Anonymous Transactions with Revocation and Auditing in Hyperledger Fabric

Dmytro Bogatov[†]
Boston University
dmytro@bu.edu

Angelo De Caro
IBM Research, Zürich
angelo@zurich.ibm.com

Kaoutar Elkhiyaoui
IBM Research, Zürich
kao@zurich.ibm.com

Björn Tackmann
DFINITY
bjoern.tackmann@ieee.org

***Abstract*—In permissioned blockchain systems, participants are admitted to the network by receiving a credential from a certification authority. Each transaction processed by the network is required to be authorized by a valid participant who authenticates via their credential. Use case settings where privacy is a concern thus require proper privacy-preserving authentication and authorization mechanisms.**

Anonymous credential schemes are cryptographic mechanisms that allow a user to authenticate while showing only those attributes necessary in a given setting, which makes these schemes a great tool for authorizing transactions in permissioned blockchain systems based on the user's attributes. As in most setups of such systems there is one distinct certification authority for each organization in the network, the use of plain anonymous credential schemes still leaks the association of a user to their issuing organization. Camenisch, Drijvers, and Dubovitskaya (CCS 2017) therefore suggest the use of delegatable anonymous credential schemes, which allows to hide even that remaining piece of information.

We implement private transaction authorization in Hyperledger Fabric based on delegatable anonymous credentials. To this end, we provide a production-grade open-source implementation of the Camenisch et al. scheme with several optimizations. We then extend Fabric to support the scheme as an additional mechanism for authorizing transactions. Our solution supports revocation and auditing, making it ready for real-world deployment. Our performance measurements show that the scheme, while incurring an overhead in comparison to the less privacy-preserving ones, is practical for settings with enhanced privacy requirements.

I. Introduction

Blockchain systems allow two or more mutually distrustful parties to perform transactions by appending them to a shared ledger, without the need to rely on a trusted third party. The first and still most prominent use of blockchains is in the area of cryptocurrencies, where each transaction transfers fungible tokens between two or more parties. Blockchain systems used for cryptocurrencies are usually *permissionless*, meaning that joining the system does not require the parties to register their identity; everyone can participate.

[†] Work done while working at IBM Research, Zürich

Network and Distributed Systems Security (NDSS) Symposium 2020
23–26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

Many other application scenarios for blockchains, however, require the participants to be registered, and access to the blockchain system to be *permissioned*. For instance, use cases in the financial domain are restricted by know-your-customer (KYC) or anti-money-laundering (AML) regulations. Elections require the set of eligible voters to be known in order to prevent illegitimate voters from submitting votes, or any voter from double-voting. Enterprise blockchain systems accelerate processing of transactions in business networks with known participants. All above use cases require the transactions to be properly authorized by some member of the network. Note that *permissioned* does not mean *centralized*: the trust is still distributed among the participants of the network, the difference with permissionless networks is that joining the network becomes an explicit operation. For instance, instead of a centralized certification authority for all participants, a permissioned blockchain network uses multiple such authorities, one per organization, resulting in a federated model.

Use cases that require the submission of transactions to be authorized often still require the identity of the submitter to be hidden. The most salient example is elections, where re-voting (as a measure against coercion [1]) inherently requires the submitter of a vote to be anonymous. Financial use cases, where the transaction history of a user can leak sensitive personal information through usage patterns, are another good example. In such cases, the use of anonymous credential systems such as Identity Mixer [13] allow participants to authorize transactions by only revealing the attributes necessary for that particular transaction (such as being a registered voter or having passed KYC checks), while keeping all other attributes (such as name, address, or age) hidden.

Unfortunately, even the use of anonymous credentials can be insufficient. The reason is that each organization has its own certificate authority, and anonymity is only guaranteed relative to that authority. In other words, the particular certificate authority that issued a user's credential will still be obvious from the use of the credential for authorizing the transaction; in certain use cases, even this leakage may not be acceptable. A naïve approach to tackle this is to have one global certificate authority issuing anonymous credentials. This, however, means that all credentials are issued by the same central entity, essentially eliminating the federated management model that permissioned blockchains are supposed to bring.

As first observed by Camenisch, Drijvers, and Dubovitskaya [11], this is where *delegatable credentials* come in handy: in a delegatable credential scheme, a root authority delegates issuance of credentials to intermediate authorities in

such a way that using the credentials only reveals the root authority. In particular, the issuance of credentials for each organization can be delegated to a different certification authority. This allows to keep the management largely decentralized, while at the same time hiding the particular authority that issued a given credential.

In this paper, we use the work of Camenisch, Drijvers, and Dubovitskaya [11] as a stepping stone and provide an implementation of the delegatable credentials developed there for the widely-used permissioned blockchain platform Hyperledger Fabric [2]. Our contributions are three-fold:

- We provide a production-grade implementation of delegatable anonymous credentials in Go with several optimizations over [11]. For the submission, we made the library available in an anonymized repository [4]. It is also publicly available as open source.
- We extend the scheme of [11] by adding mechanisms for revocation of credentials and auditing of authorizations. The protocols are efficient, as they are based solely on ElGamal encryption [18] and Schnorr proofs [31].
- We enable private transactions via delegatable anonymous credentials in Hyperledger Fabric. This includes both the design of the relevant protocol parts and their implementation, which we also plan to provide as open source, based on the above library.

II. Related work

The most immediately related work is [11], which our paper builds on. That paper presents the cryptographic anonymous credentials scheme, proves its security, and provides initial performance numbers. It also discusses, but only on a general and conceptual level, the use of anonymous credentials in permissioned blockchains. Our paper extends [11] in three main directions: (a) we provide a production-grade implementation as open source, which includes multiple performance optimizations ([11] implemented just enough to run a simple performance test), (b) we integrate anonymous credentials in the Hyperledger Fabric protocols, which in fact requires a different approach than described in [11], (c) we cover practically-relevant functionality such as revocation and auditing.

After the publication of [11], two further papers on delegatable credentials were published, namely by Blömer and Bobolz [10] and by Crites and Lysyanskaya [16]. Both claim stronger security properties compared with [11] by also supporting an anonymous delegation phase; this feature is however not required in our setting where the user and the intermediate authority know each other. On the flip side, the scheme in [10] supports only a fixed number of attributes that is determined during setup, whereas we want to be able to dynamically add attributes per intermediate authority. Furthermore, the paper does not describe a full instantiation of the protocol and the credential presentation, and when instantiated, appears less efficient than the one for [11]. The scheme in [16] does not support attributes, which makes it unsuitable for our application.

Sovrin [34] also combines anonymous credentials with a permissioned blockchain system. While we use anonymous credentials to authorize transactions within a blockchain systems, the Sovrin platform for self-sovereign identity instead uses the blockchain as a building block for the use of anonymous credentials, similarly to the anonymous decentralized credentials work of Garman, Green, and Miers [19]. The two approaches thus serve different use cases. In the context of Sovrin, there is also an implementation of [11] in Rust [23], which appears to be in its earlier stages.

A growing segment of the research literature on blockchain systems aims to improve the confidentiality of transactions using techniques such as zero-knowledge proofs (e.g. [8], [20], [29], [36]), different types of state channels (e.g. [3], [17]), or multi-party computation (e.g. [9]). While the underlying cryptographic machinery, particularly in the work on zero-knowledge proofs, is similar to the protocols we use here, achieving confidentiality of transactions is orthogonal to achieving privacy of participants, and eventually privacy-friendly permissioned blockchain systems will have to combine both.

III. Background

A. Blockchain systems

The purpose of a blockchain is to implement an immutable[1] append-only ledger that is maintained by a network of mutually distrustful parties. As a data structure, the ledger is a chain of blocks such that each block refers to its predecessor by including its hash, implementing a total order on the blocks. The parties continuously extend the chain by running a consensus mechanism (e.g., proof of work or PBFT) to decide on the respective next block. Blocks contain transactions that have been submitted by clients for inclusion in the ledger.

Blockchains are either *permissionless* or *permissioned*. In a permissionless blockchain such as Bitcoin [27] or Ethereum [35], anyone can run a peer that joins the network, participates in consensus, and validates transactions. Clients can submit their transactions anonymously (or rather: pseudonymously). Trust in such networks is established via consensus mechanisms that are based on proofs of work (e.g., [27], [35]) or proofs of stake (e.g., [15], [25]), which penalize misbehaving parties either by requiring them to expend a lot of computational power in the case of proof of work or losing their money in the case of proof of stake.

Permissioned blockchains, on the other hand, leverage identity management mechanisms to counter misbehavior, foster trust, and facilitate governance. Most permissioned blockchain systems build on variants (e.g., [21], [33]) of the well-studied and performant PBFT [14] algorithm to reach consensus. Permissioned blockchains are particularly well-suited for applications where participant identities are required either inherently or by regulation, or those with high performance requirements. This includes enterprise applications in logistics and supply-chain management, but also use cases in the financial and governmental domains. Examples of prominent permissioned blockchain platforms include Hyperledger Fabric [2] and Quorum [24].

[1] There has been work on implementing mutable ledgers, while preserving other properties such as the requirement for consensus for modification (cf. [5], [30]). All blockchain systems used in practice implement immutable ledgers.

B. Hyperledger Fabric

Fabric is a permissioned blockchain platform that is developed under the umbrella of the Hyperledger project within the Linux Foundation. Fabric is widely known for its modular and scalable architecture. We briefly describe the architecture of Fabric, focusing on those components relevant to transaction authorization and anonymous transactions through Identity Mixer in particular. We refer to the original paper [2] for a detailed description of the complete protocol and system.

A Fabric deployment involves multiple mutually distrustful organizations. Each organization corresponds to one trust domain and manages one complete stack of platform components. The components that are online during transaction processing are orderers, peers, and clients. *Clients* invoke transactions and observe their results; they constitute the link between the blockchain and the outside world. *Peers* execute and validate transactions; they process the application data. *Orderers* receive transactions, put them into blocks, run a consensus algorithm to determine their order, and distribute the blocks to the peers. Orderers ignore the transaction contents, they merely put then in order. Each organization also runs a *membership service provider (MSP)*, which is responsible for maintaining and managing identities of all participants of that organization. This includes issuing credentials for authentication and authorizations and their revocation when the need arises.

Fabric has a unique three-phase transaction flow called *Execute-Order-Validate*. Each chaincode (i.e., smart contract) has associated *endorsers*, peers that execute this chaincode. The *endorsement policy* associated with the chaincode specifies the minimum requirements for replicated execution. For instance, a sample endorsement policy could specify that at least one peer from each organization participating in the network must endorse.

In the *Execute* phase, a client invokes a chaincode (i.e., smart contract) by sending a transaction proposal to the endorsers of that chaincode. The endorsers execute the chaincode and sign the chaincode's read and write sets. After collecting enough endorsements (i.e., signatures on consistent read/write sets), the client constructs a transaction that contains the proposal, the read/write sets and the endorsements and signs it using its MSP identity (i.e., a credential obtained from an MSP).

In the *Order* phase, the client sends the signed transaction together with some metadata to the ordering service, which includes the transaction in a block and broadcasts the ordered transactions to the peers in the network.

In the *Validate* phase, the peers verify if each transaction received from the ordering service satisfies the endorsement policy of its chaincode. The peers also update their local state according to the write sets specified in the transaction.

One main advantage of this architecture in comparison with standard Order-Execute architectures is its scalability. First, the different tasks have different resource requirements (compute, memory, network) and can be run on specialized nodes. Second, resource-heavy tasks such as executing chaincodes can be scaled out by adding more nodes that process transactions in parallel. Another advantage is that chaincode can be programmed in general-purpose programming languages, because — in contrast to Order-Execute — non-determinism does not violate the consistency of the overall system, but only affects the liveness of that particular chaincode.

C. Authentication, authorization, and Identity Mixer in Fabric

The default Fabric MSP is based on X.509 certificates — an identity is an X.509 certificate and its validation/revocation follows the X.509 standard. This approach is efficient, flexible and scalable — organizations may have hierarchical CAs which translate to hierarchical MSPs. Each transaction (as a data structure) has two specific fields related to transaction authorization: the *Creator* (i.e., identity of the client invoking the transaction) and *Signature* (i.e., authorization of the transaction) fields. As each transaction thus carries the identity of its origin in the form of a certificate and a signature, the X.509 implementation compromises the anonymity and the privacy of clients.

To remedy this issue, Fabric uses Identity Mixer (idemix for short), an anonymous credentials scheme based on the protocols in [12]. The idemix-based MSP protocol enables clients to sign transactions anonymously. Instead of an X.509 certificate, an idemix MSP issues a special credential containing a set of attributes. To sign a transaction, the holder of an idemix identity generates a non-interactive zero-knowledge (NIZK) proof that she received a credential from idemix that certifies her attributes. More specifically, if *Alice* is a member of an organization *Org* whose members are authorized to submit certain transactions, then *Alice* proves that she possesses an idemix credential from her MSP that attests that she is a member of *Org*.

As discussed in the introduction, even the use of anonymous credentials is sometimes not sufficient from a privacy perspective. Namely, the current implementation of idemix leaks the identity of the MSP that issued the anonymous credential. To mitigate this leakage, we implement a stronger mechanism based on the delegatable anonymous credentials scheme from [11].

D. Notation

Here we present the notation used to describe the schemes. Let $\mathbb{Z}_q$ be a set of natural numbers from 0 to $q-1$. Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be the groups of prime order q, such that there exists an efficient bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. Let FEXP and $\hat{t}$ be the final exponentiation and Miller's loop operations respectively, such that $e = \mathrm{FEXP} \circ \hat{t}$. Let $\leftarrow\!\!\$$ describe an operation of random sampling, and let pk and sk describe public and secret keys respectively. When the subscript is omitted, the keys are implicitly considered as the user's. Let system parameters sp be a set of values shared among all entities. sp is implicitly passed to all algorithms and implicitly supplied to hash routines in zero-knowledge proofs. sp includes group generators, pairing functions, authorities' public keys and some extra scheme-specific values.

E. Groth signatures

The delegatable anonymous credential scheme in [11] uses Groth signatures introduced in [22], as they are structure

preserving (and therefore allow for efficient NIZK proofs) and have a particular algebraic structure that will allow for building certificate chains by signing public keys without leaving the algebraic representation. The following is a brief description of GROTH algorithms.

- SETUP $\rightarrow_{\$} \mathsf{sp}$
  Let $\Lambda^{\star} = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e)$ and $y_i \leftarrow_{\$} \mathbb{G}_2$ for $i = 1, \ldots, n$, where n is the maximum size of the message. Output system parameters $\mathsf{sp} = (\Lambda^{\star}, \{y_i\}_{i=1}^{n})$.
- GENERATE$(\mathsf{sp}) \rightarrow_{\$} (\mathsf{sk}, \mathsf{pk})$
  Output secret and public keys $\mathsf{sk} \leftarrow_{\$} \mathbb{Z}_q$ and $\mathsf{pk} = g_1^{\mathsf{sk}}$.
- SIGN$(\mathsf{sk}; \vec{m}) \rightarrow_{\$} \sigma$
  Let $\vec{m} = (m_1, \ldots, m_n) \in \mathbb{G}_2^n$. Set the following

  $$\rho \leftarrow_{\$} \mathbb{Z}_q^{\star} \quad r := g_1^{\rho} \quad s := (y_1 \cdot g_2^{\mathsf{sk}})^{\frac{1}{\rho}} \quad t_i := (y_i^{\mathsf{sk}} \cdot m_i)^{\frac{1}{\rho}}$$

  Output the signature $\sigma = (r, s, t_1, \ldots, t_n)$.
- VERIFY$(\mathsf{pk}, \sigma, \vec{m}) \rightarrow \{0, 1\}$
  Let $\vec{m} \in \mathbb{G}_2^n$ and $\sigma = (r, s, t_1, \ldots, t_n) \in \mathbb{G}_1 \times \mathbb{G}_2^{n+1}$, output the result of

  $$e(r, s) = e(g_1, y_1) \cdot e(\mathsf{pk}, g_2) \wedge \bigwedge_{i=1}^{n} e(r, t_i) = e(\mathsf{pk}, y_i) \cdot e(g_1, m_i)$$

- RANDOMIZE$(\sigma) \rightarrow_{\$} \sigma'$

  $$\rho' \leftarrow_{\$} \mathbb{Z}_q \qquad r' := r^{\rho'} \qquad s' := s^{\frac{1}{\rho'}} \qquad t_i' := t_i^{\frac{1}{\rho'}}$$

  Output randomized signature $\sigma' = (r', s', t_1', \ldots, t_n')$.

Notice that the public keys are in $\mathbb{G}_1$ whereas the messages are in $\mathbb{G}_2$. To be able to support chaining (and thereby delegation) using GROTH we need to switch the key space and message space. That is, we move from one delegation level to the next by swapping $\mathbb{G}_1$ and $\mathbb{G}_2$.

We call these schemes in the following $\text{GROTH}_{\mathsf{odd}}$ and $\text{GROTH}_{\mathsf{even}}$ for respective levels of credentials.

F. Delegatable anonymous credentials scheme

In the scheme from [11], all parties (the root, intermediate authorities and the users) generate pairs of secret and public keys. A 1-Level delegatee (an intermediate authority or a user) contacts the root to obtain a credential (i.e., signature) to bind her public key to her attributes. Once a 1-Level delegatee gets her credentials, she herself becomes a delegator and can issue credentials for 2-Level delegates. This delegation process may continue for an arbitrary number of levels increasing the length of the credential chain.

The holder of a credential uses non-interactive zero-knowledge (NIZK) proofs to sign messages anonymously. Notably, signing a message m consists of proving in zero-knowledge the following facts:

- the signer owns the credentials;
- the Schnorr-like generated signature is valid for message m;
- inductively, all adjacent levels are legitimate (one was delegated from the other); and
- at the end of the induction, the top-level public key is that of the root authority.

During the proof generation, the signer can choose which attributes to disclose and which to keep secret. It is possible, although not very useful, to reveal or hide all attributes. Thanks to the properties of zero-knowledge proofs, this signing process is randomized and every time it is executed it yields new and unlinkable signatures.

More formally, the scheme SCHEME consists of the following algorithms:

- KEYGEN $\rightarrow_{\$} (\mathsf{sk}, \mathsf{pk})$
  Generates a pair of secret and public keys as $\mathsf{sk} \leftarrow_{\$} \mathbb{Z}_q$ and $\mathsf{pk} = g^{\mathsf{sk}}$.
- NEWCREDS$(\mathsf{pk}_{\mathsf{root}}, \mathsf{sk}_{\mathsf{root}}) \rightarrow \mathsf{cred}$
  Produces an empty credentials data structure with the root's key pair in it.
- DELEGATE$(\mathsf{cred}, \mathsf{sk}_{\mathsf{delegator}}, \mathsf{pk}_{\mathsf{delegatee}}, \vec{a}) \rightarrow_{\$} \mathsf{cred}'$
  Produces the credentials of the next level that bind attributes $\vec{a}$ to public key $\mathsf{pk}_{\mathsf{delegatee}}$.
- PROVE$(\mathsf{cred}, \mathsf{sk}_{\mathsf{prover}}, \mathsf{pk}_{\mathsf{root}}, \vec{a}_{\mathsf{revealed}}, m) \rightarrow_{\$} \mathfrak{P}_{\mathsf{cred}}$
  Generates a zero-knowledge proof that shows the validity of the credentials under the public key of the root authority, proves the ownership of the credentials and simultaneously signs the message.
- VERIFY$(\mathfrak{P}_{\mathsf{cred}}, \mathsf{pk}_{\mathsf{root}}, \vec{a}_{\mathsf{revealed}}, m) \rightarrow \{0, 1\}$
  Verifies the proof under the public key of the root authority.

1) Delegation: Let us look at the delegation procedure in more details.

- A root authority $\mathcal{R}$ generates a key pair $(\mathsf{pk}_r, \mathsf{sk}_r)$.
- An intermediate authority $\mathcal{I}$ generates a key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ and requests *1-Level* credentials supplying her public key pk_i and her attribute vector $\vec{a_i}$.
- $\mathcal{R}$ signs $\mathcal{I}$'s public key and vector of attributes and returns the resulting signature $\sigma_i = \text{SIGN}_{\mathsf{sk}_r}(\mathsf{pk}_i \| \vec{a_i})$.
- $\mathcal{I}$ now has *1-Level* credential $\mathsf{cred}_i = (\sigma_i, \vec{a_i}, \mathsf{pk}_i, \mathsf{sk}_i)$.
- To request a *2-Level* credential, a delegatee (user $\mathcal{U}$) generates a keypair $(\mathsf{pk}_u, \mathsf{sk}_u)$ and sends pk_u to the delegator $\mathcal{I}$.
- Delegator $\mathcal{I}$ computes $\sigma_u = \text{SIGN}_{\mathsf{sk}_i}(\mathsf{pk}_u \| \vec{a_u})$ and sends both signatures, public keys and attribute vectors to the delegatee $\mathcal{U}$. Accordingly, *2-Level* credentials of user $\mathcal{U}$ are defined as $\mathsf{cred}_u = (\sigma_i, \sigma_u, \vec{a_i}, \vec{a_u}, \mathsf{pk}_i, \mathsf{pk}_u, \mathsf{sk}_u)$.
- This process can repeat, the resulting *L-Level* credential is $\mathsf{cred}_L = (\langle \sigma_i, \vec{a_i}, \mathsf{pk}_i \rangle_{i=1}^{L}, \mathsf{sk}_L)$.

Note that to be able to prove in ZK that a given delegatable credential is valid, the underlying signature scheme needs to be structure preserving; this is the reason for using Groth signatures introduced in [22].

Algorithm 1 Improved proof generation. Green code is refactored, red code corrects mistakes in the original code.

procedure CREDPROVE($\langle r_i, s_i, \langle t_{i,j}\rangle_{j=1}^{n_i+1}\rangle_{i=1}^{L}$, csk, $\langle \mathsf{cpk}_i\rangle_{i=1}^{L}$, $\langle a_{i,j}\rangle_{i=1,\ldots,L;j=1\ldots,n_i}$, D, $\mathsf{sk}_{\mathsf{nym}}$, m)
  for $i = (1, \ldots, L)$ **do**
    $\rho_{\sigma_i} \leftarrow_{\$} \mathbb{Z}_q$, $r'_i := r_i^{\rho_{\sigma_i}}$, $s'_i := s_i^{\frac{1}{\rho_{\sigma_i}}}$
    for $j = 1, \ldots, n_i + 1$ **do**
      $t'_{i,j} := t_{i,j}^{\frac{1}{\rho_{\sigma_i}}}$
  $\langle \rho_{s_i}, \langle \rho_{t_{i,j}}\rangle_{j=1}^{n_i+1}\rangle_{i=1}^{L}, \langle \rho_{a_{i,j}}\rangle_{(i,j)\notin D}, \langle \rho_{\mathsf{cpk}_i}\rangle_{i=1}^{L-1}, \rho_{\mathsf{csk}}, \rho_{\mathsf{nym}} \leftarrow_{\$} \mathbb{Z}_q$
  for $i = (1, \ldots, L)$ **do**
    if $i \bmod 2 = 1$ **then**
      $g_1 := \mathsf{sp}.g_1$, $g_2 := \mathsf{sp}.g_2$, $y := \mathsf{sp}.y_1$
    else
      $g_1 = \mathsf{sp}.g_2$, $g_2 = \mathsf{sp}.g_1$, $y = \mathsf{sp}.y_2$
    $\mathsf{com}_{i,1} := e\left(g_1, r_i\right)^{\rho_{\sigma_i}\cdot\rho_{s_i}} \left[\cdot e\left(g_1^{-1}, g_2\right)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
    $\mathsf{com}_{i,2} := e\left(g_1, r_i\right)^{\rho_{\sigma_i}\cdot\rho_{t_{i,1}}} \cdot e\left(g_1, g_2^{-1}\right)^{\rho_{\mathsf{cpk}_i}} \left[\cdot e\left(y_1, g_2\right)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
    for $j = (1, \ldots, n_i)$ **do**
      if $(i, j) \in D$ **then**
        $\mathsf{com}_{i,j+2} := e\left(g_1, r_i\right)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}} \left[\cdot e\left(y_{j+1}, g_2\right)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
      else
        $\mathsf{com}_{i,j+2} := e\left(g_1, r_i\right)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}} \cdot e\left(g_1, g_2^{-1}\right)^{\rho_{a_{i,j}}} \left[\cdot e\left(y_{j+1}, g_2\right)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
  $\mathsf{com}_{\mathsf{nym}} := g_1^{\rho_{\mathsf{cpk}_L}} h^{\rho_{\mathsf{nym}}}$
  $c := \text{HASH}(\mathsf{ipk}, \langle r'_i, \langle \mathsf{com}_{i,j}\rangle_{j=1}^{n_i+2}\rangle_{i=1}^{L}, \mathsf{com}_{\mathsf{nym}}, \langle a_{i,j}\rangle_{(i,j)\in D}, m)$
  for $i = (1, \ldots, L)$ **do**
    if $i \bmod 2 = 1$ **then** $g := g_1$
    else $g = g_2$
    $\mathfrak{p}_{s_i} := g^{\rho_{s_i}} s'^{c}_i$, $\left[\mathfrak{p}_{\mathsf{cpk}_i} := g^{\rho_{\mathsf{cpk}_i}} \mathsf{cpk}_i^{c}\right]_{i\neq L}$, $\left[\mathfrak{p}_{\mathsf{csk}} := \rho_{\mathsf{cpk}_L} + c \cdot \mathsf{csk}\right]_{i=L}$, $\left[\mathfrak{p}_{\mathsf{nym}} := \rho_{\mathsf{nym}} + c \cdot \mathsf{sk}_{\mathsf{nym}}\right]_{i=L}$
    for $j = 1, \ldots, n_i + 1$ **do**
      $\mathfrak{p}_{t_{i,j}} := g^{\rho_{t_{i,j}}} t'^{c}_{i,j}$
    for $j : (i, j) \notin D$ **do**
      $\mathfrak{p}_{a_{i,j}} := g^{\rho_{a_{i,j}}} a_{i,j}^{c}$
  return $c, \langle r'_i, \mathfrak{p}_{s_i}, \langle \mathfrak{p}_{t_{i,j}}\rangle_{j=1}^{n_i+1}\rangle_{i=1}^{L}, \langle \mathfrak{p}_{a_{i,j}}\rangle_{(i,j)\notin D}, \langle \mathfrak{p}_{\mathsf{cpk}_i}\rangle_{i=1}^{L-1}, \mathfrak{p}_{\mathsf{nym}}, \mathfrak{p}_{\mathsf{csk}}$

Algorithm 2 Improved proof verification. Green code is refactored, red code corrects mistakes in the original code.

procedure CREDVERIFY($c, \langle r'_i, \mathfrak{p}_{s_i}\langle \mathfrak{p}_{t_{i,j}}\rangle_{j=1}^{n_i+1}\rangle_{i=1}^{L}, \langle \mathfrak{p}_{a_{i,j}}\rangle_{(i,j)\notin D}, \langle \mathfrak{p}_{\mathsf{cpk}_i}\rangle_{i=1}^{L-1}, \mathfrak{p}_{\mathsf{csk}}, \langle a_{i,j}\rangle_{(i,j)\in D}, D, \mathsf{pk}_{\mathsf{nym}}, m$)
  for $i = (1, \ldots, L)$ **do**
    if $i \bmod 2 = 1$ **then**
      $g_1 := \mathsf{sp}.g_1$, $g_2 := \mathsf{sp}.g_2$, $y := \mathsf{sp}.y_1$
    else
      $g_1 = \mathsf{sp}.g_2$, $g_2 = \mathsf{sp}.g_1$, $y = \mathsf{sp}.y_2$
    $\mathsf{com}_{i,1} := e\left(\mathfrak{p}_{s_i}, r'_i\right) \cdot e\left(y_1, g_2\right)^{-c} \left[\cdot e\left(g_1^{-1}, \mathfrak{p}_{\mathsf{cpk}_i}\right)\right]_{i\neq 1} \left[\cdot e\left(g_1, \mathsf{ipk}\right)^{-c}\right]_{i=1}$
    $\mathsf{com}_{i,2} := e\left(\mathfrak{p}_{t_{i,1}}, r'_i\right) \left[\cdot e\left(y_1, \mathfrak{p}_{\mathsf{cpk}_{i-1}}\right)\right]_{i\neq 1} \left[\cdot e\left(y_1, \mathsf{ipk}\right)^{-c}\right]_{i=1} \left[\cdot e\left(\mathfrak{p}_{\mathsf{cpk}_i}, g_2^{-1}\right)\right]_{i\neq L} \left[\cdot e\left(g_1, g_2^{-1}\right)^{\mathfrak{p}_{\mathsf{csk}}}\right]_{i=L}$
    for $j = (1, \ldots, n_i)$ **do**
      if $(i, j) \in D$ **then**
        $\mathsf{com}_{i,j+2} := e\left(\mathfrak{p}_{t_{i,j+1}}, r'_i\right) \cdot e\left(a_{i,j}, g_2\right)^{-c} \left[\cdot e\left(y_{j+1}, \mathfrak{p}_{\mathsf{cpk}_{i-1}}\right)\right]_{i\neq 1} \left[\cdot e\left(y_{j+1}, \mathsf{ipk}\right)^{-c}\right]_{i=1}$
      else
        $\mathsf{com}_{i,j+2} := e\left(\mathfrak{p}_{t_{i,j+1}}, r'_i\right) \cdot e\left(\mathfrak{p}_{a_{i,j}}, g_2^{-1}\right) \left[\cdot e\left(y_{j+1}, \mathfrak{p}_{\mathsf{cpk}_{i-1}}\right)\right]_{i\neq 1} \left[\cdot e\left(y_{j+1}, \mathsf{ipk}\right)^{-c}\right]_{i=1}$
  $\mathsf{com}_{\mathsf{nym}} := g_1^{\mathfrak{p}_{\mathsf{csk}}} h^{\mathfrak{p}_{\mathsf{nym}}} \mathsf{pk}_{\mathsf{nym}}^{-c}$
  $c' := \text{HASH}(\mathsf{ipk}, \langle r'_i, \langle \mathsf{com}_{i,j}\rangle_{j=1}^{n_i+2}\rangle_{i=1}^{L}, \mathsf{com}_{\mathsf{nym}}, \langle a_{i,j}\rangle_{(i,j)\in D}, m)$
  return $c = c'$

2) Proof generation: Proving the knowledge and validity of credentials is equivalent to generating the following NIZK:

$$\mathfrak{P}_{\mathsf{cred}} \leftarrow\!\!\$\, \mathrm{NIZK}\{(\sigma_1, \ldots, \sigma_L, \mathsf{pk}_1, \ldots, \mathsf{pk}_L, \langle a' \rangle_{\mathsf{hidden}}, \sigma_m) :$$
$$\bigwedge_{i=1,3,\ldots}^{L} \textsc{Groth}_{\mathsf{odd}}.\textsc{Verify}(\mathsf{pk}_{i-1}; \sigma_i; \mathsf{pk}_i, a'_{i,1}, \ldots, a'_{i,n_i})$$
$$\bigwedge_{i=2,4,\ldots}^{L} \textsc{Groth}_{\mathsf{even}}.\textsc{Verify}(\mathsf{pk}_{i-1}; \sigma_i; \mathsf{pk}_i, a'_{i,1}, \ldots, a'_{i,n_i})$$
$$\wedge\, \textsc{Schnorr}.\textsc{Verify}(\mathsf{pk}_L; \sigma_m; m)\}$$

Note that additionally to proving the validity of the credential chain, proof generation signs an input message m using a Schnorr-like procedure. This allows the prover to show that she knows the secret key corresponding to the last level credential.

Proof generation algorithm [11, Figure 4] is a non-trivial math-heavy routine. We only provide a quick overview and we refer interested readers to the original paper [11].

All GROTH signatures are randomized every time a new proof is generated. The code is split into halves — for odd and even levels of credentials. The first part computes a set of commitments — one per each s and t value from GROTH. The commitments are then hashed along with the public information and the message. The last part computes $\mathfrak{p}$-values using the hash value, the verifier will use them to reconstruct the commitments.

3) Proof verification: Verification is conceptually simpler [11, Figure 5]. The code is again split in halves of odd and even levels. $\mathfrak{p}$-values are used to reconstruct all commitments. These commitments are then hashed the same way as in proof generation. The hashes are then compared and verification succeeds if they are equal. Given the collision resistance of cryptographic hashes, any discrepancy in the hash input (whether a commitment or any public parameter including the message) will yield a different value that will fail the verification.

IV. IMPROVEMENTS TO CREDENTIALS SCHEME

While implementing the delegatable anonymous credentials scheme, we discovered several simplifications and optimizations that improved readability and performance of our code. This section presents our improvements.

A. Refactored pseudocode

Following the pseudocode [11, Figures 4 and 5] precisely we have found out that the verification always fails. We were able to spot mistakes and provide a corrected version in Algorithms 1 and 2. Additionally, we have refactored the pseudocode by adjusting g_1, g_2 and y values on each loop iteration, simplifying the code and reducing its size in half.

B. Parallelization

We have noticed that the most heavy operations in the code are computations of commitments. We have also noticed that these computations are independent, and therefore can be easily parallelized. Instead of computing the commitments eagerly, our algorithm schedules a computation and puts it in a queue. Before computing a hash of the commitments, the program waits for the last computation to finish, signaling that the commitment set is computed. We find this task granularity optimal in this scenario — the task takes long enough to neglect a cost of spawning an extra thread and it is small enough that the system can uniformly disperse the tasks among available resources.

C. Miller's loop and final exponentiation

Camenisch, Drijvers, and Dubovitskaya [11] mention that when computing a product of pairings it makes sense to compute Miller's loop first on all pairs, multiply them, and only then apply final exponentiation. However, the authors used this tactic only on a fraction of computations. We have discovered a way to extend this optimization and apply it globally.

The idea is to convert every pairing product to a set of Miller's loops and apply final exponentiation once per such product. The trick is to use bilinearity of Miller's loop to put exponents inside the pairings. For example, the following are equivalent:

$$\prod_i e\left(a_i, b_i\right)^{c_i} = \textsc{fexp}\left(\prod_i \hat{t}(a_i^{c_i}, b_i)\right) = \textsc{fexp}\left(\prod_i \hat{t}(a_i, b_i^{c_i})\right)$$

Since exponentiations are cheaper in $\mathbb{G}_1$ than in $\mathbb{G}_2$, we decided to exponentiate elements in $\mathbb{G}_1$. See Algorithm 3.

Algorithm 3 e-product optimization

Require: $a_i \in \mathbb{G}_1$, $b_i \in \mathbb{G}_2$, $c_i \in \mathbb{Z}_q \cup \perp$ for $L = 1, \ldots, n$
Ensure: $\textsc{eProduct}(\langle a_i, b_i, c_i \rangle_{i=1}^n) = \prod_{i=1}^n e(a_i, b_i)^{c_i}$

procedure $\textsc{eProduct}(\langle a_i, b_i, c_i \rangle_{i=1}^n)$
  $r := 1_T \in \mathbb{G}_T$ ▷ an *identity element*
  for $i = (1, \ldots, n)$ **do**
    if $c_i \neq \perp$ **then**
      $a_i := a_i^{c_i}$
  for $i = (1, 3, \ldots, n)$ **do**
    if $a_{i+1} \neq \perp$ **then**
      ▷ $\hat{t}_2$ is a more efficient version of $\hat{t} \cdot \hat{t}$
      $r := r \cdot \hat{t_2}\left(a_i, b_i, a_{i+1}, b_{i+1}\right)$
    else
      $r := r \cdot \hat{t}\left(a_i, b_i\right)$
  return $\textsc{fexp}\left(r\right)$

D. Attempt to aggregate commitments

In an attempt to improve performance even further, we considered reducing the number of FEXP calls in the commitment phase by multiplying the commitments and supplying to the hash the aggregate instead of the individual commitments. In this manner, instead of calling FEXP to compute each commitment, FEXP is only called once. However, when this applied naïvely it brings about a security flaw: one could easily find different tuples of commitments (which are essentially Pedersen commitments [28] and thus malleable) that yield the same aggregate, making the NIZK proof malleable. The workaround is to aggregate only the commitments that do not share any base. We have implemented the optimization and discovered that it does not result in any tangible improvement for small and medium input sizes (size as in number of levels and attributes).

Algorithm 4 Pseudonym and public key possession proof algorithms

1: **procedure** MAKENYM(sk)
2: $\mathsf{sk}_{\mathsf{nym}} \leftarrow_{\$} \mathbb{Z}_q$
3: $\mathsf{pk}_{\mathsf{nym}} := g^{\mathsf{sk}} h^{\mathsf{sk}_{\mathsf{nym}}}$
4: **return** $\mathsf{sk}_{\mathsf{nym}}$, $\mathsf{pk}_{\mathsf{nym}}$
5: **procedure** SIGNNYM($\mathsf{pk}_{\mathsf{nym}}, \mathsf{sk}_{\mathsf{nym}}, \mathsf{sk}, m$)
6: $\rho_1, \rho_2 \leftarrow_{\$} \mathbb{Z}_q$
7: $\mathsf{com} := g^{\rho_1} h^{\rho_2}$
8: $c := \text{HASH}(\mathsf{com}, \mathsf{pk}_{\mathsf{nym}}, m)$
9: $\mathfrak{p}_{\mathsf{sk}} := \rho_1 + c \cdot \mathsf{sk}$
10: $\mathfrak{p}_{\mathsf{skNym}} := \rho_2 + c \cdot \mathsf{sk}_{\mathsf{nym}}$
11: **return** $c, \mathfrak{p}_{\mathsf{sk}}, \mathfrak{p}_{\mathsf{skNym}}$
12: **procedure** VERIFYNYM($\mathsf{pk}_{\mathsf{nym}}, m, c, \mathfrak{p}_{\mathsf{sk}}, \mathfrak{p}_{\mathsf{skNym}}$)
13: $\mathsf{com} = g^{\mathfrak{p}_{\mathsf{sk}}} h^{\mathfrak{p}_{\mathsf{skNym}}} \mathsf{pk}_{\mathsf{nym}}^{-c}$
14: **return** $c = \text{HASH}(\mathsf{com}, \mathsf{pk}_{\mathsf{nym}}, m)$
15: **procedure** PROVEPK(sk, pk, nonce)
16: $\rho \leftarrow_{\$} \mathbb{Z}_q$
17: $\mathsf{com} := g^{\rho}$
18: $c := \text{HASH}(\mathsf{com}, \mathsf{pk}, \mathsf{nonce})$
19: $\mathfrak{p} := \rho + c \cdot \mathsf{sk}$
20: **return** $c, \mathfrak{p}$
21: **procedure** VERIFYPK($c, \mathfrak{p}, \mathsf{pk}, \mathsf{nonce}$)
22: $\mathsf{com} = g^{\mathfrak{p}} \mathsf{pk}^{-c}$
23: **return** $c = \text{HASH}(\mathsf{com}, \mathsf{pk}, \mathsf{nonce})$

V. INTEGRATION WITH HYPERLEDGER FABRIC

In this section we explain how the building blocks defined earlier work together within Fabric. We assume that all parties have access to a set of system parameters sp including the root authority public key (see Section III-D), and have generated their pairs of keys. The keys are always generated as $\mathsf{sk} \leftarrow_{\$} \mathbb{Z}_q$ and $\mathsf{pk} := g^{\mathsf{sk}}$ in the group generated by g.

A. Including pseudonyms in proof

In Fabric, a transaction has two special fields that are used in tandem to establish its authenticity. A *Creator* field that contains the identity of the transaction author, and a *Signature* field that holds a signature of the rest of the transaction by its author. Fabric specifications require that *Creator* and *Signature* be validated individually. Integrating Identity Mixer directly introduces two security flaws: namely, if *Creator* is a NIZK of the credential validity and *Signature* is a regular signature with the author's secret key, then (**1**) there is no guarantee that the keys used to generate the NIZK and the signature are the same, and (**2**) the regular signature itself would leak the identity of the signer by going through all users' public keys and testing whether the signature verifies.

To solve the above problems, we generate a *Pedersen commitment* (called pseudonym) to the secret key and place it in both fields. This pseudonym ensures that the same secret key is used to produce *Creator* and *Signature* fields. Notably, *Creator* contains a modified NIZK proof that shows that the prover knows the secret key used to construct the pseudonym and that it is the same secret key underlying the credentials. *Signature*, on the other hand, is a Schnorr-like proof of knowledge of the secret committed in the pseudonym, in which the content of the transaction is leveraged to compute the challenge.

The verifier first checks whether *Creator* and *Signature* include the same pseudonym. If so, then it verifies the validity of the content of those fields independently; otherwise it rejects. See Algorithm 4 for more details.

B. Submitting transactions

A user authorizes the execution of chaincode by providing a NIZK proof and a linked signature on the proposal, as described in Section V-A. During this process, the user can decide to selectively disclose attributes, which are made available to the chaincode so access control can be implemented as needed by the application.

The protocol has the following global stages (see Algorithm 5).

At the **setup** stage (line 2), the parties generate their secret and public keys.

The **delegation** stage starts by a *credential request* from the delegatee to the delegator in which the former proves that she knows the secret key corresponding to her public key, using a classical non-interactive Schnorr proof (see Algorithm 4). To ensure the freshness of the proof the delegator (i.e., verifier) provides a nonce that would be used to compute the challenge in the proof. If the provided proof is valid, then the delegator signs, using GROTH, the public key and the attributes of the delegatee. We note that it is up to the delegator to determine the delegatee's valid attributes. This process of credential issuance can be repeated an arbitrary number of times increasing the length of the credential chain. In more concrete terms, the first level of the delegation corresponds to the root authority issuing credentials to intermediate authorities that in turn delegate the credentials further down the hierarchy (lines 2–5). On the last level of the credential chain, we find users who submit transactions to Fabric.

The **transaction** stage (lines 14–23) has the user generate randomized proofs and signatures to authenticate the content of her transactions anonymously. Namely, the user generates a pseudonym (i.e. Pedersen commitment) to commit to her secret key (see Section V-A). Then she generates a proof in which she discloses her attributes as needed and shows the following: (**1**) the user knows valid credentials, and (**2**) the pseudonym commits to the secret key matching the credentials. Finally, she signs the content of the transaction with the secret key in the pseudonym (lines 14 and 18).

Algorithm 5 Delegation, revocation, auditing and transaction submission protocols

	i-**Level CA**		$(i+1)$-**Level CA**
	..... Repeated for L rounds of delegation (from the Root CA to Intermediate CAs to the User)		
2 :	$\mathsf{sk}_i \leftarrow\!\!\$\, \mathbb{Z}_q, \mathsf{pk}_i := g^{\mathsf{sk}_i}$		$\mathsf{sk}_{i+1} \leftarrow\!\!\$\, \mathbb{Z}_q, \mathsf{pk}_{i+1} := g^{\mathsf{sk}_{i+1}}$
3 :	$\mathsf{nonce} \leftarrow\!\!\$\, \{0,1\}^\lambda$	$\xrightarrow{\mathsf{nonce}}$	$\mathfrak{P}_{\mathsf{pk}} \leftarrow\!\!\$\, \textsc{ProvePK}(\mathsf{sk}_{i+1}, \mathsf{pk}_{i+1}, \mathsf{nonce})$
4 :	$\textsc{VerifyPK}(\mathfrak{P}_{\mathsf{pk}}, \mathsf{pk}_{i+1}, \mathsf{nonce})$	$\xleftarrow{\mathfrak{P}_{\mathsf{pk}}, \mathsf{pk}_{i+1}}$	
5 :	$\sigma_i \leftarrow\!\!\$\, \textsc{Groth.Sign}_{\mathsf{sk}_i}(\mathsf{pk}_{i+1} \| \vec{a}_{i+1})$	$\xrightarrow{\sigma_{i+1}}$	$\mathsf{cred}_{i+1} := (\sigma_{i+1}, \vec{a}_{i+1}, \mathsf{pk}_{i+1}, \mathsf{sk}_{i+1})$
6 :	**Revocation authority**		**User**
	..... On each epoch, user requests a non-revocation signature		
7 :	$\mathsf{sk}_{\mathsf{rev}} \leftarrow\!\!\$\, \mathbb{Z}_q, \mathsf{pk}_{\mathsf{rev}} := g^{\mathsf{sk}_{\mathsf{rev}}}$		$\mathsf{sk}_u \leftarrow\!\!\$\, \mathbb{Z}_q, \mathsf{pk}_u := g^{\mathsf{sk}_u}$
8 :	$\mathsf{nonce} \leftarrow\!\!\$\, \{0,1\}^\lambda$	$\xrightarrow{\mathsf{nonce}}$	
9 :	$\textsc{VerifyPK}(\mathfrak{P}_{\mathsf{pk}}, \mathsf{pk}_u, \mathsf{nonce})$	$\xleftarrow{\mathfrak{P}_{\mathsf{pk}}, \mathsf{pk}_u}$	$\mathfrak{P}_{\mathsf{pk}} \leftarrow\!\!\$\, \textsc{ProvePK}(\mathsf{sk}_u, \mathsf{pk}_u, \mathsf{nonce})$
10 :	$\sigma_{\mathsf{rev}} \leftarrow\!\!\$\, \textsc{NRSign}_{\mathsf{sk}_{\mathsf{rev}}}(\mathsf{pk}_u, \mathsf{epoch})$	$\xrightarrow{\sigma_{\mathsf{rev}}}$	
11 :	**Verifier**		**User**
12 :		(from the delegation stage)	$\mathsf{cred}_u := (\langle \sigma_j, \vec{a_j}, \mathsf{pk}_j \rangle_{j=1}^{L}, \mathsf{sk}_u)$
13 :		(computed once)	$\mathsf{enc}, \rho := \textsc{AuditEnc}(\mathsf{pk}_{\mathsf{aud}}, \mathsf{pk}_u)$
	..... User submits a transaction		
14 :			$\mathsf{sk}_{\mathsf{nym}}, \mathsf{pk}_{\mathsf{nym}} \leftarrow\!\!\$\, \textsc{MakeNym}(\mathsf{sk}_u)$
15 :			$\mathfrak{P}_{\mathsf{rev}} \leftarrow\!\!\$\, \textsc{NRProve}(\sigma_{\mathsf{rev}}, \mathsf{sk}, \mathsf{sk}_{\mathsf{nym}}, \mathsf{epoch})$
16 :			$\mathfrak{P}_{\mathsf{audit}} \leftarrow\!\!\$\, \textsc{AuditProve}(\mathsf{enc}, \rho, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_{\mathsf{nym}}, \mathsf{sk}_{\mathsf{nym}})$
17 :			$\mathfrak{P}_{\mathsf{cred}} \leftarrow\!\!\$\, \textsc{CredProve}(\mathsf{cred}_u, D, \perp)$
18 :			$\sigma_{\mathsf{nym}} \leftarrow\!\!\$\, \textsc{SignNym}(\mathsf{pk}_{\mathsf{nym}}, \mathsf{sk}_{\mathsf{nym}}, \mathsf{sk}_u, \mathsf{tx})$
19 :	$(\mathfrak{P}_{\mathsf{cred}}, \mathfrak{P}_{\mathsf{rev}}, \mathfrak{P}_{\mathsf{audit}}, \mathsf{enc}, \mathsf{tx}, \mathsf{pk}_{\mathsf{nym}}) := m$	$\xleftarrow{m, \sigma_{\mathsf{nym}}}$	$m := (\mathfrak{P}_{\mathsf{cred}}, \mathfrak{P}_{\mathsf{rev}}, \mathfrak{P}_{\mathsf{audit}}, \mathsf{enc}, \mathsf{tx}, \mathsf{pk}_{\mathsf{nym}})$
20 :	$\textsc{VerifyNym}(\mathsf{pk}_{\mathsf{nym}}, \mathsf{tx}, \sigma_{\mathsf{nym}})$		
21 :	$\textsc{NRVerify}(\mathfrak{P}_{\mathsf{rev}}, \mathsf{pk}_{\mathsf{nym}}, \mathsf{epoch})$		
22 :	$\textsc{AuditVerify}(\mathfrak{P}_{\mathsf{audit}}, \mathsf{enc}, \mathsf{pk}_{\mathsf{nym}})$		
23 :	$\textsc{CredVerify}(\mathfrak{P}_{\mathsf{cred}}, D, \mathsf{pk}_{\mathsf{nym}}, \perp)$		

Verifiers consequently validate the transaction by first verifying the disclosed attributes and then checking that the signature and the proof refer to the same pseudonym and that they are valid under the public key of the root authority (lines 20 and 23).

C. Revocation

Anonymous credentials pose the challenge of revocation. Classical mechanisms of revocation lists inherently fail as they require the credential holder to reveal her public key. One could use primitives such as zero-knowledge sets [26] or accumulators combined with zero-knowledge proofs [6] to simulate revocation lists, but such methods can be computationally prohibitive and negatively impact the transaction throughput once they are integrated into Fabric. Yet real systems cannot do without user revocation. While we can safely assume that organizations in Fabric will not be revoked frequently, users on the other hand may have their authorization to submit transactions denied at any moment (e.g., a failure to pay a monthly subscription, or an employee leaving her company).

To support user revocation in Hyperledger Fabric we combine *epoch-based whitelisting* mechanisms and signatures in a way that yields efficient proofs of membership. Namely, we divide the timeline into *epochs* that define the validity periods of user credentials. For each epoch, a non-revoked user will be issued an *epoch credential* (a signature) that binds her public key to the epoch. When a user submits a transaction, she provides along with it a *proof of non-revocation* that consists of proving in zero-knowledge that she knows a signature that binds her public key to the current epoch. User credentials that are valid for a certain epoch are automatically revoked the moment the epoch expires. It should be noted that an epoch expires either naturally (epoch elapses) or manually

Algorithm 6 Non-revocation proof generation and verification algorithms

1: **procedure** NRPROVE($\sigma, \mathsf{sk}, \mathsf{sk}_{\mathsf{nym}}, \mathsf{epoch}$)
2: $(r', s', t'_1, t'_2) \leftarrow_{\$}$ GROTH.RANDOMIZE(σ)
3: $\langle \rho \rangle_{1\ldots4} \leftarrow_{\$} \mathbb{Z}_q$
4: $\mathsf{com}_1 := e\left(r', g_2^{\rho_1}\right) \cdot e\left(g_1^{-1}, g_2^{\rho_2}\right)$
5: $\mathsf{com}_2 := e\left(r', g_2^{\rho_3}\right)$
6: $\mathsf{com}_3 := g_1^{\rho_2} h^{\rho_4}$
7: $c := \text{HASH}(r', s', \mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{epoch})$
8: $\mathfrak{p}_1 := g_2^{\rho_1} {t'_1}^{c}$
9: $\mathfrak{p}_2 := \rho_2 + \mathsf{sk} \cdot c$
10: $\mathfrak{p}_3 := g_2^{\rho_3} {t'_2}^{c}$
11: $\mathfrak{p}_4 := \rho_4 + \mathsf{sk}_{\mathsf{nym}} \cdot c$
12: **return** $c, \langle \mathfrak{p} \rangle_{1\ldots4}, r', s'$
13: **procedure** NRSIGN($\mathsf{sk}_{\mathsf{rev}}, \mathsf{pk}, \mathsf{epoch}$)
14: **return** GROTH.SIGN($\mathsf{sk}_{\mathsf{rev}}; \mathsf{pk} \| g^{\mathsf{epoch}}$)
15: **procedure** NRVERIFY($c, \langle \mathfrak{p} \rangle_{1\ldots4}, r', s', \mathsf{pk}_{\mathsf{nym}}, \mathsf{epoch}$)
16: **if** $e\left(r', s'\right) \neq e\left(g_1, y_1\right) \cdot e\left(\mathsf{pk}_{\mathsf{rev}}, g_2\right)$ **then**
17: **return** false
18: $\mathsf{com}_1 := e\left(r', \mathfrak{p}_1\right) \cdot e\left(g_1^{-1}, g_2\right)^{\mathfrak{p}_2} \cdot e\left(\mathsf{pk}_{\mathsf{rev}}, y_1\right)^{-c}$
19: $\mathsf{com}_2 := e\left(r', \mathfrak{p}_3\right) \cdot e\left(\mathsf{pk}_{\mathsf{rev}}, y_2\right)^{-c} \cdot e\left(g_1, g_2^{\mathsf{epoch}}\right)^{-c}$
20: $\mathsf{com}_3 := g_1^{\mathfrak{p}_2} h^{\mathfrak{p}_4} \mathsf{pk}_{\mathsf{nym}}^{-c}$
21: $c' := \text{HASH}(r', s', \mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{epoch})$
22: **return** $c = c'$

(authorized parties advance the epoch by putting a special message on the ledger).

In this section, we describe two alternative solutions that differ in their generality. The first one straightforward but requires that revocations are handled by the same entity that issues credentials. The second is more complex but allows revocations and credentials be handled by different entities.

Epoch as an attribute: For the simple solution, we implement revocation using delegatable credentials in such a way that users in the last level of delegation have epoch identifiers as attributes. A user thus needs to request new delegatable credentials from her issuer every time an epoch expires to be able to submit transactions. The proof of non-revocation in this implementation leverages the proof generation depicted in Algorithm 1 such that one of the disclosed attributes is the identifier of the current epoch.

Explicit proof of non-revocation: Although the above solution requires no additional cryptographic implementation, it suffers from the limitation that the credential issuer must always be the same as the revocation authority. Depending on the setting, this may actually be a desirable feature. However, to accommodate settings where credential issuers are different from revocation authorities, we decouple the credentials for user attributes from epoch credentials (i.e., credentials to submit transactions in an epoch). To obtain authorization for the current epoch, a user contacts the revocation authority (proving the possession of her public key) and the latter returns a GROTH signature of the user's public key and the epoch identifier (Algorithm 5, lines 8–10). When a user wishes to submit a transaction, she generates a proof of non-revocation that proves knowledge of an epoch credential and the associated secret key (Algorithm 5, line 15). Verifiers in the blockchain verify the proof of non-revocation (Algorithm 5, line 21) and if it is valid, then they verify the user signature on the transaction content. Algorithm 6 depicts the details of the generation and the verification of non-revocation proof. Note that in the proof of non-revocation the user leverages the pseudonym to show that the secret key mapped to the epoch credential is the one used to generate the signature.

D. Auditing

It is desirable to allow an authorized external entity — an *auditor* — to examine the origin of transactions committed to the ledger. The natural approach is for the transaction author to embed her identifier (the public key) encrypted under the auditor's public key into the transaction. Following this approach gives rise to two challenges: We must enforce (**1**) that the user encrypts her own public key, and (**2**) that she uses the public key of the authorized auditor. Schnorr zero-knowledge proofs [31] coupled with ElGamal encryption [18] allow us to address these challenges relatively efficiently.

Algorithm 7 depicts the details of how to integrate verifiable encryption with delegatable credentials to support auditability. When submitting a transaction, the user encrypts her public key using ELGAMAL scheme under the public key of the auditor (Algorithm 5, line 13), and generates a zero-knowledge proof that this encryption is valid (Algorithm 5, line 16). Finally, the verifier checks this proof as a part of transaction validation (Algorithm 5, line 22). If the auditor decides to learn the identity of the author of a given transaction, all she needs to do is decrypt the ciphertext in the transaction. This process is guaranteed to succeed and to correctly yield the author's public key.

It should be noted that our algorithms only cover settings where there is one single auditor for all users in the system.

VI. IMPLEMENTATION AND EVALUATION

We have provided the first production-ready open-sourced implementation of delegatable credentials. It is generic and produces valid credentials and proofs for any number of levels and attributes. The project is tested with over 400 tests and they cover 100% of the code. We note that this is a significant improvement over the original code which is only a prototype computing a single hard-coded credential. We also note that the original code is not open-sourced.

All benchmarks were run on macOS 10.14.6 using 2.5 GHz Intel Core i7 8-cores CPU. We have used Apache Milagro Cryptographic Library (AMCL) [32] with a 254-bit Barreto-Naehrig curve [7] for low-level operations such as pairings, exponentiations and PRG operations.

Algorithm 7 Auditing proof generation and verification algorithms

procedure AUDITPROVE($\mathsf{enc}, \rho, \mathsf{pk}, \mathsf{sk}, \mathsf{pk}_{\mathsf{nym}}, \mathsf{sk}_{\mathsf{nym}}$)
  $\langle \rho \rangle_{1\ldots3} \leftarrow_{\$} \mathbb{Z}_q$
  $\mathsf{com}_1 := g^{\rho_1} \mathsf{pk}_{\mathsf{aud}}^{\rho_2}$
  $\mathsf{com}_2 := g^{\rho_2}$
  $\mathsf{com}_3 := g^{\rho_1} h^{\rho_3}$
  $c := \text{HASH}(\mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{enc}, \mathsf{pk}_{\mathsf{nym}})$
  $\mathfrak{p}_1 := \rho_1 + c \cdot \mathsf{sk}$
  $\mathfrak{p}_2 := \rho_2 + c \cdot \rho$
  $\mathfrak{p}_3 := \rho_3 + c \cdot \mathsf{sk}_{\mathsf{nym}}$
  return $c, \langle \mathfrak{p} \rangle_{1\ldots3}$

procedure AUDITENC($\mathsf{pk}_{\mathsf{aud}}, \mathsf{pk}$) ▷ ELGAMAL
  $\rho \leftarrow_{\$} \mathbb{Z}_q$
  $\mathsf{enc} := (\mathsf{enc}_1, \mathsf{enc}_2) := (\mathsf{pk} \cdot \mathsf{pk}_{\mathsf{aud}}^{\rho}, g^{\rho})$
  return enc, ρ

procedure AUDITVERIFY($c, \mathsf{enc}, \langle \mathfrak{p} \rangle_{1\ldots3}, \mathsf{pk}_{\mathsf{nym}}$)
  $\mathsf{com}_1 := g^{\mathfrak{p}_1} \mathsf{pk}_{\mathsf{aud}}^{\mathfrak{p}_2} \mathsf{enc}_1^{-c}$
  $\mathsf{com}_2 := g^{\mathfrak{p}_2} \mathsf{enc}_2^{-c}$
  $\mathsf{com}_3 := g^{\mathfrak{p}_1} h^{\mathfrak{p}_3} \mathsf{pk}_{\mathsf{nym}}^{-c}$
  $c' := \text{HASH}(\mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{enc}, \mathsf{pk}_{\mathsf{nym}})$
  return $c = c'$

A. Delegatable credentials scheme

We have run extensive benchmarks of every operation of the scheme on multiple parameter values. We stress that our evaluation results differ from the ones in the original paper [11]. First, the implementations are in different languages and run on different processors. These differences play a big role when benchmarking cryptographic primitives, which mostly involve bit manipulations. Second, we have obtained the original code of [11] and we have noticed differences in benchmark methodologies. The original code pre-computes some values (pairings) during the signature phase, and therefore this time is not included in the proof generation and verification stages. Our benchmarks involve no pre-computations to produce more fair results. Overall, given that our code is production-ready, generic and open-sourced, we want our benchmarks to be treated independently of the previous work.

In the following, L stands for the number of delegation levels, n stands for the number of attributes per level, which we set to be the same for every level for simplicity. Note that the most sensitive overhead is due to verification, since it is the operation that will be run by the Fabric network. In Fabric, having $L = 2$ and $n = 2$ covers most use-cases.

1) Optimizations: First of all, we wanted to demonstrate the improvement due to our optimizations. We have run the benchmarks with all combinations of e-product and parallelization optimizations (see Table I). Results show that for the most commonly-used parameter values the improvement is almost an order of magnitude.

e-product	Parallelization	CREDPROVE		CREDVERIFY	
		Big	Small	Big	Small
disabled	disabled	3 007	896	4 836	2 026
enabled	disabled	1 733	470	2 718	1 202
disabled	enabled	1 495	391	2 117	584
enabled	enabled	1 097	259	1 414	234
Improvement ($\approx$ times)		2.7	3.5	3.4	8.7

TABLE I: Optimizations benchmark for $L = 2$ and $n = 2$ (small) and $L = 5$ and $n = 3$ (big). The values are in milliseconds.

2) Different parameters: With optimizations enabled we have run the operations for multiple combinations of levels and attributes. In Table II we put the proof generation and verification times along with the generated proof size for $L \in \{1, 2, 3, 5, 10\}$ and $n = (0, \ldots, 4)$. In all cases all attributes are hidden — the overhead difference when all attributes are revealed is minimal. We can confirm that the overhead and proof size grow linearly with L and n.

L \ n	0	1	2	3	4
1	55 ms 53 ms 364 B	67 ms 59 ms 500 B	82 ms 71 ms 636 B	102 ms 101 ms 772 B	119 ms 119 ms 908 B
2	119 ms 79 ms 767 B	176 ms 126 ms 1.2 kB	257 ms 228 ms 1.6 kB	384 ms 369 ms 2.0 kB	423 ms 529 ms 2.4 kB
3	170 ms 127 ms 1.2 kB	270 ms 283 ms 1.7 kB	394 ms 1 322 ms 2.2 kB	531 ms 1 666 ms 2.8 kB	632 ms 2 407 ms 3.3 kB
5	338 ms 1 355 ms 2.0 kB	548 ms 1 678 ms 2.9 kB	733 ms 2 259 ms 3.8 kB	1 100 ms 2 772 ms 4.8 kB	1 326 ms 2 904 ms 5.7 kB
10	754 ms 1 848 ms 4.0 kB	1 374 ms 2 540 ms 6.0 kB	1 912 ms 3 228 ms 8.0 kB	2 365 ms 4 115 ms 9.9 kB	3 000 ms 5 076 ms 12 kB

TABLE II: Parameters benchmark. In each cell the top value is a proof generation overhead, the middle value is a proof verification overhead and the bottom value is the proof size.

B. Integration

In this section we present the results of preliminary benchmarks with the real Fabric code. First, we analyze the performance of helper procedures (key and signature generation, revocation and auditing), then we run real transactions on a local deployment of Fabric using our new protocols.

1) Components: Table III depicts the performance results for some of the helper methods. All routines have been run 100 times and the resulting time is averaged. Note that the revocation routines are considerably slower due to use of pairing in proofs. Moreover, a good part of computations is done over $\mathbb{G}_2$ which is slower in AMCL than $\mathbb{G}_1$ — see the difference between GROTH_1 and GROTH_2. Our future work is to apply the optimizations we used with delegatable credentials scheme to this procedure as well.

Note that using pseudonyms, enabling auditing and proving possession of the secret key are almost free operations relative to credential proofs.

2) Transactions: To understand the relative overhead of using our new idemix with Fabric, we have put together a preliminary integration of the components into the main code. We have changed all the old idemix calls to proof generation and verification to the new routines. We are using two levels of credentials with two attributes on each level. In the real deployment, most of the attributes will be on the last level, but since odd and even levels take different time to process, we decided to uniformly disperse the attributes.

As an example application, we have crafted a simple chaincode that puts a hashed message on the ledger along with its timestamp. Such an application can be used to commit the messages without revealing them, such as for timestamping documents. In this application there are two organizations with two peers each, and the endorsement policy mandates that at least one endorser from each organization approves.

Procedure	Time	Procedure	Time
GROTH_1.SIGN	21	GROTH_2.SIGN	48
GROTH_1.VERIFY	60	GROTH_2.VERIFY	67
GROTH_1.RANDOMIZE	13	GROTH_2.RANDOMIZE	31
AUDITENCRYPT	4	NRSIGN	36
AUDITPROVE	7.5	NRPROVE	107
AUDITVERIFY	12	NRVERIFY	154
MAKENYM	2.5	PROVEPK	11
SIGNNYM	2.5	VERIFYPK	11
VERIFYNYM	4	KEYGEN	6

TABLE III: Running time of helper procedures in milliseconds.

These experiments were run with a local deployment of Fabric on a single machine as described earlier, with all Fabric components (all instances of orderers, peers, and clients) running as Docker containers on one machine. We stress that this is a preliminary benchmark that should not be extrapolated to a distributed setting. For instance, the most time-consuming operations are the cryptographic computations, and during transaction validation the single machine performs the tasks of four machines running in parallel in a distributed scenario. However, it helps to understand the performance impact of using the old (i.e., without delegation) and new (i.e., with delegation) idemix implementations. It also allowed us to suggest some improvements to the way Fabric validates the identities.

We have run 100 transactions (plus 10 for warmup) sequentially. A transaction begins with a user generating a proposal, and it ends when all peers have updated their local state. We have averaged the transaction overhead for all runs for each of the three cases — no idemix, old idemix and new idemix. According to our results, without idemix a transaction takes $235\,\text{ms}$, with the old idemix it takes $860\,\text{ms}$ and with the new idemix the time is $2\,252\,\text{ms}$.

We have dived into the Fabric code to track which entities and at which stages perform the most heavy operations. We have, therefore, defined a critical path of the transaction flow — the lower bound of the overhead if all replicated operations (such as endorsements) occur in parallel and, in a practical deployment, take the same time to finish. To estimate the total overhead we counted only proof generation and verification, which is a part of identity validation, as these are the most computation-heavy operations. As a part of a transaction life cycle, the user prepares a *transaction proposal* containing information about the chaincode to be executed and the arguments to use. In addition, the users signs the proposal. She sends it over to endorsers, who, in parallel, validate the proposal's creator and check the access policies, which also requires validating the proposal's creator. Checking an access policy involves verifying that the creator of the proposal or transaction has produced a valid signature and that her identity indeed satisfies the given access policy. Endorsers send back their endorsements, the user composes them into a transaction, signs it and sends it to the ordering service. The ordering service authenticates each transaction, assembles them in blocks, and broadcasts the blocks to the network. Each peer in the network validates the transaction, which includes validating the transaction's creator and checking policies, which again requires validating the creator.

In this critical path there is one proof generation and seven verification operations. To estimate the critical path cost we add the overhead of a transaction not using idemix and get $2\,088\,\text{ms}$. We note that this value is close to the actual measured time. The difference is attributed to the fact that the critical path is a lower bound. In practice, the latency of a set of parallel tasks is the latency of the slowest one.

Recall that the validation of transactions is the task that has to be performed by each peer in the Fabric transaction flow; this is the only operation that cannot be scaled out. Based on our measurements, this operation clocks at $228\,\text{ms}$, which means that we expect around 4 transactions per second *if all validators run on standard laptop computers*. As our algorithm parallelizes smoothly, running the system on fast server processors with more cores is expected to improve performance near proportional.

The default MSP in Fabric uses X.509 certificates for identities and the operations over these certificates are relatively fast. With X.509 it may be reasonable to take a conservative approach at verification and validate identities every time an operation with this identity takes place. With idemix, however, given the cost of such validation, there is a need to optimize this part. For example, note that the access policy check requires identity validation and therefore proof verification. Peers should be configured to cache validation results if the same identity is being validated. This optimization alone would save four proof verifications, or almost a second of latency, cutting the overhead in half.

We conclude that with optimization on Fabric, the throughput of the new idemix will get closer to practical.

VII. CONCLUSION

The possibility to perform transactions privately and anonymously will be crucial for the use of blockchain technology in many use cases in the financial and governmental domains, as well as all use cases that involve personal data. Anonymous transaction authorization, as achieved through

delegatable anonymous credentials in Hyperledger Fabric, can therefore be seen as one key enabler for blockchain technology in privacy-sensitive use cases.

The enhanced privacy guarantees incur a price in terms of computational complexity in the transaction generation and achievable throughput. For this reason, we have identified points for optimization to make the new idemix performance closer to practical.

The code of the cryptographic library implementing the anonymous credential scheme is already available as open source under MIT license. The integration into Fabric is not yet publicly available. Our goal is to make it a part of the standard Fabric distribution, and we are working with the Fabric community toward this goal.

Future work

Our near-term future work will include improving the benchmarks of the Fabric integration by running them in a fully distributed setting with industry-standard compute nodes. We are also pursuing open-sourcing the integration code, preferably as part of the Hyperledger Fabric codebase.

While our work is an important step toward improving privacy in permissioned blockchains, both security and performance of our current solution can be further improved. In our current implementation, the root certificate authority is still a central party. Although it does not play an active role in the online protocols and does not issue any certificates to users, our aim is to implement a threshold protocol in which the organizations participating in the blockchain system jointly produce the first-level signatures, further distributing the trust.

In Fabric, every transaction is executed (endorsed) only by a subset of the peers, which allows for parallel execution and addresses potential non-determinism. A flexible endorsement policy specifies which peers, or how many of them, need to vouch for the correct execution of a given smart contract. Currently, the endorsement policy reveals the identity of the involved peers. A possible way to remove this leakage is to equip the peers with idemix credentials and replace the endorsement policy by a commitment to it. Then, after collecting all the required endorsements, the client can prove in zero-knowledge the knowledge of valid signatures that satisfy the endorsement policy.

Finally, another important direction is toward increasing the throughput by speeding up the validation phase. There are several possible routes in this direction: as discussed in Section VI-B2, modifications to the Fabric software that help to avoid unnecessary duplication of computation are the first step. A further possibility is to modify the transaction signature such that the expensive validation of the credential delegation has to happen only in the endorsement phase, and not during validation.

VIII. Acknowledgments

This work has been supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477 PRIViLEDGE. We thank the authors of [11] for giving us access to their source code.

References

[1] D. Achenbach, C. Kempka, B. Löwe, and J. Müller-Quade, "Improved coercion-resistant electronic elections through deniable re-voting," in *JETS '15*, USENIX, 2015.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18, 2018, 30:1–30:15. DOI: 10.1145/3190508.3190538.

[3] E. Androulaki, C. Cachin, A. D. Caro, and E. Kokoris-Kogias, "Channels: Horizontal scaling and confidentiality on permissioned blockchains," in *ESORICS*, J. Lopez, J. Zhou, and M. Soriano, Eds., ser. LNCS, vol. 11098, Springer, 2018, pp. 111–131.

[4] *Anonymized Repository for Delegatable Anonymous Credentials library*, https://anonymous.4open.science/r/1335dc45-3197-4d98-87b9-6478edea78d0/, Accessed: 2019-09-13.

[5] G. Ateniese, B. Magri, D. Venturi, and E. Andrade, "Redactable blockchain – or – rewriting history in bitcoin and friends," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017.

[6] F. Baldimtsi, J. Camenisch, M. Dubovitskaya, A. Lysyanskaya, L. Reyzin, K. Samelin, and S. Yakoubov, "Accumulators with applications to anonymity-preserving revocation," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017, pp. 301–315.

[7] P. S. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *International Workshop on Selected Areas in Cryptography*, Springer, 2005, pp. 319–331.

[8] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 459–474.

[9] F. Benhamouda, A. D. Caro, S. Halevi, T. Halevi, C. Jutla, Y. Manevich, and Q. Zhang, "Initial public offering (IPO) on permissioned blockchain using secure multiparty computation," in *IEEE Blockchain*, IEEE, 2019.

[10] J. Blömer and J. Bobolz, "Delegatable attribute-based anonymous credentials from dynamically malleable signatures," in *Applied Cryptography and Network Security*, ser. LNCS, vol. 10892, Springer, 2018, pp. 221–239.

[11] J. Camenisch, M. Drijvers, and M. Dubovitskaya, "Practical uc-secure delegatable credentials with attributes and their application to blockchain," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 683–699.

[12] J. Camenisch, M. Drijvers, and A. Lehmann, "Anonymous attestation using the strong diffie hellman assump-

tion revisited," in *Trust and Trustworthy Computing*, Springer International Publishing, 2016, pp. 1–20.
[13] J. Camenisch and E. van Heerweeghen, "Design and implementation of the *idemix* anonymous credential system," in *ACM Conference on Computer and Communication Security*, ACM, 2002, pp. 21–30.
[14] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Third Symposium on Operating Systems Design and Implementation*, 1999.
[15] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theoretical Computer Science*, vol. 777, pp. 155–183, 2019.
[16] E. C. Crites and A. Lysyanskaya, "Delegatable anonymous credentials from mercurial signatures," in *Topics in Cryptography — CT-RSA*, ser. LNCS, vol. 11405, Springer, 2019, pp. 535–555.
[17] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," in *Advances in Cryptology — EUROCRYPT (1)*, ser. LNCS, Springer, 2019, pp. 625–656.
[18] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.
[19] C. Garman, M. Green, and I. Miers, "Decentralized anonymous credentials," in *NDSS*, Internet Society, 2014.
[20] ——, "Accountable privacy for decentralized anonymous payments," in *Financial Cryptography and Data Security*, J. Grossklags and B. Preneel, Eds., ser. LNCS, vol. 9603, Springer, 2016, pp. 81–98.
[21] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: A scalable and decentralized trust infrastructure," in *DSN*, 2019, pp. 568–580.
[22] J. Groth, "Efficient fully structure-preserving signatures for large messages," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2015, pp. 239–259.
[23] L. Harchandani, *Delegatable anonymous credentials in rust*, https://github.com/lovesh/signature-schemes/tree/delegatable/delg_cred_cdd, Sep. 2019.
[24] O. Harris. Quorum, [Online]. Available: https://www.goquorum.com/.
[25] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology — CRYPTO*, J. Katz and H. Shacham, Eds., ser. LNCS, IACR, vol. 10401, Springer, 2017, pp. 357–388.
[26] S. Micali, M. Rabin, and J. Kilian, "Zero-knowledge sets," in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, Oct. 2003, pp. 80–91.
[27] S. Nakamoto. (2009). Bitcoin: A peer-to-peer electronic cash system, [Online]. Available: https://bitcoin.org/bitcoin.pdf.
[28] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, 1991, pp. 129–140.
[29] A. Poelstra, A. Back, M. Friedenbach, G. Maxwell, and P. Wuille, "Confidential assets," in *Financial Cryptography and Data Security*, A. Zohar, I. Eyal, V. Teague, J. Clark, A. Bracciali, F. Pintore, and M. Sala, Eds., ser. LNCS, vol. 10958, Springer, 2018, pp. 43–63.
[30] I. Puddu, A. Dmitrienko, and S. Capkun, *μchain: How to forget without hard forks*, Cryptology eprint archive, report 2017/106, Feb. 2017.
[31] C. P. Schnorr, "Efficient identification and signatures for smart cards," in *Advances in Cryptology — CRYPTO*, G. Brassard, Ed., ser. LNCS, vol. 435, Springer, 1989, pp. 239–252.
[32] M. Scott, "The apache milagro crypto library," [Online]. Available: https://github.com/MIRACL/amcl.
[33] C. Stathakopoulou, T. David, and M. Vukolić, *Mir-BFT: High-throughput BFT for blockchains*, arXiv:1906.05552, Jun. 2019.
[34] P. J. Windley. Sovrin, [Online]. Available: https://sovrin.org/.
[35] G. Wood. Ethereum: A secure decentralised generalised transaction ledger, [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.
[36] K. Wüst, K. Kostiainen, V. Capkun, and S. Capkun, *PRCash: Fast, private and regulated transactions for digital currencies*, Cryptology eprint archive: report 2018/412, May 2018.