Anonymous Transactions with Revocation and Auditing in Hyperledger Fabric

Extended version

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

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

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

Björn Tackmann
IBM Research, Zürich
bta@zurich.ibm.com

***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 her credential. Use case settings where privacy is a concern thus require proper privacy-preserving authentication and authorization mechanisms.**

Anonymous credential schemes allow a user to authenticate while showing only those attributes necessary in a given setting. This makes them a great tool for authorizing transactions in permissioned blockchain systems based on the user's attributes. As in most setups of such systems where 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 her issuing organization. Camenisch, Drijvers and Dubovitskaya (CCS 2017) therefore suggest the use of a delegatable anonymous credential scheme to also hide that remaining piece of information.

In this paper we extend the base protocol with revocation and auditability, two functionalities that are necessary for real-world adoption, and improve the core functionality of the Camenisch et al. scheme. We present a complete protocol and provide its production-grade open-source implementation including the scheme and the proposed extensions, ready to be integrated with Hyperledger Fabric. Our distributed-setting performance measurements show that the integration of the scheme with Hyperledger Fabric, while incurring an overhead in comparison to the less privacy-preserving solutions, is practical for settings with stringent 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.

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 aforementioned use cases require the transactions to be properly authorized by a 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 example, 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 call for a transaction authorization often still require the identity of the transaction origin to be hidden. The most salient example is elections, where re-voting (as a measure against coercion [1]) inherently requires voters 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 like Identity Mixer [14] allows participants to submit transactions while revealing only the attributes necessary to authorize that particular transaction (such as being a registered voter or having passed KYC checks), and 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 still will be leaked from the authorized transactions. In certain use cases even this leakage is not acceptable, for example, the leakage of a patient being treated in a particular hospital department. 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.

This is where *delegatable credentials* come in handy: in a delegatable credential scheme, a root authority delegates issuance of credentials to intermediate authorities in 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 helps keeping the management largely decentralized, while at the same time hides the particular authority that issued a given credential.

In this paper, we build a production-grade delegatable credentials system for Hyperledger Fabric [2], based on the core protocol of [12]. Our contributions are four-fold:

- We extend the scheme of [12] by adding mechanisms for credentials revocation and authorizations auditing. The new extensions are efficient as they are based solely on ElGamal encryption [21] and Schnorr proofs [33]. We also provide a security definition for delegatable anonymous credentials with revocation and auditing in the UC framework, and prove the full scheme secure.
- We provide ans opens-source a production-grade implementation in Go that includes a set of optimizations over the core protocol of [12], allowing for significant performance gains.
- We enable auditable and 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.
- We present a comprehensive evaluation of the scheme and the proposed extensions. Namely, we design a Fabric prototype that measures the incurred computational overhead, the gains from our optimizations, and network usage. Our prototype runs in a fully distributed setting faithfully executing all parts of the protocol.

II. RELATED WORK

The most immediately related work is [12], which our paper builds on. That paper presents an instantiation of delegatable anonymous credentials, 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 [12] in three main directions: (a) we provide a production-grade implementation as open source, which includes multiple performance optimizations ([12] 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 [12]; (c) we cover practically-relevant functionalities such as revocation and auditing.

After the publication of [12], two further papers on delegatable credentials were published, namely by Blömer and Bobolz [11] and by Crites and Lysyanskaya [18]. Both claim stronger security properties compared to [12] 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 [11] 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, which when instantiated, appears to be less efficient than the one in [12]. The scheme in [18] does not support attributes, which makes it unsuitable for our application.

Sovrin [38] also combines anonymous credentials with a permissioned blockchain system. While we use anonymous credentials to authorize transactions on a blockchain, the Sovrin platform instead leverages the blockchain to produce anonymous credentials, in the vein of previous work on decentralized anonymous credentials of Garman, Green, and Miers [22]. The two approaches thus serve two different purposes. In the context of Sovrin, there is also an implementation of [12] in Rust [26], 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. [4], [9], [23], [32], [40]), different types of state channels (e.g. [3], [20]) or multi-party computation (e.g. [10]). While the underlying cryptographic machinery, particularly in the work on zero-knowledge proofs, is similar to what 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

The purpose of a blockchain is to implement an immutable 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, enforcing thus 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 [31] or Ethereum [39], 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., [31], [39]) or proofs of stake (e.g., [16], [29]), 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 to counter misbehavior, foster trust and aid governance. Most permissioned blockchain systems (e.g., [24], [37]) build on variants of the well-studied and efficient PBFT [15] 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 [27].

A. Hyperledger Fabric

Fabric is a permissioned blockchain platform 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 it, focusing on those components relevant to transaction authorization. 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 them in order. Each organization also runs a *membership service provider (MSP)*, which maintains and manages identities of all participants of that organization. This includes issuing credentials for authentication and authorization 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) identifies *endorsers*, peers that execute this chaincode. The *endorsement policy* associated with the chaincode specifies the minimum requirements for replicated execution. 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 by sending a transaction proposal to the endorsers of that chaincode. The endorsers execute the chaincode and sign the chaincode's read/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 orders the transaction in a block and broadcasts the block to the peers in the network.

In the *Validate* phase, the peers verify that each transaction in the block 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 valid transactions.

B. 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 for transaction authorization: the *Creator* (i.e., identity of the client invoking the transaction) and the *Signature* (i.e., authorization of the transaction). As each transaction carries the identity of its origin as 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 [13]. 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 provide a Fabric-tailored implementation of delegatable anonymous credentials based on the work of [12]. This implementation ensures that the only information leaked by a transaction is the root CA common to all network participants. Additionally, the implementation supports efficient revocation and comes with auditing capabilities that allow authorized parties to trace the transactions back to their authors achieving some level of accountability.

C. Notation

Let $\mathbb{Z}_q$ be the set of natural numbers in $[0;q)$ where q is a large prime. Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be three groups of order q, such that there exists an efficient bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Let g_i be a random generator for $\mathbb{G}_i$ for $i \in \{1, 2\}$. Let $\textsc{fexp}$ and $\hat{t}$ be the final exponentiation and Miller's loop operations respectively, such that $e = \textsc{fexp} \circ \hat{t}$. Let $\leftarrow\!\!\$$ describe the operation of random sampling. Let sp denote the public parameters available to all algorithms in the system. These include the description of the bilinear groups and hash functions. Let NIZK$\{w : x\}$ denote a non-interactive zero-knowledge proof for statement x and witness w (i.e., private input).

IV. DELEGATABLE ANONYMOUS CREDENTIALS

A scheme for delegatable anonymous credentials involves the following:

Root authority A *trusted* entity that provides credentials to intermediate authorities.

Intermediate authority Each intermediate authority presents its public key and its attributes to a parent (root or other intermediate) authority. The latter verifies that the intermediate authority knows the secret key and that it holds the presented attributes, and in turn provides the corresponding credential. An intermediate authority is allowed to issue credentials to other intermediate authorities or users.

Users A user requests credentials from the intermediate authority of her organization. Prior to credential generation, the intermediate authority checks the legitimacy of the user's public key and the attributes.

All participants (root, intermediate authorities and users) start by generating their pairs of secret and public keys. A *Level*-1 delegatee (usually an intermediate authority) contacts the root to obtain a credential (i.e., signature) to bind its public key to its attributes. Once a *Level*-1 delegatee gets its credentials, it becomes a delegator itself and can thereafter issue credentials for *Level*-2 delegates. This delegation process may continue for an arbitrary number of levels, increasing the length of the credential chain.

The holder of a credential typically uses non-interactive zero-knowledge (NIZK) proofs to sign messages *anonymously*. More precisely, signing a message m consists of proving in zero-knowledge that (**1**) the signer owns the credentials; (**2**) the Schnorr-like generated signature is valid for message m; (**3**) inductively, all adjacent levels are legitimate (one was delegated from the other); and (**4**) at the end of the induction, the top-level public key is that of the root authority. During the proof generation, the signer chooses which attributes to disclose and which to keep secret. It is possible to reveal or hide all attributes, albeit not very useful.

A. Algorithms

A delegatable anonymous credential scheme consists of the following algorithms:

- $\textsc{KeyGen}(\mathsf{sp}) \rightarrow_{\$} (\mathsf{csk}, \mathsf{cpk})$: this algorithm is called with the system parameters to generate a pair of secret and public keys for the caller.
  We denote the public and the secret keys of the root authority cpk_0 and csk_0 respectively, and its credentials $\mathsf{cred}_0 = \mathsf{cpk}_0$.
- $\textsc{Delegate}(\mathsf{csk}_i, \mathsf{cred}_i, \mathsf{cpk}_{i+1}, \vec{a}_{i+1}) \rightarrow_{\$} \mathsf{cred}_{i+1}$: a level i authority invokes this algorithm with its secret key csk_i and credentials cred_i to produce credentials of the next level $i+1$ that bind attributes $\vec{a}_{i+1}$ to public key cpk_{i+1}.
- $\textsc{Present}(\mathsf{csk}_L, \mathsf{cred}_L, \mathsf{cpk}_0, \langle a_{i,j} \rangle_{(i,j)\in D}, m) \rightarrow_{\$} \mathfrak{P}_{\mathsf{cred}}$: a user calls this algorithm with her secret csk_L, her credentials cred_L, the root public key cpk_0, attributes $\langle a_{i,j} \rangle_{(i,j)\in D}$ she wishes to reveal and a message m to be signed. (D is the set of indices of attributes in the delegation chain that a user wishes to disclose.) The algorithm returns a zero-knowledge proof that (**1**) shows the validity of cred_L under cpk_0; (**2**) proves that secret key csk_L matches cred_L and disclosed attributes $\langle a_{i,j} \rangle_{(i,j)\in D}$; (**3**) and signs m.
- $\textsc{Verify}(\mathfrak{P}_{\mathsf{cred}}, \mathsf{cpk}_0, \langle a_{i,j} \rangle_{(i,j)\in D}, m) \rightarrow \{0,1\}$: this algorithm verifies the correctness of proof $\mathfrak{P}_{\mathsf{cred}}$ relative to disclosed attributes $\langle a_{i,j} \rangle_{(i,j)\in D}$, message m and public key cpk_0.

B. Instantiation of the scheme

Camenisch, Drijvers, and Dubovitskaya [12] introduce a delegatable anonymous credential scheme that supports an arbitrary number of delegation levels, thanks to a combination of Groth signatures [25] and non-interactive zero-knowledge proofs. Groth signatures come with two appealing features: (**1**) they sign *vectors of messages* efficiently; and (**2**) they are structure preserving. The latter property is particularly important as it enables the generation of certificate chains by signing public keys without leaving the algebraic representation, and proving in zero-knowledge statements about signed public keys without necessarily knowing the underlying secret keys.

1) Groth signatures: Groth signature [25] consists of the following algorithms:

- $\textsc{Setup}(n) \rightarrow_{\$} \mathsf{sp}$: on input of integer n, output system parameters $\mathsf{sp} = (\Lambda^\star, \{y_{2,i}\}_{i=1}^n)$ whereby $\Lambda^\star = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, e)$ and $y_{2,i} \leftarrow_{\$} \mathbb{G}_2$ for $1 \le i \le n$.
- $\textsc{KeyGen}(\Lambda^\star) \rightarrow_{\$} (\mathsf{sk}, \mathsf{pk})$: on input of group description $\Lambda^\star$, output secret and public keys $\mathsf{sk} \leftarrow_{\$} \mathbb{Z}_q$ and $\mathsf{pk} = g_1^{\mathsf{sk}}$.
- $\textsc{Sign}(\mathsf{sk}; \vec{m}) \rightarrow_{\$} \sigma$: on input of secret key sk and vector $\vec{m} = (m_1, \ldots, m_n) \in \mathbb{G}_2^n$, do
  $$\rho \leftarrow_{\$} \mathbb{Z}_q^\star \quad r := g_1^\rho \quad s := (y_{2,1} \cdot g_2^{\mathsf{sk}})^{\frac{1}{\rho}} \quad t_i := (y_{2,i}^{\mathsf{sk}} \cdot m_i)^{\frac{1}{\rho}}$$
  and output signature $\sigma = (r, s, t_1, \ldots, t_n)$.
- $\textsc{Verify}(\mathsf{pk}; \sigma; \vec{m}) \rightarrow \{0,1\}$: on input of public key pk, signature $\sigma = (r, s, t_1, \ldots, t_n) \in \mathbb{G}_1 \times \mathbb{G}_2^{n+1}$ and vector $\vec{m} \in \mathbb{G}_2^n$, output the result of
  $$e(r,s) = e(g_1, y_{2,1}) \cdot e(\mathsf{pk}, g_2) \wedge \bigwedge_{i=1}^{n} e(r, t_i) = e(\mathsf{pk}, y_{2,i}) \cdot e(g_1, m_i)$$
- $\textsc{Randomize}(\sigma) \rightarrow_{\$} \sigma'$: on input of signature $\sigma = (r, s, t_1, \ldots, t_n) \in \mathbb{G}_1 \times \mathbb{G}_2^{n+1}$, do
  $$\rho' \leftarrow_{\$} \mathbb{Z}_q \qquad r' := r^{\rho'} \qquad s' := s^{\frac{1}{\rho'}} \qquad t'_i := t_i^{\frac{1}{\rho'}}$$
  and 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 signatures, we need to switch the key space and the 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 $\textsc{Groth}_1$ and $\textsc{Groth}_2$ where $\textsc{Groth}_i$ signs messages in $\mathbb{G}_i$, $i \in \{1,2\}$.

2) Description: Let L denote the length of the delegation chain, i.e., the length of the path from the root authority to any user in the system.

Let n_i for $1 \le i \le L$ denote the number of attributes $a_{(i,1)}, \ldots, a_{(i,n_i)}$ authorized at the i_{th} delegation level.

Let N_1 denote $\max\limits_{\substack{2 \le i \le L \\ i \text{ even}}} n_i$ whereas N_2 denote $\max\limits_{\substack{1 \le i \le L \\ i \text{ odd}}} n_i$.

Setup: Root authority calls

$$\textsc{Groth}_1.\textsc{Setup}(N_1) \rightarrow_{\$} \left(\Lambda^\star, \{y_{1,i}\}_{i=1}^{N_1}\right)$$
$$\textsc{Groth}_2.\textsc{Setup}(N_2) \rightarrow_{\$} \left(\Lambda^\star, \{y_{2,i}\}_{i=1}^{N_2}\right)$$

sets $\mathsf{sp} = \left(\Lambda^\star, \{y_{1,i}\}_{i=1}^{N_1}, \{y_{2,i}\}_{i=1}^{N_2}\right)$ and finally announces its credential $\mathsf{cred}_0 = \mathsf{cpk}_0$ whereby:

$$\textsc{Groth}_2.\textsc{KeyGen}(\Lambda^\star) \rightarrow_{\$} (\mathsf{csk}_0, \mathsf{cpk}_0)$$

Delegation: An intermediate authority of *Level*-1 calls $\textsc{Groth}_1.\textsc{KeyGen}(\Lambda^\star) \rightarrow_{\$} (\mathsf{csk}_1, \mathsf{cpk}_1)$ to first generate its secret and public keys. It then requests *Level*-1 credentials from the root authority by supplying cpk_1, a zero-knowledge proof that it knows the corresponding secret key csk_1 and an attribute vector $\vec{a}_1$.

The root authority verifies the zero-knowledge proof, and if it is valid returns a Groth signature

$$\textsc{Groth}_2.\textsc{Sign}(\mathsf{csk}_0; \mathsf{cpk}_1, \vec{a}_1) \rightarrow_{\$} \sigma_1$$

The intermediate authority now has *Level*-1 credential $\mathsf{cred}_1 = (\sigma_1, \vec{a}_1, \mathsf{cpk}_1)$.

Similarly, a *Level*-2 intermediate authority obtains a *Level*-2 credential by first executing $\textsc{Groth}_2.\textsc{KeyGen}(\Lambda^\star) \rightarrow_{\$} (\mathsf{csk}_2, \mathsf{cpk}_2)$ and then sending a credential request to a *Level*-1 authority. The credential request consists of public key cpk_2, a zero-knowledge proof that the requestor knows the corresponding secret key csk_2, and a vector of attributes $\vec{a}_2$.

The *Level*-1 authority accordingly checks the zero-knowledge proof, runs $\textsc{Groth}_1.\textsc{Sign}(\mathsf{csk}_1; \mathsf{cpk}_2, \vec{a_2}) \rightarrow_{\$} \sigma_2$ and returns a *Level*-2 credential $\mathsf{cred}_2 = (\sigma_1, \vec{a_1}, \mathsf{cpk}_1, \sigma_2, \vec{a_2}, \mathsf{cpk}_2)$ to the requestor.

This process can repeat L times, the resulting *Level*-L credential is $\mathsf{cred}_L = (\langle \sigma_i, \vec{a_i}, \mathsf{cpk}_i \rangle_{i=1}^{L})$.

Credential presentation: To sign a message m while disclosing attributes $\langle a_{i,j} \rangle_{(i,j)\in D}$, a *Level*-$L$ user generates the following NIZK proof:

$$\begin{aligned}
&\mathfrak{P}_{\mathsf{cred}} \leftarrow_{\$} \mathrm{NIZK}\{(\sigma_{1,\dots,L}, \mathsf{cpk}_{1,\dots,L}, \langle a_{i,j} \rangle_{(i,j)\notin D}, \sigma_m) : \\
&\bigwedge_{i=2,4,\dots}^{L} \textsc{Groth}_1.\textsc{Verify}(\mathsf{cpk}_{i-1}; \sigma_i; \mathsf{cpk}_i, a_{i,1}, \dots, a_{i,n_i}) \\
&\bigwedge_{i=1,3,\dots}^{L} \textsc{Groth}_2.\textsc{Verify}(\mathsf{cpk}_{i-1}; \sigma_i; \mathsf{cpk}_i, a_{i,1}, \dots, a_{i,n_i}) \\
&\qquad\qquad\qquad \wedge \textsc{Schnorr}.\textsc{Verify}(\mathsf{cpk}_L; \sigma_m; m)\}
\end{aligned}$$

In addition to proving the validity of the credential chain, the user signs an input message m using a Schnorr-like procedure. This allows the user to show that she knows the secret key corresponding to the last-level credentials.

Verification: Upon receipt of proof $\mathfrak{P}_{\mathsf{cred}}$, a verifier checks its correctness with respect to public key cpk_0 of the root authority, message m and the disclosed attributes $\langle a_{i,j} \rangle_{(i,j)\in D}$.

For more details on the implementation of the zero-knowledge proofs interested readers can refer to Algorithm 3 in Appendix B. This algorithm also includes details on how to integrate this instantiation with Hyperledger Fabric.

Towards real-world adoption: Anonymous credentials provide a generic solution to privacy-preserving transaction authorization in permissioned blockchains. Nonetheless, on their own they fall short of addressing the requirements of revocation and auditability. For instance, the above instantiation does not allow a verifier of the blockchain to tell if the credentials used to sign the transaction are still valid (not revoked); neither does it allow authorized parties (e.g., auditors) to trace the origin of the transactions posted in the ledger.

The following section extends the protocol to address these shortcomings.

V. AUDITABLE DELEGATABLE ANONYMOUS CREDENTIALS WITH REVOCATION

Revocation: Classical mechanisms for revocation are at odds with anonymous credentials, whereas privacy-friendly alternatives — such as as zero-knowledge sets [30] or accumulators combined with zero-knowledge proofs [7] — are too computationally prohibitive to be integrated into Hyperledger Fabric.

To enable *efficient* and *privacy-preserving* revocation we couple *epoch-based whitelisting* with signatures in a way that yields efficient proofs of *non-revocation*. Namely, we divide the timeline into *epochs* that define the validity periods of the credentials. For each epoch, a non-revoked participant is issued an *epoch handle* (a signature) that binds her public key to the epoch. When a participant presents her credentials, she provides along with them a proof of non-revocation that consists of proving in zero-knowledge that she holds a signature linking her public key to the current epoch. Credentials that are valid for a certain epoch are automatically revoked the moment the epoch expires. An epoch expires either naturally (epoch elapses) or manually (authorized parties advance the epoch by putting a special message on the ledger).

We define epochs in terms of blockchain height, which ensures that transactions of revoked parties are going to be rejected by the verifiers in the blockchain.

For ease of exposition, we assume that only the credentials of users are revoked (i.e. *Level*-L credentials). We contend that such an assumption is fair as organizations in Fabric will not be revoked as frequently as users, who, on the other hand, may have their authorization to submit transactions denied at any moment (e.g., a failure to pay a monthly subscription, an employee leaving her company, etc.). We note though that the proposed mechanisms can be generalized to accommodate settings in which intermediate authorities are also revoked.

Let $sid = (\mathcal{R}, \mathcal{AU}, \mathcal{T}, L, \mathsf{Param}, sid')$ be the session identifier.

1) Setup. On input $(\mathsf{SETUP}, \langle n_i \rangle_i)$ from root $\mathcal{R}$.
   - Output $(\mathsf{SETUP}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, pp', \mathsf{Present}, \mathsf{Verify}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$.
   - Store algorithms Present and Verify and parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\mathrm{de}}, \mathcal{L}_{\mathfrak{p}}, \mathcal{L}_{\mathrm{au}} \leftarrow \emptyset$. If $\mathcal{AU}$ is corrupt set $pp \leftarrow pp'$, else set $pp \leftarrow \mathsf{Param}()$.
   - Output $\mathsf{SETUPDONE}$ to $\mathcal{R}$.

   On input SETUP from $\mathcal{AU}$, output $(\mathsf{SETUP}, \mathcal{AU})$ to $\mathcal{A}$, wait for response; output $\mathsf{SETUPDONE}$ to $\mathcal{AU}$.
2) $\mathsf{Advance}$. On input $\mathsf{ADVANCE}$ from $\mathcal{T}$, set $\mathcal{L}_{\mathfrak{p}} \leftarrow \emptyset$, $\mathcal{L}_{\mathrm{de}} \leftarrow \{\langle \mathcal{P}_i, \vec{a}_1, \ldots, \vec{a}_l \rangle \in \mathcal{L}_{\mathrm{de}} : l < L\}$.
3) $\mathsf{Delegate}$. On input $(\mathsf{DELEGATE}, ssid, \vec{a}_1, \ldots, \vec{a}_l, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $l \leq L$ and $\vec{a}_l \in \mathbb{A}_l^{n_l}$.
   - If $l = 1$: check $sid = (\mathcal{P}_i, \mathcal{AU}, \mathcal{T}, L, sid')$, else abort.
   - If $l > 1$, check that $\langle \mathcal{P}_i, \vec{a}_1, \ldots, \vec{a}_{l-1} \rangle \in \mathcal{L}_{\mathrm{de}}$, else abort.
   - Output $(\mathsf{ALLOWDEL}, ssid, \mathcal{P}_i, \mathcal{P}_j, l)$ to $\mathcal{A}$; wait for input $(\mathsf{ALLOWDEL}, ssid)$ from $\mathcal{A}$.
   - Add an entry $\langle \mathcal{P}_j, \vec{a}_1, \ldots, \vec{a}_l \rangle$ to $\mathcal{L}_{\mathrm{de}}$.
   - Output $(\mathsf{DELEGATE}, ssid, \vec{a}_1, \ldots, \vec{a}_l, \mathcal{P}_i)$ to $\mathcal{P}_j$.
4) $\mathsf{Present}$. On input $(\mathsf{PRESENT}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \{\perp\})^{n_i}$ for $i = 1, \ldots, L$.
   - Check that an entry $\langle \mathcal{P}_i, \vec{a}'_1, \ldots, \vec{a}'_L \rangle$ exists in $\mathcal{L}_{\mathrm{de}}$ such that $\vec{a}_i \preceq \vec{a}'_i$ for $i = 1, \ldots, L$.
   - If $\mathcal{AU}$ honest, set $\mathfrak{p} \leftarrow \mathsf{Present}(pp, m, \vec{a}_1, \ldots, \vec{a}_L; \perp)$, else $\mathfrak{p} \leftarrow \mathsf{Present}(pp, m, \vec{a}_1, \ldots, \vec{a}_L; \mathcal{P}_i)$. Abort if $\mathsf{Verify}(pp, \mathfrak{p}, m, \vec{a}_1, \ldots, \vec{a}_L) = 0$.
   - Store $\langle m, \vec{a}_1, \ldots, \vec{a}_L, \mathfrak{p} \rangle$ in $\mathcal{L}_{\mathfrak{p}}$ and $\langle \mathfrak{p}, \mathcal{P}_i \rangle$ in $\mathcal{L}_{\mathrm{au}}$.
   - Output $(\mathsf{PROOF}, \mathfrak{p})$ to $\mathcal{P}_i$.
5) Verify. On input $(\mathsf{VERIFY}, \mathfrak{p}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from $\mathcal{P}_i$.
   - If $\langle m, \vec{a}_1, \ldots, \vec{a}_L, \mathfrak{p} \rangle \notin \mathcal{L}_{\mathfrak{p}}$, $\mathcal{R}$ is honest, and for $i = 1, \ldots, L$, there is no corrupt $\mathcal{P}_j$ with $\langle \mathcal{P}_j, \vec{a}'_1, \ldots, \vec{a}'_i \rangle \in \mathcal{L}_{\mathrm{de}}$ and $\vec{a}_j \preceq \vec{a}'_j$ for $j = 1, \ldots, i$, set $f \leftarrow 0$.
   - Else, output $(\mathsf{VERIFY}, \mathfrak{p})$ to $\mathcal{A}$; expect response $(\mathsf{VERIFY}, \mathcal{P})$. Set $f \leftarrow \mathsf{Verify}(pp, \mathfrak{p}, m, \vec{a}_1, \ldots, \vec{a}_L)$. If $\mathcal{P}$ corrupt $\wedge f$ then add $\langle \mathfrak{p}, \mathcal{P} \rangle$ to $\mathcal{L}_{\mathrm{au}}$.
   - Output $(\mathsf{VERIFIED}, f)$ to $\mathcal{P}_i$.
6) Audit. On input $(\mathsf{AUDIT}, \mathfrak{p})$ from $\mathcal{AU}$, if $\langle \mathfrak{p}, \mathcal{P} \rangle \notin \mathcal{L}_{\mathrm{au}}$, output $(\mathsf{AUDIT}, \mathfrak{p})$ to $\mathcal{A}$. Upon obtaining $(\mathsf{AUDIT}, \mathcal{P})$ from $\mathcal{A}$, where $\mathcal{P}$ is corrupted, store $\langle \mathfrak{p}, \mathcal{P} \rangle$ in $\mathcal{L}_{\mathrm{au}}$. If now there is a valid record $\langle \mathfrak{p}, \mathcal{P} \rangle$ in $\mathcal{L}_{\mathrm{au}}$, then output $(\mathsf{RESULT}, \mathcal{P})$ to $\mathcal{AU}$. Else, output $\perp$ to $\mathcal{AU}$.

Fig. 1: Extended credentials functionality $\mathcal{F}_{\mathrm{dac+}}$.

Audit: To enable auditing, the transaction author embeds her identifier (the public key) encrypted under the auditor's public key into the transaction using a semantically secure encryption. For this solution to be viable, it must ensure that the user (**1**) encrypts her own public key and (**2**) uses the public key of the authorized auditor. Zero-knowledge proofs such as [33] coupled with ElGamal encryption [21] allow us to address these challenges relatively efficiently.

For the sake of simplicity, we only focus on settings where just a single auditor is present for all the users in the system. The proposed solution could be easily enhanced to support scenarios with multiple auditors. Namely, users will have their auditor's public key as an attribute and the proof of correct encryption will show that the correct public key is being used.

A. Security definition

We define the security of our extended scheme based on the functionality $\mathcal{F}_{\mathrm{dac}}$ from [12]. We model revocation by introducing a message $\mathsf{ADVANCE}$ that can be input by a special party $\mathcal{T}$, and that effects in all last-level delegations as well as generated proofs becoming invalid. This input models an epoch switch. We model audit by providing an input AUDIT to an auditor $\mathcal{AU}$, which upon input of a credential proof $\mathfrak{p}$ outputs the party $\mathcal{P}$ that presented $\mathfrak{p}$. Properly modeling audit also requires to account for the case where $\mathcal{AU}$ is corrupted. This is achieved by allowing $\mathcal{A}$ to input the parameters pp' so that Present can include the identity of the origin of each proof $\mathfrak{p}$, which is necessary since a corrupt auditor will be able to decrypt this information anyway. The complete functionality $\mathcal{F}_{\mathrm{dac+}}$ is specified in Fig. 1.

B. Revocation

We describe two alternative solutions that differ in their generality. The first one is straightforward but requires revocations being handled by the same authorities that issue user credentials. The second is more complex but allows revocations and credentials to be handled by different authorities. See Section VIII-B for performance analysis of the latter approach.

Epoch as an attribute: 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 case uses the proof generation depicted in Algorithm 3 such that one of the disclosed attributes is the identifier of the current epoch. Note that in this case only the last-level credentials are being regenerated in each epoch.

Explicit proof of non-revocation: The solution above requires no additional cryptographic implementation, however, it suffers from the limitation that the credential issuer must always be the same as the revocation authority. To accommodate settings where credential issuers are different from revocation authorities, we decouple the credentials for user attributes from epoch credentials. To obtain authorization for the current epoch, a user contacts the *revocation authority* with a proof of her public key possession. The revocation authority in turn responds with a Groth signature of the user's public key and the epoch identifier. When the user wishes to submit a transaction, she generates a proof of non-revocation that shows the knowledge of an epoch handle and the associated secret key. Verifiers in the blockchain check the non-revocation proof and if valid, verify the user's signature on the transaction content. In more formal terms, we augment the protocol in Section IV-B2 with the following.

Let g denote a generator of the bilinear group in which the public keys of users (i.e., public keys associated with *Level-L* credentials) reside, and let f denote a generator of the other bilinear group.

Revocation setup: The revocation authority computes its pair of Groth secret and public keys $(\mathsf{rsk}, \mathsf{rpk} = f^{\mathsf{rsk}}) \leftarrow\!\$\ \textsc{Groth.KeyGen}(\Lambda^\star)$ and publishes rpk.

Generation of non-revocation credentials: Upon receipt of a credential request for public key cpk and current epoch, revocation authority verifies that the requestor knows the secret key matching cpk, and computes

$$\begin{aligned}\varepsilon &:= g^{\textsc{Hash}(\mathsf{epoch})}\\ \sigma &\leftarrow\!\$\ \textsc{Groth.Sign}(\mathsf{rsk}; \varepsilon, \mathsf{cpk})\end{aligned}$$

and returns non-revocation credentials (σ, cpk).

Proof generation: A user signs a message m and proves that she is not revoked during the current epoch by outputting a tuple $(m, \langle a_{i,j}\rangle_{(i,j)\in D}, \mathfrak{P})$ such that:

$$\begin{aligned}\mathfrak{P} \leftarrow\!\$\ \mathsf{NIZK}\{&(\sigma_{1,\ldots,L}, \mathsf{cpk}_{1,\ldots,L}, \langle a_{i,j}\rangle_{(i,j)\notin D}, \sigma_m, \sigma):\\ &\bigwedge_{i=2,4,\ldots}^{L} \textsc{Groth}_1.\textsc{Verify}(\mathsf{cpk}_{i-1}; \sigma_i; \mathsf{cpk}_i, a_{i,1}, \ldots, a_{i,n_i})\\ &\bigwedge_{i=1,3,\ldots}^{L} \textsc{Groth}_2.\textsc{Verify}(\mathsf{cpk}_{i-1}; \sigma_i; \mathsf{cpk}_i, a_{i,1}, \ldots, a_{i,n_i})\\ &\wedge \textsc{Schnorr.Verify}(\mathsf{cpk}_L; \sigma_m; m)\\ &\wedge \textsc{Groth.Verify}(\mathsf{rpk}; \sigma; \varepsilon, \mathsf{cpk}_L)\}\end{aligned}$$

C. Audit

Our auditable anonymous delegatable credentials extends the scheme described in Section IV-B2 by adding an *Audit setup* and enhancing the credential presentation with verifiable encryption.

Audit setup: The authorized auditor computes a pair of ElGamal secret and public keys $(\mathsf{ask}, \mathsf{apk} = g^{\mathsf{ask}})$ and then announces apk. We assume there are mechanisms in place to verify that the auditor is legitimate and knows the secret key ask.

Proof Generation: A user signs a message m in an auditable manner and outputs a tuple $(m, \langle a_{i,j}\rangle_{(i,j)\in D}, \mathsf{enc}, \mathfrak{P})$ such that:

$$\begin{aligned}\mathfrak{P} \leftarrow\!\$\ \mathsf{NIZK}\{&(\sigma_{1,\ldots,L}, \mathsf{cpk}_{1,\ldots,L}, \langle a_{i,j}\rangle_{(i,j)\notin D}, \sigma_m, \sigma, \rho):\\ &\bigwedge_{i=2,4,\ldots}^{L} \textsc{Groth}_1.\textsc{Verify}(\mathsf{cpk}_{i-1}; \sigma_i; \mathsf{cpk}_i, a_{i,1}, \ldots, a_{i,n_i})\\ &\bigwedge_{i=1,3,\ldots}^{L} \textsc{Groth}_2.\textsc{Verify}(\mathsf{cpk}_{i-1}; \sigma_i; \mathsf{cpk}_i, a_{i,1}, \ldots, a_{i,n_i})\\ &\wedge \textsc{Schnorr.Verify}(\mathsf{cpk}_L; \sigma_m; m)\\ &\wedge \textsc{Groth.Verify}(\mathsf{rpk}; \sigma; \varepsilon, \mathsf{cpk}_L)\\ &\wedge \mathsf{enc} = (\mathsf{cpk}_L \cdot \mathsf{apk}^{\rho}, g^{\rho})\}\end{aligned}$$

If the auditor decides to learn the identity of the origin of a message m, all she needs to do is to decrypt ciphertext enc. This process is guaranteed to succeed and correctly yield the right public key thanks to the soundness of $\mathfrak{P}$.

Details on the implementation of this extension is provided in Algorithms 4 to 6 in Appendix B. Algorithm 2 puts all the components together and includes elements of the integration with Hyperledger Fabric.

D. Security statement

In Appendix A, we prove that our extended protocol realizes the functionality specified in Fig. 1.

Theorem 1. Delegatable credentials protocol $\Pi_{\mathrm{dac+}}$ securely realizes $\mathcal{F}_{\mathrm{dac+}}$ in the $(\mathcal{F}_{\mathrm{smt}}, \mathcal{F}_{\mathrm{ca}}, \mathcal{F}_{\mathrm{crs}}, \mathcal{F}_{\mathrm{clock}})$-hybrid model, provided that

- SIGNNYM (Algorithm 4) is a strongly unforgeable signature,
- the auditing encryption is semantically secure,
- NIZK is a simulation-sound extractable non-interactive zero-knowledge proof.

Our instantiated protocol is covered by the security statement since both Schnorr (used in binding the pseudonym) and Groth signatures are existentially unforgeable, ElGamal encryption is semantically secure, and Schnorr proofs are simulation-sound extractable.

The full proof of Theorem 1 could be found in Appendix A.

VI. OPTIMIZED IMPLEMENTATION

While implementing our extended protocol, we discovered several enhancements and optimizations over the scheme in [12] that improve the readability and performance of our code. This section presents our improvements.

A. Corrected and refactored pseudocode

Following the pseudocode [12, Figures 4 and 5] precisely we have found out that the verification always fails. We were able to spot the mistakes and provide a corrected version in Algorithm 3. 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 heaviest operation in the code is the computation of commitments. Moreover, we have found that commitments can be computed independently of one another, and therefore can be easily parallelized. Instead of computing the commitments eagerly, our program schedules the computation and puts it in a queue. Before hashing 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 as the computation takes long enough to neglect a cost of spawning an extra thread and is small enough that the system can uniformly disperse its load among available resources.

C. Miller's loop and final exponentiation

Camenisch, Drijvers, and Dubovitskaya [12] mention that when computing a product of pairings it makes sense to compute Miller's loop first on some 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 a product. The trick is to use bilinearity of Miller's loop to put exponents inside the pairings. For example, the following computations are equivalent:

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

Since exponentiations are cheaper in $\mathbb{G}_1$ than in $\mathbb{G}_2$ (specifically, when using AMCL library [34]), we decided to exponentiate elements in $\mathbb{G}_1$. See Algorithm 1.

Algorithm 1 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 $L = 1, \dots, n$

Ensure: $\text{EPRODUCT}(\langle a_i, b_i, c_i \rangle_{i=1}^{n}) = \prod_{i=1}^{n} e(a_i, b_i)^{c_i}$

```
procedure EPRODUCT($\langle a_i, b_i, c_i \rangle_{i=1}^{n}$)
    $r := 1_T \in \mathbb{G}_T$                          ▷ an identity element
    for $i = (1, \dots, n)$ do
        if $c_i \neq \perp$ then
            $a_i := a_i^{c_i}$
    for $i = (1, 3, \dots, n)$ 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 FEXP $(r)$
```

VII. INTEGRATION WITH HYPERLEDGER FABRIC

This section presents our protocol and explains how the building blocks defined earlier work together within Fabric. We assume that all parties have access to system parameters sp and public key cpk_0 of the root authority, and that they have generated their pairs of secret and public keys. The keys are always generated as $\mathsf{sk} \leftarrow\!\!\$\, \mathbb{Z}_q$ and $\mathsf{pk} := g^{\mathsf{sk}}$ where g is a group generator of either $\mathbb{G}_1$ or $\mathbb{G}_2$ depending on the delegation level.

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 delegatable credentials 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 that leverages the content of the transaction to compute the challenge and shows knowledge of the secret key committed in the pseudonym.

The verifier first checks whether *Creator* and *Signature* include the same pseudonym. If so, 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 a chaincode by providing a NIZK proof and a linked signature on the proposal, as described in Section VII-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 2).

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 where 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

Algorithm 2 $\Pi_{\mathrm{dac+}}$: delegation, revocation, auditing and transaction submission protocols

```
1 :  Level-$i$ CA                                                                  Level-$(i+1)$ CA
..................... Repeated for $L$ rounds of delegation (from the Root CA to Intermediate CAs to the User) .....................
2 :  $\mathsf{csk}_i \leftarrow\$ \mathbb{Z}_q, \mathsf{cpk}_i := g^{\mathsf{csk}_i}$                                  $\mathsf{csk}_{i+1} \leftarrow\$ \mathbb{Z}_q, \mathsf{cpk}_{i+1} := f^{\mathsf{csk}_{i+1}}$
3 :  $\mathsf{nonce} \leftarrow\$ \{0,1\}^\lambda$          $\xrightarrow{\mathsf{nonce}}$          $\mathfrak{P}_{\mathsf{pk}} \leftarrow\$ \textsc{ProvePK}(\mathsf{csk}_{i+1}, \mathsf{cpk}_{i+1}, \mathsf{nonce})$
4 :  $\textsc{VerifyPK}(\mathfrak{P}_{\mathsf{pk}}, \mathsf{cpk}_{i+1}, \mathsf{nonce})$          $\xleftarrow{\mathfrak{P}_{\mathsf{pk}}, \mathsf{cpk}_{i+1}}$
5 :  $\sigma_{i+1} \leftarrow\$ \textsc{Groth.Sign}(\mathsf{csk}_i; \mathsf{cpk}_{i+1}, \vec{a}_{i+1})$          $\xrightarrow{\sigma_{i+1}}$          $\mathsf{cred}_{i+1} := (\sigma_{i+1}, \vec{a}_{i+1}, \mathsf{cpk}_{i+1})$
6 :  Revocation authority                                                          User
..................... On each epoch, user requests a non-revocation handle .....................
7 :  $\mathsf{rsk} \leftarrow\$ \mathbb{Z}_q, \mathsf{rpk} := g^{\mathsf{rsk}}$                                  $\mathsf{csk} \leftarrow\$ \mathbb{Z}_q, \mathsf{cpk} := g^{\mathsf{csk}}$
8 :  $\mathsf{nonce} \leftarrow\$ \{0,1\}^\lambda$          $\xrightarrow{\mathsf{nonce}}$          $\mathfrak{P}_{\mathsf{pk}} \leftarrow\$ \textsc{ProvePK}(\mathsf{csk}, \mathsf{cpk}, \mathsf{nonce})$
9 :  $\textsc{VerifyPK}(\mathfrak{P}_{\mathsf{pk}}, \mathsf{cpk}, \mathsf{nonce})$          $\xleftarrow{\mathfrak{P}_{\mathsf{pk}}, \mathsf{cpk}}$
10 : $\sigma \leftarrow\$ \textsc{NRSign}(\mathsf{rsk}; \mathsf{cpk}, \mathsf{epoch})$          $\xrightarrow{\sigma}$          $\sigma, \mathsf{epoch}$
11 : Verifier                                                                      User
12 :                                        (from the delegation stage)          $\mathsf{cred} := (\langle \sigma_j, \vec{a}_j, \mathsf{cpk}_j \rangle_{j=1}^{L})$
..................... User submits a transaction .....................
13 :                                                                                $\mathsf{enc}, \rho := \textsc{AuditEnc}(\mathsf{apk}, \mathsf{cpk})$
14 :                                                                                $\mathsf{sk}_{\mathsf{nym}}, \mathsf{pk}_{\mathsf{nym}} \leftarrow\$ \textsc{MakeNym}(\mathsf{csk})$
15 :                                                                                $\mathfrak{P}_{\mathsf{rev}} \leftarrow\$ \textsc{NRProve}(\sigma, \mathsf{csk}, \mathsf{sk}_{\mathsf{nym}}, \mathsf{epoch})$
16 :                                                                                $\mathfrak{P}_{\mathsf{audit}} \leftarrow\$ \textsc{AuditProve}(\mathsf{enc}, \rho, \mathsf{cpk}, \mathsf{csk}, \mathsf{pk}_{\mathsf{nym}}, \mathsf{sk}_{\mathsf{nym}})$
17 :                                        (no need to sign a message)          $\mathfrak{P}_{\mathsf{cred}} \leftarrow\$ \textsc{CredProve}(\mathsf{cred}, D, \mathsf{sk}_{\mathsf{nym}}, \mathsf{csk}, \perp)$
18 :                                                                                $\sigma_{\mathsf{nym}} \leftarrow\$ \textsc{SignNym}(\mathsf{pk}_{\mathsf{nym}}, \mathsf{sk}_{\mathsf{nym}}, \mathsf{csk}, \mathsf{tx})$
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 : $\textsc{CredVerify}(\mathfrak{P}_{\mathsf{cred}}, D, \mathsf{pk}_{\mathsf{nym}}, \perp)$
```

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 13–23) 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 VII-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. As part of the transaction the user also includes the proof of possessing a non-revocation handle (line 15) and an encryption of her public key under auditor's key, along with the proof of its correctness (lines 13 and 16). If the user does not have a non-revocation handle for the current epoch, she requests it from the authority (lines 8–10). Finally, she signs the content of the transaction with the secret key in the pseudonym (lines 14 and 18).

Verifiers consequently validate the transaction by checking that the proofs and signatures refer to the same pseudonym and that they are valid with respect to the disclosed attributes (lines 20 and 23).

VIII. BENCHMARKS

We have provided the first production-ready open-sourced implementation of delegatable credentials scheme and our extensions. It is generic and produces valid credentials and proofs for any number of levels and attributes for both groups: $\mathbb{G}_1$ and $\mathbb{G}_2$, for odd and even levels. The project is tested with over 470 tests and they cover 100 % of the code. We note that this is a significant improvement over the original code, which was only a prototype computing a single hard-coded credential. We also note that the original code is not open-sourced.

All benchmarks (unless otherwise specified) were run on `c2-standard-60` GCE VM running Ubuntu 18.04 (60 vCPUs, Intel Cascade Lake 3.1 GHz, 240 GB RAM). We have used Apache Milagro Cryptographic Library (AMCL) [34] with a 254-bit Barreto-Naehrig curve [8] for low-level operations such as pairings, exponentiations and PRG operations.

A. Anonymous and auditable delegatable credentials with revocation

We have run comprehensive benchmarks of every operation of the scheme and our extensions using multiple parameter values. We stress that our evaluation results differ from the ones in [12]. First, the implementations are written in different languages and run on different processors. These differences are significant when benchmarking cryptographic primitives, which mostly involve bit manipulations. Second, we have obtained the original code of [12] and we have noticed distinctions in benchmark methodologies. The code in [12] 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 most fair results. Third, our scheme includes pseudonym commitments, which add noticeable overhead for small values of L and n. 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. All benchmarked operations were run 100 times. Note that the most sensitive overhead is due to verification, since it is the operation that will be run by the entire Fabric network. In Fabric, having $L = 2$ and $n = 2$ covers most use-cases. We noticed that the overhead value is very sensible, thus for fairness we present the results with the highest overhead.

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 fivefold.

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 $n \in [0; 4]$. 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.

e-product	Parallelization	CREDPROVE		CREDVERIFY	
		Big	Small	Big	Small
disabled	disabled	2 873	843	1 523	948
enabled	disabled	1 312	341	853	372
disabled	enabled	1 480	357	890	352
enabled	enabled	890	191	391	197
Improvement (≈ times)		3.2	4.4	3.9	**4.8**

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

L \ n	0	1	2	3	4
1	41 ms 89 ms 398 B	51 ms 110 ms 534 B	63 ms 116 ms 670 B	72 ms 153 ms 806 B	82 ms 173 ms 942 B
2	94 ms 124 ms 801 B	138 ms 158 ms 1.2 kB	192 ms 198 ms 1.6 kB	255 ms 262 ms 2.0 kB	315 ms 310 ms 2.4 kB
3	173 ms 188 ms 1.2 kB	273 ms 249 ms 1.7 kB	367 ms 329 ms 2.3 kB	516 ms 387 ms 2.8 kB	616 ms 427 ms 3.3 kB
5	333 ms 276 ms 2.0 kB	542 ms 342 ms 2.9 kB	661 ms 391 ms 3.9 kB	891 ms 500 ms 4.8 kB	1 146 ms 648 ms 5.7 kB
10	822 ms 457 ms 4.0 kB	1 177 ms 638 ms 6 kB	1 652 ms 860 ms 8 kB	2 115 ms 1 053 ms 10 kB	2 666 ms 1 234 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.

Extensions: Table III depicts the performance results for the helper methods. Each method was run in both $\mathbb{G}_1$ and $\mathbb{G}_2$ (judged by the number of operations in a group). Note that operations in $\mathbb{G}_2$ are considerably slower in AMCL and that revocation routines are relatively slower due to the use of pairing in proofs. Our future work is to apply the optimizations we used with delegatable credentials scheme to this procedure as well.

Also note that adding pseudonyms, enabling auditing and proving possession of the secret key incur little overhead relative to the cost of credential proof generation.

Against older idemix: We have run the benchmarks against the non-delegatable idemix implementation currently in Fabric and against the Fabric MSP with no anonymity (see Section III-B). The default (non-idemix) Fabric MSP simply uses X.509 certificates and ECDSA algorithms [36] for signatures and verifications. The current idemix implementation

Procedure	Time $\mathbb{G}_1$	Time $\mathbb{G}_2$	Procedure	Time $\mathbb{G}_1$	Time $\mathbb{G}_2$
Groth.KeyGen	1.6	4.7	Groth.Sign	16	41
Groth.Randomize	11	23	Groth.Verify	53	62
Schnorr.Sign	1.6	4.8	Schnorr.Verify	2	9.6
AuditEncrypt	3	9.4	NRSign	14	30
AuditProve	5.8	24	NRProve	66	88
AuditVerify	9.2	39	NRVerify	127	149
MakeNym	2.1	9.4	ProvePK	3.1	9.4
SignNym	2.2	9.9	VerifyPK	2	9.5
VerifyNym	3.5	14	KeyGen	1.5	4.2

TABLE III: Running time of extensions in milliseconds.

in Fabric [13] uses BBS+ signatures [6]. A user in this construction essentially proves the knowledge of a signature on her attributes. This mechanism however, does not support delegation.

We have run a simple workload — generating secrets, signing and verifying identities — for all three mechanisms. For the default MSP we have run ECDSA algorithms available in Go `crypto` module using the P-384 curve — the most secure option available in Fabric. For the Fabric idemix MSP we have run the entire workload against the actual Go code in the official repository [17] using five attributes. Lastly, we have run the workload with our solution using a single level and five attributes.

Experimental results show the relative costs of using more privacy-preserving solutions. Default MSP takes $21\,\mathrm{ms}$, idemix MSP in Fabric takes $108\,\mathrm{ms}$ and our solution takes $210\,\mathrm{ms}$. Reasonably, the more anonymity a solution offers, the more expensive it is. We believe that this overhead is acceptable, given that privacy-preserving MSP operations are tailored for applications that see value in trading gains in performance for gains in privacy.

B. Revocation authority overhead

The revocation functionality requires a single (possibly distributed) revocation authority. A legitimate concern is that the revocation authority could become a bottleneck in a real-world deployment, as at the beginning of each epoch users need to update their revocation credentials to be able to submit transactions. We contend that in most cases this will not be an issue for the following reasons. First, since the users require the handle to submit transactions, we can safely assume that they will only request it when they are about to submit a transaction. Therefore, the load is more likely to be distributed, especially for long epochs. For short epochs a user may not even need the handle if she does not wish to submit a transaction. Second, the overhead of issuing the non-revocation handle is $15\,\mathrm{ms}$ to $30\,\mathrm{ms}$, which is much smaller than the time it takes to process an anonymous transaction. This means that a faster revocation authority does not necessarily result in any improvement on the perceived performance of the network (i.e., transaction latency and throughput).

To validate our intuition, we have designed a minimalistic server in Go that uses our library to process requests for non-revocation handles. We observed a stable 200 requests per second throughput on our testing machine. We note that the real deployment will likely use a replicated service, which will scale horizontally.

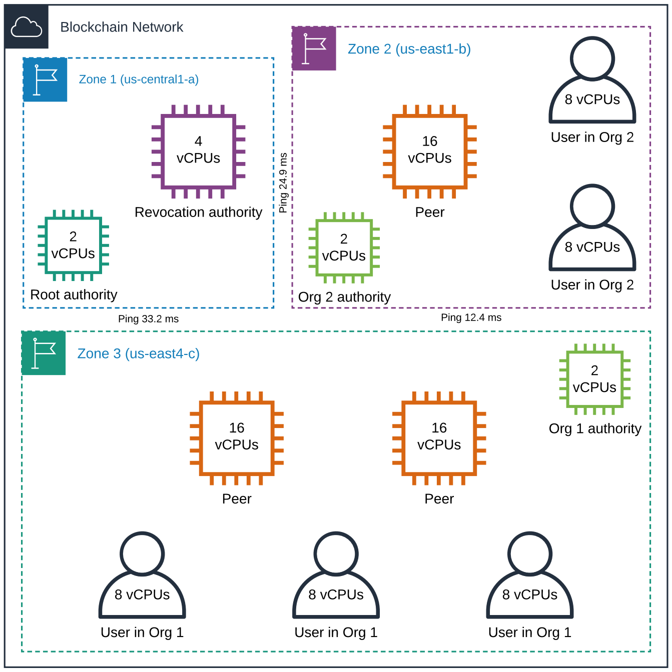

Fig. 2: Network architecture

IX. BLOCKCHAIN PROTOTYPE

We have built a standalone Hyperledger Fabric prototype to empirically assess the computation and network overhead of our implementation. We believe that this new prototype is of independent interest, and therefore, we open-sourced it together with a log parser, an analyzer and plotting scripts [5]. We also note that although our prototype faithfully mimics the processing and network components of Fabric, it is still an idealized version of the latter. Thus, the numbers we present here are a lower bound of the expected integration cost.

Our prototype integrates the cryptographic protocols of credentials delegation and transaction processing (recall Algorithm 2). More precisely, in the *setup* phase, root, revocation and auditing authorities, intermediate organizations and users generate their secret and public key pairs. The root delegates the credentials to organizations, which delegate them further to their respective users.

In the *transaction* phase, all users submit a configured number of transactions. Transactions are submitted sequentially for a single user but in parallel among all users. Users wait a configurable amount of time, sampled from Poisson distribution, before submitting the next transaction. Transaction processing requires executing the chaincode and computing read/write sets, which involves running a Docker container. We model this stage by waiting $50\,\mathrm{ms}$ — average time it takes to execute the simplest chaincode. In the *auditing* phase, an auditor goes over all transactions decrypting the user public keys.

When making design decisions, we chose very conservative options to measure the worst-case overhead. For example, we wait for all peers to validate a transaction, not for $50\,\% + 1$. We also have organizations and users associated with two attributes each to closely emulate typical Fabric deployment. The prototype has two modes of operation — local and distributed.

Local setting: In the *local* mode the blockchain is run on a single powerful machine (see Section VIII specs) with Fabric components modeled as threads running event loops. The purpose of this setting is very fine-grained control over the simulated network and detailed reporting.

Users, peers and authorities faithfully execute cryptographic protocols and the transfer of objects is carefully recorded. For each transaction, prototype reports the processing time — total and broken up into stages. It also reports all network traffic for all transferred objects in a run. Network log is further processed with scripts to produce statistics and plots.

The simulated network is composed of a global switch to which every party is connected, and a local switch per each party. The bandwidth of all switches is configurable — lower value for local and higher value for global. Once a party schedules an object transfer, such as user sending a transaction proposal to an endorser, the sender waits the time it takes to transfer the object through the pipeline. A pipeline consists of local switches of sender and receiver and a global switch. If any switch is occupied the sender waits in a queue.

Distributed setting: In the *distributed* mode the components (authorities, users and peers) are running each on its own VM and they transfer the objects over the real network. The purpose of this setup is running the entire protocol presented in this work in the most real-world setting. Experiments in this mode were run on `n1-standard` GCE VMs running Ubuntu 18.04 (Intel Broadwell 2.2 GHz).

To mimic realistic deployment we have used three geographically different regions for different components. Root (2 vCPUs) and revocation (4 vCPUs) authorities are deployed in Zone 1 (`us-central-1`). Sets of peers (16 vCPUs), users (8 vCPUs) and organization (*Level*-1) authorities (2 vCPUs) are deployed in Zone 2 (`us-east1-b`) and Zone 3 (`us-east4-c`). Ping times between zones are $12.5\,\mathrm{ms}$, $24.9\,\mathrm{ms}$ and $33.2\,\mathrm{ms}$. VMs talk to each other through remote procedure calls (RPC) asynchronously. We have depicted the setup in Fig. 2.

A. Results

We have run a number of experiments using *local* setting to study network behavior and *distributed* setting to study the effects of varying the number of peers, users, required endorsements and enabling / disabling revocation and auditing. Our default setting is 2 organizations, 5 users, 3 peers, 2 required endorsements and revocation with auditing enabled (see Fig. 2). In all experiments the users submit 100 transactions each (while waiting between transactions in the *local* setting).

Local setting: In *local* setting, we fine-tuned the bandwidth of our virtual network switches. Predictably, when the bandwidth is set to a higher value, the network imposes no overhead, whereas when the bandwidth is low, network becomes the bottleneck. The experiment has shown that the largest and most frequent object traveling in the network is a $4.9\,\mathrm{KiB}$ transaction. By tuning the bandwidth parameters in a binary search fashion, we found that approximately at $40\,\mathrm{KiB/s}$ locally and $100\,\mathrm{KiB/s}$ globally the network overhead stops affecting the transaction processing time. Typical Fabric deployment uses at least a $1\,\mathrm{GiB/s}$ network, which means that the increased object sizes of our idemix will have no effect on network performance.

We then have studied the effect of revocation on the network overhead. On top of benchmarking the revocation server in isolation (see Section VIII-B), we have run an experiment with frequent revocations. We have set the epoch length to 5 seconds and observed no significant change in the experiment results. Non-revocation request is just a small stage in a transaction submission, and the transaction throughput is much smaller than that of revocation processing.

To visualize network usage, we have plotted the network log in Fig. 3. The plot is generated from $20\,\mathrm{ms}$ intervals, each bar and tick representing data from a single interval. Bars on the bottom show the objects traveling in the network in a given interval. Lines on the top show the latency. Ideal latency (green) is the time it takes to transfer the object over an unsaturated network, i.e, object size over bandwidth. Real latency (red) shows how much time it actually took for the slowest object in the interval. Among other things, the figure shows that despite short epochs (5 seconds), revocation requests do not result in any spikes in latency.

Distributed setting: In the *distributed* setting, we started with a hypothetical best-case scenario: a single user and a single peer. The time it takes to generate, validate and commit a single transaction is below a second on average ($855\,\mathrm{ms}$). Disabling auditing and revocation saves additional $256\,\mathrm{ms}$, see Fig. 4 *minimal* category. Out of this, $164\,\mathrm{ms}$ is spent on endorsement, whereas validation takes $432\,\mathrm{ms}$. The rest is taken by user's actions: credential proof generation, signing, collecting and verifying endorsements, etc.

Auditing takes $4\,574\,\mathrm{ms}$, which corresponds to the decryption of the ElGamal ciphertexts (500 decryptions, about $9\,\mathrm{ms}$ per decryption).

We have run the default experiment (5 users, 3 peers, 2 endorsements, 20 seconds epochs and both extensions enabled) and have studied the effects of changing the settings on the transaction processing overhead. The default experiment involves 500 transactions and takes $1\,555\,\mathrm{ms}$ per transaction. The first group of experiments examines the overheads of extensions. Disabling revocation saves $15\,\%$, disabling auditing saves another $8\,\%$, see Fig. 4 *extensions* category.

We have found that the number of required endorsements does not significantly affect the overhead, see Fig. 4 *endorsements* category. This is expected since endorsements are processed in parallel and take a small fraction of transaction processing time.

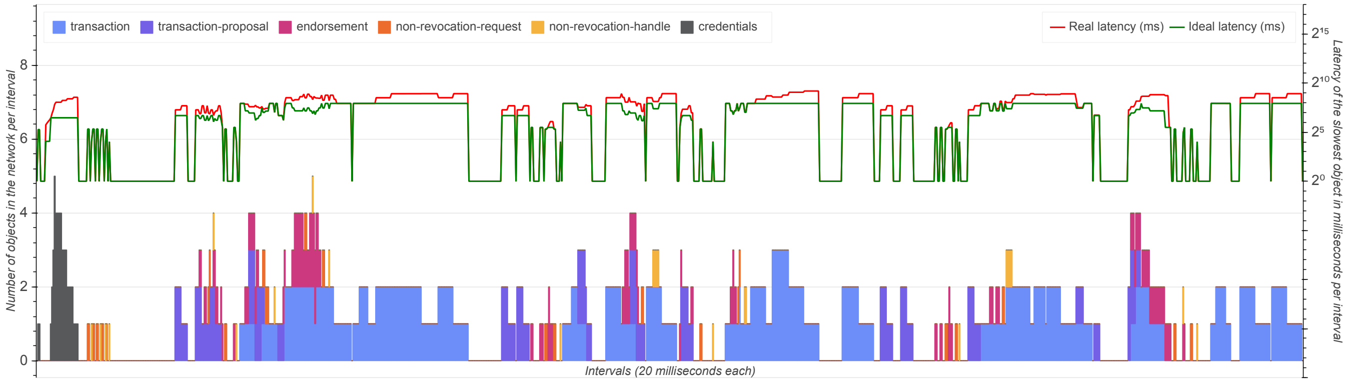

Fig. 3: Network log visualization (subset is shown, 18 transactions). Interval size is $20\,\mathrm{ms}$. Experiment involves 5 users, 3 peers, 2 endorsements, $20\,\mathrm{KiB/s}$ and $50\,\mathrm{KiB/s}$ local and global bandwidths, and epoch length 5 seconds. Bars show objects in the network, lines show latencies (green for ideal, red for real). Latency scale is logarithmic.

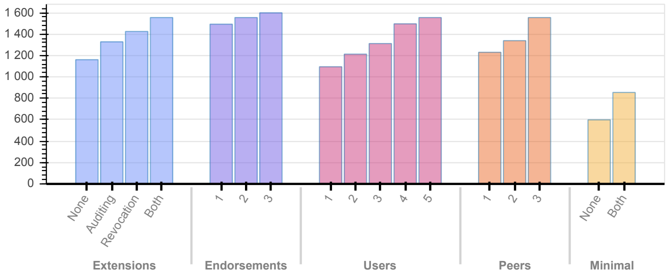

Fig. 4: Distributed experiments results. *Extensions* category runs default setting with some, all or no extensions enabled. *Endorsements*, *users* and *peers* categories vary the number of endorsements, users and peers respectively. *Minimal* category runs a single user, peer and endorsement with both or no extensions enabled. *Both* and *None* refer to enabling and disabling both revocation and auditing. Average transaction overhead is reported in milliseconds.

The number of users influences the overhead substantially, see Fig. 4 *users* category. Each new user increases the number of transactions validated by a single peer at a time. Since each peer eventually validates all transactions, the number of users is linearly correlated with the overhead regardless of the number of peers. Figure 4 indeed depicts linear relationship with a difference between 3 and 4 users attributed to different ping times between zones (recall Fig. 2).

Finally, we have found that the number of peers is also positively correlated with the overhead, see Fig. 4 *peers* category. This is also expected since a transaction is completed when the *last* peer validates it, and each new peer increases the variance in validation overhead. The difference between 2 and 3 peers is also attributed to inter-zone ping time.

In this section, we have shown the most relevant and representative experiments and results. We encourage the community to use our open-sourced prototype [5] to run more experiments in larger settings.

X. CONCLUSION

The possibility to perform transactions privately and anonymously is crucial to the use of blockchain technology in many financial and governmental use cases, as well as all use cases that involve personal data. Anonymous transaction authorization, as achieved through our implementation and extensions, is a 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 identified points for optimization to make the performance of delegatable credentials 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

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, we plan to implement a threshold protocol in which the organizations participating in the blockchain system jointly produce the first-level signatures, further distributing the trust (e.g., [35]).

In Fabric, every transaction is executed (endorsed) only by a subset of the peers, which allows 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 future line of work would be to remove this leakage. The idea is to equip the peers with idemix credentials and use commitments to obfuscate the endorsement policy. Then, after collecting all the required endorsements, the client can prove in zero-knowledge the knowledge of valid signatures that satisfy the obfuscated endorsement policy.

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] E. Androulaki, J. Camenisch, A. D. Caro, M. Dubovitskaya, K. Elkhiyaoui, and B. Tackmann, *Privacy-preserving auditable token payments in a permissioned blockchain system*, Cryptology ePrint Report 2019/1058, Nov. 2019.

[5] *Anonymized Repository for Fabric Network and Crypto Simulator*, https://anonymous.4open.science/r/3238a9fe-0a23-4a8a-8bd0-08b26d1ee255/, Accessed: 2020-02-12.

[6] M. H. Au, W. Susilo, and Y. Mu, "Constant-size dynamic k-TAA," in *International conference on security and cryptography for networks*, Springer, 2006, pp. 111–125.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] J. Camenisch, M. Drijvers, and A. Lehmann, "Anonymous attestation using the strong Diffie-Hellman assumption revisited," in *Trust and Trustworthy Computing*, Springer International Publishing, 2016, pp. 1–20.

[14] 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.

[15] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Third Symposium on Operating Systems Design and Implementation*, 1999.

[16] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theoretical Computer Science*, vol. 777, pp. 155–183, 2019.

[17] H. community, *Hyperledger fabric*, https://github.com/hyperledger/fabric, May 2020.

[18] 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.

[19] M. Drijvers, "Composable anonymous credentials from global random oracles," PhD thesis, ETH Zürich, Zürich, Switzerland, 2018.

[20] 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.

[21] 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.

[22] C. Garman, M. Green, and I. Miers, "Decentralized anonymous credentials," in *NDSS*, Internet Society, 2014.

[23] ——, "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.

[24] 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.

[25] 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.

[26] L. Harchandani, *Delegatable anonymous credentials in rust*, https://github.com/lovesh/signature-schemes/tree/delegatable/delg_cred_cdd, Sep. 2019.

[27] O. Harris. “Quorum,” [Online]. Available: https://www.goquorum.com/.

[28] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, “Universally composable synchronous computation,” in *Theory of Cryptography*, A. Sahai, Ed., ser. LNCS, vol. 7785, Springer, 2013, pp. 477–498.

[29] 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.

[30] 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.

[31] S. Nakamoto. (2009). “Bitcoin: A peer-to-peer electronic cash system,” [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[32] 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.

[33] 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.

[34] M. Scott, “The Apache Milagro Crypto Library,” [Online]. Available: https://github.com/MIRACL/amcl.

[35] A. Sonnino, M. Al-Bassam, S. Bano, S. Meiklejohn, and G. Danezis, “Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers,” *arXiv preprint arXiv:1802.07344*, 2018.

[36] N. I. of Standards and Technology, *FIPS PUB 186-4: Digital Signature Standard*. National Institute for Standards and Technology, Jul. 2013. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[37] C. Stathakopoulou, T. David, and M. Vukolić, *Mir-BFT: High-throughput BFT for blockchains*, arXiv:1906.05552, Jun. 2019.

[38] P. J. Windley. “Sovrin,” [Online]. Available: https://sovrin.org/.

[39] G. Wood. “Ethereum: A secure decentralised generalised transaction ledger,” [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[40] 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.

APPENDIX A

SECURITY ANALYSIS

Our theorem proving the security of the extended protocol builds directly on the proof of the core protocol from [12]. A significantly extended version of that proof appears in [19]. The main differences are as follows. (**1**) We write the scheme as a combination of standard signatures and NIZK, instead of sibling signatures and NIZK as used in [12], [19]. This is possible as we restrict ourselves to the case where the length of each delegation chain is fixed. (**2**) We need the NIZK to be non-malleable, as otherwise $\mathcal{F}_{\mathrm{dac}+}$ cannot identify the correct credential owner during an auditing query. This, however, is already implied by simulation-sound extractability. (**3**) We use a clock functionality [28] to model the advancement of epochs for the revocation scheme. We skip the parts of the description of the protocol $\Pi_{\mathrm{dac}+}$ and the proof that are identical to [12], and only discuss the differences that appear due to the revocation and auditing features.

Setup: In addition to root authority $\mathcal{R}$, auditor $\mathcal{AU}$ creates a Diffie-Hellman key pair and registers the public key. The auditor also registers a proof-of-knowledge of the private key like root authority $\mathcal{R}$, at functionality $\mathcal{F}_{\mathrm{ca}}$. We use the same scheme for $\mathcal{AU}$ as [12] uses for $\mathcal{R}$, so that we also achieve online extractability.

Advance: Upon input, the epoch counter $\mathcal{T}$ provides an input to $\mathcal{F}_{\mathrm{clock}}$, which advances the epoch.[1]

Delegate: Delegation is almost the same, except for the last delegation step (the one to the end user) where the delegator includes as one attribute the current epoch obtained from $\mathcal{F}_{\mathrm{clock}}$. In this step, the delegator also deposits the delegate's public key with $\mathcal{AU}$.

Present: There are three modifications during presentation. The first is that the user generates a new pseudonym and proves consistency. The second is that a credential proof is only generated if a relevant credential exists *for the present epoch*, and the attribute that encodes the current epoch is always disclosed. The third one is that, as explained in Section V-C, the user encrypts their public key under the auditor's public key using AUDITENC and then proves consistent encryption using AUDITPROVE.

Verify: The changes are dual to the above ones. The receiver, in addition to the standard credential validation, checks the consistency of the pseudonym, that the epoch attribute in the credential proof is valid, and the consistency of the auditing proof.

---

[1] Other parties interact with $\mathcal{F}_{\mathrm{clock}}$ to read the epoch. They technically also provide input to $\mathcal{F}_{\mathrm{clock}}$, which is required for modeling a synchrony assumption such as epochs in the otherwise asynchronous UC framework [28].

Let $sid = (\mathcal{R}, \mathcal{AU}, \mathcal{T}, L, \mathsf{Param}, sid')$ be the session identifier.

1) **Setup.** On input $(\mathsf{SETUP}, \langle n_i \rangle_i)$ from root $\mathcal{R}$.
    - Output $(\mathsf{SETUP}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, pp', \mathsf{Present}, \mathsf{Verify}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$.
    - Store algorithms Present and Verify and parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\text{de}}, \mathcal{L}_{\mathfrak{p}}, \mathcal{L}_{\text{au}} \leftarrow \emptyset$. If $\mathcal{AU}$ is corrupt set $pp \leftarrow pp'$, else set $pp \leftarrow \mathsf{Param}()$.
    - Output SETUPDONE to $\mathcal{R}$.

    On input SETUP from $\mathcal{AU}$, output $(\mathsf{SETUP}, \mathcal{AU})$ to $\mathcal{A}$, wait for response; output SETUPDONE to $\mathcal{AU}$.
2) **Advance.** On input ADVANCE from $\mathcal{T}$, set $\mathcal{L}_{\mathfrak{p}} \leftarrow \emptyset$, $\mathcal{L}_{\text{de}} \leftarrow \{\langle \mathcal{P}_i, \vec{a}_1, \ldots, \vec{a}_l \rangle \in \mathcal{L}_{\text{de}} : l < L\}$.
3) **Delegate.** On input $(\mathsf{DELEGATE}, ssid, \vec{a}_1, \ldots, \vec{a}_l, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $l \leq L$ and $\vec{a}_l \in \mathbb{A}_l^{n_l}$.
    - If $l = 1$: check $sid = (\mathcal{P}_i, \mathcal{AU}, \mathcal{T}, L, sid')$, else abort.
    - If $l > 1$, check that $\langle \mathcal{P}_i, \vec{a}_1, \ldots, \vec{a}_{l-1} \rangle \in \mathcal{L}_{\text{de}}$, else abort.
    - Output $(\mathsf{ALLOWDEL}, ssid, \mathcal{P}_i, \mathcal{P}_j, l)$ to $\mathcal{A}$; wait for input $(\mathsf{ALLOWDEL}, ssid)$ from $\mathcal{A}$.
    - Add an entry $\langle \mathcal{P}_j, \vec{a}_1, \ldots, \vec{a}_l \rangle$ to $\mathcal{L}_{\text{de}}$.
    - Output $(\mathsf{DELEGATE}, ssid, \vec{a}_1, \ldots, \vec{a}_l, \mathcal{P}_i)$ to $\mathcal{P}_j$.
4) **Present.** On input $(\mathsf{PRESENT}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \{\perp\})^{n_i}$ for $i = 1, \ldots, L$.
    - Check that an entry $\langle \mathcal{P}_i, \vec{a}'_1, \ldots, \vec{a}'_L \rangle$ exists in $\mathcal{L}_{\text{de}}$ such that $\vec{a}_i \preceq \vec{a}'_i$ for $i = 1, \ldots, L$.
    - If $\mathcal{AU}$ honest, set $\mathfrak{p} \leftarrow \mathsf{Present}(pp, m, \vec{a}_1, \ldots, \vec{a}_L; \perp)$, else $\mathfrak{p} \leftarrow \mathsf{Present}(pp, m, \vec{a}_1, \ldots, \vec{a}_L; \mathcal{P}_i)$. Abort if $\mathsf{Verify}(pp, \mathfrak{p}, m, \vec{a}_1, \ldots, \vec{a}_L) = 0$.
    - Store $\langle m, \vec{a}_1, \ldots, \vec{a}_L, \mathfrak{p} \rangle$ in $\mathcal{L}_{\mathfrak{p}}$ and $\langle \mathfrak{p}, \mathcal{P}_i \rangle$ in $\mathcal{L}_{\text{au}}$.
    - Output $(\mathsf{PROOF}, \mathfrak{p})$ to $\mathcal{P}_i$.
5) **Verify.** On input $(\mathsf{VERIFY}, \mathfrak{p}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from $\mathcal{P}_i$.
    - If $\langle m, \vec{a}_1, \ldots, \vec{a}_L, \mathfrak{p} \rangle \notin \mathcal{L}_{\mathfrak{p}}$, $\mathcal{R}$ is honest, and for $i = 1, \ldots, L$, there is no corrupt $\mathcal{P}_j$ with $\langle \mathcal{P}_j, \vec{a}'_1, \ldots, \vec{a}'_i \rangle \in \mathcal{L}_{\text{de}}$ and $\vec{a}_j \preceq \vec{a}'_j$ for $j = 1, \ldots, i$, set $f \leftarrow 0$.
    - Else, output $(\mathsf{VERIFY}, \mathfrak{p})$ to $\mathcal{A}$; expect response $(\mathsf{VERIFY}, \mathcal{P})$. Set $f \leftarrow \mathsf{Verify}(pp, \mathfrak{p}, m, \vec{a}_1, \ldots, \vec{a}_L)$. If $\mathcal{P}$ corrupt $\wedge f$ then add $\langle \mathfrak{p}, \mathcal{P} \rangle$ to $\mathcal{L}_{\text{au}}$.
    - Output $(\mathsf{VERIFIED}, f)$ to $\mathcal{P}_i$.
6) **Audit.** On input $(\mathsf{AUDIT}, \mathfrak{p})$ from $\mathcal{AU}$, if $\langle \mathfrak{p}, \mathcal{P} \rangle \notin \mathcal{L}_{\text{au}}$, output $(\mathsf{AUDIT}, \mathfrak{p})$ to $\mathcal{A}$. Upon obtaining $(\mathsf{AUDIT}, \mathcal{P})$ from $\mathcal{A}$, where $\mathcal{P}$ is corrupted, store $\langle \mathfrak{p}, \mathcal{P} \rangle$ in $\mathcal{L}_{\text{au}}$. If now there is a valid record $\langle \mathfrak{p}, \mathcal{P} \rangle$ in $\mathcal{L}_{\text{au}}$, then output $(\mathsf{RESULT}, \mathcal{P})$ to $\mathcal{AU}$. Else, output $\perp$ to $\mathcal{AU}$.

Fig. 5: Extended credentials functionality $\mathcal{F}_{\text{dac+}}$ (restated Fig. 1).

Audit: Given a credential proof, the auditor first checks its validity. If the credential proof is valid, the auditor then extracts the ciphertext that encrypts the user's key and decrypts it.

Theorem 1. Delegatable credentials protocol $\Pi_{\text{dac+}}$ securely realizes $\mathcal{F}_{\text{dac+}}$ in the $(\mathcal{F}_{\text{smt}}, \mathcal{F}_{\text{ca}}, \mathcal{F}_{\text{crs}}, \mathcal{F}_{\text{clock}})$-hybrid model, provided that

- SIGNNYM (Algorithm 4) is a strongly unforgeable signature,
- the auditing encryption is semantically secure,
- NIZK is a simulation-sound extractable non-interactive zero-knowledge proof.

The proof holds for static corruption of $\mathcal{AU}$.

Proof. We extend the proof of [12] to the functionality we added to the scheme. In **Setup**, the additional setup phase of auditor $\mathcal{AU}$ is proved analogously to that of the root authority. This includes the extraction of the private key if $\mathcal{AU}$ is corrupt; in that case the simulator sets pp to include the auditor's public key as well as public keys for all parties. If $\mathcal{AU}$ is honest, algorithm Param provides a fresh random key. **Advance** in $\mathcal{F}_{\text{dac+}}$ means that all issued credential proofs become invalid, and that the last-level delegations are deleted from $\mathcal{L}_{\text{de}}$. The same effect appears in the protocol, where the epoch advanced and inputs with old credential proofs to VERIFY will fail, as will the presentation of credentials that have been issued in an earlier epoch. **Delegate** behaves the same as before.

In the presentation phase, the credential proof $\mathfrak{p}$ returned by the functionality contains multiple additional elements (which in $\mathcal{F}_{\text{dac+}}$ are generated by the algorithm Present). The first two are $\mathsf{pk}_{\mathsf{nym}}$ and σ_{nym}, the pseudonym generated for this presentation and the signature on m. The next two are enc and $\mathfrak{P}_{\mathsf{audit}}$, the encryption of the user's public key pk under the auditor's public key apk, and the NIZK proving the correctness of this encryption. Algorithm Present generates the credential proof by building a fresh delegation chain with fresh keys and only the specified attributes; the only exception is that if $\mathcal{AU}$ is corrupt, then the correct user's public key, as indicated by the additional argument to Present, is chosen from pp and encrypted under the auditor's key. If $\mathcal{AU}$ is honest, then Present includes an encryption of a random message under the simulated auditor's public key in $\mathfrak{p}$. Present sets the additional values as follows: $\mathsf{pk}_{\mathsf{nym}}$ and σ_{nym} are set to a fresh pseudonym and a signature relative to $\mathsf{pk}_{\mathsf{nym}}$ and the also fresh user public key. If $\mathcal{AU}$ is corrupt then the encryption of the user public key under apk and the corresponding zero-knowledge proof are computed as in the scheme using the values from pp. If $\mathcal{AU}$ is honest, then (as discussed) a random encryption is chosen and the proof is simulated. This simulation requires that the encryption scheme is semantically secure and the NIZK is zero-knowledge to ensure that the consistency proof for the encryption is indistinguishable from a real proof, and that as in [12] fresh delegations are indistinguishable from the real world where the same delegations are used for multiple presentations.

In the verification phase, in both the real and the ideal cases, the verification algorithm is used to verify $\mathfrak{p}$. While in the ideal case with honest auditor the auditing proof is simulated, this will also successfully verify in Verify. The main difference is that $\mathcal{F}_{\mathrm{dac+}}$ prevents forgeries ideally whereas the protocol merely relies on the verification of the zero-knowledge proofs. The functionality also ensures that, for credential proofs that are accepted, their holders are known, therefore auditing will succeed. In the ideal world, the simulator knows the private key of $\mathcal{AU}$ (since it is chosen by the simulator if $\mathcal{AU}$ is honest, or extracted if $\mathcal{AU}$ is corrupt), and can therefore obtain the public key of the credential holder. This difference is indistinguishable by the simulation-sound extractability of the zero-knowledge proofs and the unforgeability of the signature scheme. Note that, in contrast with [12], we allow verification to succeed only for credential proofs $\mathfrak{p}$ that have either been generated by $\mathcal{F}_{\mathrm{dac+}}$ or are valid for corrupt parties. This in particular means that credential proofs are non-malleable, but non-malleability is already implied by simulation-sound extractability.

When honest $\mathcal{AU}$ inputs a credential proof $\mathfrak{p}$, the embedded ciphertext is decrypted. For credential proofs generated by an honest $\mathcal{P}_i$ this will always succeed. For those not generated by an honest $\mathcal{P}_i$, the functionality lets the adversary decide on the identity of the holder; the adversary can choose any corrupted party. The simulator can decrypt the auditing field of the credential proofs using the secret key of the auditor (which in case of a dishonest auditor has been extracted during setup). Indistinguishability again follows by the zero-knowledge property of the NIZK.

$\square$

APPENDIX B
ALGORITHMS

Algorithm 3 Improved proof generation and verification. Green is enhanced, red corrects mistakes in the original code.

```
procedure CredProve($\langle r_i, s_i, \langle t_{i,j}\rangle_{j=1}^{n_i+1}\rangle_{i=1}^{L}, \mathsf{csk}, \langle \mathsf{cpk}_i\rangle_{i=1}^{L}, \langle a_{i,j}\rangle_{i=1,\dots,L;j=1\dots,n_i}, D, \mathsf{sk}_{\mathsf{nym}}, m$)
    for $i = (1, \dots, L)$ 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 $j = 1, \dots, n_i + 1$ 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 $i = (1, \dots, L)$ 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
            $g_1 = \mathsf{sp}.g_2,\ g_2 = \mathsf{sp}.g_1,\ y = \mathsf{sp}.y_2$
        $\mathsf{com}_{i,1} := e(g_1, r_i)^{\rho_{\sigma_i}\cdot\rho_{s_i}} \left[\cdot e\left(g_1^{-1}, g_2\right)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
        $\mathsf{com}_{i,2} := e(g_1, r_i)^{\rho_{\sigma_i}\cdot\rho_{t_{i,1}}} \cdot e\left(g_1, g_2^{-1}\right)^{\rho_{\mathsf{cpk}_i}} \left[\cdot e(y_1, g_2)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
        for $j = (1, \dots, n_i)$ do
            if $(i, j) \in D$ then
                $\mathsf{com}_{i,j+2} := e(g_1, r_i)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}} \left[\cdot e(y_{j+1}, g_2)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
            else
                $\mathsf{com}_{i,j+2} := e(g_1, r_i)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}} \cdot e\left(g_1, g_2^{-1}\right)^{\rho_{a_{i,j}}} \left[\cdot e(y_{j+1}, g_2)^{\rho_{\mathsf{cpk}_{i-1}}}\right]_{i\neq 1}$
    $\mathsf{com}_{\mathsf{nym}} := g_1^{\rho_{\mathsf{cpk}_L}} h^{\rho_{\mathsf{nym}}}$
    $c := \textsc{Hash}(\mathsf{sp}.\mathsf{cpk}_0, \langle r'_i, \langle \mathsf{com}_{i,j}\rangle_{j=1}^{n_i+2}\rangle_{i=1}^{L}, \mathsf{com}_{\mathsf{nym}}, \langle a_{i,j}\rangle_{(i,j)\in D}, m)$
    for $i = (1, \dots, L)$ do
        if $i \bmod 2 = 1$ then $g := g_1$
        else $g = g_2$
        $\mathfrak{p}_{s_i} := g^{\rho_{s_i}} {s'}_i^{c},\ \left[\mathfrak{p}_{\mathsf{cpk}_i} := g^{\rho_{\mathsf{cpk}_i}} \mathsf{cpk}_i^{c}\right]_{i\neq L},\ \left[\mathfrak{p}_{\mathsf{csk}} := \rho_{\mathsf{cpk}_L} + c\cdot\mathsf{csk}\right]_{i=L},\ \left[\mathfrak{p}_{\mathsf{nym}} := \rho_{\mathsf{nym}} + c\cdot\mathsf{sk}_{\mathsf{nym}}\right]_{i=L}$
        for $j = 1, \dots, n_i + 1$ do
            $\mathfrak{p}_{t_{i,j}} := g^{\rho_{t_{i,j}}} {t'}_{i,j}^{c}$
        for $j : (i, j) \notin D$ do
            $\mathfrak{p}_{a_{i,j}} := g^{\rho_{a_{i,j}}} a_{i,j}^{c}$
    return $c, \langle r'_i, \mathfrak{p}_{s_i}, \langle \mathfrak{p}_{t_{i,j}}\rangle_{j=1}^{n_i+1}\rangle_{i=1}^{L}, \langle \mathfrak{p}_{a_{i,j}}\rangle_{(i,j)\notin D}, \langle \mathfrak{p}_{\mathsf{cpk}_i}\rangle_{i=1}^{L-1}, \mathfrak{p}_{\mathsf{nym}}, \mathfrak{p}_{\mathsf{csk}}$
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 $i = (1, \dots, L)$ 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
            $g_1 = \mathsf{sp}.g_2,\ g_2 = \mathsf{sp}.g_1,\ y = \mathsf{sp}.y_2$
        $\mathsf{com}_{i,1} := e(\mathfrak{p}_{s_i}, r'_i) \cdot e(y_1, g_2)^{-c} \left[\cdot e\left(g_1^{-1}, \mathfrak{p}_{\mathsf{cpk}_i}\right)\right]_{i\neq 1} \left[\cdot e(g_1, \mathsf{sp}.\mathsf{cpk}_0)^{-c}\right]_{i=1}$
        $\mathsf{com}_{i,2} := e\left(\mathfrak{p}_{t_{i,1}}, r'_i\right) \left[\cdot e\left(y_1, \mathfrak{p}_{\mathsf{cpk}_{i-1}}\right)\right]_{i\neq 1} \left[\cdot e(y_1, \mathsf{sp}.\mathsf{cpk}_0)^{-c}\right]_{i=1} \left[\cdot e\left(\mathfrak{p}_{\mathsf{cpk}_i}, g_2^{-1}\right)\right]_{i\neq L} \left[\cdot e\left(g_1, g_2^{-1}\right)^{\mathfrak{p}_{\mathsf{csk}}}\right]_{i=L}$
        for $j = (1, \dots, n_i)$ do
            if $(i, j) \in D$ then
                $\mathsf{com}_{i,j+2} := e\left(\mathfrak{p}_{t_{i,j+1}}, r'_i\right) \cdot e(a_{i,j}, g_2)^{-c} \left[\cdot e\left(y_{j+1}, \mathfrak{p}_{\mathsf{cpk}_{i-1}}\right)\right]_{i\neq 1} \left[\cdot e(y_{j+1}, \mathsf{sp}.\mathsf{cpk}_0)^{-c}\right]_{i=1}$
            else
                $\mathsf{com}_{i,j+2} := e\left(\mathfrak{p}_{t_{i,j+1}}, r'_i\right) \cdot e\left(\mathfrak{p}_{a_{i,j}}, g_2^{-1}\right) \left[\cdot e\left(y_{j+1}, \mathfrak{p}_{\mathsf{cpk}_{i-1}}\right)\right]_{i\neq 1} \left[\cdot e(y_{j+1}, \mathsf{sp}.\mathsf{cpk}_0)^{-c}\right]_{i=1}$
    $\mathsf{com}_{\mathsf{nym}} := g_1^{\mathfrak{p}_{\mathsf{csk}}} h^{\mathfrak{p}_{\mathsf{nym}}} \mathsf{pk}_{\mathsf{nym}}^{-c}$
    $c' := \textsc{Hash}(\mathsf{sp}.\mathsf{cpk}_0, \langle r'_i, \langle \mathsf{com}_{i,j}\rangle_{j=1}^{n_i+2}\rangle_{i=1}^{L}, \mathsf{com}_{\mathsf{nym}}, \langle a_{i,j}\rangle_{(i,j)\in D}, m)$
    return $c = c'$
```

Algorithm 4 Pseudonym and public key possession proof algorithms

```
procedure $\textsc{MakeNym}(\mathsf{csk})$
    $\mathsf{sk_{nym}} \xleftarrow{\$} \mathbb{Z}_q$
    $\mathsf{pk_{nym}} := g^{\mathsf{csk}} h^{\mathsf{sk_{nym}}}$
    return $\mathsf{sk_{nym}}, \mathsf{pk_{nym}}$
procedure $\textsc{SignNym}(\mathsf{pk_{nym}}, \mathsf{sk_{nym}}, \mathsf{csk}, m)$
    $\rho_1, \rho_2 \xleftarrow{\$} \mathbb{Z}_q$
    $\mathsf{com} := g^{\rho_1} h^{\rho_2}$
    $c := \textsc{Hash}(\mathsf{com}, \mathsf{pk_{nym}}, m)$
    $\mathfrak{p}_{\mathsf{csk}} := \rho_1 + c \cdot \mathsf{csk}$
    $\mathfrak{p}_{\mathsf{skNym}} := \rho_2 + c \cdot \mathsf{sk_{nym}}$
    return $c, \mathfrak{p}_{\mathsf{csk}}, \mathfrak{p}_{\mathsf{skNym}}$
procedure $\textsc{VerifyNym}(\mathsf{pk_{nym}}, m, c, \mathfrak{p}_{\mathsf{csk}}, \mathfrak{p}_{\mathsf{skNym}})$
    $\mathsf{com} = g^{\mathfrak{p}_{\mathsf{csk}}} h^{\mathfrak{p}_{\mathsf{skNym}}} \mathsf{pk_{nym}}^{-c}$
    return $c = \textsc{Hash}(\mathsf{com}, \mathsf{pk_{nym}}, m)$
procedure $\textsc{ProvePK}(\mathsf{csk}, \mathsf{cpk}, \mathsf{nonce})$
    $\rho \xleftarrow{\$} \mathbb{Z}_q$
    $\mathsf{com} := g^{\rho}$
    $c := \textsc{Hash}(\mathsf{com}, \mathsf{cpk}, \mathsf{nonce})$
    $\mathfrak{p} := \rho + c \cdot \mathsf{csk}$
    return $c, \mathfrak{p}$
procedure $\textsc{VerifyPK}(c, \mathfrak{p}, \mathsf{cpk}, \mathsf{nonce})$
    $\mathsf{com} = g^{\mathfrak{p}} \mathsf{cpk}^{-c}$
    return $c = \textsc{Hash}(\mathsf{com}, \mathsf{cpk}, \mathsf{nonce})$
```

Algorithm 5 Non-revocation proof generation and verification algorithms

```
procedure $\textsc{NRProve}(\sigma, \mathsf{csk}, \mathsf{sk_{nym}}, \mathsf{epoch})$
    $(r', s', t'_1, t'_2) \xleftarrow{\$} \textsc{Groth.Randomize}(\sigma)$
    $\langle \rho \rangle_{1\ldots4} \xleftarrow{\$} \mathbb{Z}_q$
    $\mathsf{com}_1 := e\left(r', g_2^{\rho_1}\right) \cdot e\left(g_1^{-1}, g_2^{\rho_2}\right)$
    $\mathsf{com}_2 := e\left(r', g_2^{\rho_3}\right)$
    $\mathsf{com}_3 := g_1^{\rho_2} h^{\rho_4}$
    $c := \textsc{Hash}(r', s', \mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{epoch})$
    $\mathfrak{p}_1 := g_2^{\rho_1} {t'_1}^{c}$
    $\mathfrak{p}_2 := \rho_2 + \mathsf{csk} \cdot c$
    $\mathfrak{p}_3 := g_2^{\rho_3} {t'_2}^{c}$
    $\mathfrak{p}_4 := \rho_4 + \mathsf{sk_{nym}} \cdot c$
    return $c, \langle \mathfrak{p} \rangle_{1\ldots4}, r', s'$
procedure $\textsc{NRSign}(\mathsf{rsk}, \mathsf{cpk}, \mathsf{epoch})$
    $\varepsilon := \textsc{Hash}(\mathsf{epoch})$
    return $\textsc{Groth.Sign}(\mathsf{rsk}; \mathsf{cpk}, g^{\varepsilon})$
procedure $\textsc{NRVerify}(c, \langle \mathfrak{p} \rangle_{1\ldots4}, r', s', \mathsf{pk_{nym}}, \mathsf{epoch})$
    if $e(r', s') \neq e(g_1, y_1) \cdot e(\mathsf{rpk}, g_2)$ then
        return false
    $\varepsilon := \textsc{Hash}(\mathsf{epoch})$
    $\mathsf{com}_1 := e(r', \mathfrak{p}_1) \cdot e\left(g_1^{-1}, g_2\right)^{\mathfrak{p}_2} \cdot e(\mathsf{rpk}, y_1)^{-c}$
    $\mathsf{com}_2 := e(r', \mathfrak{p}_3) \cdot e(\mathsf{rpk}, y_2)^{-c} \cdot e(g_1, g_2^{\varepsilon})^{-c}$
    $\mathsf{com}_3 := g_1^{\mathfrak{p}_2} h^{\mathfrak{p}_4} \mathsf{pk_{nym}}^{-c}$
    $c' := \textsc{Hash}(r', s', \mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{epoch})$
    return $c = c'$
```

Algorithm 6 Auditing proof generation and verification algorithms

```
procedure $\textsc{AuditProve}(\mathsf{enc}, \rho, \mathsf{cpk}, \mathsf{csk}, \mathsf{pk_{nym}}, \mathsf{sk_{nym}})$
    $\langle \rho \rangle_{1\ldots3} \xleftarrow{\$} \mathbb{Z}_q$
    $\mathsf{com}_1 := g^{\rho_1} \mathsf{apk}^{\rho_2}$
    $\mathsf{com}_2 := g^{\rho_2}$
    $\mathsf{com}_3 := g^{\rho_1} h^{\rho_3}$
    $c := \textsc{Hash}(\mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{enc}, \mathsf{pk_{nym}})$
    $\mathfrak{p}_1 := \rho_1 + c \cdot \mathsf{csk}$
    $\mathfrak{p}_2 := \rho_2 + c \cdot \rho$
    $\mathfrak{p}_3 := \rho_3 + c \cdot \mathsf{sk_{nym}}$
    return $c, \langle \mathfrak{p} \rangle_{1\ldots3}$
procedure $\textsc{AuditEnc}(\mathsf{apk}, \mathsf{cpk})$                ▷ ElGamal
    $\rho \xleftarrow{\$} \mathbb{Z}_q$
    $\mathsf{enc} := (\mathsf{enc}_1, \mathsf{enc}_2) := (\mathsf{cpk} \cdot \mathsf{apk}^{\rho}, g^{\rho})$
    return $\mathsf{enc}, \rho$
procedure $\textsc{AuditVerify}(c, \mathsf{enc}, \langle \mathfrak{p} \rangle_{1\ldots3}, \mathsf{pk_{nym}})$
    $\mathsf{com}_1 := g^{\mathfrak{p}_1} \mathsf{apk}^{\mathfrak{p}_2} \mathsf{enc}_1^{-c}$
    $\mathsf{com}_2 := g^{\mathfrak{p}_2} \mathsf{enc}_2^{-c}$
    $\mathsf{com}_3 := g^{\mathfrak{p}_1} h^{\mathfrak{p}_3} \mathsf{pk_{nym}}^{-c}$
    $c' := \textsc{Hash}(\mathsf{com}_1, \mathsf{com}_2, \mathsf{com}_3, \mathsf{enc}, \mathsf{pk_{nym}})$
    return $c = c'$
```