's program to a zero-knowledge constraint system, which in turn can prove that every program instruction has been executed correctly on a given input. A related approach works by implementing a complete von Neumann architecture as a zero-knowledge program, effectively implementing a Turing-complete computer that can verify arbitrary programs. This is the approach used by vnTinyRAM [12] and later Cairo [14].

A prominent use case of Turing-complete zero-knowledge abstractions is verifiable computation. Mouris and Tsoutsos [15] introduce Zilch, a framework specifically for verifiable computation with ZKPs. Zilch uses zMIPS, a zero-knowledge oriented Microprocessor without Interlocked Pipelined Stages (MIPS) architecture, and ZeroJava, a high-level Java-like domain-specific language that is compiled to zMIPS. In verifiable computation, a user sends a secret input to a third party which will perform computations with this input and returns its output, together with a proof that the computations where performed correctly. Zilch's instruction set provides a relatively efficient way of generating this proof. Because ZeroJava is a high-level language, it is significantly easier to design these proofs than with manually implemented arithmetic constraints.

Alternative languages like zero-knowledge proof description language (ZKPDL) [9] and Circom [13] are then again tied to a specific zero-knowledge architecture. Circom encodes constraints for R1CS, and can be used with several different R1CS-based ZKP systems. Whereas it does not impose any overhead on top of the underlying R1CS system, it only abstracts over a specific family of ZKP systems. ZKPDL is not based on an NP-complete subsystem, but instead works by specifying the raw (discrete log) relations between public inputs and secret inputs.

Circuitree can provide a *declarative* option for verifiable computation, as an alternative to the previously mentioned imperative systems. Compared to the previously mentioned frameworks, which use their own domain-specific imperative language with limitations, Circuitree implements Datalog, an existing declarative language.

### C. DATALOG AND REBAC

Datalog is a declarative programming language and a subset of Prolog. Compared to imperative languages, where the programmer gives the program exact instructions, declarative languages allow the programmer to define assertions (also called facts) and logical rules which generate new assertions based on existing ones. It is then possible to either find all assertions the program entails, or query the program to determine whether the assertions and rules entail a specific query (also called a question).

Datalog's main application is database querying and data retrieval; it has certain restrictions, most importantly concerning negation, that ensure that a Datalog reasoning process always terminates. It is because of these restrictions that infinite loops are impossible in Datalog, which makes Datalog not Turing-complete. These restrictions make Datalog suitable for declaring access control rules [24], [25].

Masoumzadeh and Joshi [26] describe a methodology for enforcing access control in an ontology-based social networking system (SNS), which they call OSNAC (ontology-based social network access control). Their approach embeds the permissions as data in the SNS. Similar to OSNAC is ReBAC (relationship-based access control) [27]. We draw inspiration from both. Permissions can be defined explicitly (i.e. asserted), or derived from data via logic rules. This allows one to enforce permissions with a simple database query.

OSNAC assumes a centralized database, where the service provider has complete control over the data. For privacy-minded applications, it would be desirable to have an OSNAC-like system, where permissions are enforced *without* a service provider. Peer-to-peer networks such as PeerSoN [28], Glycos [29], or LibreSocial [30] all provide private and secure SNSs. Their access control mechanisms are relatively simple, in the sense that every user that should have access is listed with the protected data item. This technique is called an access control list (ACL). We propose to replace the ACLs with ZKPs, by declaring the access control rules in Circuitree instead. This keeps the data and access permissions confidential, while the prover's authorization claim can be verified without disclosing any additional information.

In such a system, a database query suffices to establish whether a user is allowed to write data. However, if the permission data or the data required to deduce the permission needs to stay secret, as would be the case in a privacy-oriented p2p network, the query engine would have to be able to reason about encrypted data. In the next section, we will describe how Circuitree solves this issue.

## III. APPLICATIONS

Circuitree can be used as a building block for identity management. Applications that require identity management range from banking, healthcare, and government services to education and transportation [31], [32]. Circuitree could play a role in the new World Wide Web Consortium (W3C) recommendation about *Linked data proofs* [33], allowing for a broader feature set. Circuitree can also be applied to other domains, like verifiable computation. We show that other problems like access control for p2p networks and logic problems are, in essence, a verifiable computation problem that can be solved by Circuitree.

### A. IDENTITY MANAGEMENT

ZKPs are notably useful in identity management, where they typically appear in anonymous authentication systems. In such a system, a user is able to authenticate with a system without disclosing their identity. Especially the context of blockchain technology has seen a recent interest spike as a cornerstone for identity management [32]. The idea is to store a personal identity in a secure form (e.g., committed or encrypted) on a blockchain. At any later time, parts of the identity can be extracted and transformed so that the data owner can reveal the result. The correctness of the result is then verified, given the original blockchain and a small proof.

Another example is the Signal private group system [4], wherein participants prove that they have the necessary access rights to alter an existing group definition without disclosing their identity or even their access level.

The rules for identity management could potentially be declared in Circuitree to generate such proofs. This approach is more flexible and high-level than the manual implementation of every potential proof. The outcome is similar to role-based encryption (RBE) [34], which enforces role-based access control at decryption time. Depending on the application at hand, one could opt for either Circuitree or RBE, or both could even be seen as complementary. Applications may use a form of RBE for read access and Circuitree for write access.

### B. LINKED DATA

The W3C shows interest in standardizing ''Linked data proofs''. Linked Data is the principle of interlinking structured data on the Web, and linked data proofs would enable extracting partial information from potentially confidential Linked Data [33]. The W3C draft specification is based on BBS+ signatures [35], a pairing-based signature scheme that allows efficient ZKPs, to disclose only a subset of multiple signed messages [36]. At the time of writing, the draft does not mention providing proofs for transformed data, and to the best of our knowledge, BBS+ signatures are not suitable for this purpose. Circuitree trivially reveals data, namely the query, on top of applying transformations. We believe it could be an interesting alternative approach to linked data proofs.

### C. VERIFIABLE COMPUTATION

In verifiable computation, a party requests a third party to perform certain computations and return the result, alongside a proof that the computations were in fact performed correctly. This is useful to allow devices with too little computational power to outsource heavy computations to a semi-trusted third party, or to allow a party to distribute workload between untrusted third parties. Circuitree provides a verifiable computation framework for Datalog. A party can reason using a Datalog reasoner and prove to a third party that they performed computations within the domain of the system.

### D. ACCESS CONTROL

Circuitree verifies the correctness of a reasoning process. This can be used for access control in p2p networks, where no single authority can decide whether a peer has the authority to write certain data to a network. An example of such an

```
vaccinatedCert(Person) :- hasVaccine1(Person), hasVaccine2(Person).
vaccinatedCert(Person) :- hasVaccineJJ(Person).
recoveringCert(Person) :- confirmedCase(Person, Infection),
                          hasDate(Infection, Date),
                          daysSince(Date, Days),
                          Days >= 11, Days <= 180.
testCert(Person) :- hasPCRTest(Person, Test),
                    hasDate(Test, Date) isNegative(Test),
                    daysSince(Date, Days),
                    Days <= 2.
testCert(Person) :- hasRATTest(Person, Test),
                    hasDate(Test, Date),
                    isNegative(Test),
                    daysSince(Date, Days),
                    Days <= 1.
hasCert(Person) :- vaccinatedCert(Person).
hasCert(Person) :- recoveringCert(Person).
hasCert(Person) :- testCert(Person).
ok(Person) :- hasCert(Person).
ok(Person) :- hasAge(Person, Age), Age < 10.
```

**LISTING 1. An example of a Datalog program that checks whether a person conforms to the requirements of the CST.**

application is Glycos [29]. Glycos is a p2p framework for building privacy-friendly online social networks (OSNs), which aims to provide equivalent building blocks to classical OSN services based on client-server and web architectures. In Glycos, in order to write data, a peer has to convince the network that they have the authority to do so. With Circuitree, it could do this by proving that the encrypted data in the network and the rules defined on the SNS entail their authority. The peer can use the subset of data on the network that they know the plaintext of as input assertion set to Circuitree, to generate a proof that it does, in fact, have write access to the network. With this in mind, Circuitree can efficiently enforce an access control policy anonymously while maintaining the flexibility of declaring access control rules in a high-level language.

The process described above is an alternative approach to the traditional verifiable computation, where a third party performs computations efficiently and returns data, providing a proof that the data were computed correctly. For Circuitree, the computing party is the end user who provides a proof to the system. For access control in Glycos, the performed computation is that of a Datalog reasoner. The result should evidently always be the fact that the user is allowed to write data. In this case, the result is irrelevant since a user will always claim that they have this authority, regardless of whether this is true. However, the proof that they computed this result correctly does hold meaning since the user cannot lie about this.

### *E. LOGIC PROBLEMS*

Because Datalog is a logic language, albeit not Turing-complete, we can also apply Circuitree to logic problems. An example of such a logic problem would be a privacy-preserving application for EU Digital COVID Certificates for COVID-19. During the COVID-19 pandemic, these certificates were designed to prove to a third party that a person meets certain criteria, like vaccination status or whether they were recently infected or tested. However, the current certificates contain sensitive information in plaintext, including, but not limited to, the certificate holder's full name, date of birth, vaccination status, which vaccine they received and if they have recently been infected. By means of ZKPs, this information does not need to be transferred but can instead be replaced with a proof that the certificate holder matches all required criteria. Based on the rules imposed by each country, Circuitree can be applied to design such ZKPs through a high-level logic language, namely Datalog.

Listing 1 demonstrates a small Datalog program that checks whether a person conforms to the requirements of the CST, the Belgian EU Digital COVID Certificate for COVID-19. A user can use Circuitree to generate a proof in zero-knowledge to prove their conformity without revealing any personal data.

Because Datalog is a high-level language, it is easy to create additional rules for other regulations as well. The end-user would be able to select the regulation for which a proof is needed, and the application would generate the proof on-the-fly.

Note that this proof requires arithmetic reasoning, which is future work for Circuitree. Additionally, the prover application will need a way to introduce the input assertions in a sound way, for example by means of an embedded signature verification algorithm.

Circuitree could also be applied to generate ZKP for similar problems. It could, for example, generate a proof that can prove a person meets the criteria to take a loan from a bank without having to disclose a large amount of personal data like current balance or income.

These problems can once again be seen as a form of verifiable computation, where a user proves to a third party that they computed the permissions correctly.

## IV. CIRCUITREE OVERVIEW

Our Circuitree implementation is based on a slightly modified version of Bulletproofs [17]. Our choice for Bulletproofs is motivated by its transparent setup and the size of its proofs. Small proofs are a desirable property in p2p applications. In a commercial setting, one could trade a transparent setup for a boost in performance by using a ZK-SNARK, or one could trade the small proofs for a boost in performance by using a ZK-STARK.

An R1CS system of constraints is usually declared using one or more so-called "gadgets", which are modular and reusable pieces of proof-embedded logic. In its turn, the gadget encodes logical statements as (arithmetic) constraints. In our case, given a set of rules, these logical statements are the program flow of a Datalog reasoner. More specifically, the set of constraints declared by this reasoner gadget should be valid if and only if the assertions and rules expressed in Datalog entail the query result that we wanted to prove.

The arithmetic constraints form a rank-1 constraint system (R1CS). A R1CS is a mathematical model which only recognizes addition, subtraction, and multiplication operations on integers modulo some large prime $p$. This implies that we have to convert our logical reasoning operations to those arithmetic operations. It should be noted that the equations in the R1CS are never actually computed. This means that the data are never revealed and cannot be extracted from the system, as opposed to a regular program, where the data pass through the program. However, this also implies that it is impossible to do a conditional branch in a R1CS (because the branch condition is never computed), nor is it possible to use recursion. Because our gadget's performance is tied to the number of constraints, the challenge is to design the reasoner gadget so that it uses the lowest number of arithmetic constraints possible, given these restrictions. We do not make assumptions on the input for the reasoner gadget. In practice, the reasoner gadget will receive input from either a decryption gadget or a commitment. The decryption gadget proves that a certain encrypted input can be decrypted and decoded to be fed into the reasoner gadget. A commitment means that the processed data was committed to earlier in time and binds the committing party to a certain value in a privacy-preserving way. The hidden value cannot feasibly be changed, and can be optionally revealed at a later time. Using a commitment in a proof is known as the commit-then-prove paradigm. The composition of the reasoning gadget with its complementary gadgets is visualized in Fig. 2. Proof-of-decryption is considered out-of-scope for this article, especially now that new symmetric encryption algorithms are being designed with proof-of-decryption in mind [37].

Our running example is a hypothetical access control model for p2p social networks, wherein the p2p network will enforce write permissions in zero-knowledge, i.e. without

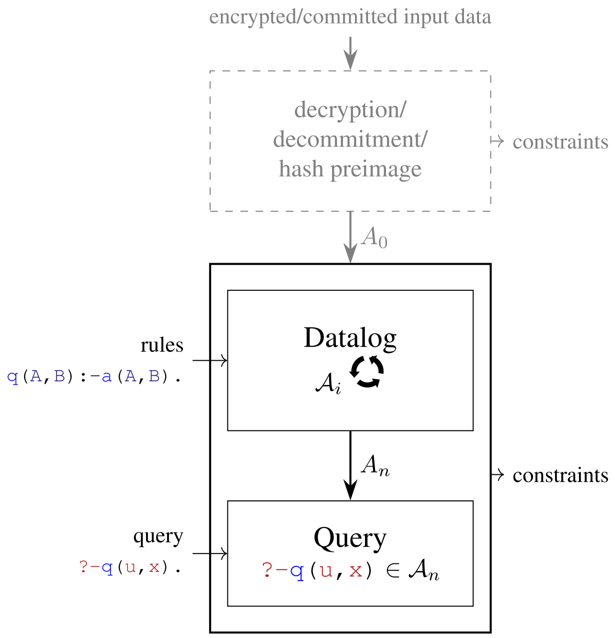

encrypted/committed input data
decryption/ decommitment/ hash preimage
constraints
$A_0$
rules
q(A,B):-a(A,B).
Datalog
$\mathcal{A}_i$
$A_n$
constraints
query
?-q(u,x).
Query
?-q(u,x) $\in \mathcal{A}_n$

**FIGURE 2. Schematic overview of the reasoner gadget and its context. Above is a gadget responsible to securely provide the input data, usually a decryption gadget. The data are then fed from the decryption gadget into the reasoner gadget, which takes the ruleset and outputs the necessary constraints, corresponding to the iterative reasoning of the Datalog engine.**

access to the plaintext information. A peer uploading data to the network will need to attach a proof to their data that proves they are allowed to upload the data. For example, assume that the `can_write` property gives a `Person` object the permission to write to a `Wall` object and that the network has a Datalog rule like

```
can_write(APerson, AWall) :-
  has_wall(AnotherPerson, AWall),
  has_friend(AnotherPerson, APerson).
```

This rule indicates that Bob can write to Alice's wall if Alice has Bob as a friend. However, these data are not readily available on the network but can be derived with the rule above. This means that if Bob wants to write to Alice's wall, he needs to attach a ZKP that proves that the network's data and rules entail `can_write(bob,aliceWall)`.

In Section V, we first "naively" implement a reasoner and show that it has a worse-than-exponential behavior in terms of arithmetic constraints. This naive implementation provides an insight into Datalog reasoners and R1CS systems, which will help understand Section VI. With regard to verifiable computation, this naive implementation demonstrates how a Datalog reasoner would be executed in an imperative system like Zilch. In Section VI, we present an optimization to this reasoner, which first generates the path of the reasoning tree, and then proves that every step taken in the tree is valid. We show that this approach has a cubic behavior in terms of arithmetic constraints.

## V. NAIVE IMPLEMENTATION

The input for the reasoner gadget represents a dataset, for example from a p2p network containing Datalog

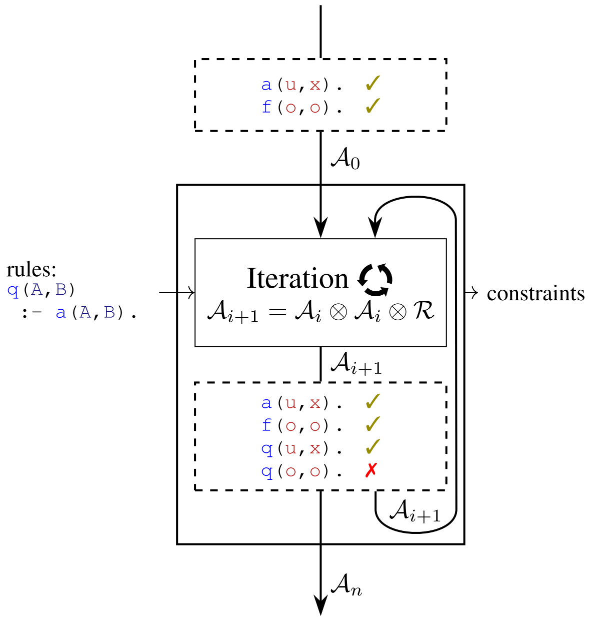


**FIGURE 3. "Naive" reasoner gadget. All rules are applied to every assertion in the input assertion set $\mathcal{A}_i$, which outputs the set $\mathcal{A}_{i+1}$. Every assertion carries a "validity flag" (illustrated by ✓ or ✗), indicating whether the used rule was applicable.**

facts, e.g. `has_wall(alice, aliceWall)` or `has_friend(alice, bob)`. We need to reason over these data by applying the plaintext rules provided by the application. We assume that the initial assertion set $\mathcal{A}_0$ is the input of the reasoner gadget, and the ruleset $\mathcal{R}$ is defined in the application.

### A. REASONER STRUCTURE

When queried, a fixpoint Datalog reasoner applies each rule in the ruleset $\mathcal{R}$ recursively on each matching combination of assertions from $\mathcal{A}_0$, generating new assertions in every recursive step. These new assertions will be added to $\mathcal{A}_0$ to create a new assertion set $\mathcal{A}_1$, which in its turn will then be used in the next iteration of the reasoner. This process is applied recursively until the assertion set remains unchanged, i.e. $A_i = A_{i+1}$. The reasoner returns the set of results that match the query. For our application, the empty set is a failure, any other result is a success. Because Circuitree does not support wildcards at this stage, a successful result will always be a single set containing only the query, and we can stop our iterations once the result is found.

However, because we are reasoning in zero-knowledge, the operations we are allowed to apply are limited. The limitations are: we cannot perform a conditional branch, we cannot use (unbounded) recursion, and (because of the former two) we do not know when our reasoner halts. Instead of branching, we follow each possible path and store and track the validity of a condition in a "validity flag". This flag tracks whether a generated assertion is correct, i.e., whether a plain Datalog engine would have produced it.

Circuitree's recursive step is implemented by unwinding the would-be recursive calls, which means that, for each Datalog iteration, we replicate the computational logic. Since we cannot branch based on the comparison of two consecutive assertion sets, it is impossible to halt the reasoner that way, nor is it possible to compute the recursion depth $n$ after which to halt, because Datalog's boundedness is not solvable in general without the plaintext input database [38]. To solve this, the prover is responsible for providing a maximum number of iterations $n$ in clear text, after which the query should be found in the assertion set. If the query is not found within this number of iterations, the result is a failure. When further restricting the Datalog language, it would be possible to compute the maximum number of iterations based on the ruleset, eliminating the need for the prover to provide $n$.

### B. RULE APPLICATION

Circuitree supports rules of any cardinality but, for the sake of simplicity, we only consider binary rules in this article. These are rules with exactly two subgoals, e.g. `q(U, X) :- f(O, O), b(A, R).` In Section VII-E we discuss rules of different cardinality.

In order to apply a binary rule $r_0$ at iteration $i$, we take all the combinations $\mathcal{A}_i \otimes \mathcal{A}_i$ of each assertion in $\mathcal{A}_i$, and pass these combinations one by one to Algorithm 1 together with the rule. Because we cannot encode a conditional branch in R1CS, the result of this application is always a new assertion, independent of whether the assertions match the rule. A "validity flag" is added that tracks whether the rule matches: a new assertion is valid if and only if its parent assertions are valid, and the parent assertions and their arguments match the rule generating the new assertion.

To give a concrete example, assume we have the following initial assertion set:

*Example 1 (Alice's profile and her friend Bob):*

$$\mathcal{A}_0 = \{a_0, a_1, a_2\}$$

with

$a_0 =$ `has_wall(alice, aliceWall)`,

$a_1 =$ `has_friend(alice, bob)`, and

$a_2 =$ `has_wall(bob, bobWall)`.

Because these assertions are in the initial assertion set, we assume them to be true, and thus set the validity flag for these assertions true. Also assume the ruleset $\mathcal{R}$ containing only the previously introduced rule $r_0$:

```
can_write(APerson, AWall) :-
  has_wall(AnotherPerson, AWall),
  has_friend(AnotherPerson, APerson).
```

To apply $r_0$, we need to take the combinations $\mathcal{A}_0 \otimes \mathcal{A}_0$ as arguments for the rule's subgoals. These combinations are:

$$\begin{aligned} \{\ &(a_0, a_0), \quad (a_0, a_1), \quad (a_0, a_2), \\ &(a_1, a_0), \quad (a_1, a_1), \quad (a_1, a_2), \\ &(a_2, a_3), \quad (a_2, a_3), \quad (a_2, a_3)\} \end{aligned}$$

We then pass each of these combinations and rule $r_0$ as arguments to Algorithm 1, which will generate nine new assertions.

**Algorithm 1** Apply Rule $r_j$ to Assertions $a_x$ and $a_y$, Yielding the Assertion $a_r$. Unification Is Enforced on Algorithm 5. Arguments That Should Be Unified Are Compared, and Their Comparison Is Conjoined in the Validity Flag. The Unified Argument Can Also Be Present in the Rule's Head Through the Index `arg_r`.

```
1: function apply_rule(assertions $a_x$, $a_y$, binary rule $r_j$)
2:     let $a_r \leftarrow$ new assertion{}
3:     $a_r$.predicate $\leftarrow r_j$.predicate
4:     let rule_matches? $\leftarrow$ ($r_j$.subgoals[0] $\overset{?}{=} a_x$.predicate) and ($r_j$.subgoals[1] $\overset{?}{=} a_y$.predicate)
5:     for all (arg_r, arg1, arg2) $\leftarrow r_j$.args_to_unify do
6:         rule_matches? $\leftarrow$ rule_matches? and $a_x$.args[arg1] $\overset{?}{=} a_y$.args[arg2]
7:         if arg_r $\neq$ null then
8:             $a_r$.args[arg_r] $\leftarrow a_x$.args[arg1]        $\triangleright$ $a_x$.args[arg1] is equal to $a_y$.args[arg2]
9:         end if
10:    end for
11:    $a_r$.valid? $\leftarrow a_x$.valid? and $a_y$.valid? and rule_matches?
12:    return $a_r$
13: end function
```

$\mathcal{A}_1 =$ {~~`can_write(aliceWall, aliceWall)`,~~
**`can_write(bob, aliceWall)`**,
~~`can_write(bobWall, aliceWall)`,~~
~~`can_write(aliceWall, bob)`,~~
~~`can_write(bob, bob)`,~~
~~`can_write(bobWall, bob)`,~~
~~`can_write(aliceWall, bobWall)`,~~
~~`can_write(bob, bobWall)`,~~
~~`can_write(bobWall, bobWall)`,~~}

**FIGURE 4. The assertions that result from applying the $r_0$ rule to the dataset from example 1. Of these assertions, only `can_write(bob, aliceWall)` is valid and therefore depicted in bold. The other assertions are considered invalid and are therefore struck out with a red line.**

The resulting assertions are depicted in Fig. 4. Of the resulting assertions, only `can_write(bob, aliceWall)` has a validity flag that evaluates to true.

During iteration $i$, we apply each rule in $\mathcal{R}$ to each combination of assertions in $\mathcal{A}_i$, and add these results to $\mathcal{A}_i$ to generate $\mathcal{A}_{i+1}$. We do this as many times as defined in the maximum number of iterations $n$. In order to confirm whether the query $q$ is in fact true, we use the following formula, where $a$.valid? is $a$'s validity flag:

$$\bigvee_{a \in \mathcal{A}_n} a.\text{valid?} \wedge (a \overset{?}{=} q)$$

After the maximum number of iterations, the final assertion set will have been generated. The reasoning results in a success if the query is found as an assertion in the assertion set, and that assertion is valid.

## C. COMPUTATIONAL COMPLEXITY

For $\mathcal{A}_i$ an assertion set and $|\mathcal{R}|$ the number of rules in the ontology, we can compute the size of the next iteration of the assertion set $A_{i+1}$:

$$\begin{aligned} |\mathcal{A}_{i+1}| &= |\mathcal{A}_i \otimes \mathcal{A}_i| \times |\mathcal{R}| \\ &= |A_i|^2 \times |\mathcal{R}| \end{aligned}$$

where $\mathcal{A}_i \otimes \mathcal{A}_i$ is the cartesian product of $\mathcal{A}_i$ with itself. For $n$ iterations, we can generalize the assertion set size in one iteration to:

$$\begin{aligned} |\mathcal{A}_n| &= |\mathcal{A}_{n-1} \otimes \mathcal{A}_{n-1}| \times r \\ &= |\mathcal{A}_{n-1}|^2 \times r \\ &= (|\mathcal{A}_{n-2} \otimes \mathcal{A}_{n-2}| \times r)^2 \times r \\ &= (|\mathcal{A}_{n-2}|^2 \times r)^2 \times r \\ &= |\mathcal{A}_{n-2}|^4 \times r^3 \\ &= (|\mathcal{A}_{n-3}|^2 \times r)^4 \times r^3 \\ &= |\mathcal{A}_{n-3}|^8 \times r^7 \\ &= \dots \\ &= |\mathcal{A}_0|^{2^n} \times r^{2^n-1} \end{aligned}$$

This shows that the final assertion set size grows superexponentially in the initial assertion set size. Because the size of the assertion sets impacts the size of our proof and computation time linearly, this performance is less than desirable, which is why we propose improvements in Section VI.

## VI. IMPROVED IMPLEMENTATION

Datalog problems are a subset of NP. This means that a solution of a Datalog problem can be verified in polynomial time. We base our improved version of the reasoner on this insight. Our improved version will no longer be a reasoner, but a verifier that can verify whether a single result holds true for our initial assertion set and rules.

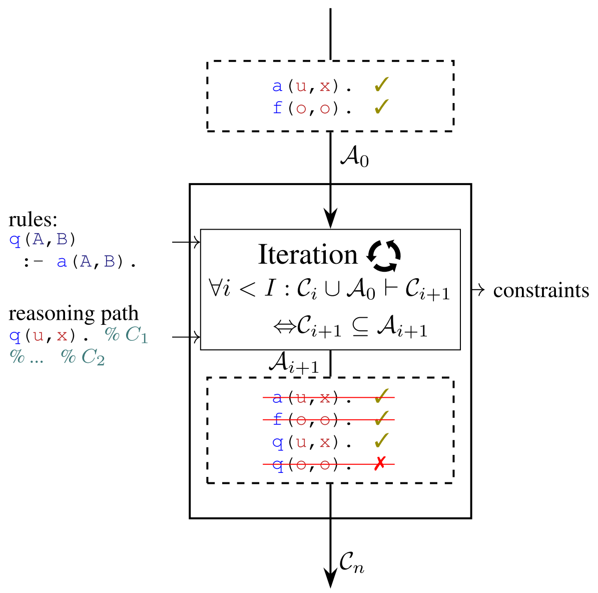


**FIGURE 5. “Improved” reasoner gadget. Like in the “naive version”, all rules are applied to every assertion in the input set, yielding the output set $\mathcal{A}_{i+1}$. Instead of feeding $\mathcal{A}_{i+1}$ directly back into the next iteration, we use a previously generated assertion set $\mathcal{C}_{i+1}$ as the input for the iteration, which has the unnecessary (—) and invalid (✗) assertions pruned.**

As explained in Section V, our naive reasoner’s most significant performance issue is the superexpontential growth of our assertion sets. The main cause of the assertion set’s combinatorial explosion is that we combine each assertion set with itself and do this many times over. Ideally, we minimize the size of each assertion set. In a R1CS, we cannot compare and branch, so it is impossible to remove assertions with the false flag.

However, we can achieve smaller assertion set sizes by doing a preliminary reasoning phase using a standard Datalog reasoner, and logging the assertion sets used in each iteration. As an additional argument to the proof, these assertions are passed per iteration. We then only have to compare each iteration with the assertions resulting from the rule applications on the assertions of the previous iteration. This approach effectively prunes the intermediate assertion sets, leaving only the minimal number of assertions to evaluate, instead of the ever-growing generated assertion sets from the naive reasoning gadget.

More specifically, the new method is as follows: first, we prove that the input assertion set $\mathcal{A}_0$ contains the first assertion set $\mathcal{C}_0$ from the assertions generated by the preliminary reasoner. Then we prove for each iteration $i$ that $\mathcal{C}_i \cup \mathcal{A}_0 \vdash \mathcal{C}_{i+1}$ by applying rules in the same manner as shown in the naive reasoner, and proving that the generated assertion set $\mathcal{A}_{i+1}$ fully contains $\mathcal{C}_{i+1}$. Finally, we prove that the query is in $\mathcal{C}_n$, the assertion set of the last iteration generated by the preliminary reasoner. This process is illustrated in Fig. 5.

For each iteration, we start our reasoning with an assertion set of minimal size instead of our generated assertion set containing a large number of invalid assertions. As a result, our assertion sets no longer grow superexponentially in our improved implementation. Each generated assertion set will now have a maximum size of $|\mathcal{C}|^2 \times |\mathcal{R}|$, where $\mathcal{C}$ is the largest assertion set.

In the naive implementation, the assertion set size was the biggest issue performance-wise. With that issue resolved, we can analyze our implementation more accurately. We will do this by counting the number of comparisons that happen in the constraint system. This correlates to the number of multipliers used in the R1CS, which in turn is the factor with the biggest influence on our proof’s performance.

Denote by $\mathcal{C}$ the largest assertion set, $\mathcal{A}$ the input assertions, $\mathcal{R}$ the ruleset and $I$ the number of iterations, then the total number of comparisons in the reasoner gadget can be computed as:

$$\begin{aligned} &O(|\mathcal{C}| \times |\mathcal{A}| && (\text{prove } \mathcal{C}_0 \subseteq \mathcal{A}) \\ &\quad +|\mathcal{C}|^2 \times |\mathcal{R}| && (\text{rule application}) \\ &\quad\quad \times(|\mathcal{C}| + |\mathcal{A}|) && (\text{subset comparison}) \\ &\quad\quad \times I) && (\text{iterations}) \qquad (1) \end{aligned}$$

We can use this formula to reason about other potential trade-offs and improvements.

## VII. IMPLEMENTATION

In the previous sections, we have omitted some details for the sake of simplicity. This section will give some in-depth remarks to clarify these details.

### A. FROM LOGIC TO ARITHMETIC

Until now, we have casually used logical operators and the term “comparison”. Because Bulletproofs works with R1CSs instead of Boolean logic and we are working with integer values instead of facts, we cannot use these concepts in practice. However, it is easy enough to map these concepts and values to arithmetic constraints. Suppose we assume 0 as true in our constraint system and any other value as false, then:

- a comparison becomes a subtraction,
- a disjunction becomes a multiplication, and
- a conjunction becomes a (randomized) addition.

When we want to compare two assertions, we subtract the predicates and each corresponding argument and take a randomized sum of the results. When we want to find out whether an assertion is in an assertion set, we compare the assertion to each assertion in the set, and take the disjunction of the results by multiplying them.

### B. DUMMY VALUES

For the underlying proof system to correctly process the input data, some metadata needs to accompany the proof. It would be possible to transmit the exact sizes of all assertion sets $\mathcal{C}_i$, each assertion and each iteration. However, this would leak a large amount of metadata. Instead, we only supply the size of the largest assertion set that is encountered, and the largest number of arguments that are ever encountered in any assertion. The prover ensures that each assertion set

$\mathcal{C}_i$ is as large as the largest assertion set (or another larger number) by padding the empty assertion slots with "dummy" assertions. These dummy values are system-wide, constant, random numbers.

We also pad every assertion to a number of arguments equal to the largest number of arguments of a single assertion. This allows us to compare assertions with a different number of arguments, and this also means that we do not have to supply more information-leaking metadata.

This provokes a small change in our proof of entailment in iterations: $\mathcal{C}_i \cup \mathcal{A}_0 \vdash \mathcal{C}_{i+1}$ is valid if each element of $\mathcal{C}_{i+1}$ is either in the generated assertion set $\mathcal{A}_{i+1}$ or a dummy value.

### C. NEGATION

Currently, Circuitree does not support negation for several reasons. Firstly, as one of its restictions, Datalog itself only supports certain kinds of negation. Having a Prolog-like negation would make our Datalog implementation non-computable.

Secondly, negation in Datalog is non-trivial. A negation in Datalog means taking the complement of an assertion set, which is each assertion not in that assertion set, which amounts to an infinite number of assertions.

Thirdly, since Circuitree has been designed for p2p applications, most of which assume an open world. The Open World Assumption states that, if a fact is not found in the database, it is not necessarily false.

This means that negation is a complex issue for Circuitree. Limited forms of negation are possible under certain restrictions. As such, we regard negation in Circuitree as a topic for future work.

### D. PRELIMINARY REASONER

The prover uses the preliminary reasoner to query the plaintext assertion set and obtain a history of how data are produced throughout its iterations. Each assertion remembers its provenance, i.e., the assertions from which it was generated. This way, the data are structured tree-like, with as root the query result and as leafs the required assertions in the initial assertion sets. Each non-leaf node in the tree is a generated assertion, whose children are the assertions it was generated from. By doing a breadth-first collection of this data, we can obtain the assertion sets for each iteration of the reasoner.

In a standard Datalog reasoner, the same assertion is only generated once. However, in our ZKP reasoner, we prune the assertions at every iteration. This means that we can only use assertions generated in the previous iteration. Because of this, we always need to prove every branch of the tree fully until we reach the leaves. We cannot shortcut this reasoning process with previously generated assertions.

It would be possible to include all the previous assertion sets in each iteration, since they stay valid, but this would generally result in a bigger performance loss than if we would prove identical subtrees multiple times. Instead, we prove the entire reasoning tree and include the leaves in the iterations provided by the preliminary reasoner. Because of this,

```
can_write(Bob, WallA) :-
  has_friend(Alice, Bob),
  has_wall(Alice, WallA).
can_write(Bob, WallA) :-
  has_member(GroupA, Bob),
  has_wall(GroupA, WallA).
has_member(Group, Alice) :-
  has_admin(Group, Alice).
```

**LISTING 2. The rules used for the "realistic" scenario. Participants can write on a "wall" if they are a member of the wall's group, or on a personal wall if the participants are friends. For a dataset {`has_admin(group, bob)`,`has_wall(group, groupWall)`}, the query `can_write(bob, groupWall)` will result in a reasoning of two iterations.**

we compare the values in each iteration $i$ with both the generated values $\mathcal{A}_i$, as the input data set $\mathcal{A}_0$.

### E. RULE CARDINALITY

In our implementations, we have assumed rules with a cardinality of two, which means they consist of two subgoals. Circuitree supports rules with any number of subgoals. However, for rules with more than two subgoals, we have to take a larger number of combinations, increasing our assertion set size significantly. With $S$ the largest number of subgoals, generalizing eq. (1) yields:

$$O(|\mathcal{C}| \times |\mathcal{A}| + |\mathcal{C}|^S \times |\mathcal{R}| \times (|\mathcal{C}| + |\mathcal{A}|) \times I)$$

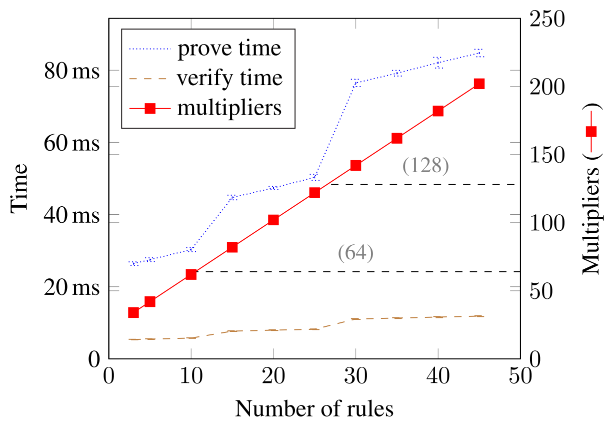
This means that rules with large numbers of subgoals cause a significant decrease in worst-case performance. However, it is possible to keep the exponent constant by splitting the rule subgoals and adding a degree of indirection. We can take two subgoals of the rule, and create a new rule for them. After one iteration step, an assertion will be generated that represents the conjunction of these values, but as one subgoal. The impact of this solution is at most one iteration step and rule per subgoal after the second, which in our formula will increase both $I$ and $R$. Especially with larger assertion set sizes, this solution is preferable to an increase in subgoals.

## VIII. EVALUATION

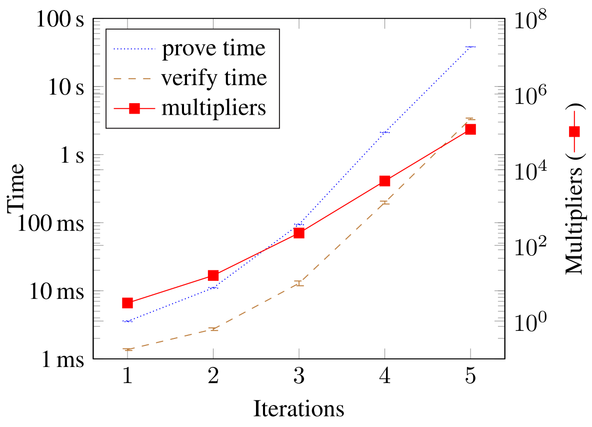
### A. PERFORMANCE

This section describes the performance of our Rust-based implementation of the improved version of Circuitree. We benchmark two different scenarios: the "realistic" scenario and the "treelike" scenario, and plot the results in Fig. 6. The realistic scenario is a system with rules for a group-based message board system, as depicted in Listing 2. The "treelike" scenario is a system with rules that generate exponentially many assertions, as depicted in Listing 3.

For the realistic benchmark, we evaluate the performance for a realistic scenario and how it is influenced by adding extra rules to the system. These extra rules will not lead to our query, but represent rules that are present in the system for other potential queries. These rules do not accomplish anything, but they will need to be applied like every other



(a) Performance of the "realistic" scenario. Jumps in prove and verify time are due to Bulletproofs padding the multipliers up to the next power of two.



(b) Performance of the worst-case "treelike" scenario, in which we double the assertion set in every iteration, thanks to an exponential amount of rules.

**FIGURE 6. Performance measurements for two different scenarios. Even the worst-case "treelike" scenario, in which the intermediate assertion sets grow exponentially, is still viable for 3 or 4 iterations.**

```
% Last iteration
pred0(X) :- pred1(X), pred2(X).
% First iteration
pred1(X) :- leaf0(X), leaf1(X).
pred2(X) :- leaf2(X), leaf3(X).
```

**LISTING 3. The rules used for the "treelike" scenario. Every predicate depends on two other predicates, and the leaf predicates are the input data for the system. In order to require $n$ iterations, $2^{n-1} - 1$ rules and $2^n$ leaves are generated. In this example, we require $n = 2$ iterations, as such $2^n = 4$ leaves. After $n$ iterations, the reasoning results in the query** `pred0(foo)`.

rule in Circuitree, and therefore impact the total performance. Since only assertions relevant to the query impact the gadget's performance, we do not vary the total number of assertions. As expected per eq. (1) and as measured and depicted in Fig. 6(a), our proof and verification time scales linearly with the number of rules in the system. With 45 rules in the system, the proof time is only 80 ms. Depending on the performance required by the application, other ZKP systems should be benchmarked and considered.

The treelike performance is a worst-case scenario for binary rules, with a varying number of iterations. In this scenario, our reasoning tree is a full binary tree, which means it has $2^{I-1}$ leaf assertions, $2^I - 1$ total assertions over all iterations and $2^{I-1} - 1$ rules in the system. Figure 6(b) shows that with four iterations or fewer, the computation time is acceptable considering, for example, the latency of a web application. As of five iterations, the query is derived using 31 distinct rules, and 32 leaf assertions, which we believe to be excessive for realistic applications, and will become noticeable by an end-user.

The benchmarks show that the system's performance is mostly based on the complexity of the reasoning tree and only minimally by the number of rules in the system. Furthermore, because only necessary data are used in the proof, the total amount of data in the system is irrelevant to the performance.

By keeping this information in mind, an application developer can aim to optimize their system by reducing the complexity of the reasoner. For example, the developer could add frequently used derived assertions to their system as data to drastically minimize the number of iterations needed.

## B. VERIFIABLE COMPUTATION SYSTEMS

Cairo [14] and similar high-level zero-knowledge systems are written, compiled and interpreted as imperative languages. Datalog is a declarative language, which is interpreted entirely differently from imperative languages. Datalog has a reasoner, whereas imperative languages use instructions. Because existing verifiable computation frameworks are designed with imperative languages in mind, it is not easy to fairly assess their relative performance.

Comparing an imperative language to a declarative querying language is a more or less meaningless question. As an example, it is similar to comparing performance in C to SQL. The two languages achieve different things in different ways. One could argue to write a database engine in C in order to compare them, but this then raises the question of what optimizations should be implemented, and whether it is still a C program and not just another querying language. Similarly, implementing a Datalog reasoner and Circuitree's optimizations in e.g. Cairo, can be seen as simply implementing Circuitree on Cairo, which would tell us very little about the difference between the two.

Furthermore, we count our optimization described in Section VI as part of Circuitree's contribution. If we were to write a Datalog reasoner in an existing system, it would arguably result in a R1CS system similar to our naive implementation. In this article, we have already made the comparison between the naive and improved version.

Finally, even though existing systems are on a higher level than constraints in R1CS, most of them do not support existing, high-level programming languages. In other words, the programmer cannot just write a program in e.g. C and pass this to Cairo, as would be the ideal scenario in verifiable computation. Indeed, this is exactly what Circuitree provides: Datalog as a high-level programming language, where the programmer does not interact with any zero-knowledge elements directly, and can even verify the execution of existing programs.

To conclude, it is very hard to present a fair comparison between our declarative system and an existing imperative system, because of the way such programming languages are processed by the computer.

## IX. CONCLUSION

This article presents the design, implementation and evaluation of *Circuitree*, a Datalog verifier that can prove statements about secret data in zero-knowledge. Datalog, a logic-based query language, is well suited to model access control rules. Instead of simulating a full Datalog engine in zero-knowledge as discussed in our naive approach, we increase performance and efficiency by only verifying that each reasoning step was carried out correctly.

We discuss several applications for Circuitree. One such application is access control in a p2p network with a knowledge graph. Because Circuitree uses binary predicates, it can be efficiently implemented on existing knowledge graphs. Another application is an implementation of a secure and privacy-preserving EU Digital COVID Certificate.

First results show that our approach allows for fast and efficient proofs and proof verification.

## X. FUTURE WORK

Although Circuitree's performance is acceptable for our current goals, when put in a real-world scenario, it might be necessary to investigate ways to further increase its performance. One of these ways would be to adapt the system to use a more efficient ZKP protocol, or one with more features. Examples are Groth16 [39] and Aurora [40]. Porting Circuitree to other R1CS-based proof systems should be relatively straight forward. It could also prove useful into researching whether it is viable to design a ZKP protocol that, instead of arithmetic or boolean values, can use Datalog primitives like assertions and lists.

Currently, Circuitree only supports a restricted fragment of Datalog. Extensions, such as negation, lists and mathematical operations should be considered for future addition. Additional features make Circuitree more powerful, and would allow for fair grounds of comparison with related systems by tackling existing and well-known problems. For example, it would be possible to compare a traveling salesman problem (TSP) implementation on Circuitree, to one on TinyRAM [10]. Research has to be conducted in order to investigate the possibility and performance impact of adding such features to Circuitree.

## ACKNOWLEDGMENT

*(Tom Godden and Ruben De Smet are co-first authors.)*

## REFERENCES

[1] P. Bischoff. *Report: 267 Million Phone Numbers & Facebook User IDs Exposed Online*. Comparitech. Accessed: Apr. 27, 2020. [Online]. Available: https://www.comparitech.com/blog/information-security/267-million-phone-numbers-exposed-online/

[2] E. Graham-Harrison and C. Cadwalladr. *Revealed: 50 Million Facebook Profiles Harvested for Cambridge Analytica in Major Data Breach*. [Online]. Available: https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election

[3] R. Satter. *U.S. Court: Mass Surveillance Program Exposed by Snowden was Illegal*. Accessed: Sep. 3, 2020. [Online]. Available: https://www.reuters.com/article/us-usa-nsa-spying-idUSKBN25T3CK

[4] M. Chase, T. Perrin, and G. Zaverucha, "The signal private group system and anonymous credentials supporting efficient verifiable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1445–1459.

[5] N. van Saberhagen, "Cryptonote v2.0," Tech. Rep., 2013.

[6] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification," Zerocoin Electr. Coin, Tech. Rep., 2016.

[7] D. Chaum, "Showing credentials without identification," in *Proc. Workshop Theory Appl. Cryptograph. Techn.* Berlin, Germany: Springer, 1985, pp. 241–244.

[8] M. Chase, E. Ghosh, S. Setty, and D. Buchner, "Zero-knowledge credentials with deferred revocation checks," Tech. Rep., 2020, p. 13. [Online]. Available: https://raw.githubusercontent.com/decentralized-identity/snark-credentials/master/whitepaper.pdf

[9] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, "ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash," in *Proc. USENIX Secur. Symp.*, vol. 10, 2010, pp. 193–206.

[10] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Proc. Annu. Cryptol. Conf.* Berlin, Germany: Springer, 2013, pp. 90–108.

[11] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "TinyRAM architecture specification, v0. 991," SCIPR Lab, Tech. Rep., 2013, p. 16.

[12] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 781–796.

[13] H. G. Navarro, "Design and implementation of the Circom 1.0 compiler," Universidad Complutense de Madrid, Madrid, Spain, Tech. Rep., 2020.

[14] L. Goldberg, S. Papini, and M. Riabzev. *Cairo—A Turing-Complete Stark-Friendly CPU Architecture*. Accessed: Sep. 2, 2021. [Online]. Available: https://eprint.iacr.org/2021/1063

[15] D. Mouris and N. G. Tsoutsos, "Zilch: A framework for deploying transparent zero-knowledge proofs," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 3269–3284, 2021.

[16] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language*. Accessed: Jan. 10, 2022. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[17] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *Proc. IEEE Symp. Secur. Privacy*, May 2018, pp. 315–334.

[18] O. Goldreich, S. Micali, and A. Wigderson, "How to play ANY mental game," in *Proc. 19th Annu. ACM Conf. Theory Comput.*, 1987, pp. 218–229.

[19] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM J. Comput.*, vol. 18, no. 1, pp. 186–208, Feb. 1989.

[20] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems," *J. ACM*, vol. 38, no. 3, pp. 690–728, Jul. 1991.

[21] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, "Mina: Decentralized cryptocurrency at scale," New York Univ. O(1) Labs, New York, NY, USA, Whitepaper, 2020, pp. 1–47.

[22] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable zero knowledge with, no., trusted setup," in *Proc. CRYPTO*, in Lecture Notes in Computer Science, A. Boldyreva and D. Micciancio, Eds. Cham, Switzerland: Springer, pp. 701–732.

[23] M. Hoffmann, M. Klooß, and A. Rupp, "Efficient zero-knowledge arguments in the discrete log setting, revisited," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 2093–2110.

[24] N. Li and J. C. Mitchell, "DATALOG with constraints: A foundation for trust management languages," in *Practical Aspects of Declarative Languages* (Lecture Notes in Computer Science), V. Dahl and P. Wadler, Eds. Berlin, Germany: Springer, vol. 2562, 2002, pp. 58–73.

[25] E. Pasarella and J. Lobo, "A datalog framework for modeling relationship-based access control policies," in *Proc. 22nd ACM Symp. Access Control Models Technol.*, Jun. 2017, pp. 91–102.

[26] A. Masoumzadeh and J. Joshi, "OSNAC: An ontology-based access control model for social networking systems," in *Proc. IEEE 2nd Int. Conf. Soc. Comput.*, Aug. 2010, pp. 751–759.

[27] P. W. L. Fong and I. Siahaan, "Relationship-based access control policies and their policy languages," in *Proc. 16th ACM Symp. Access Control Models Technol.*, 2011, pp. 51–60.

[28] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta, "PeerSoN: P2P social networking: Early experiences and insights," in *Proc. 2nd ACM EuroSys Workshop Social Netw. Syst.*, 2009, pp. 46–52.

[29] R. De Smet, A. Dooms, A. Braeken, and J. Pierson, "Glycos: The basis for a peer-to-peer, private online social network," in *Privacy and Identity Management Fairness, Accountability, and Transparency in the Age of Big Data* (IFIP Advances in Information and Communication Technology), E. Kosta, J. Pierson, D. Slamanig, S. Fischer-Hübner, and S. Krenn, Eds. Cham, Switzerland: Springer, vol. 547, 2019, pp. 123–136.

[30] K. Graffi and N. Masinde, "LibreSocial: A peer-to-peer framework for online social networks," in *Proc. Int. Conf. Cloud Edge Comput., Big Data Blockchain*, 2020, vol. 33, no. 8, Art. no. e6150.

[31] S. Z. R. Rizvi, P. W. L. Fong, J. Crampton, and J. Sellwood, "Relationship-based access control for an open-source medical records system," in *Proc. 20th ACM Symp. Access Control Models Technol.*, Jun. 2015, pp. 113–124, doi: 10.1145/2752952.2752962.

[32] L. Lesavre, P. Varin, P. Mell, M. Davidson, and J. Shook, "A taxonomic approach to understanding emerging blockchain identity management systems," 2019, *arXiv:1908.00929*.

[33] *BBS+ Signatures 2020*. Accessed: May 14, 2021. [Online]. Available: https://w3c-ccg.github.io/ldp-bbs2020/

[34] L. Zhou, V. Varadharajan, and M. Hitchens, "Achieving secure role-based access control on encrypted data in cloud storage," *IEEE Trans. Inf. Forensics Security*, vol. 8, no. 12, pp. 1947–1960, Dec. 2013.

[35] D. Boneh, X. Boyen, and H. Shacham, "Short group signatures," in *Proc. Annu. Int. Cryptol. Conf.* Berlin, Germany: Springer, 2004, pp. 41–55.

[36] M. Lodder and T. Looker. Jun. 15, 2020. *BBS+ Signature Scheme*. [Online]. Available: https://mattrglobal.github.io/bbs-signatures-spec/

[37] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," presented at the 30th USENIX Secur. Symp., 2021.

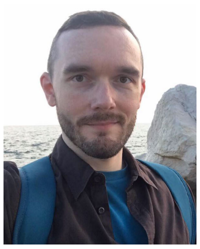
[38] G. G. Hillebrand, P. C. Kanellakis, H. G. Mairson, and M. Y. Vardi, "Undecidable boundedness problems for datalog programs," *J. Log. Program.*, vol. 25, no. 2, pp. 163–190, 1995.

[39] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proc. EUROCRYPT*, in Lecture Notes in Computer Science, M. Fischlin and J.-S. Coron, Eds. Berlin, Germany: Springer, pp. 305–326.

[40] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "Aurora: Transparent succinct arguments for R1CS," in *Proc. EUROCRYPT*, Y. Ishai and V. Rijmen, Eds. Cham, Switzerland: Springer, 2019, pp. 103–128.

**TOM GODDEN** received the M.Sc. degree in informatics from the Vrije Universiteit Brussel, in 2020. He is currently conducting research for his Ph.D. degree in engineering. His research interest includes user empowerment through privacy and security. He is also the designer of Circuitree, a system for verifiable computation in datalog, with the goal of applying it for privacy-preserving enforcement of access control. He is aiding in the design of Glycos, which sprouted the initial idea of Circuitree.

**RUBEN DE SMET** (Student Member, IEEE) received the M.Sc. degree in engineering from the Vrije Universiteit Brussel, in 2018, where he is currently pursuing the Ph.D. degree in the ETRO Department. His research interests include privacy enhancing technologies, specifically in online social networks, using zero-knowledge proofs, elliptic curves, and symmetric cryptography. He is the designer of Glycos, a peer-to-peer, private, online social network platform, and the organizer of the Belgium Rust Meetup Group.



**CHRISTOPHE DEBRUYNE** received the Ph.D. degree from the Vrije Universiteit Brussel, in 2013. He is currently an Assistant Professor in data representation and engineering with the Université de Liège. He also investigates methods and tools for data integration with knowledge graph technologies and the management thereof. His research interests include semantics, data integration, and knowledge graphs. He began to conducted a Ph.D. on collaborative ontology engineering. After his Ph.D., he spent several years working as a Research Fellow with the Trinity College Dublin, where he applied his research in both academic and industry-driven projects.

**THIBAUT VANDERVELDEN** received the M.Sc. degree in engineering with a specialization in embedded electronics and ICT from the Vrije Universiteit Brussel (VUB), in 2019, where he is currently pursuing the Ph.D. degree. He evaluates performance of IoT communication protocols, developed using the emerging Rust programming language with VUB. His current interests include security and privacy protocols for IoT, embedded programming, and low-power wireless protocols.

**KRIS STEENHAUT** (Member, IEEE) received the master's degree in engineering sciences, in 1984, the master's degree in applied computer sciences, in 1986, and the Ph.D. degree in engineering sciences from the Vrije Universiteit Brussel (VUB), in 1995. She is currently a Professor with the Department of Electronics and Informatics (ETRO) and the Department of Engineering technology (INDI), Faculty of Engineering, VUB, Belgium. Her research interests include the design, implementation and evaluation of wireless sensor networks for building automation, environmental monitoring, autonomous ground vehicle applications, mobility control and smart grids, and taking into account security and privacy aspects.

**AN BRAEKEN** received the M.Sc. degree in mathematics from the University of Gent, in 2002, and the Ph.D. degree in engineering sciences from the Research Group Computer Security and Industrial Cryptography (COSIC), KU Leuven, in 2006. She became a Professor with the Industrial Sciences Department, Erasmushoge School Brussel, in 2007. Since 2013, she has been with the Vrije Universiteit Brussel. Her current interests include security and privacy protocols for IoT, cloud and fog, blockchain, and 5G security. She is the coauthor of over 120 publications. She has been a member of the program committee for numerous conferences and workshops and a member of the editorial board for *Security and Communications Magazine*. In addition, since 2015, she has been an expert reviewer for several EU calls. She has cooperated and coordinated more than 12 national and international projects.