AuxChannel: Enabling Efficient Bi-Directional Channel for Scriptless Blockchains

Zhimei Sui  
Monash University  
Melbourne, Australia  
zhimei.sui1@monash.edu

Joseph K. Liu  
Monash University  
Melbourne, Australia  
joseph.liu@monash.edu

Jiangshan Yu  
Monash University  
Melbourne, Australia  
jiangshan.yu@monash.edu

Man Ho Au  
The University of Hong Kong  
Hong Kong, China  
allenau@cs.hku.hk

Jia Liu  
Fetch.ai  
UK  
jia.liu@fetch.ai

ABSTRACT

Payment channels have been a promising solution to blockchain scalability. While payment channels for script-empowered blockchains (such as Bitcoin and Ethereum) have been well studied, developing payment channels for scriptless blockchains (such as Monero) is considered challenging. In particular, enabling bi-directional payment on scriptless blockchains remains an open challenge.

This work closes this gap by providing AuxChannel, the first bi-directional payment channel protocol for scriptless blockchains, meaning that building payment channels only requires the support of verifiably encrypted signature (aka adaptor signature) on the underlying blockchain. AuxChannel leverages verifiably encrypted signature to create a commitment for each off-chain payment and deploys a verifiable decentralised key escrow service to resolve dispute. To enable efficient construction of AuxChannel, we introduce a new cryptographic primitive, named Consecutive Verifiably Encrypted Signature (CVES), as a core building block and it can also be of independent interest for other applications. We provide and implement a provably secure instantiation on Schnorr-based CVES. We also provide a formal security analysis on the security of the proposed AuxChannel.

KEYWORDS

Scriptless Blockchain, Payment channel, Scalability

1 INTRODUCTION

Payment channel [21] is a promising solution to improve the limited throughput of blockchain (in particular, Proof-of-Work blockchain such as Bitcoin). It allows users to make multiple transactions without committing all of them on-chain. In a typical payment channel, only two transactions, namely the first one representing the initial state and second one representing the final state of all transactions in between, are added to the blockchain while a large number of intermediate transactions can be executed between the participants without needing to be recorded on-chain. This greatly improves the blockchain throughput as only two transactions (representing a large number of transactions) need to go through the slow and expensive process of the distributed blockchain consensus [29].

Payment channels can be uni-directional or bi-directional. A uni-directional payment channel only enables one party to pay the other within a limited lifespan, whereas a bi-directional payment channel allows duplex payments. Thus, the uni-directional payment channel will be closed if the payer’s balance is insufficient to make a new payment. In contrast, bi-directional payment channels are more flexible, as it allows parties to pay each other and rebalance the channel.

Deploying either type of payment channels requires the underlying blockchain to support script. For unidirectional payment channel, the script is mainly used to unlock the payer’s fund when the payee is not responding.

In particular, with unidirectional payment channel, as a payer performs incremental payment to a payee (by creating a transaction signed by the payer), the payee always has an incentive to close the channel with the latest state (where the payee has largest balance) by cross-signing the received payment. However, it is possible that the payee is not responding the request to close the channel, for example when the payee has never received any payment. In this case, the payee’s balance is locked. To resolve this issue, a timelock function is desired to unlock the payer’s deposit. Time-lock function (such as hashed time lock [30]) defines a timer, such that upon timeout the payer will get full refund of its channel balance unless the payee has reacted to it by closing the channel before timeout.

However, time lock is not sufficient to resolve dispute in bidirectional payment channels. Suppose Alice and Bob are establishing a bidirectional payment channel together. Initially, both parties deposit 5 coins in the channel respectively. After several sessions of payments, Alice has 10 coins and Bob has 0 coin. To close the channel, the final state (Alice has 10 coins and Bob has nothing) should be posted on chain. Two misbehaviours could happen: (1) if closing a channel requires the permission from both parties, Bob can refuse to close the channel, as he loses nothing but Alice would lose all the money; and (2) if it only requires a single side permission, Bob has the motivation to close the channel by using a revoked state (a state is revoked by another successful channel update triggered by a transaction within the channel), where Bob has more coins than his final balance at the closing state. A script contract to specify dispute policy is required to resolve the above cases.

Nevertheless, some blockchains do not support script language, and therefore cannot implement such dispute mechanism. We call such blockchains scriptless blockchains. Deploying a bi-directional payment channel on scriptless blockchains, such as Monero and ZCash [43], remains an open challenge.
Our Contributions. Our papers contains the following merits:

1. We propose AuxChannel in enabling efficient bi-directional payment channel for scriptless blockchains with unlimited lifespan. AuxChannel guarantees that neither parties can prevent the channel from closing or steal coins from the counterpart. We facilitate this by a secret release mechanism on a script-enabled blockchain. Namely, both parties commit their secrets to the script-enabled blockchain. When a party is not following the protocol, the counterpart can request the script-enabled blockchain to obtain the other’s secret to recover the full signature for validating the payment. This naive approach is still not practical, as it not only incurs $O(n)$ (where $n$ is the number of off-chain transactions) communication complexity with the script-enabled blockchain, but also requires both parties to update their secrets for each new payment through the smart contract on the script-enabled blockchain, thus making it expensive, slow, and non-scalable. We reduce the communication complexity to $O(1)$ and does not need the on-chain update for each payment by using novel cryptographic techniques. In AuxChannel, both parties only need to commit their initial secrets on the script-enabled blockchain. Later if a party is not following the protocol, the counterpart can request for the initial secret and self-derive the latest state to recover the full signature and thus validate the payment and close the channel.

2. In order to construct AuxChannel, we further propose a new cryptographic primitive called Consecutive Verifiably Encrypted Signature (CVES) (as known as Consecutive Adaptor Signature). CVES allows a signer to encrypt his signatures by using a sequence of his encryption keys. These encrypted signatures are verifiable, and more importantly the encryption-decryption key-pairs are consecutive, which means the latest key-pairs are derived from their previous session (consecutive). A verifier can ensure both the correctness of the encrypted signature and the consecutiveness of signer’s encryption keys. The key generation function is one-way, meaning that the previous key-pairs can be used to generate the next one but not the opposite flow. (We remark that CVES can be also regarded as the consecutive version of adaptor signature [18], in which a new “adapted secret” can be generated in a verifiable and consecutive way from its ancestor.) CVES is not just the core building block of AuxChannel but we believe it is also of independent interest for further research.

2 RELATED WORK

This work focuses on the payment channel scheme, and introduces a new bi-directional payment channel protocol for scriptless blockchains. We then summarize some works related to payment channel protocols below.

Existing payment channel protocols for scriptless blockchain. While providing bi-directional payment channels on scriptless blockchains remains unsolved, ways to provide unidirectional payment channel for scriptless blockchain have been explored.

Currently there are three attempts to provide unidirectional payment channel for scriptless blockchain, namely the use of time proof (e.g. DLSAG [28] and Z-Channel [44]) and timed commitment (e.g. PayMo [38]), as all of them provide the time-lock function without requiring the support of script language. Both DLSAG channel for Monero and Z-Channel for Zerocash deploy time proof, which is a verifiable commitment as a predefined timelock. It is contained in the input of a transaction. The output of the transaction can only be spent after the predefined time proof. To avoid repetition, we use DLSAG channel as an example to specify how the time commitment works in a payment channel.

DLSAG channel makes it possible for Monero to support payment channels, while it has the limited lifespan which means a channel has to be closed within a predefined time. This limits the number of transactions that can be processed in the channel. Further, it only supports one-way payment. This may exhaust the balance in the channel quickly (even before the predefined channel lifespan) and the channel parties will have to close and re-establish a channel between them. Also, this solution requires an update on the mining software, which means a hard fork in the blockchain is needed to support time-proof verification. Thus it makes it difficult for being adopted in Monero. Even for that, it is only designed specifically for Monero, but not other scriptless blockchains.

Another approach is timed commitment. PayMo uses timed commitment to provide payment channel for Monero users. A user creates a timed commitment transaction such that the receiver of the commitment can force the opening of the commitment and learn the signature only after a pre-specified time. PayMo has extra computation requirement for opening the timed commitment, while it can be mitigated by a third party service [36]. Thus PayMo is fully compatible with the transaction in Monero. However, it still has limited lifespan and supports one-way payment only.

Existing bi-directional payment channel protocols for blockchains with limited scripts. Bolt [20], a bi-directional payment channel protocol, focuses on solving the linkability issues in channels and can be built on ZCash. Deploying Bolt requires that the zero-knowledge proof can be verified on the underlying blockchain, which is not compatible with other scriptless blockchains, such as Monero. A recent work from Aumayr et al. [2] formalizes the generalized payment channel for Blockchains with limited scripts, which requires the underlying blockchain to support adaptor signature scheme, relative time-lock and constant number of Boolean operations. Another recent work, Sleepy Channels [4], supports bi-directional payment channel with limited life-span, and also have the requirement of supporting time-lock on-chain. In addition, running Sleepy Channels between untrusted parties requires extra collateral, which could be equal to the channel capacity for each channel party, to guaranteed the secure and fast channel closure. While our work focuses on building bi-directional payment channels with unlimited lifespan for scriptless blockchains, such that building payment channels only requires the underlying blockchain to support verifiably encrypted signature (aka adaptor signature).

Other related works. In 2013, Satoshi Nakamoto, the author and creator of Bitcoin whitepaper and project, described a high-frequency transactions technique in a personal email [32]. Participants can close the channel with any intermediate state and double-spend the off-chain funding input. According to its design, this
technique only works on a strong assumption that all participants follow the protocol honestly. This idea is considered as the prototype of payment channel schemes. This is the first off-chain transaction idea on blockchain with obvious disadvantages, which has no punishment and on-chain deposit designed. At the same year, Spilman-styled payment channel [34] was proposed to fill the insecure design in Satoshi’s high-frequency transactions technique idea. This protocol makes a secure two-party off-chain trades without the honest players assumption. While this protocol only allows the uni-directional coin transfer with limited channel lifetime, this design increases the cost of creating a payment channel and is not desirable in real payment scenarios. Duplex Channel [40] is a simple attempt of bi-directional payment channel, which is consisted of two opposite directional Spilman-styled payment channels between two channel parties. This is not only quite straightforward, but also inherits the design of limited channel lifetime. While the direction of designing a bi-directional payment channel protocol is worth to do a depth-in research. In 2016, Poon-Dryja channel [30] realizes untrusted bi-directional payment channel without limited lifetime on Bitcoin. The well-known payment channel network project, Lightning Network, is implemented based on the Poon-Dryja channel protocol. The core technique of Poon-Dryja channel, Hash-Time Lock Contract (HTLC), is widely adopted in many protocols. HTLC is a useful technique among on-chain, off-chain and cross-chain applications, and requires a programmable language supporting by its underlying blockchain. After Poon-Dryja channel, there are two worth mentioned payment channel protocols, Eltoo [15] and Generalized Bitcoin-Compatible Channels [2]. These two protocols are also based on the HTLC technique.

Atomic swap is one of the cross-chain coin swap protocols, which also adopts HTLC technique as its core building block. According to its requirements of programmable language, atomic swap has the same issues for privacy-preserving blockchains. BTC-XMR atomic swaps [22] protocol enables the coins transfer between a script-enabled blockchain and a scriptless blockchain. It requires time-lock function only on Bitcoin side. Another recent cross-chain coins swap protocol for scriptless blockchain, JugglingSwap [33], introduces a trusted third party to control the time of secret releasing. Comparing to this work, AuxChannel does not require the involvement of the trusted third party for each off-chain payment by using the CVES scheme.

The anonymous multi-hop locks (AMHLs) [24], which aims to lock several atomic transactions among several users in a payment path, is similar to the our proposed CVES. To derive all the decryption keys, AMHLs requires all the on-path users’ participation. Also, the number of decryption keys is fixed at the setup phase. Thus, under the bi-directional payment channel scenario, AMHLs cannot prevent a malicious participant from locking the channel and also suffers from a limited life cycle. The functionality of AMHLs do not satisfy the requirements of AuxChannel, while CVES does.

3 PRELIMINARY
This section gives the definitions of Verifiable Encrypted Signature and One-time Verifiable Encrypted Signature, which are the two preceding concepts of our new primitive Consecutive Verifiable Encrypted Signature.

3.1 Verifiably Encrypted Signature
Verifiably encrypted signature (VES) [9, 13] scheme allows a signer’s signature to be encrypted using a third party’s encryption key (the third party is also known as adjudicator), and the resulting ciphertext can be publicly verified that it is really an encryption of the signer’s signature.

Note that VES is different from signcryption (sign-and-encrypt), though they share a similar name. In VES scheme, a signature is encrypted under the signer’s encryption key or a third party’s public key. We refer the third party as the adjudicator, which follows the term adopted in Boneh’s paper [9]. The encrypted signature can be verified by others, and the corresponding signature can only be revealed by the signer or the trusted third party. A similar cryptographic primitive is signcryption, where a signer encrypts its signature under the receiver’s public key. However, differ from VES scheme, only the receiver can decrypt and verify this encrypted signature.

An original VES scheme is defined with an ordinary underlying signature scheme (Gen, Sign, Verify) and four additional algorithms: EncGen is an algorithm to generate a (public key encryption) key pair for the adjudicator; EncSign is an algorithm for generating an encrypted signature from a message, the encryption key of the adjudicator and the signing key of the signer; EncVerify is an algorithm for convincing the verifier that the encrypted signature of a message is valid by using the verification key of the signer and the encryption key of the adjudicator; DecSig is an algorithm for recovering the encrypted signature by using the decryption key of the adjudicator. The formal definition of the verifiably encrypted signature scheme is given as follows.

Definition 1. (Verifiably Encrypted Signature Scheme). A verifiably encrypted signature (VES) scheme is defined with an ordinary underlying signature scheme (Gen, Sign, Verify) and four additional algorithms EncGen, EncSign, EncVerify, and DecSig:

- Gen(λ) → (sk, pk): a probabilistic key generation algorithm which takes the input security parameter λ and outputs a key pair (sk, pk), where sk is the signing key and pk is its corresponding verification key.
- Sign(sk, m) → σ: a probabilistic signing algorithm that, given a signing key sk and a message m, produces a signature σ.
- Verify(pk, σ, m) → {0, 1}: a deterministic verification algorithm that given a verification key pk, a signature σ and a message m, and returns 1 or 0.
- EncGen(λ) → (dk, ek): a probabilistic encryption key generation algorithm for adjudicator with an input, the security parameter λ, and outputs a valid key pair (dk, ek), such that dk is the decryption key and ek is the corresponding encryption key.
- EncSign(sk, ek, m) → ˆσ: a probabilistic encrypted signing algorithm, which on the input of the signer’s signing key sk, adjudicator’s encryption key ek and a message m, outputs an encrypted signature ˆσ.
- EncVerify(pk, ek, ˆσ, m) → {0, 1}: a deterministic verification algorithm that given the signer’s verification key pk, adjudicator’s encryption key ek, an encrypted signature ˆσ and a message m, returns 1 or 0.
• DecSig(\(dk, \hat{\sigma}\)) \(\rightarrow\) \(\{\sigma, \bot\}\): a deterministic signature decryption algorithm that given the adjudicator’s decryption key \(dk\) and an encrypted signature \(\hat{\sigma}\), outputs a decrypted signature \(\sigma\) or \(\bot\). This process is also known as adjudication.

VES should satisfy three security properties: validity, unforgeability, and opacity, which are described as follows:

• Validity: It requires that for any message, if the encrypted signature is generated by the EncSig, it should pass through the verification of EncVerify. It also requires that for any message, if the encrypted signature generated by the EncSig is decrypted by DecSig, the resulting signature should pass through the verification of Verify.

• Unforgeability: It is hard to forge a valid encrypted signature for any message from the verification key and encryption key only.

• Opacity: The signature for a message cannot be recovered from an encrypted signature without the decryption key.

3.2 One-Time Verifiable Encrypted Signature

The one-time verifiably encrypted signature (one-time VES), as known as Adapater Signature, is formalized by Fournier [18]. The main difference for one-time VES over normal VES is that the decryption key of one-time VES can be recovered once the original signature is revealed. Therefore the encryption-decryption key pairs can be used once (and thus called “one-time” VES). We also remark that in many context, one-time VES is more known as adaptor signature (e.g. [3, 17, 31, 35]). Although the name of the signature (and the names of the algorithms) are different, they are exactly the same. We use the notion of one-time VES in the rest of this paper, as this name reflects more the nature of its properties in our context.

One-time VES scheme is defined under the basis of the original VES scheme by adding two algorithms: RecKey and Rec. RecKey is an algorithm for extracting the “recovery” key from the encrypted signature, which is later used in Rec to recover the decryption key from the decrypted signature. Here, we only give two additional algorithms, while referring readers to Section 3.1 for the rest of the other 7 algorithms which are exactly the same as VES:

Definition 2. (One-time Verifiably Encrypted Signature). A one-time verifiably encrypted signature (one-time VES) scheme is a VES scheme (Gen, Sign, Verify, EncGen, EncSig, DecSig, RecGen, RecKey, Rec) with two additional algorithms RecKey and Rec:

• RecKey(ek, \(\hat{\sigma}\)) \(\rightarrow\) rk: a deterministic recovery key extraction algorithm, which given the encryption key ek and an encrypted signature \(\hat{\sigma}\) returns a recovery key rk.

• Rec(\(\sigma\), rk) \(\rightarrow\) dk: a deterministic decryption key recovery algorithm, which given a signature \(\sigma\) and the recovery key rk returns the decryption key dk.

The one-time VES scheme has the security properties, validity, unforgeability and recoverability. The first two are the same as VES and we omit here. Informally speaking, recoverability requires that it is easy to recover a decryption key by knowing the encrypted signature and its original signature.

4 CONSECUTIVE VERIFIABLY ENCRYPTED SIGNATURE (A.K.A. CONSECUTIVE ADAPTOR SIGNATURE)

4.1 Introduction of CVES

We propose a new cryptographic primitive, called Consecutive Verifiably Encrypted Signature (CVES) (a.k.a Consecutive Adapater Signature), which is a generalization of one-time VES such that CVES enables a sequence of one-time encryption-decryption key pairs. Given the initial decryption key, CVES is able to generate the next encryption-decryption key pair and so on. But this process cannot be reverted. That is, given the most updated decryption key only, no one can compute its “ancestors”. More importantly, CVES allows anyone to publicly verify that the new encryption key is correctly generated by its immediate ancestor, from the proof given by the owner of the corresponding decryption key.

CVES inherits all the algorithms defined in one-time VES and introduces three additional algorithms: KeyUpdate generates a new encryption-decryption key-pair by taking the previous decryption key; ConsProof generates a proof from the knowledge of two consecutive decryption keys which can convince any verifier that the corresponding encryption keys are consecutive (that is, the new encryption key is well-formed or generated from KeyUpdate); ConsVerify verifies the proof generated by ConsProof.

We now present the formal definition of CVES scheme. We use \((dk^0, ek^0)\) to denote the initiate decryption-encryption key pair, while \((dk^1, ek^1)\) to denote the \(i\)-th key pair, after executing KeyUpdate \(i\) times for any \(i > 0\). We also use KeyUpdate\(^i\) to denote recursively executing KeyUpdate \(i\) times.

Definition 3 (Consecutive Verifiably Encrypted Signature). A consecutive verifiably encrypted signature (CVES) scheme is a one-time VES scheme (Gen, Sign, Verify, EncGen, EncSig, EncVerify, DecSig, RecKey, Rec) with three additional algorithms KeyUpdate, ConsProof and ConsVerify:

• KeyUpdate(dk\(^{i-1}\)) \(\rightarrow\) \((dk^i, \hat{dk}^i, ek^i)\): a deterministic algorithm with an input, a decryption key \(dk^{i-1}\) and returns its successive decryption key \(dk^i\), a truncated decryption key \(\hat{dk}^i\) and the corresponding encryption key \(ek^i\).

• ConsProof(dk\(^{i-1}\), \(\hat{dk}^i\)) \(\rightarrow\) \(P^i\): a probabilistic algorithm, which on inputs two decryption keys \(dk^{i-1}\) and \(dk^i\), outputs a proof \(P^i\).

• ConsVerify(ek\(^{i-1}\), ek\(^i\), \(P^i\)) \(\rightarrow\) \(\{0, 1\}\): a deterministic algorithm, which on inputs two consecutive encryption keys \(ek^{i-1}\), \(ek^i\) and a proof of consecutiveness \(P^i\), outputs 0 or 1.

We also note that there are some minor differences with one-time VES for the other algorithms. In CVES, we only require Rec to output the truncated decryption key \(dk^i\) instead of the full decryption key \(dk^i\), as in one-time VES. Furthermore, EncGen is executed only once to generate the initial decryption-encryption key-pair \((dk^0, \hat{dk}^0, ek^0)\) including the truncated decryption key \(dk^0\), and the subsequent decryption-encryption key-pairs can be generated by executing KeyUpdate.
4.2 Security of CVES

CVES scheme inherits all the secure properties from one-time VES scheme, i.e., validity, unforgeability, recoverability, along with three additional properties, consecutiveness, consecutive verifiability and one-wayness. The informal definitions of the first three have been given in the previous section, and the later three definitions are described informally as follows:

- **Consecutiveness**: It requires that the decryption-encryption key-pair used in the \( i \)-th session of CVES is derived from the decryption key of the \((i-1)\)-th session from KeyUpdate, where \( i > 0 \).
- **Consecutive verifiability**: Given two encryption keys and the corresponding proof, anyone can be convinced if the two keys are consecutive, meaning that the new encryption key is generated from KeyUpdate taking the input of the previous corresponding decryption key.
- **One-wayness**: given the \( i \)-th decryption key, no one can derive any of the \( j \)-th decryption key, where \( 0 \leq j < i \).

We then give the formal definition of CVES security properties. We first introduce the concept of validity, recoverability and consecutiveness into a single definition of correctness, as defined below:

**Definition 4** (Correctness of CVES). A CVES is correct if the following properties hold. For any \( i > 0 \) and any message \( m^i \), we require

\[
\begin{align*}
(sk, pk) &\leftarrow Gen(\lambda) \\
(dk^0, \hat{dk}^0, ek^0) &\leftarrow EncGen(\lambda) \\
(dk^{i-1}, \hat{dk}^{i-1}, ek^{i-1}) &\leftarrow KeyUpdate^{i-1}(dk^0) \\
(dk^i, \hat{dk}^i, ek^i) &\leftarrow KeyUpdate(dk^{i-1}) \\
\sigma^i &\leftarrow Sign(sk, m^i) \\
\hat{\sigma}^i &\leftarrow EncSign(sk, ek^i, m^i) \\
P_i &\leftarrow ConsProof(dk^{i-1}, dk^i) \\
1 &\leftarrow ConsVerify(ek^{i-1}, ek^i, P_i) \\
1 &\leftarrow EncVerify(pk, ek^i, \hat{\sigma}^i, m^i) \\
\hat{dk}^i &\leftarrow Rec(\sigma^i, RecKey(\hat{\sigma}^i)) \\
1 &\leftarrow Verify(pk, DecSig(\hat{dk}^i, \hat{\sigma}^i), m^i)
\end{align*}
\]

Next, before giving the definition of unforgeability, we first introduce two oracles below, which can be accessed by adversaries in the unforgeability experiment.

**Signing Oracle**: Let \( O \) be a signing oracle, which takes the verification key \( pk \) and the message \( m \) as inputs, and outputs a signature \( \sigma \) such that \( 1 \leftarrow Verify(pk, \sigma, m) \).

**Encrypting Oracle**: Let \( E \) be an encrypting oracle which takes the verification key \( pk \), the encryption key \( ek^i \) for session \( i \) and a message \( m \) as inputs, and outputs an encrypted signature \( \hat{\sigma}^i \) such that \( 1 \leftarrow EncVerify(pk, ek^i, \hat{\sigma}^i, m) \).

**Definition 5** (Unforgeability of CVES). A CVES is unforgeable, if the advantage \( Adv = Pr[Exp_{A}^{\text{unforgeability}} = 1] \), where \( Exp_{A}^{\text{unforgeability}} \) is defined in Figure 1, of any PPT adversary in forging a signature \( \sigma^i \) given access to signing oracle \( O \) and encrypting oracle \( E \) is negligible. The probability is taken over the coin tosses of the key generation algorithm \( Gen \) and \( EncGen \), the oracles \( O \) and \( E \). The adversary is additionally constrained that \( m^i \neq m \), where \( m \) is an input to \( E \) and \( O \) accessed by the adversary.

**Definition 6** (One-wayness of CVES). A CVES is one-way, if the advantage \( Adv = Pr[Exp_{A}^{\text{one-way}} = 1] \), where \( Exp_{A}^{\text{one-way}} \) is defined in Figure 1, of any PPT adversary in recovering \( dk^{i-1} \) from \( dk^i \) is negligible. The probability is taken over the coin tosses to the reversion algorithm of the adversary.

**Definition 7** (Consecutive Verifiability). A CVES is consecutive verifiable, if the advantage \( Adv = Pr[Exp_{A}^{\text{verifiability}} = 1] \), where \( Exp_{A}^{\text{verifiability}} \) is defined in Figure 1, of any PPT adversary in yielding a proof of consecutiveness on two unconsecutive decryption keys is negligible. The probability is taken over the coin tosses of the ConsProof algorithm and of the adversary.

The opacity property defined in VES guarantees that a decrypted signature is unforgeable. Similar to one-time VES, CVES does not have the opacity property as well, as the recoverability property makes it possible to recover the \( i \)-th session decryption key once the \( j \)-th session decryption key is revealed, where \( i > j \geq 0 \).

A CVES is secure, if it satisfies the correctness, unforgeability, one-wayness, consecutive verifiability defined in Definition 4, 5, 6, 7.

4.3 Schnorr CVES Construction

**Overview Idea**: Now we present a Schnorr-based CVES construction. The overview idea is that. Our construction is motivated by the Schnorr-based one-time VES of [18]. The main difference is the consecutiveness of our construction, which requires two additional elements to achieve our purpose: 1) The design of a one-way function is to derive the next decryption key from the previous one; 2) The proof to prove the correctness of the key generated, such that the verification only takes the public information (encryption keys).

Since our system is based on the Schnorr ecosystem, which is in discrete logarithm (DL) setting, the first naive idea of constructing such a one-way function and the corresponding proof, is to use another DL proof system. Exponentiation function over a cyclic group is already a popular one-way function, while it also provides an efficient zero-knowledge proof for its input given the output. Unfortunately, in our case it cannot be adopted in an efficient way. The main reason is the difference of the domains of the input and output of the exponentiation function. The input domain is usually an integer, while the output domain is a group element (which is not necessarily to be an integer). In our setting, we require the domain of the input and output for the one-way function to be the same. This is necessary because the one-way function is used to generate the decryption key of the next session (from the decryption key of the previous session), which obviously should lie within the same domain of the previous decryption key. Although there are some techniques to convert the output of the exponentiation function to the same domain of the input element, they are not efficient when a proof is also required.

To be exact, we need a one-way permutation with the ability to construct an efficient proof for our scheme. Instead of using an exponentiation function, we in turn use a square function over
with length $\lambda_1$ which is another security parameter. The order of the group $Z'_N$ is with unknown order $\phi(n)$ (again we can use MPC to generate $n$ such that no one knows its factorization). The one-way function can be set as $x \mapsto x^2 \mod n$.

The complete construction of Schnorr-based CVES scheme is shown in Figure 2. (The detailed explanation and underlying instantiation of the NIZK in the algorithms ConsVerify and ConsProof is given in Appendix B.)

We also remark that $N$ can be used as a system parameter (e.g. used by a group of users) while $n$ is used between the prover and the verifier. In practice, we simply set $N = n$, if this number is generated using MPC or other trusted method so that no one knows its factorization. We then give the intuition of why we can set $N = n$: According to the algorithm KeyUpdate in Figure 2, the public key used in zero-knowledge proof is constructed as $Y' = h^y'$ mod $N$, where $y' \equiv y^2 \mod n$. Thus, the distribution of $Y'$, where $y' \in [0, n)$, would affect the distribution of $Y'$. Ideally, we hope $Y'$ is uniformly distributed from $Z'_N$, which can be guaranteed when $n = \phi(N)$, where $\phi(N)$ is the group order of $Z'_N$. As $\phi(N)$ is unknown, to ensure that $(h^y \mod N)$ can cover all group elements in $Z'_N$, $n$ should be greater than $\phi(N)$. Thus if we set $n = N > \phi(N)$, and when $y'$ is from $[0, n)$, and $n = N$ is large enough, $Y' = (h^y \mod N)$ is statistically indistinguishable from $Y''$, which is uniformly in $Z'_N$.

**Theorem 1.** The Schnorr CVES is correct, and satisfies the unforgeability, one-wayness, and consecutive verifiability properties, if the DL problem and factorization problem are hard, and the underlying NIZK system is correct and sound.

The proof is given in Appendix A.

### 4.4 Performance Evaluation of Schnorr CVES

We have instantiated and implemented the algorithms KeyUpdate, ConsVerify and ConsProof (in Figure 2). The detailed explanation and underlying instantiation of the NIZK in the algorithms ConsVerify and ConsProof are given in Appendix B.

Our implementation is based on a zero-knowledge library, emmy [41]. For simplicity, we set $n = N = p^q'$, and adopt an unknown group $Z'_N$ and a cyclic group $\mathbb{G}_p$ with the prime order $p$, where $|N| = 2048$ and $p = 2^{252} + 277423177737235353581937799883648943$.

Suppose a signer has generated an encryption-decryption key-pair $(ek^{i+1}, dk^{i+1})$ and shared $\hat{ek}^{i+1}$ with a verifier, where $i \in \mathbb{N}$. He then performs $(\hat{dk}^{i}, \hat{dk}^{i}, \hat{ek}^{i}) \leftarrow$ KeyUpdate$(\hat{dk}^{i-1})$ to updates the encryption and decryption keys. To convince the verifier that $\hat{dk}^{i-1}$ and $\hat{dk}^{i}$ are consecutive, the signer creates a proof $P^{i} \leftarrow$ ConsProof$(\hat{dk}^{i-1}, \hat{dk}^{i})$, and shares $P^{i}$ and $\hat{ek}^{i}$ with the verifier. Our experiments are run on a macOS with processor 2.6 GHz 6-Core Intel Core i7 and memory 16GB 2400 MHz DDR4. Each algorithm is executed for 100 times. The result shows that on average it requires about 33 ms to create the proof, and the corresponding verification.
We propose AuxChannel, a bi-directional payment channel protocol.

5 OVERVIEW OF AUXCHANNEL

We propose AuxChannel, a bi-directional payment channel protocol with unlimited life span for scriptless blockchains. We present the basic idea of AuxChannel with a step by step approach, before presenting it formally in Section 6.

5.1 Problem Statement

Recall the scenario we considered: two parties, Alice and Bob, establish a bi-directional payment channel over a scriptless blockchain. Initially, each of them deposits 5 coins in the channel. After several sessions of payments, Alice has 10 coins and Bob exhausted his balance. To close the channel, the final state (Alice has 10 coins and Bob has no coin) should be posted on the blockchain.

Depending on the rule of closing a channel, two misbehaviours could happen: (1) if closing a channel requires the permission from both parties, Bob can ignore the request as he loses nothing but Alice would lose all 10 coins; and (2) if it only requires a single side permission, Bob can close the channel by using a revoked state, where Bob can claim more than his actual balance. To protect a channel from being attacked by the two misbehaviours above, the protocol should provide two guarantees, guaranteed channel closing and guaranteed balance payout for channel parties [16].

Please note that the above challenge does not exist on unidirectional channels, where a payer makes incremental payment to a payee: allowing the payee to close a channel alone is sufficient as the payee is incentivised to always close the channel (i.e., guaranteed channel closing) with the latest actual balance (i.e., guaranteed balance payout).

As in other layer 2 solutions supporting bi-directional payments (such as Lightning Network [30]), the key to address the above challenges is to provide the ability to resolve dispute upon observing the misbehaviours. Resolving disputes on the blockchain is easy for blockchains supporting script language, but extremely challenging, if not impossible, for scriptless blockchains.

requires about 348 ms. The communication latency is transferring 16.9 KB data, including the required proof and encryption key ($P^{i+1}, e^{i+1}$).

Figure 2: The construction of Schnorr-based CVES algorithms.
5.2 Strawman Design

One natural step is to leverage a script-enabled blockchain (denoted \( L_2 \)) to help scriptless blockchain (denoted \( L_1 \)) to resolve any dispute.

First attempt. This leads to the first attempt of our design, where both channel parties lock enough funds on the script-enabled blockchain as "insurance". Upon the detection of any misbehavior, the victim can resolve the dispute by claiming all insurance on \( L_2 \).

As this only provides compensate to the victim on \( L_2 \), rather than correcting the misbehavior on \( L_1 \), the victim will lose its fund on \( L_2 \) (regardless of requiring permissions from one or both parties to close a channel) and the malicious attacker still gets the money on \( L_1 \) (if closing a channel only requires the permission from one party). Thus, to deincentivise such misbehaviors, each party should lock a fund as an insurance on \( L_2 \), such that the locked amount is no smaller than the channel capacity on \( L_1 \) to cover the maximum potential loss of a victim.

This approach has two issues. First, it requires (the smart contract of) \( L_2 \) to monitor all events on \( L_1 \) and retrieve evidence to resolve the dispute. While there are existing attempts to enable communication across blockchains [1], they introduce significant additional cost and with open challenges [42]. Second, it requires additional deposit on \( L_2 \) as insurance, where the total deposit is at least twice as large as the channel capacity. In other words, channel parties have to lock 3C to establish a channel with capacity C, which greatly reduces the liquidity and makes it impractical.

Second attempt. In our second attempt, rather than locking the deposit for covering the potential loss and deincentivise misbehaviors, we require both parties’ involvement to close a channel. In this way, closing a channel requires the permission from both parties, eliminating the possibility of claiming balances with a revoked state and providing guaranteed balance payout. In addition, we envision a distributed and verifiable "key escrow" service on \( L_2 \) to keep and release some secret (for example, by using [6]) required to close the channel.

In particular, each payment in the channel is created through a verifiable encrypted signature scheme, where the "key escrow" service is leveraged to keep a secret to recover the full signature for validating the payment. In the normal case, when both parties are honest, they can cooperate to close the channel with the latest state. In the case where one party is not responding to the request of closing a channel, the "key escrow" service can help releasing the secret to close the channel with latest channel state. Note that each encrypted signature should use a unique encryption key, as otherwise when an honest party releases the decryption key, the malicious counterparty can decrypt and validate any past revoked payment.

This approach still has some issues. While both parties only need to pay a small fee to the distributed and verifiable key escrow service (rather than locking additional significant amount of insurance), channel parties need to interact with the key escrow service for each payment. This not only incurs \( O(n) \) communication complexity with the key escrow service, but also requires the channel parties to update the secret for each new payment through the smart contract on \( L_2 \), making it expensive, slow, and non-scalable.

5.3 AuxChannel

We address the remaining challenges in our second attempt by leveraging our newly introduced CVES as a core building block. In particular, thanks to the consecutiveness property, both channel parties only need to provide the initial secret to the key escrow service (reducing the communication complexity to \( O(1) \)) when establishing a channel, processing all payments without interacting with the key escrow service, and only asking the key escrow service to release the secret to close the channel when one party is not cooperating.

For the ease of understanding, we only present the basic idea of AuxChannel below and leave the formal and detailed presentation to the next section. Loosely speaking, AuxChannel has four phases: channel establishment, channel update, channel closure, and channel dispute. For simplicity, we assume all the messages are correctly verified before being accepted.

In the channel establishment phase, each channel party executes EncGen() to generate a pair of decryption key and encryption key, and share the output with the distributed and verifiable key escrow service. The escrow service will include a commitment on the received secret in its smart contract with the two channel parties on \( L_2 \).

Each of the channel parties performs EncSign(·) to create an encrypted signature on a mutually agreed transaction, such that the full signature can be recovered by using the associated decryption key kept by the escrow service on \( L_2 \). To validate a transaction, one recovers the full signature on the transaction from the encrypted signature of its counterparty, and creates a mutually signed transaction by adding its own signature to the recovered full signature.

Each new payment redistributes the channel balance as agreed by both parties. Payments can be performed via channel update phase, where for each payment the payer updates the decryption and encryption key pair by performing KeyUpdate(·), and creates an encrypted signature on the new payment by performing EncSign(·) with the signing key and the updated encryption key.

To close a channel, channel parties share their latest decryption key with each other, perform DecSig(·) with the counterparty’s decryption key, and validate the transaction representing the latest state of the channel balance by creating a multi-signature on the transaction.

In the case where one party is not responding to the channel closure request, the other party can resolve the dispute by asking the key escrow service to release the secret required to close the channel. Thanks to the consecutiveness property of our introduced CVES, if the initial secret shared with the escrow service by one party is released, the other party can derive the latest secret recover the full signature to close the channel, presenting a neat and efficient solution to the guaranteed channel closing.

As a malicious party may request the secret to be released for a revoked state in order to get more coins than its actual balance, it is important for the key escrow service to give enough time for the counterparty to verify and react to the secret releasing request on \( L_2 \). This can be achieved by using standard solutions, such as a time-lock contract. We defer more details to the next section.
6 FORMAL AUXCHANNEL DESCRIPTION

6.1 System Model

We consider two blockchains, $L_1$, which does not support script language, and $L_2$, which is script-enabled. For simplicity, we assume that if a valid transaction is propagated to the blockchain network, it cannot be censored and will be immediately included in the "permanent" part of the blockchain.

Two participants want to establish a payment channel on $L_1$. Once a channel is established, the two participants become "channel parties". For simplicity, this paper always uses channel parties to refer to them. We assume that both channel parties have accounts with sufficient coins on $L_1$ and $L_2$ in participating AuxChannel. We assume that they are always online, even though this assumption can be removed by deploying watchtower-like services [5, 21, 26]. We assume a distributed and verifiable key escrow service (KES), such as [6], on $L_2$. The KES keeps secret keys for its clients and releases the key when a pre-defined condition is met. We assume that the communication channels among channel parties and the Key Escrow Service (KES) are authenticated e.g. through cryptographically secure digital signature.

6.2 Formalized AuxChannel

Let $\mathcal{P}_A$ and $\mathcal{P}_B$ be the two channel parties with signing-verification key-pairs $(sk_A, pk_A)$ and $(sk_B, pk_B)$ on $L_1$, and $(sk'_A, pk'_A)$ and $(sk'_B, pk'_B)$ on $L_2$, respectively, and they have exchanged public keys with each other. Let $Tx := In||Out$ be the construction of the transaction on $L_1$, where $In$ denotes the input of $Tx$ and $Out$ denotes the output of $Tx$. We present AuxChannel with four phases, namely establishment, update, close, and dispute.

Establishment To establish a channel, two channel parties collectively create a funding transaction $Tx_f$ to make their deposit on $L_1$. To initialise the establishment, each of the parties generates initial decryption keys $(\hat{d}_k^0, \hat{d}_k^1)$ and encryption key $ek^0_\mathcal{P}$ by executing $EncGen(\lambda)$, where $\mathcal{P} \in \{A, B\}$. They then interact with KES to share their $(cid, \hat{d}_k^0, \hat{d}_k^1)$, where $cid$ is a unique identifier of the channel. KES verifies the received messages and records $(cid, ek^0_\mathcal{P}, ek^1_\mathcal{P})$ on $L_2$ as a confirmation of providing key escrow service to both parties. The conditions to release decryption keys by the KES will be presented in the dispute phase. Upon seeing the confirmation from KES:

1. Both channel parties create a funding transaction $Tx_f := (pk_A : bal^0_A, pk_B : bal^0_B)||(pk_M : bal_M)$
   to transfer their balance $(bal^0_A$ of address $pk_A$) to a multi-signature account (identified by using $pk_M$), where the balance $bal_M$ is the capacity of the to be established channel. They also create an encrypted signature $\sigma_{pk} := EncSign(sk_A, ek^1_A, TX^0_c)$ over a newly created commitment transaction $TX^0_c := (pk_M : bal_M)||(pk_A : bal^1_A, pk_B : bal^1_B)$.

2. If the received $\hat{\sigma}_{pk}$ passed the verification $ConsVerify(pk_A, ek^0_A, \hat{\sigma}_{pk}, TX^0_c)$, they exchange locally signed $TX_f$ and post the cross-signed $TX_f$ on $L_1$ indicating the completion of channel establishment.

Update To update a channel from the channel state index $i - 1$ to $i$, where $i \in \mathbb{N}^+$, both channel parties agree on the commitment transaction $TX^i_c$ and their decryption keys $(ek^i_A, ek^i_B)$. The update phase is presented as follows:

1. Both channel parties update their decryption keys $(d_k^i, d_k^1)$ and encryption key $ek^i_\mathcal{P}$ by executing $KeyUpdate(d_k^{i-1})$. They also create the proof of encryption keys' consecutiveness $\sigma_i^\mathcal{P}$, an encrypted signature $\hat{\sigma}_{pk}$, and an newly created commitment transaction $TX^i_c := (pk_M : bal_M)||(pk_A : bal^1_A, pk_B : bal^1_B)$, and exchange message $(ek^i_A, \hat{\sigma}_{pk}, P_B)$.

2. If the received $(ek^i_A, \hat{\sigma}_{pk}, P_B)$ passed the two verifications $EncVerify(pk_A, ek^0_A, \hat{\sigma}_{pk}, TX^0_c)$ and $ConsVerify(ek^{i-1}_B, ek^i_B, \sigma_i^\mathcal{P})$, where $ek^{i-1}_B$ is received and stored from the counter-party at the previous state, they create and exchange a signature $\sigma_i^\mathcal{P}$ over the channel state index $i$ and both channel parties' decryption keys $(ek^i_A, ek^i_B)$ by using $sk_M$.

Close The channel can be closed collectively by the two parties. Both parties exchange their latest decryption keys, $d_k^i$ and $d_k^i$, and derive the corresponding signature $\sigma_i^\mathcal{P} := DecSign(d_k^i, \hat{\sigma}_{pk})$ on $TX^i_c$, where $\hat{d}_k_i$ is the truncated $d_k^i$ (defined in Section 4). Thus, either of them can close the channel with a latest state by posting a cross signed $TX^i_c$.

Dispute Both channel parties resolve their dispute through $L_2$ and the KES will release the secret per their predefined condition, as follows.

In the predefined condition, any channel party can request the KES to release the counter-party's initial decryption key by uploading a trigger transaction to $L_2$, which starts a timer for the counter party to react before timeout. Assuming that $\mathcal{P}_B$ triggers the timer by posting a trigger transaction $TX^i_{up,B}$, which includes the decryption key $d_k^i$, and the message $(\sigma_i^A, \sigma_i^B, i, ek^i_A, ek^i_B)$.

Before the timeout occurs, if the index $i$ represents the latest state of the channel, $\mathcal{P}_A$ releases her decryption key $d_k^i$ by posting a release transaction $TX^i_r,A$; otherwise, $\mathcal{P}_B$ uploads an update transaction $TX^i_{up,A}$, where $j > i$, and KES releases $d_k^0_A$ on $L_2$. Upon timeout, if $\mathcal{P}_A$ did not have any response, KES releases $d_k^0_A$.

If $\mathcal{P}_A$ has released her decryption key $d_k^i$, the timer, $\mathcal{P}_B$ can perform $DecSign(d_k^i, \hat{\sigma}_{pk})$ to derive the commitment transaction $TX^i_c$ signed by $\mathcal{P}_A$. Similarly, as $d_k^i$ is already posted on $L_2$, $\mathcal{P}_A$ can also derive the commitment transaction $TX^i_c$ signed by $\mathcal{P}_B$.

Thus, either of them can close the channel with the latest state by posting a cross signed $TX^i_c$.

If the KES has released $d_k^0_A$ (for $\mathcal{P} \in \{A, B\}$) on $L_2$ according to the conditions above, then the corresponding counter party can derive the decryption keys $(d_k^i, d_k^i)$ of any state $i' \in \{1, j\}$ through...
(dk_p^i, dk_e^i, ek_p^i) ← KeyUpdate(dk_p^{i-1}), and then derive the commitment transaction Tx_c^i signed by P. This enables the counter party to close the channel with any channel state. In other words, when a party is not following the protocol, the counter party is able to close the channel at any chosen state (to its advantage) with the help of KES. This is considered a punishment to the misbehaved party.

6.3 Security Analysis

Similar to Perun [16], we also employ the following security goals of AuxChannel:

1. Guaranteed channel closing: a channel can be closed by any of the channel parties;
2. Guaranteed balance payout for end users: when a channel is closed, either both channel parties retrieve their latest balances or the honest party obtains no less than his latest balance.

As AuxChannel employs Schnorr CVES scheme to update channel party’s decryption keys and encrypted signatures at each state, we reduce the security of AuxChannel to the security of CVES scheme in Theorem 2.

Theorem 2. AuxChannel achieves the two security goals if the underlying CVES scheme is secure.

We present the proof sketch of Theorem 2 as follows:

Proof. We prove it by contradiction. Let P_A be the adversary A, who has the ability to break the security goals of AuxChannel, we can make use of P_A to break the security of Schnorr-based CVES scheme.

For the first security goal: guaranteed channel closing. As we assume that P_A has the ability to prevent the channel from closing, P_A would not release any of her decryption keys. Thus, KES sends the initial decryption keys dk_A^0 to P_B. As P_A has the ability to prevent the channel from closing, the dk_A^0 cannot help P_B to recover a valid \( \sigma_A^{i'} \) at any intermediate state \( i' \), where \( i' \in [0, i] \). Thus, the channel can be closed without P_A's cooperation.

We can make use of P_A to generate a proof \( P_A^{i'+1} \) on two decryption keys dk_A^{i'} and dk_A^{i'+1}, where:

\[
(dk_A^{i'}, dk_A^{i'+1}, ek_A^{i'}) \leftarrow A(dk_A^{i'})
\]

\[
(dk_A^{i'+1}, dk_A^{i'+2}, ek_A^{i'+1}) \leftarrow A(dk_A^{i'+1})
\]

\[
\sigma_A^{i'+1} \leftarrow \text{EncSign}(sk_A, ek_A^{i'+1}, Tx_c^{i'+1})
\]

\[
P_A^{i'+1} \leftarrow A(dk_A^{i'+1}, \sigma_A^{i'+1})
\]

\[
1 \leftarrow \text{ConsVerify}(ek_A^{i'+1}, P_A^{i'+1})
\]

\[
\sigma_A^{i'+1} \leftarrow \text{DecSign}(dk_A^{i'+1}, \sigma_A^{i'+1})
\]

\[
1 \leftarrow \text{Verify}(pk_B, \sigma_B^{i'+1}, Tx_c^{i'+1})
\]

Or we can also make use of P_A to create a valid encrypted signature \( \sigma_A^{i'+1} \), but P_B cannot derive a valid \( \sigma_A^{i'+1} \) from \( \sigma_A^{i'+1} \), where:

\[
(dk_A^i, dk_A^{i'+1}, ek_A^i) \leftarrow \text{KeyUpdate}(dk_A^{i-1}),
\]

\[
P_A^{i'+1} \leftarrow \text{ConsProof}(dk_A^i, dk_A^{i'+1})
\]

\[
1 \leftarrow \text{ConsVerify}(ek_A^i, P_A^{i'+1})
\]

\[
\sigma_A^{i'+1} \leftarrow \text{DecSign}(dk_A^{i'+1}, \sigma_A^{i'+1})
\]

\[
1 \leftarrow \text{Verify}(pk_B, \sigma_B^{i'+1}, Tx_c^{i'+1})
\]

Thus, P_A breaks the consecutive verifiability and the correctness of Schnorr CVES scheme.

For the second security goal: guaranteed payout of channel parties. We assume that P_A has the highest balance at an intermediate channel state \( i' \), and she has the ability to close the channel at state \( i' \) with an non-negligible possibility. That is P_A can derive \( \sigma_B^{i'} \), and post a cross-signed \( Tx_c^{i'} \) on L1. Thus, she steals coins from P_B.

Thus, we can make use of P_A to forge a valid signature \( \sigma_B^{i'} \) by using \( pk_B, ek_B^{i'}, \hat{\sigma}_B^{i'} \), and the proof \( P_B^{i'} \), where:

\[
\sigma_B^{i'} \leftarrow \text{Sign}(pk_B, \sigma_B^{i'}, \hat{\sigma}_B^{i'}, P_B^{i'})
\]

\[
1 \leftarrow \text{Verify}(pk_B, \sigma_B^{i'}, Tx_c^{i'})
\]

We can also make use of P_A to reverse \( dk_B^{i'} \) from \( dk_B^{i} \), where:

\[
dk_B^{i'} \leftarrow A(dk_B^{i})
\]

\[
\sigma_B^{i'} \leftarrow \text{DecSign}(dk_B^{i'}, \sigma_B^{i'})
\]

\[
1 \leftarrow \text{Verify}(pk_B, \sigma_B^{i'}, Tx_c^{i'})
\]

No matter which scenario happens, P_A breaks unforgeability or one-wayness of Schnorr CVES scheme.

In summary, if one of the channel parties can break the security goal, either guaranteed channel closing or guaranteed payout of channel parties, of AuxChannel, it also breaks the security of CVES scheme, which is contradict to the assumptions. Theorem 2 is proved then.

\[\Box\]

7 DISCUSSION AND CONCLUSION

Payment channel networks (PCN) enable offchain payments between users that have not established a payment channel between them, by routing payments via a path of payment channels. This section provides an intuition on how to make a payment via multi-hop AuxChannel.

normally, when making a payment via multi-hop channels, two core requirements should be met. First, all payments in the path should be atomic, meaning that either they all succeed or they all failed. For Bitcoin-compatible blockchains, atomicity is usually guaranteed by using Hash-time Lock Contract (HTLC) [30], the coins locked in HTLC can only be released when some conditions are met. Second, all the on-path channels are unlockable even when the parties involved are corrupted or malicious [27, 39]. This is normally guaranteed by making penalty for the malicious [25], which is used for compensating parties who incurred loss by locking funds.
Assuming that Alice (A) has a channel with Bob (B), who has a channel with Carol (C), who has a channel with Dave (D), the path of these channels could be considered like: A ↔ B ↔ C ↔ D. The multi-hop payment from A to D requires that each intermediate channel, namely the channels between A and B, B and C, and C and D, to lock a payment. Such that once D unlocks the payment from C, C can unlock the payment from B, and B can unlock the payment from A. Thus, A obtains a witness as receipt of paying to D. Otherwise, B and C can unlock their payments (between (A and B) and (B and C)) after a pre-defined time, and A obtain the receipt as well. Under this scenario, A can still prove that she has paid to D by using the receipt, and D has motivation to unlock the payment from C, or he never get paid. Thus, all the intermediate payments are atomic and unbreakable.

Supporting payment channel network in AuxChannel requires addressing two challenges: first, how to lock an encrypted signature within a channel; second, for B and C, how to unlock the locked signature sequentially. For the channel between Dave and Carol: Alice sequentially. For the channel between Dave and Carol:

Assuming that Alice wants to pay x coins to Dave via the path of AuxChannel: A ↔ B ↔ C ↔ D. The multi-hop payment has two rounds. First, all intermediate channels are locked from Dave to Alice sequentially.

- Dave produces an unlocking-locking key pair \( (z_D, Z_D) \), where \( Z_D = g^{tD} \) and a timed commitment \( \text{Comm}_{D'} \) on \( Z_D \), and forwards \( (Z_D, \text{Comm}_D) \) to Carol, such that Carol can force open \( \text{Comm}_D \) and obtain \( Z_D \) after a pre-specified time \( t_D \).
- Dave and Carol make locked signatures \( \sigma^*_D \) and \( \sigma^*_C \) on their newest channel balances by using same locking key \( Z_D \) respectively.
- Dave and Carol make their signatures \( \sigma_D^\prime \) and \( \sigma_C^\prime \) on the message \( i \| \| e_k \| \| e_{k_D}^D \), where \( i \) is the newest channel state index, by using their signing keys on the script-enabled blockchain respectively, and exchange \( \text{Enc}_{Z_D}(\sigma_D^\prime) \) and \( \text{Enc}_{Z_C}(\sigma_C^\prime) \), where \( \text{Enc}_{Z_D}() \) can be any encryption algorithms w.r.t the encryption key \( Z_D \), such that \( \sigma_D^\prime = \text{Dec}_{Z_D}(\text{Enc}_{Z_D}(\sigma_D^\prime)) \), where \( \text{Dec}_{Z_D} \) is the corresponding decryption key w.r.t the decryption key \( Z_D \).

Carol picks \( Z_C \) and produces the locking key \( Z_C = g^{tD} \cdot g^{tC} \), and repeats the above procedure with Bob, who then picks \( z_B \) and produces the locking key \( Z_B = g^{tD} \cdot g^{tC} \cdot g^{tB} \), and repeats the above procedure with Alice.

Second, Dave releases his encrypted signature \( \sigma_D^\prime \), and Carol calculates \( z_C = \sigma_D^\prime \cdot g^{tD} \) and produces \( \sigma_C = z_C \cdot Z_D \). Dave and Carol obtain \( \sigma_C^\prime = \text{Dec}_{Z_D}(\text{Enc}_{Z_C}(\sigma_C^\prime)) \) and \( \sigma_D^\prime = \text{Dec}_{Z_D}(\text{Enc}_{Z_D}(\sigma_D^\prime)) \) respectively, and the channel between Dave and Carol is updated. Carol then unlocks the payment and updates the channel with Bob, who then unlocks the payment and updates the channel with Alice.

Under the worst case that Dave does not release his encrypted signature \( \sigma_D^\prime \), Carol can force open \( \text{Comm}_{D'} \) and obtain \( z_B \) after the time \( t_D \). As Carol is paid from Bob, she is motivated to release her encrypted signature \( \sigma_C^\prime \) to Bob; Otherwise, Bob and Alice can force open \( \text{Comm}_{C} \) and \( \text{Comm}_{B} \) and obtain \( z_C \) and \( z_B \) after the time \( t_C \) and \( t_B \) respectively to unlock their payment and update their channel.

However, there are still some issues when running payment channel network for scriptless blockchain. For example, some privacy issues on Lightning Network [30], the payment channel network of Bitcoin, has been exploited by some recent works [7, 25], while most scriptless blockchains have privacy functionality, such as Monero, these issues may prevent payment channel network from being deployed on the privacy-preserving blockchains. These issues are non-trivial and require depth-in analysis. We put them as open questions for future work.

In conclusion, we propose AuxChannel, the first bi-directional payment channel protocol for scriptless blockchains. It supports duplex payments in a channel with unlimited lifespan. The design concept is also a new direction for enabling payment channel with scriptless blockchains. We also see CVES as an independent interest as this may be applied to other applications.

REFERENCES

A SUPPLEMENTARY OF CVES

This section gives the security analysis of our proposed Schnorr-based CVES scheme in Section 4.3.

Lemma A.1. The Schnorr CVES is correct.

Proof. In our Schnorr CVES (Figure 2), we have that sk is (x, X), and the corresponding verification key pk is X := g^x. A decryption key dk’ = (y, y’, y’’, Y) w.r.t the encryption key ek’ = (Y, Y’), where y := y mod p and Y := h^y mod N, Y := g^y. Since g^y*1 = g^y=x = g^((r+cx)-ex) = g^r = R, c := H(m||R||X)), and R = g^r, it implies that EncVerify(pk, ek’, x, m’) = 1. Similarly, as R = g^r, c := H(m||R||X) and g^y*1 = g^y=ex = g^R = R, Verify(pk, sk, m, x) = 1 always holds. Thus the encrypted signature σ1 := (R, s) and the original signature σ1 := (R, s) are valid.

As an encrypted signature σ’ = (R, s), the corresponding truncated decryption key dk = (y, Y), and the original signature σ is (R, s) = (R, s) + s = (y, y’), we have s = s = s + y = y = η(y), and Y = g^y = g^y=η. Then Rec(σ, σ) outputs σ = (y, Y) always holds, That is the Schnorr CVES is recoverable.

Also, the i-th decryption key dk = (y, Y) and (i + 1)-th decryption key dk = (y, Y’), y = g^y = g^y=η and encryption key ek = (Y, Y’) = (y, h^x = Y’ = h^y = Y = g^y = Y) mod N and ek = (y) is (y, h^x = Y = g^y = Y) mod N = g^y = Y mod N = g^y = Y mod N, it implies that ConsVerify(ek, ek, ConsProof(dk, dk)) = 1. That is the Schnorr CVES is correct.

Lemma A.2. Assuming that the discrete logarithm (DL) problem is hard and the underlying NIZK system is secure, the Schnorr CVES scheme is unforgeable under the random oracle model.

Proof. Assuming that there exists an adversary A who can break the unforgeability, then given a DL problem instance (X, g) ∈ ℓ, p, we can construct a simulator A that can make use of A to output an integer x such that X = g^x.

A picks X from the problem instance (while its DL x is unknown) and sets pk := X. For the initial and later stage decryption and encryption key-pairs and the proof on two consecutive encryption keys, S generates according to the algorithms EncGen, KeyUpdate and ConsProof. S gives (pk, ek, p, g) to A.

The game of forging a valid signature runs in three stages by A: first A chooses a certain session and records the public keys (pk, ek, p, g), and then A chooses a message m, finally A outputs a forged signature σ := (R, s) on m, such that

R = g^x - H(m||R||X), where X is the pk given to A. Remark that m was not queried to O before and that signature is valid.

A simulates the random oracle H as normal (keep a table to make the consistency of input and output), and the encoring oracle E (with input the public keys pk, ek, and the chosen message m) as follows:

- Extract X and Y from pk and ek;
- Pick c ∈ ℓ and s ∈ ℓ, randomly;
- Set R := g^y = R = R;
- Assign c to the value of the output of the random oracle query H(m||R||X);
- Output σ = (R, s).

Similar to the simulation of E, A also simulates the signing oracle O (with input the public keys pk and the chosen message m) as follows:

- Extract X and Y from pk and ek;
- Pick c ∈ ℓ and s ∈ ℓ, randomly;
- Set R := g^y = R;
- Assign c to the value of output encoring oracle query E(X, Y, m) and c to the value of the output of the random oracle query H(m||R||X), if H(m||R||X) has not been queried or assigned any value before in other oracles. Otherwise, repeat the process by choosing another different c;
- Output σ = (R, s).

A can query the random oracle for qa times, the encoring oracle for qo times, and the signing oracle for qf times. At the end, A outputs a forged signature σ := (R, s) on its chosen message m, where m is inputted to encoring oracle, but not inputted to the signing oracle.

S then sends the random tape to the point that A is making the random oracle query for the value c := H(m||R||X), and use another random tape to supply a different value c := H(m||R||X), so that A outputs a different signature σ := (R, s). [This is called the Forking Lemma.]

Now there are two forged signatures of m’ outputted by A: σ := (R, s) and σ := (R, s’) and both satisfy the equation R := g^y = R := g^y = R. Then we have the following equations: r = s + c’x mod p and r = s + c’x mod p for two unknowns r and x, where R = g^r. By solving these two equations, S can output x which is the solution for the DL problem instance.

Lemma A.3. Assuming that the SQROOT problem is hard, the Schnorr CVES scheme satisfies one-wayness.

Proof. We prove this lemma by contradiction. Assuming that there exists an adversary A who can break the one-wayness of Schnorr CVES scheme. Given a SQROOT problem instance (φ, N), where φ ∈ QN, we can construct a simulator S that can make use of A to output an integer φ such that φ = g^x = N.

S picks φ from the SQROOT problem instance (φ, N) and sets dk := (φ, y, Y), where Y := φ mod p and Y := h^x mod p, and the corresponding encryption key is ek := (Y, Y’). For the signing key and verification key-pair (sk, pk), where sk := (x, X) and pk := X, the initial and later stage decryption and encryption keys, S generates according to the algorithms Gen, EncGen and KeyUpdate. S then gives (sk, dk, g, h, N) to A.
The game of reversing a decryption key from its previous session runs in two stages: first, $\mathcal{A}_0$ chooses a certain session to attack and records the decryption key $dk^i = (\varphi, y^*, Y^*)$ and encryption key $ek^i := (\hat{Y}, Y)$ (while the decryption key $dk^{i-1} = (\hat{y}, y)$ is unknown, such that $\hat{y} := y \mod N$, $y := \hat{y} \mod p$, and $Y = \hat{y}^2$), where $\hat{Y} := h^{\phi} \mod N$, then $\mathcal{A}_1$ outputs $dk^{i-1} = (\hat{y}^*, y^*, Y^*)$, which satisfies:

\[
(dk^{i+1}, \hat{d}k^{i+1}, \hat{e}k^{i+1}) \rightarrow \text{KeyUpdate}(dk^{i-1})
\]

\[
\hat{d}_i \rightarrow \text{EncSign}(sk, \hat{e}k^i, m^i)
\]

\[
\hat{e}_i \rightarrow \text{DecSign}(\hat{d}k^{i+1}, \hat{e}k^i)
\]

\[
1 \rightarrow \text{Verify}(pk, m^i, \hat{e}k^i)
\]

where $dk^i = (\hat{y}^*, y^*, Y^*)$, $\hat{d}k^i = (\hat{y}^*, \hat{y}^*, \hat{Y}^*)$, $\hat{e}k^i = (\hat{Y}^*, y^*, \hat{Y}^*)$, $\hat{e}_i := (\hat{R}, \hat{s})$ and $\hat{e}_i := (\hat{R}^*, \hat{s}^*)$. Let $\hat{R} = \hat{y}^2$. We can imply that $\hat{y}^* = \hat{y}^* \mod p$ (from 2) $\Rightarrow y^* = (\hat{y}^2)^\varphi \mod p$. Let $\hat{R} = \hat{y}^2$, then $\hat{R} = \hat{R}^*$ (from 3) $\Rightarrow r = (\hat{r} + cy^i) \mod p$. $\hat{R} = (\hat{r} + cx) \mod p$ (from 3) $\Rightarrow s = (r - y^i + cx) \mod p$, where $c = H(m||R||X)$. 

As all the inputs of equation 3 are generated according to the algorithms Gen, EncGen and KeyUpdate, according to the correctness of Schnorr we have $1 \rightarrow \text{Verify}(pk, \hat{e}k^i, \hat{d}_i)$, which implies that $\hat{R} = \hat{y}^2X^c - C$. Thus, $\hat{R} = \hat{R}^*$ (from 4) $\Rightarrow \hat{R} = \hat{y}^2X^c - \hat{y}^2\hat{Y}^c$. On the other side, $\hat{R} = \hat{y}^2X^c$ (from 5) $\Rightarrow \hat{R} = \hat{y}^2X^c - \hat{y}^2\hat{Y}^c$. From the above equation, we have $\hat{R} = \hat{y}^2X^c - \hat{y}^2\hat{Y}^c = \hat{y}^2Y^c - c^c = c^c \Rightarrow H(m||R||X) = H(m^i||R^i||X)$. Remark that, for the same input, the hash function $H()$ produces a same output. Thus, if $c = c^c$, it implies that $m^i||R||X = m^i||R^i||X$. We then have:

\[
\hat{R} = \hat{R}^* \Rightarrow \hat{R}^* = \hat{R}^* \Rightarrow \hat{y}^* = \hat{y}^*
\]

\[
\Rightarrow \varphi \mod p = \hat{y}^* \mod p = (\hat{y}^2)^\varphi \mod p
\]

That is $S_3$ can output $\hat{y}$, which is the solution for the SQROOT problem instance $(\varphi, N)$. \hfill \Box

**Lemma A.4.** Assuming that the underlying NIZK system used in our ConsProof and ConsVerify is secure, the Schnorr CVES is consecutively verifiable.

**Proof.** We prove this lemma by contradiction. Assuming that, for a session $i$ with the decryption key $dk^i$ and encryption key $ek^i$, there exists an adversary $\mathcal{A}$ who can break the consecutively verifiability of Schnorr CVES scheme.

We can construct a simulator $S_2$ who can make use of $\mathcal{A}$ to output a proof on a chosen NIZK problem instance $(\chi, w)$, which includes a chosen NP problem instance $\chi := (\hat{Y}, y^*, Y^*)$ and a five-element tuple $w := (\hat{y}, \hat{y}^*, y, y^*, k)$, such that NIZK.Verify$(\chi, P) = 1$, where $\chi := (\hat{y}, y^*, Y^*)$ are chosen by $\mathcal{A}$.

$S_2$ picks $\hat{y}, \hat{Y}, Y$ from an NIZK problem instance $(\chi, w)$, where $\chi := (\hat{Y}, y^*, Y^*)$, and $w := (\hat{y}, \hat{y}^*, y, y^*, k)$, then sets decryption key $dk^i := (\hat{y}, y)$ and the corresponding session key $ek^i := (\hat{Y}, Y)$. For the signing and verification key-pair $(sk, pk)$, where $sk := (x, X)$ and $pk := X$, the initial and later decryption keys, $S_2$ generates according to the algorithms Gen, EncGen and KeyUpdate. $S_2$ then gives $(dk^i, ek^i, k)$ to $\mathcal{A}$.

The game of breaking the consecutively verifiability of the Schnorr CVES scheme is defined in three stages: first $\mathcal{A}_0$ chooses a certain session and records the useful intermediate variable, then $\mathcal{A}_1$ outputs a decryption and encryption key-pair $(dk^{i+1}, \hat{d}k^{i+1}, \hat{e}k^{i+1})$, where $dk^{i+1} = (\hat{y}^*, y^*, \hat{Y}^*)$, $\hat{d}k^{i+1} = (\hat{y}^*, \hat{Y}^*)$, $\hat{e}k^{i+1} = (\hat{Y}^*, y^*)$, $y^* = \hat{y}^* \mod p$, $Y^* = \hat{y}^* \mod N$, on the input the decryption key $dk^i$, finally $\mathcal{A}_2$ outputs a proof $P^o$ on $dk^i$ and $dk^{i+1}$. These outputs satisfy:

\[
(dk^{i+1}, \hat{d}k^{i+1}, \hat{e}k^{i+1}) \rightarrow \text{KeyUpdate}(dk^{i-1})
\]

\[
\hat{d}^* \rightarrow \text{EncSign}(sk, \hat{e}k^{i+1}, m^*)
\]

\[
\hat{e}^* \rightarrow \text{DecSign}(\hat{d}k^{i+1}, \hat{e}k^{i+1}, \hat{d}^*)
\]

\[
1 \rightarrow \text{ConsVerify}(ek^i, \hat{e}k^{i+1}, P^o)
\]

where $dk^{i+1} = (\hat{y}^*, y^*, \hat{Y}^*)$, $\hat{d}k^{i+1} = (\hat{y}^*, \hat{Y}^*)$, $\hat{e}k^{i+1} = (\hat{Y}^*, y^*)$, $\hat{e}^* = \hat{y}^* \mod p$, $\hat{Y}^* = \hat{y}^* \mod N$. We can imply that $\hat{y}^* = \hat{y}^* \mod N$, $y^* = \hat{y}^* \mod p$, $Y^* = \hat{y}^* \mod N$. According to the above inference, for the NIZK instance $(\chi, w)$, where $\chi := (\hat{y}, \hat{y}^*, y, y^*, k)$, and the relation $R$, where $R$ is defined as: $R = \text{PoK}_{\text{CVES}} = \{(\hat{y}, \hat{y}^*, y, y^*, k) : Y = h^\phi \mod N \land Y^* = h^{\phi^*} \mod N \land Y^* = y^* \land 0 < y^* < N\}$, which implies that $\hat{y}^* = \hat{y}^* \mod N$, there exist a proof $P^o$ satisfies the equation NIZK.Verify$(\chi, P^o) = 1$, where $(\chi, w) \not\in R$, and $Pr[\text{NIZK.Verify}(\chi, P^o) = 1] > negl(\xi)$.

It is contradict to the NIZK soundness [8, 19]. That is $S_3$ can output a proof $P^o$ on the chosen NIZK instance $(\chi, w)$, which breaks the soundness property of the NIZK system. \hfill \Box

**B**

**B.1 Instantiation and Implementation of Schnorr-Based CVES**

**B.1 Instantiation**

We give the instantiation of the algorithms KeyUpdate(), ConsVerify() and ConsProof() (in Figure 2). For simplicity, we set $n = N = p'q'$, and let $dk^i := (u_i, t_i, T_i)$ and $ek^i := (U_i, T_i)$ in the following proof, where $i$ is the session number. Let $u_i \in [0, N - 1]$ and $(u_i - 1, u_i)$ satisfy: $u_i = u_i^2 - 1 \mod N$. 

Observe that $U_{i-1}, U_i, T_{i-1}, T_i$ are public elements. We denote $\text{PoK}_{\text{CVES}}$ the proof-of-knowledge for CVES scheme. Then we construct a zero-knowledge proof-of-knowledge system for $u_{i-1}$,
AuxChannel: Enabling Efficient Bi-Directional Channel for Scriptless Blockchains

PoK$_{CVES}^1$ = \{(u$_{i-1}$, u$_i$, t$_{i-1}$, t$_i$, k) : U$_{i-1}$ = h$^{\rho_{i-1}}$ mod N \wedge U$_i = h^{\rho_i}$ mod N \wedge U$_i (h^N)^k = U_{i-1}^b$ mod N \wedge T$_i = g^\eta_i \}.

PoK$_{CVES}^1$ implies that u$_i = u$_{i-1}$ + kN (or u$_i = u$_{i-1}$ mod N) and t$_i = u$_{i-1}$ mod p.

Also, we need to prove u$_i$ is from 0 to N - 1. This can be done with any range proof technique. Thus, we have the second part for PoK$_{CVES}^1$:

PoK$_{CVES}^2$ = \{(u$_i$) : U$_i = h^{u_i}$ mod N \wedge 0 < u$_i$ < N\}, (11)

The construction of PoK$_{CVES}^1$ has two parts as described above. They are constructed separately, but can be executed in parallel in its actual implementation. First, we present the proof system for PoK$_{CVES}^1$. Let $\lambda_3, \lambda_4$ be some security parameters which determine the soundness and (statistical) zero-knowledge of the proof.

Define a cryptographic hash function $H : \{0,1\}^* \rightarrow \{0,1\}^{\lambda_3}$. The first part of the zero-knowledge proof is constructed as follows:

• (Commitment). In a session $i$, the prover chooses $\rho_{i-1}, \rho_i, \rho_k$ uniformly at random from $[0, 2\lfloor |N|^{\lambda_3} + \lambda_4 - 1\rfloor]$, and computes four commitments:

\begin{align*}
Y_{i1} &= g^{\rho_{i-1}} U_{i1}^b \mod N, \quad Y_{i2} = h^{\rho_i} U_{i2} \mod N, \\
Y_{i3} &= g^{\rho_k} U_{i3} \mod N, \quad Y_{i4} = h^{\rho_k} U_{i4} \mod N.
\end{align*}

• (Challenge). The verifier picks $\lambda_3$-bit challenge uniformly at random. In the non-interactive form, the $\lambda_3$-bit challenge can be computed as the hash of the commitments and the public parameters, i.e., $\omega_i = H(g, h, T_{i1}, T_{i2}, U_{i1}, U_{i2}, Y_{i1}, Y_{i2}, Y_{i3}, Y_{i4})$.

• (Response). The prover computes the response:

$$z_{i-1} = \rho_{i-1} - \omega_i \cdot u_{i-1}, \quad z_i = \rho_i - \omega_i \cdot u_i, \quad z_k = \rho_k - \omega_i \cdot k,$$

as the proof.

• (Verify). The verifier outputs 1 if it holds that:

$$Y_{i1} = h^{u_{i-1}} U_{i1}^{z_{i-1}} \mod N, \quad Y_{i2} = g^{z_{i-1}} T_{i1}^{-\omega_i} \mod N,$$

Completeness: if $z_{i-1} = \rho_{i-1} - \omega_i \cdot u_{i-1}, z_i = \rho_i - \omega_i \cdot u_i$, and $z_k = \rho_k - \omega_i \cdot k$, then $Y_{i1} = g^{\rho_{i-1}} U_{i1}^{z_{i-1}} \mod N$. Let $\omega_i = H(g, h, T_{i1}, T_{i2}, U_{i1}, U_{i2}, Y_{i1}, Y_{i2}, Y_{i3}, Y_{i4})$. (12)

$$\omega_i = H(g, h, U_{i1}, U_{i2}, T_{i1}, T_{i2})$$

$$\left(\rho_{i-1}, \rho_i, \rho_k\right) \leftarrow [0, 2\lfloor |N|^{\lambda_3} + \lambda_4 - 1\rfloor]$$

$$\omega_i = H(g, h, U_{i1}, U_{i2}, T_{i1}, T_{i2}, U_{i3}, U_{i4}).$$

$$\omega_i = H(g, h, U_{i1}, U_{i2}, T_{i1}, T_{i2}, U_{i3}, U_{i4}).$$

$$\omega_i = H(g, h, U_{i1}, U_{i2}, T_{i1}, T_{i2}, U_{i3}, U_{i4}).$$

The proof that $\omega_i$ is chosen from $[0, 2\lfloor |N|^{\lambda_3} + \lambda_4 - 1\rfloor]$ , and $z_{i-1}, z_i, z_k$ are chosen from $[0, 2\lfloor |N|^{\lambda_3} + \lambda_4 - 1\rfloor]$, then the resulting $Y_{i1}, Y_{i2}, Y_{i3}, Y_{i4}$ are random, and the output is distributed statistically close to the real transcript.

Soundness: Let $P'$ be a (possible malicious) prover that convinces the verifier with probability $1 - negl(\xi')$, where $negl(\xi')$ is a negligible probability with $\xi'$-bit inputs. We construct the extractor as follows:

$P'(h)$:

1. Run the prover $P'$ to obtain an initial message tuple $(Y_{i1}, Y_{i2}, Y_{i3}, Y_{i4})$ and the first challenge $\omega_i$, using the two tuples. Then we have a response tuple $(z_{i-1}^{\omega_i}, z_i^{\omega_i}, z_k^{\omega_i} \leftarrow [0, 2\lfloor |N|^{\lambda_3} + \lambda_4 - 1\rfloor]$).

2. Rewind the prover $P'$ to its state after the first message.

3. Run the prover $P'$ to obtain the second challenge $\omega_i$ by using $(Y_{i1}, Y_{i2}, Y_{i3}, Y_{i4})$, and get second response tuple $(z_{i-1}^{\omega_i}, z_i^{\omega_i}, z_k^{\omega_i})$.

4. Compute and output $u_i = z_i^{\omega_i} - z_i'^{\omega_i} \mod N$, $u_{i-1} = z_{i-1}^{\omega_i} - z_{i-1}'^{\omega_i} \mod N$. It remains to argue that $\omega_i.b - \omega_i.a \cdot z_i'^{\omega_i} - z_{i-1}'^{\omega_i}$ (resp. $\omega_i.b - \omega_i.a \cdot z_i'^{\omega_i} - z_{i-1}'^{\omega_i}$) so that the extractor can compute this without knowing $\phi(N)$.

We sketch the proof that $\omega_i.b - \omega_i.a \cdot z_i'^{\omega_i} - z_{i-1}'^{\omega_i}$ under the strong RSA assumption. First, we have $Y_{i3} = h^{z_i u_i} U_{i3}^{\omega_i} \mod N$ and $Y_{i3} = h^{z_i u_i} U_{i3}^{\omega_i}$ mod N. Thus, we have $h^{z_i u_i} U_{i3}^{\omega_i} = h^{z_i u_i} U_{i3}^{\omega_i}$ (mod N) Denote by $a$ and $b$ the values $z_i'^{\omega_i} - z_i'^{\omega_i}$ and $\omega_i.b - \omega_i.a$, thus, we have $h^{a} = U_{i}^{b}(mod N)$.

Suppose on a contrary, $\beta$ does not divide $\alpha$. Let $f = gcd(a, b)$ and $\alpha = fa' \beta$, $\beta = fb'$. This implies $h^{a'} = U_{i}^{b'}(mod N)$ (otherwise we can set up $N$ as a product of two safe-primers and $f$ would help factoring $N$. In other words, we should choose $N$ such that it is a product of two safe primes.)

Since $gcd(a', b') = 1$, there exists integers $X$, $Y$ such that $Xa' + Yb' = 1$. Therefore, $h^{Xa'} = h^{Yb'} (mod N) = (U_{i}^{h^{Xa'}})^{Yb'}(mod N)$.

Therefore, $(U_{i}^{h^{Xa'}})^{Yb'}(mod N)$ is a solution to the strong RSA problem on $(N, h)$. In other words, if $\omega_i.b - \omega_i.a \cdot z_i'^{\omega_i} - z_{i-1}'^{\omega_i}$, we can solve the strong RSA problem.

The proof that $\omega_i.b - \omega_i.a \cdot z_i'^{\omega_i} - z_{i-1}'^{\omega_i}$ is exactly the same and is thus omitted.

The extraction then fails if $z_i'^{\omega_i} = z_i'^{\omega_i}$ or $z_{i-1}'^{\omega_i} = z_{i-1}'^{\omega_i}$, which happens with probability $\frac{1}{2N^2}$. Therefore, the knowledge error here is $negl(\xi') = \frac{1}{2N^2(\lambda_3 + \lambda_4)}$.

The second part PoK$_{CVES}^2$ requires that at each session, the prover proves that his/her updated decryption key $up_{i,1}$ lies in the interval $[0, N - 1]$. This proof can be done by various existing range proof systems, for example $[10–12, 14]$. As the second part PoK$_{CVES}^2$ is not the contribution of the paper, we omit the instantiation of proving $up_{i,1}$ lies in the interval $[0, N - 1]$. 