MoNet: A Fast Payment Channel Network for Scriptless Cryptocurrency Monero

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

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

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

Xianrui Qin
The University of Hong Kong, Hong Kong
Email: xrqin@cs.hku.hk

***Abstract*—We propose MoNet, the first bi-directional payment channel network with unlimited lifetime for Monero. It is fully compatible with Monero without requiring any modification of the current Monero blockchain. MoNet preserves transaction fungibility, i.e., transactions over MoNet and Monero are indistinguishable, and guarantees anonymity of Monero and MoNet users by avoiding any potential privacy leakage introduced by the new payment channel network. We also propose a new crypto primitive, named Verifiable Consecutive One-way Function (VCOF). It allows one to generate a sequence of statement-witness pairs in a consecutive and verifiable way, and these statement-witness pairs are one-way, namely it is easy to compute a statement-witness pair by knowing any of the pre-generated pairs, but hard in an opposite flow. By using VCOF, a signer can produce a series of consecutive adaptor signatures CAS. We further propose the generic construction of consecutive adaptor signature as an important building block of MoNet. We develop a proof-of-concept implementation for MoNet, and our evaluation shows that MoNet can reach the same transaction throughput as Lightning Network, the payment channel network for Bitcoin. Moreover, we provide a security analysis of MoNet under the Universal Composable (UC) security framework.**

I. INTRODUCTION

Monero is the top one privacy-preserving cryptocurrency with a market cap of over 2.6 Billion US dollars[1]. It dedicates to chasing its strong privacy-preserving functionalities (anonymity and fungibility [1], [2]), but still suffers from performance issues, including low throughput, high transaction fees and slow confirmation.

Payment channel network has been known as a promising solution to solve the above issues. However, as most payment channel network protocols require the support of script languages for the underlying blockchains, it is very challenging for scriptless blockchains, including Monero, to adopt this type of solutions. Another challenge in designing a payment channel network for Monero is the requirement of fungibility, which requires that any third party observer cannot distinguish a transaction of a payment channel network from a standard transaction of the underlying blockchain. For example, Bitcoin Lightning Network [3] allows both channel parties to establish a channel by transferring coins into a P2WSH 2-of-2 multi-signature address, which can easily be identified and is vulnerable to bribery attacks [4]. As Monero is a privacy-preserving blockchain, fungibility should be guaranteed when adopting a payment channel network solution.

Related Work. There are some Monero-compatible payment channels, DLSAG channel [5], PayMo [1], Sleepy Channel [6] and AuxChannel [7], proposed recently. Both DLSAG channel and PayMo are uni-directional channels with limited lifespan. In addition, DLSAG channel proposes a new primitive for Monero to enable dual-key transactions with time flag on-chain, which not only harms the Monero token's fungibility but also requires a hard fork of Monero ledger. *Sleepy channel* is a bi-directional payment channel protocol, and requires only the support of digital signatures. However, it has limited lifetime and requires additional collateral (which reduces liquidity) for both channel parties to incentivize the fast channel closure, thus limiting its usability. AuxChannel is the only bi-directional payment channel with unlimited lifespan for scriptless blockchain, which only requires the support of adaptor signature for the underlying blockchain. It processes channel dispute on a script-enabled chain by employing a distributed Key Escrow Service (KES) to guarantee the honest party's payout. However, AuxChannel is a generic construction that cannot be applied to Monero directly, and it does not consider the potential privacy leakage from the use of the second chain, and does not provide a mechanism for enabling payment network to support multi-hop payments. This work addresses this unclear design and provides a formal security model and analysis for our proposed system.

A. Our Contributions

This work designs a payment channel network system for Monero, that does not compromise Monero token's anonymity and fungibility. Our contributions are summarized as follows:

Verifiable Consecutive One-way function (VCOF) and Generic Consecutive Adaptor Signature (CAS). We propose the verifiable consecutive one-way function (VCOF), which generates a sequence of statement-witness pairs in a single direction, and provides the one-wayness property, where it is

---

* Jiangshan Yu is the corresponding author.

[1] Data collected on 15 Jan 2022 from https://coinmarketcap.com/.

easy to compute on the input, but hard to invert given the image of a random input, can be publicly verified. We further construct the generic construction of consecutive adaptor signature (CAS) by using VCOF, which is an important building block of MoNet.

MoNet. We construct an efficient payment channel network, MoNet, for Monero. In particular, we propose MoChannel to enable bi-directional payment channels (with no limit on the lifetime) between both channel parties, and build our MoNet on top of the MoChannel to enable multi-hop payments. Both MoChannel and MoNet also preserve the anonymity and fungibility of Monero. We also construct the security model for MoNet, and prove that MoNet is UC-secure.

Implementation and Optimization. We develop and evaluate a proof-of-concept implementation of MoNet. Our evaluation shows that MoChannel enables Monero to process approximately $2.34D$ transactions per second (TPS), where D is the number of established channel on Monero. For example, as of Jan 2022, there are more than $80,000$ channels in Lightning Network for Bitcoin. If MoChannel is of the same scale, then it can support Monero to process over $180,000$ transactions per second. We further optimize MoChannel's performance by using pre-computation to calculate and verify a batch of statement-witness pairs, improving the throughput to approximately $1,100,000$ TPS, which reaches the same throughput level of lightning network[2]. In terms of multi-hop payment, MoNet can process a multi-hop payment within $68.68ms \cdot n_h$ via n_h hops with 4G WAN latency of 60ms.

II. PRELIMINARY

A. Anonymous Multi Hop Locks (AMHL)

Anonymous Multi Hop Locks (AMHL) proposed by Malavolta et al. [8] allows $n+1$ channel parties to setup n payment locks, denoted by $l_1, \dots, l_n$, in a path. AMHL guarantees that the lock l_i can be "unlocked" if and only if l_{i+1} is also released, where $i \in [1, \dots, n-1]$. It enforces the atomicity of multi-hop payments and also offers better on-chain privacy. A recent work [1] by Thyagarajan et al, constructs a LRS-compatible AMHL scheme.

B. Signature Scheme and Adaptor Signature Scheme

Signature scheme SIG allows a signer to authenticate messages, and adaptor signature $aSIG$ is a signature concealed by a secret, it allows a signer to pre-sign a message under her secret key, such that this pre-signature can be *adapted* into a valid signature for anyone who knows the pre-signature and the corresponding secret, or anyone who knows the pre-signature and signature can *extract* a secret from them. Figure 1 is the generic construction of digital signature, where $\mathcal{H}_{SIG}$ is the challenge hash function, P_1 and P_2 are two algorithms employed by the signing algorithm producing the corresponding signature, and V_0 is the verifier algorithm employed by the verification algorithm. Erwig et al. [9] further constructs a generic transformation from signature schemes into adaptor signatures schemes, as shown in Figure 2, where f_{shift}, f_{adapt} and f_{ext} are a randomness shift function, an adapt operation function, and a witness extraction function respectively.

```
$\mathsf{Gen}_{sk}(\lambda)$
    return $(sk, pk)$

$\mathsf{Sign}_{sk}(m)$
    $(R, st) \leftarrow P_1(sk)$
    $h := \mathcal{H}_{SIG}(R, m)$
    $s := P_2(sk, R, h, st)$
    $\sigma := (h, s)$
    return $\sigma$

$\mathsf{Vrfy}_{sk}(m, \sigma))$
    parse $(h, s) \leftarrow \sigma$
    $R := V_0(pk, h, s)$
    return $h \overset{?}{=} \mathcal{H}_{SIG}(R, m)$
```

Fig. 1. Generic Construction of Digital Signature

```
$\mathsf{PSign}_{sk}(m, Y)$
    $(R_{pre}, St) \leftarrow P_1(sk)$;
    $R_{sign} := f_{shift}(R_{pre}, Y)$;
    $h := \mathcal{H}_{SIG}(R_{sign}, m)$
    $\hat{s} \leftarrow P_2(sk, R_{pre}, h, St)$
    $\hat{\sigma} := (h, \hat{s})$
    return $\hat{\sigma}$

$\mathsf{PVrfy}_{pk}(m, Y, \hat{\sigma})$
    parse $(h, \hat{s}) \leftarrow \hat{\sigma}$;
    $\hat{R}_{pre} := V_0(pk, h, \hat{s})$;
    $\hat{R}_{sign} := f_{shift}(\hat{R}_{pre}, Y)$;
    return $h \overset{?}{=} \mathcal{H}_{SIG}(\hat{R}_{sign}, m)$

$\mathsf{Adapt}_{pk}(\hat{\sigma}, y)$
    parse $(h, \hat{s}) \leftarrow \hat{\sigma}$;
    $s = f_{adapt}(\hat{s}, y)$;
    $\sigma := (h, s)$
    return $\sigma$;

$\mathsf{Ext}_{pk}(\sigma, \hat{\sigma}, Y)$
    parse $(h, s) \leftarrow \sigma$;
    parse $(h, \hat{s}) \leftarrow \hat{\sigma}$;
    return $y := f_{ext}(s, \hat{s})$
```

Fig. 2. Generic Adaptor Signature aSIG.

III. GENERALIZED CONSECUTIVE ADAPTOR SIGNATURE

This section introduces Verifiable Consecutive One-way Function (VCOF), and proposes a generic construction of consecutive adaptor signature (CAS) by using VCOF. Moreover, we apply CAS to 2-party linkable ring adaptor signature denoted by 2-party consecutive linkable ring adaptor signature (2P-CLRAS), which is key building block of MoNet.

A. Verifiable Consecutive One-way Function

We propose the concept of verifiable consecutive one-way function (VCOF), which produces a new statement-witness pair from the last generated statement-witness pair (*consecutive*), the last generated statement-witness pair is referred to as the "ancestors" pair, and the new statement-witness pair generation can be public verified (*verifiable*). Also it is efficient to compute the new statement-witnesses pair from its "ancestors" pair, but is computationally hard to compute in an opposite flow (*one-wayness*). A VCOF can be efficiently used in a stateful system, such as a bi-directional payment channel protocol, where a state is revoked if the channel capacity is reallocated. VCOF can be used for some stateful schemes requiring a sequence of interactions and updates.

Let $f_R : \Delta_w \to \Delta_s$ be the function to produce a statement $Y \in \Delta_s$ by inputting the witness $y \in \Delta_w$. We assume a secure one-way function $f_c : ((\Delta_s, \Delta_w), pp) \to (\Delta_s, \Delta_w)$ (f_c is also called the consecutive function) to generate a new statement-witness pair (Y', y') by taking the given statement-witness pair (Y, y) and a public parameter *pp*, where *pp* would be a value or an operation, as inputs. Let P_c and V_c be a proof system, where $P_c((Y, y), (Y', y'))$ generates a proof P on the two statement-witness pairs (Y, y) and (Y', y'), and the corresponding verification function $V_c((Y, Y'), P)$ outputs

---

[2] Data collected from https://medium.com/coinmonks/how-does-bitcoin-get-scalable-with-the-lightning-network-63591040462c

1 to accept the two pairs (Y, y) and (Y', y') or 0 to reject. The three functions f_c, P_c and V_c satisfy the following equations:

$$
\begin{aligned}
(Y', y') &\leftarrow f_c((Y, y), pp) \\
1 &\leftarrow V_c((Y, Y'), P_c((Y, y), (Y', y')))
\end{aligned}
$$

The proof system (P_c, V_c) is secure, if the following holds:

- *Correctness*: given two statement-witness pairs (Y, y) and (Y', y'), where $(Y', y') \leftarrow f_c((Y, y), pp)$, anyone can produce a proof $P \leftarrow P_c((Y, y), (Y', y'))$, where $1 \leftarrow V_c((Y, Y'), P)$.
- *Soundness*: once a proof P on (Y, y) and (Y', y') is accepted by the verification function V_c, then $(Y', y') \leftarrow f_c((Y, y), pp)$.
- *Zero-knowledge*: the inputs of V_c, $((Y, Y'), P)$, do not leak any information of (y, y').

The formal definition of verifiable consecutive one-way function is presented as follows:

*Definition 1 (Verifiable Consecutive One-way Function (*VCOF*)):* A verifiable consecutive function w.r.t a hard relation R consists of a tuple of three algorithms VCOF_R:=(SWGen, NewSW, CVrfy):

- $(Y^0, y^0) \leftarrow \mathsf{SWGen}(\lambda)$: the statement-witness generation algorithm takes the security parameter λ as input, and produces a statement-witness pair (Y^0, y^0).
- $((Y^{i+1}, y^{i+1}), P^{i+1}) \leftarrow \mathsf{NewSW}((Y^i, y^i), pp)$: the new statement-witness algorithm takes a statement-witness pair (Y^i, y^i) and a public parameter pp as inputs, and produces a new statement-witness pair (Y^{i+1}, y^{i+1}) and a proof P, where $i \in \mathbb{N}$.
- $1/0 \leftarrow \mathsf{CVrfy}(Y^i, Y^{i+1}, P^{i+1})$: the verification algorithm takes two statements (Y^i, Y^{i+1}) and the proof P^{i+1} as inputs, and outputs 1 (acceptance) or 0 (rejection).

```
SWGen($\lambda$)
    $y^0 \leftarrow \Delta_w$;
    $Y^0 \leftarrow f_R(y^0)$;
    return $(Y^0, y^0)$

NewSW($(Y^i, y^i), pp$)
    $(Y^{i+1}, y^{i+1}) \leftarrow f_c((Y^i, y^i), pp)$;
    $P^{i+1} \leftarrow P_c((Y^{i+1}, y^{i+1}), (Y^i, y^i))$;
    return $((Y^{i+1}, y^{i+1}), P^{i+1})$

CVrfy($(Y^i, Y^{i+1}), P^{i+1}$)
    return $1/0 \leftarrow V_c(Y^i, Y^{i+1}, P^{i+1})$
```

Fig. 3. Verifiable Consecutive One-way Function (VCOF)

Comparison between VCOF and Forward Secrecy. In a signature scheme, forward secrecy FS guarantees that a forward secure signature in the past remains secure even if the current key is lost. The main idea is to divided the lifetime of the public key into T intervals, and in each time interval the same public key corresponds to different secret keys. A current secret key can be used to derive the secret key in the future, but not the past. Thus, even a compromise of the current secret key dose not enable the adversary to forge signatures pertaining to the past. The main difference between VCOF and FS is that VCOF updates a statement-witness pair, while FS only updates the private key and the public key remains unchanged.

Security Model of VCOF. The verifiable consecutive one-way function should satisfy three properties, *consecutiveness*, *consecutive verifiability*, and *one-wayness*:

- *Consecutiveness*: a new statement-witness (Y^{i+1}, y^{i+1}) is derived from the given statement-witness by using the function $f_c((Y^i, y^i), pp)$.
- *Consecutive verifiability*: given two statements and an associated proof, anyone can be convinced that two witnesses associated to the given statements are consecutive, meaning that one of the statement-witness pairs is generated from the other.
- *One-wayness*: given a newly generated statement-witness pair, no one can derive the previous statement-witness.

We defer the formal definition of three properties and the corresponding proof to Appendix A.

Application of VCOF. VCOF can be combined to achieve some useful functionalities. We show one of the applications - consecutive adaptor signature (CAS) in Section III-B, which combines adaptor signature with VCOF. Briefly, CAS allows a signer to pre-sign some messages by using VCOF, and revealing one of the intermediate witnesses makes an exposure of the following signatures.

B. Generalized Consecutive Adaptor Signature

Consecutive adaptor signature (CAS) allows a signer to make multiple adaptor signatures $\hat{\sigma} = \{\hat{\sigma}^0, \ldots, \hat{\sigma}^{i_c}\}$, where $i_c \in \mathbb{N}$, on some messages $\mathbf{m} = \{m^0, \ldots, m^{i_c}\}$ by using a sequence of witnesses $\mathbf{y} = \{y^0, \ldots, y^{i_c}\}$. Sui et al, propose the concept of the consecutive verifiably encrypted signature (CVES) [7]. This section provides a generic transformation from adaptor signature and VCOF to CAS, which is the generalized version of CVES.

CAS uses VCOF to update the embedded statements (Y^{i_c}, Y^{i_c+1}) of two adaptor signatures. Once a witness is revealed, the corresponding signature and the following signatures are adaptable. Remark that, the first statement-witness pair is generated by executing $\mathsf{SWGen}(\cdot)$, and each of witnesses-statement pairs is generated from its ancestor by executing $\mathsf{NewSW}(\cdot)$. Let $i_c \in \mathbb{N}$ and $i'_c \in \mathbb{N}^+$ be the index number of signing sessions. Following the generic signature and adaptor signature construction from Section II-B, our generic construction of consecutive adaptor signature is presented in Algorithm 1.

Since the security model of CAS follows directly from CVES [7], we omit it and the corresponding proof.

C. 2-Party Consecutive Linkable Ring Adaptor Signature

A linkable ring signature (LRS) [10] preserves a signer's anonymity by hiding the signer's verification key vk among a set of verification keys $\vec{vk} = \{\mathsf{vk}_1, \ldots, \mathsf{vk}_n\}$. LRS scheme can be extended to 2-party linkable ring signature (2P-LRS) [1] scheme, which allows two signers to jointly sign a linkable ring signature by using their partial signing keys associated to a single public key, where the identity of the two signers is considered as a single signer for an observer and the corresponding verification algorithm is same as a LRS scheme.

We show that our generic consecutive adaptor signature can be applied to the 2-party linkable ring adaptor signature with aggregatable public keys denoted by 2P-CLRAS. It allows

Algorithm 1: Generic Transformation from aSIG and VCOF to Consecutive Adaptor Signature (CAS).

```
Procedure SetUp($\lambda$):
    return $param$
Procedure Gen($\lambda$):
    $sk \leftarrow \Delta_{sk}(param)$;
    $vk = f_{vk}(sk)$;
    return $(sk, vk)$
Procedure PSign$_{sk}(m^{i_c}, Y^{i_c})$:
    $(R_{pre}, St) \leftarrow P_1(sk)$;
    $R_{sign} := f_{shift}(R_{pre}, Y^{i_c})$;
    $h^{i_c} := \mathcal{H}(R_{sign}, m^{i_c})$;
    $\hat{s}^{i_c} \leftarrow P_2(sk, R_{pre}, h^{i_c}, St)$;
    $\hat{\sigma}^{i_c} := (h^{i_c}, \hat{s}^{i_c})$;
    return $\hat{\sigma}^{i_c}$
Procedure PVrfy$_{vk}(m^{i_c}, Y^{i_c}, \hat{\sigma}^{i_c})$:
    $(h^{i_c}, \hat{s}^{i_c}) := \hat{\sigma}^{i_c}$;
    $\hat{R}_{pre} := V_0(pk, h^{i_c}, \hat{s}^{i_c})$;
    $\hat{R}_{sign} := f_{shift}(\hat{R}_{pre}, Y^{i_c})$;
    return $h^{i_c} \stackrel{?}{=} \mathcal{H}(\hat{R}_{sign}, m^{i_c})$
Procedure Vrfy$_{vk}(m^{i_c}, \sigma^{i_c})$:
    $(h^{i_c}, s^{i_c}) := \sigma^{i_c}$;
    $R_{sign} := f_{shift}(pk, h^{i_c}, s^{i_c})$;
    return $h^{i_c} \stackrel{?}{=} \mathcal{H}(R_{sign}, m^{i_c})$
Procedure Adapt$_{vk}(\hat{\sigma}^{i_c}, y^{i_c})$:
    $(h^{i_c}, \hat{s}^{i_c}) := \hat{\sigma}^{i_c}$;
    $s^{i_c} = f_{Adapt}(\hat{s}^{i_c}, y^{i_c})$;
    return $(h^{i_c}, s^{i_c})$;
Procedure Ext$_{vk}(\sigma^{i_c}, \hat{\sigma}^{i_c}, Y^{i_c})$:
    $(h^{i_c}, s^{i_c}) := \sigma^{i_c}$;
    $(h^{i_c}, \hat{s}^{i_c}) := \hat{\sigma}^{i_c}$;
    return $f_{ext}(s^{i_c}, \hat{s}^{i_c})$
Procedure SWGen($\lambda$):
    $y^0 \leftarrow \Delta_w$;
    $Y^0 \leftarrow f_R(y^0)$;
    return $(Y^0, y^0)$
Procedure NewSW($((Y^{i'_c}, y^{i'_c}), pp)$):
    $(Y^{i'_c+1}, y^{i'_c+1}) \leftarrow$
      $f_c((Y^{i'_c}, y^{i'_c}), pp)$;
    $P^{i'_c+1} \leftarrow$
      $P_c((Y^{i'_c}, y^{i'_c}), (Y^{i'_c+1}, y^{i'_c+1}))$;
    return $((Y^{i'_c+1}, y^{i'_c+1}), P^{i'_c+1})$
Procedure
  CVrfy($((Y^{i_c}, Y^{i_c+1}), P^{i_c+1})$):
    return $1/0 \leftarrow$
      $V_c((Y^{i_c}, Y^{i_c+1}), P^{i_c+1})$
```

Algorithm 2: Generic 2-Party Consecutive Linkable Ring Adaptor Signature (2P-CLRAS) from both 2P-LRS and GCAS among n_s signers by using VCOF.

```
Procedure Setup($\lambda$):
    defined hash
      $H_{lrs} : \{0,1\}^* \rightarrow \Delta_c$;
    return param;
Procedure JGen($\lambda$):
    receiving $\mathsf{vk}_{\mathcal{P}'}$ from $\mathcal{P}'$ and
      generate $\mathsf{vk} = \mathsf{vk}_{\mathcal{P}} \oplus \mathsf{vk}_{\mathcal{P}'}$;
    return $(\mathsf{vk}, sk_{\mathcal{P}})$
Procedure PSign$_{\vec{\mathsf{vk}}}(sk_{\mathcal{P}}, m^{i_c}, Y^{i_c})$:
    $r_{\mathcal{P}} \leftarrow_s \Delta_r$;
    $R_{\mathcal{P}} = P_1(r_{\mathcal{P}}, \vec{\mathsf{vk}})$;
    receiving $R_{\mathcal{P}'}$ from $\mathcal{P}'$ and
      generate
      $R = R_{\mathcal{P}} \oplus_R R_{\mathcal{P}'} \oplus_R Y^{i_c}$;
    $c = H_{lrs}(m, R, \vec{vk})$;
    $\hat{z}_{\mathcal{P}} := P_2(sk_{\mathcal{P}}, \vec{vk}, r_{\mathcal{P}}, c)$;
    receiving $\hat{z}^{i_c}_{\mathcal{P}'}$ from $\mathcal{P}'$ and
      generate $\hat{z}^{i_c} = \hat{z}^{i_c}_{\mathcal{P}} \oplus \hat{z}^{i_c}_{\mathcal{P}'}$;
    return $\hat{\sigma}^{i_c} = (\hat{z}^{i_c}, c)$
Procedure PVrfy$_{\vec{\mathsf{vk}}}(m^{i_c}, \hat{\sigma}^{i_c}, Y^{i_c})$:
    parse $\hat{\sigma}^{i_c} = (\hat{z}, c)$;
    $R = V_0(\vec{\mathsf{vk}}, c, \hat{z}, m^{i_c}) \oplus_R Y^{i_c}$;
    if $c \neq H_{lrs}(m, R, \vec{vk})$ then
        return 0;
    return 1;
Procedure Vrfy$_{\vec{\mathsf{vk}}}(m^{i_c}, \sigma^{i_c})$:
    parse $\sigma^{i_c} = (z, c)$;
    $R = V_0(\vec{\mathsf{vk}}, c, z)$;
    if $c \neq H_{lrs}(m^{i_c}, R, \vec{vk})$ then
        return 0;
    return 1;
Procedure Adapt$_{\vec{\mathsf{vk}}}(\hat{\sigma}^{i_c}, y^{i_c})$ and
  Ext$_{\vec{\mathsf{vk}}}(\hat{\sigma}^{i_c}, \sigma^{i_c}, Y^{i_c})$:
    parse $\hat{\sigma}^{i} := (\hat{z}, c)$;
    $z = \hat{z} \oplus y^{i_c}$;
    $\sigma^{i_c} = (z, c)$;
    return $\sigma^{i_c}$;
Procedure SWGen($\lambda$):
    pick $y^0_{\mathcal{P}} \in \Delta_t$;
    $Y^0_{\mathcal{P}} = g^{y^0_{\mathcal{P}}}$;
    receiving $Y^0_{\mathcal{P}'}$ from $\mathcal{P}'$ and
      compute $Y^0 = Y^0_{\mathcal{P}} \oplus_R Y^0_{\mathcal{P}'}$;
    return $(Y^0, (Y^0_{\mathcal{P}}, y^0_{\mathcal{P}}))$;
Procedure NewSW($((Y^{i'_c}_{\mathcal{P}}, y^{i'_c}_{\mathcal{P}}), pp)$):
    $(Y^{i'_c+1}_{\mathcal{P}}, y^{i'_c+1}_{\mathcal{P}}) \leftarrow$
      $f_c((Y^{i'_c}_{\mathcal{P}}, y^{i'_c}_{\mathcal{P}}), pp)$;
    $P^{i'_c+1}_{\mathcal{P}} \leftarrow$
      $P_c((Y^{i'_c}\mathcal{P}, y^{i'_c}_{\mathcal{P}}, (Y^{i'_c+1}_{\mathcal{P}}, y^{i'_c+1}_{\mathcal{P}}))$;
    receiving $Y^{i_c}_{\mathcal{P}'}, P^{i'_c+1}_{\mathcal{P}'}$ from $\mathcal{P}'$
      and compute
      $Y^{i_c} = Y^{i_c}_{\mathcal{P}} \oplus_R Y^{i_c}_{\mathcal{P}'}$ and
      $P^{i_c} = P^{i_c}_{\mathcal{P}} \oplus_R P^{i_c}_{\mathcal{P}'}$;
    return
      $(Y^{i'_c+1}, (Y^{i'_c+1}_{\mathcal{P}}, y^{i'_c+1}_{\mathcal{P}}), P^{i'_c+1})$;
Procedure
  CVrfy($((Y^{i_c}_{\mathcal{P}}, Y^{i_c+1}_{\mathcal{P}}), P^{i_c+1}_{\mathcal{P}})$):
    if $V_c((Y^{i_c}_{\mathcal{P}}, Y^{i_c+1}_{\mathcal{P}}), P^{i_c+1}_{\mathcal{P}}) =$
      1 then
        return 1;
    return 0;
```

two signers with single aggregatable public key to generate a sequence of linkable ring adaptor signature together. Let A and B be two signers in a 2P-LRS scheme, and $\mathcal{P} \in \{A, B\}$ and $\mathcal{P}' = \{A, B\} \setminus \mathcal{P}$. Let $\oplus$ be a group operation in Δ_r, and $\oplus_R$ be a group operation in the domain of R. The inverse functions are defined in the corresponding group operations $\oplus$ and $\oplus_R$. We present a generic construction of 2P-CLRAS in Algorithm 2.

IV. DESCRIPTION OF MONET

The goal of this work is to build a payment channel network for Monero, a fully scriptless blockchain with strong privacy requirements on the payment channel (network) construction. This section proposes MoChannel, a bi-directional payment channel for Monero, and further constructs the payment channel network, MoNet, built upon MoChannel. For simplicity, we assume that Alice has frequent transactions with Bob, so they maintain a channel. However, Alice needs to pay Carol causally, through Bob.

A. Overview

For the ease of understanding, this section presents high-level description of MoNet. Alice and Bob join MoNet collaboratively by funding a MoChannel with some coins. They transact with each other by re-distributing their balances within the established channel, and exit MoNet by recording their final balances on Monero. By using MoNet, Alice can pay Carol, who has no channel with her but does have one with Bob. By leveraging 2P-CLRAS, it is hard for each of the channel parties to revert a history state from the latest state, but easy to infer the latest state if the counterparty tries to close the channel with a history state.

To establish a MoChannel, Alice or Bob executes 2P-CLRAS.SWGen($\cdot$) and 2P-CLRAS.JGen($\cdot$) collaboratively to generate their partial initial statement-witness pairs associated with a joint initial statement and their partial signing keys associated with an aggregatable public key. Each of them then creates two transactions. The first is funding transaction, which funds a channel by transferring Monero tokens from their private addresses respectively to the aggregatable public key address as the channel capacity. The other is commitment transaction, which spends the output of funding transaction to both parties' private addresses respectively. This transaction guarantees that each channel party can exit MoChannel even if there is no transactions within the channel. Alice and Bob perform 2P-CLRAS.PSign($\cdot$) to pre-sign commitment transaction collaboratively, and sign funding transaction and broadcast the signed funding transaction to Monero network. A channel is established between Alice and Bob, once the funding transaction is recorded on-chain.

To update the channel, Alice and Bob preforms 2P-CLRAS.PSign($\cdot$) collaboratively on a newly created transaction to redistribute their channel balances, where the difference of their balances indicates the transaction amount.

To close the channel, Alice and Bob exchange their latest witnesses, and adapt a signature from the corresponding pre-signature by executing 2P-CLRAS.Adapt$(\cdot)$. Finally, each of them can close the channel by uploading a transaction with the adapted signature to Monero.

When the channel between Alice and Bob is on a payment path for a multi-hop payment, they lock a payment collaboratively within their channel first and unlock the payment only if the receiver in their channel paid for the next hop.

B. Security Properties

We expect the following properties to be guaranteed:

For a single MoChannel:

Guaranteed channel closure. Normally, closing MoChannel requires both Alice and Bob's cooperation. This property requires that either Alice or Bob can close the channel unilaterally, which prevent the channel from being locked forever.

Guaranteed payout for honest channel parties. Assuming that there is at least one honest party within a channel, he can withdraw with no less than his latest balance.

On-chain unidentifiability. On-chain transactions cannot be identified as being for opening or closing a channel.

For a payment via multi-hop MoChannel:

Atomicity. All the payments on the path are either successful or failed, and no "half paid" scenario exists.

Unlockability. All the on-path payments can be unlocked, even if the receiver does not cooperate.

Sender/Receiver privacy. For successful payments, any intermediary cannot determine if the left (right) neighbor along the path is the actual sender (receiver) or just an honest user connected to the sender (receiver) through a path of non-compromised users.

Path privacy. In the case of a successful payment, malicious intermediaries cannot determine which users participated in the payment aside from their direct neighbors.

C. Main Construction

MoNet allows users to build payment channels, MoChannel, upon Monero and make payments to a recipient connecting directly or through a path of payment channels with the sender. This section describes how MoNet works. Moreover, we provide intuitions on how MoNet satisfies these properties, before formally proving them in the next section.

MoChannel (Figure 4). To establish a channel, Alice and Bob generate their partial signing keys $\tilde{sk}_A$, $\tilde{sk}_B$ associated with their joint verification key vk_{AB} (line 1). They also generate the initial statement-witness tuples $(S^0, (S^0_A, w^0_A))$ and $(S^0, (S^0_B, w^0_B))$ (line 2) collaboratively and create two transactions: funding transaction $Tx_f := (vk_A : bal^0_A, vk_B : bal^0_B)||(vk_{AB} : bal_C)$ and commitment transaction $Tx^0_c := (vk_{AB} : bal_C)||(vk_{A'} : bal^0_A, vk_{B'} : bal^0_B)$ (line 3). The funding transaction Tx_f indicates that Alice and Bob transfer bal^0_A and bal^0_B (their initial channel balances) from vk_A and vk_B respectively to vk_{AB} as channel capacity bal_C. The commitment Tx^0_c spends the output of Tx_f to reallocate the channel capacity from their joint account vk_{AB} to $vk_{A'}$ and $vk_{B'}$ [3]. They collaboratively produce pre-signature $\hat{\sigma}^0_{\tilde{sk}_A,\tilde{sk}_B}$ over Tx^0_c by using $\tilde{sk}_A$, $\tilde{sk}_B$ and S^0 (line 4), and sign and exchange their signatures σ_{sk_A} and σ_{sk_B} on the funding transaction Tx_f by using their signing keys sk_A and sk_B associated with vk_A and vk_B respectively (line 5-6). Each of them can upload the signed funding transaction $(\sigma_{sk_A}, \sigma_{sk_B}, Tx_f)$ to Monero, and the channel Ch_{AB} established.

To update the channel, Alice and Bob collaboratively generate a statement S^i, exchange their partial statements and proofs (S^i_A, P^i_A) and (S^i_B, P^i_B), and keep the corresponding witnesses w^i_A and w^i_B secret (line 11). Once the received statement is verified, they collaboratively produce a pre-signature $\hat{\sigma}^i_{\tilde{sk}_A,\tilde{sk}_B}$ over $Tx^i_c := (vk_{AB} : bal_C)||(vk_{A'} : bal^i_A, vk_{B'} : bal^i_B)$ by using $(\tilde{sk}_A, S^i_A)$ and $(\tilde{sk}_B, S^i_B)$ (line 12-13).

To close a channel, Alice and Bob exchange their latest witnesses w^i_A and w^i_B (line 14-17), and adapt the corresponding signature $\sigma^0_{\tilde{sk}_A,\tilde{sk}_B}$ on Tx^i_c from $\hat{\sigma}^0_{\tilde{sk}_A,\tilde{sk}_B}$, w^i_A and w^i_B (line 18). Each of Alice and Bob can upload the signed $(\sigma^i_{\tilde{sk}_A,\tilde{sk}_B}, Tx^i_c)$ to Monero, which closes the channel Ch_{AB} (line 19-20).

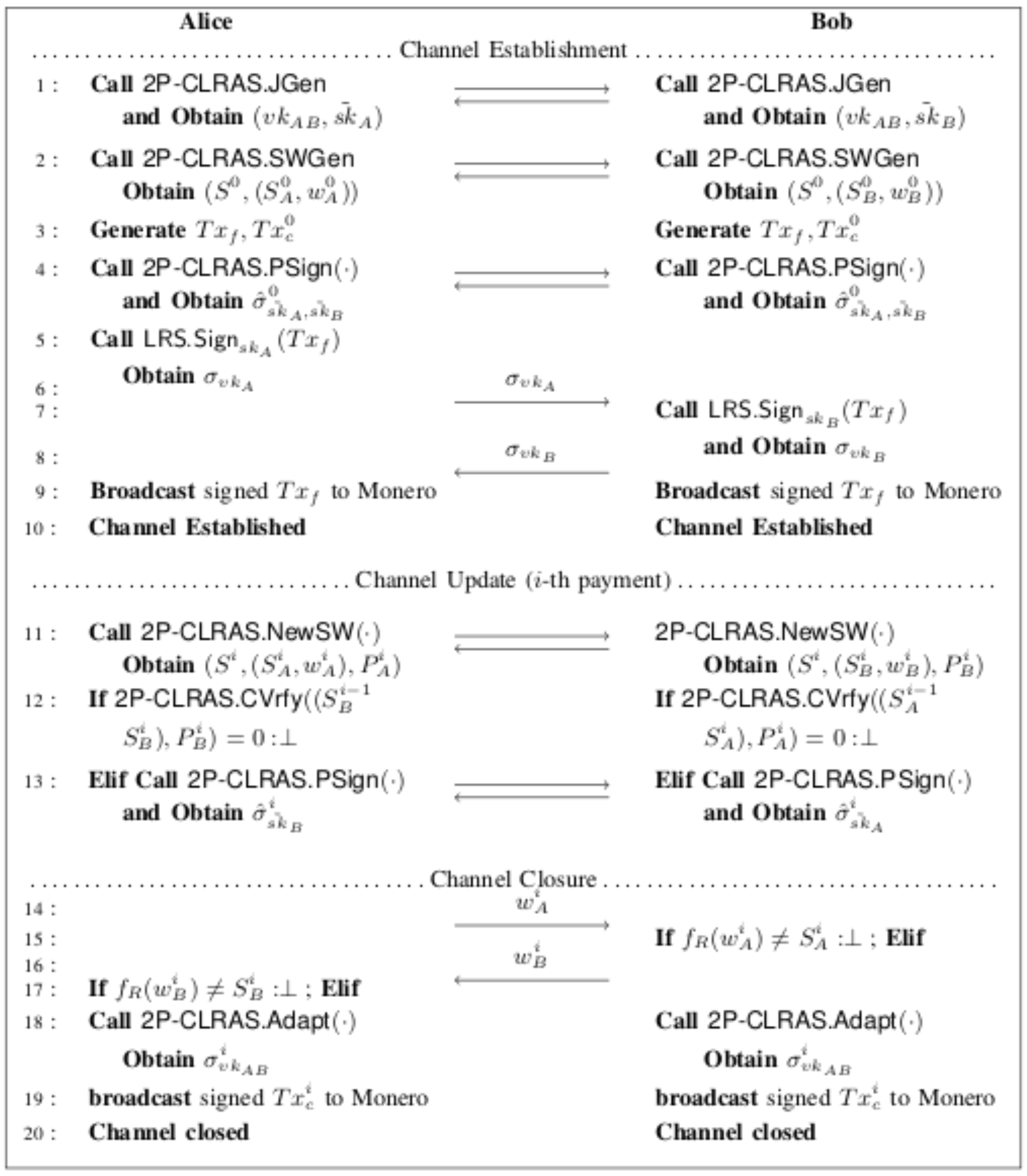
Alice
Bob
Channel Establishment
1 : Call 2P-CLRAS.JGen and Obtain $(vk_{AB}, \tilde{sk}_A)$
Call 2P-CLRAS.JGen and Obtain $(vk_{AB}, \tilde{sk}_B)$
2 : Call 2P-CLRAS.SWGen Obtain $(S^0, (S^0_A, w^0_A))$
Call 2P-CLRAS.SWGen Obtain $(S^0, (S^0_B, w^0_B))$
3 : Generate Tx_f, Tx^0_c
Generate Tx_f, Tx^0_c
4 : Call 2P-CLRAS.PSign$(\cdot)$ and Obtain $\hat{\sigma}^0_{\tilde{sk}_A,\tilde{sk}_B}$
Call 2P-CLRAS.PSign$(\cdot)$ and Obtain $\hat{\sigma}^0_{\tilde{sk}_A,\tilde{sk}_B}$
5 : Call LRS.Sign$_{sk_A}(Tx_f)$
Obtain σ_{vk_A}
6 :
σ_{vk_A}
7 : Call LRS.Sign$_{sk_B}(Tx_f)$ and Obtain σ_{vk_B}
8 :
σ_{vk_B}
9 : Broadcast signed Tx_f to Monero
Broadcast signed Tx_f to Monero
10 : Channel Established
Channel Established
Channel Update (i-th payment)
11 : Call 2P-CLRAS.NewSW$(\cdot)$ Obtain $(S^i, (S^i_A, w^i_A), P^i_A)$
2P-CLRAS.NewSW$(\cdot)$ Obtain $(S^i, (S^i_B, w^i_B), P^i_B)$
12 : If 2P-CLRAS.CVrfy$((S^{i-1}_B S^i_B), P^i_B) = 0 : \perp$
If 2P-CLRAS.CVrfy$((S^{i-1}_A S^i_A), P^i_A) = 0 : \perp$
13 : Elif Call 2P-CLRAS.PSign$(\cdot)$ and Obtain $\hat{\sigma}^i_{\tilde{sk}_B}$
Elif Call 2P-CLRAS.PSign$(\cdot)$ and Obtain $\hat{\sigma}^i_{\tilde{sk}_A}$
Channel Closure
14 :
w^i_A
15 : If $f_R(w^i_A) \neq S^i_A : \perp$; Elif
16 :
w^i_B
17 : If $f_R(w^i_B) \neq S^i_B : \perp$; Elif
18 : Call 2P-CLRAS.Adapt$(\cdot)$ Obtain $\sigma^i_{vk_{AB}}$
Call 2P-CLRAS.Adapt$(\cdot)$ Obtain $\sigma^i_{vk_{AB}}$
19 : broadcast signed Tx^i_c to Monero
broadcast signed Tx^i_c to Monero
20 : Channel closed
Channel closed

Fig. 4. MoChannel. The double arrows (e.g. line 1, 2, 4, 11, 13) denote that both channel parties execute an interactive algorithm collaboratively, and the number of interactions are decided by the corresponding algorithm.

Multi-hop Payments (Figure 5). For the sake of clarify, we describe a scenario that Alice wants to transfer x coins to Carol via Bob, who has channels with Alice and Carol respectively.

As Monero does not support any script execution, to lock a payment within a channel in MoNet, both channel parties jointly create an incomplete pre-signature, which is an adaptor signature concealed by a secret value, by using a locking key (public key), their newly generated statements and their partial

---

[3] Due to fresh key policy in Monero, $vk_{A'}$ and $vk_{B'}$ are different from vk_A and vk_B respectively.

signing key associated with the joint address in the output of funding transaction on a new commitment transaction.

To simplify notations, we assume that both channels Ch_{AB} and Ch_{BC} are in the $(i-1)$-th state when processing a multi-hop payment. The multi-hop payment in MoNet is processed as follows: 1) **Setup**. Alice, the sender of the multi-hop payment, generates some locks and unlocking keys, (Y_B, y_b) and (Y_C, y_c), for each intermediate channel, and sends $((Y_B, Y_C), y_b)$ and $((Y_C, 0), y_c)$, to Bob and Carol respectively (line 1-7). 2) **Lock**. In each on-path channel, for example channel Ch_{AB}, Alice and Bob collaboratively generate their new statements, witnesses and the corresponding proofs $(S^i_{AB}, (S^i_A, w^i_A), P^i_A)$ and $(S^i_{AB}, (S^i_B, w^i_B), P^i_B)$, and an incomplete pre-signature $\hat{\sigma}^{i*}_{AB}$ on $Tx^i_{c,AB}$ by using Y_B, which locks channel Ch_{AB} (line 8, 10). The process is identical to Bob and Carol, they collaboratively lock the channel Ch_{BC} at state i by using the lock Y_C (line 9, 11). 3) **Unlock**. Carol adapts $\hat{\sigma}^i_{vk_{BC}}$ from $\hat{\sigma}^{*i}_{vk_{BC}}$ by adding y_c, and sends $\hat{\sigma}^i_{vk_{BC}}$ to Bob (line 12-13), who then calculates y_c from $\hat{\sigma}^{*i}_{vk_{BC}}$ and $\hat{\sigma}^i_{vk_{BC}}$, and recovers $\hat{\sigma}^i_{vk_{AB}}$ from $\hat{\sigma}^{*i}_{vk_{AB}}$ by adding y_b and y_c (line 16-17). Finally, all channels updated and the payment succeed.

We discuss that how MoNet satisfies the properties in Section IV-B below.

To ensure the **sender/receiver privacy** and **path privacy**, MoNet leverages anonymous multi-hop locks (AMHLs [8], see Section II-A) scheme, which allows the sender of a multi-hop payment to generate locks and deliver messages via an anonymous communication channel [11] to each on-path user.

Resolve the Dispute. MoNet also provides solutions for resolving some potential disputes, e.g., any of channel parties does not release his witness at the channel closure phase.

We employ a distributed Key Escrow Service following AuxChannel paradigm [7] to provide the *guaranteed channel closure*, *guaranteed payout* and *unlockability* properties. Key Escrow Service can be implemented on a script-enabled platform, for example on Ethereum. Alice and Bob escrow their initial witnesses to Key Escrow Service among n_e escrowers by using publicly verifiable secret sharing scheme [12], [13], [14] before establishing a channel. Therefore, each channel party can reconstruct the other's initial witness from n_e escrowers and further compute the other's latest witness and recover a valid signature on the latest commitment transaction within the channel.

To ensure the **guaranteed channel closure**, each channel party can propose a channel dispute request to KES and reconstruct the counterparty's initial witness from escrowers if he does not cooperate to close the channel. A channel dispute request contains a timer τ^i and both channel parties's statements $(S^i_{\mathcal{P}}, S^i_{\mathcal{P}'})$ at channel state i. Notice that, both channel parties agree on and cross-sign over a timer and two puzzles with the signature scheme employed by KES at each channel update phase.

To ensure the **unlockability** when routing a multi-hop payment, the locked payment can be cancelled within each on-path channel or processed by calling KES if there is a dispute. We follow the scenario that Alice pays Carol x Monero tokens via Bob, and a successful payment updates both Ch_{AB} and Ch_{BC} from state $i-1$ to i. Alice and Bob, which are the senders in channel Ch_{AB} and Ch_{BC} respectively, can redeem their coins by updating channel Ch_{AB} or Ch_{BC} to state $i+1$ with their balances at state $i-1$. This requires the collaboration between both channel parties in channel Ch_{AB} or Ch_{BC} respectively. Moreover, if Bob or Carol does not cooperate in channel Ch_{AB} or Ch_{BC} to cancel a locked payment, Alice or Bob can redeem their coins by proposing a dispute request to KES. The channel under dispute will be closed. Usually, only the channel in the last hop is close, when suffering an unexpected locking time due to the malicious recipient (Carol), who does not unlock a payment. So other on-path participants (Alice and Bob) are rational and still want to make transactions within their channels (Ch_{AB}), they would cancel the payment within their channels.

To satisfy the **atomicity**, MoNet ensures that there is no "half paid" scenario happens. Similar to LN [3], the atomicity of a multi-hop payment is guaranteed by the cascade timers, where $\tau^i_{AB} > \tau^i_{BC}$, and the hard relationship of each lock. As both parties can redefine their channel timer at each channel state, it guarantees the cascade time-lock requirement.

As both channel parties expose their initial statement-witness pairs to KES, anyone who knows both parties' witnesses can extract an adaptor signature once the corresponding signature is recorded on-chain, which will be identified as a closing channel transaction. This is not desired and goes against the **on-chain unidentifiability** property. To guarantee this property, both parties can re-randomize their witnesses and statements by setting $S'^0_{\mathcal{P}} = S^0_{\mathcal{P}} \times g^r$ and $w'^0_{\mathcal{P}} = w^0_{\mathcal{P}} + r$ for some random value r chosen uniformly over the randomness domain and generator g.

V. SECURITY MODEL AND ANALYSIS OF MONET

This section formally defines the ideal functionality of MoNet, provides the analysis of how the ideal functionality satisfies the security properties described in Section IV-B, and proves that MoNet is secure under universal composable (UC) framework introduced by Canetti [15].

A. Security Model

UC-Security. Let $EXEC_{\Pi,\mathcal{A},\mathcal{E}}$ be the ensemble of the outputs of the environment $\mathcal{E}$ when interacting with the adversary $\mathcal{A}$ and parties running the protocol Π (over the random coins of all the involved machines).

Definition 2 (UC-Security): A protocol Π UC-realizes an ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ the ensemble $EXEC_{\Pi,\mathcal{A},\mathcal{E}}$ and $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ are computationally indistinguishable.

Attacker Model. We model participants in our protocol as interactive Turing machines that interact with a trusted functionality $\mathcal{F}$ via secure and authenticated channels. We model the attacker $\mathcal{A}$ as an interactive Turing machine that all the

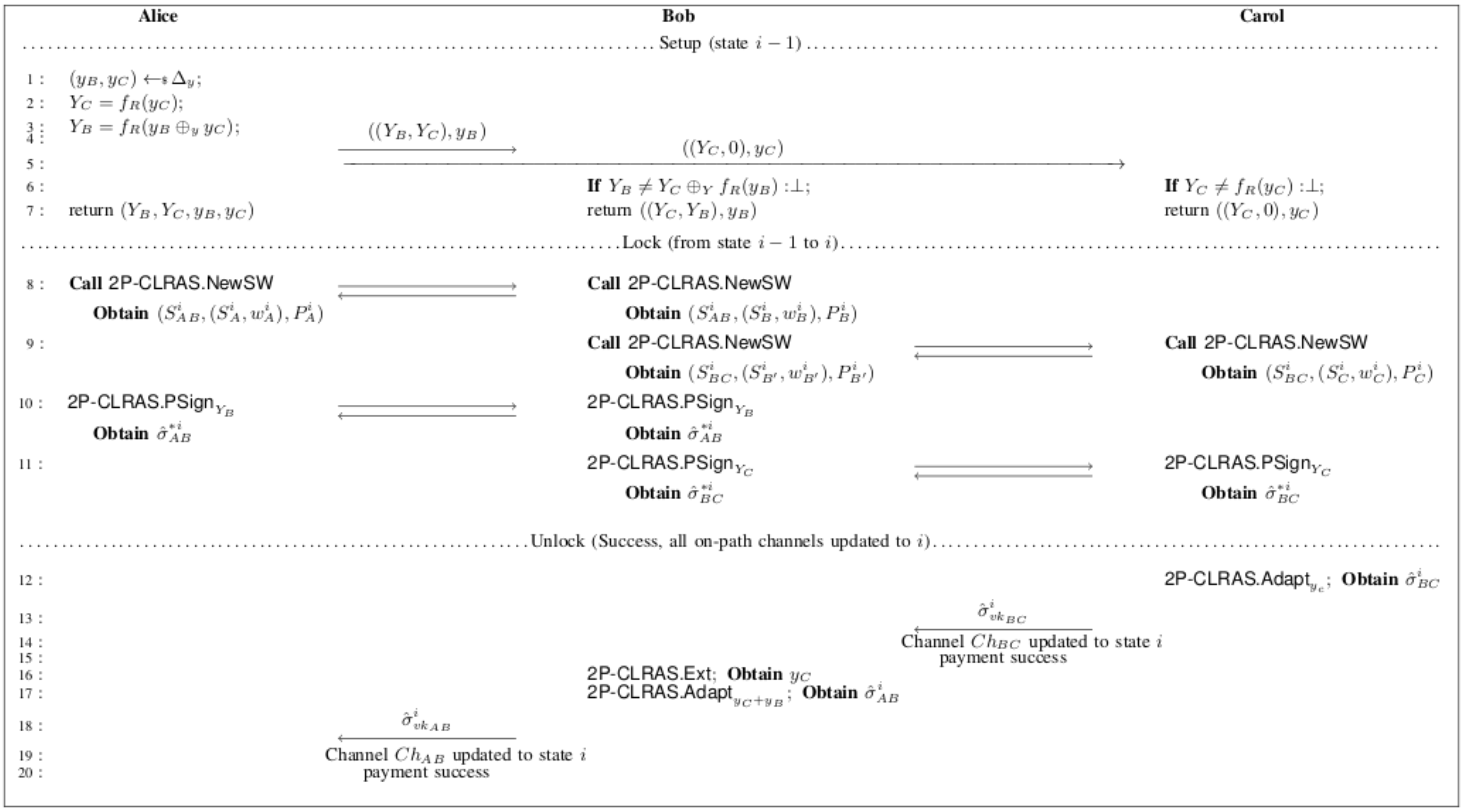
Alice
Bob
Carol
Setup (state $i-1$)
1 : $(y_B, y_C) \leftarrow_{\$} \Delta_y$;
2 : $Y_C = f_R(y_C)$;
3 : $Y_B = f_R(y_B \oplus_y y_C)$;
$((Y_B, Y_C), y_B)$
$((Y_C, 0), y_C)$
6 : **If** $Y_B \neq Y_C \oplus_Y f_R(y_B) : \perp$;
If $Y_C \neq f_R(y_C) : \perp$;
7 : return (Y_B, Y_C, y_B, y_C)
return $((Y_C, Y_B), y_B)$
return $((Y_C, 0), y_C)$
Lock (from state $i-1$ to i)
8 : **Call** 2P-CLRAS.NewSW **Obtain** $(S^i_{AB}, (S^i_A, w^i_A), P^i_A)$
Call 2P-CLRAS.NewSW **Obtain** $(S^i_{AB}, (S^i_B, w^i_B), P^i_B)$
9 : **Call** 2P-CLRAS.NewSW **Obtain** $(S^i_{BC}, (S^i_{B'}, w^i_{B'}), P^i_{B'})$
Call 2P-CLRAS.NewSW **Obtain** $(S^i_{BC}, (S^i_C, w^i_C), P^i_C)$
10 : 2P-CLRAS.PSign$_{Y_B}$ **Obtain** $\hat{\sigma}^{*i}_{AB}$
2P-CLRAS.PSign$_{Y_B}$ **Obtain** $\hat{\sigma}^{*i}_{AB}$
11 : 2P-CLRAS.PSign$_{Y_C}$ **Obtain** $\hat{\sigma}^{*i}_{BC}$
2P-CLRAS.PSign$_{Y_C}$ **Obtain** $\hat{\sigma}^{*i}_{BC}$
Unlock (Success, all on-path channels updated to i)
12 : 2P-CLRAS.Adapt$_{y_c}$; **Obtain** $\hat{\sigma}^i_{BC}$
13 : $\hat{\sigma}^i_{vk_{BC}}$
14 : Channel Ch_{BC} updated to state i
15 : payment success
16 : 2P-CLRAS.Ext; **Obtain** y_C
17 : 2P-CLRAS.Adapt$_{y_C+y_B}$; **Obtain** $\hat{\sigma}^i_{AB}$
18 : $\hat{\sigma}^i_{vk_{AB}}$
19 : Channel Ch_{AB} updated to state i
20 : payment success

Fig. 5. Multi-hop Payment in MoNet

actions (i.e. incoming and outgoing communication) of $\mathcal{P}$ is taken over by $\mathcal{A}$.

Communication Model. We model our system under a synchronous communication network, where a message broadcasted by a participant P at time $\mathcal{T}$ will be reached to all other participants within $\mathcal{T}+\tau_\Delta$, and τ_Δ denotes the network latency. In the ideal world, participants do not communicate, but only receive and forward messages from and to the functionality $\mathcal{F}_{pay}$.

Key Escrow Service. Let $\mathcal{F}_{kes}$ be a Key Escrow Service functionality that maintains a set of active contract instances, Ke:=($Ke.id$, $Ke.key$, $Ke.timer$,$\phi_{ke.id}$), with some attributes, an unique KES identifier, escrowed key, a timer and a verification function, where the escrowed key is defined as a tuple $Ke.key := \{k_a, k_b\}$, and the function $\phi_{ke.id}$ defines the validity of a dispute request message. Let $\mathcal{P}$ be one of two participants who deploy a KES instance together, and $\mathcal{P}'$ be the other. All the attributes should be initiated when deploying a KES instance except for the timer $Ke.timer$. The functionality Ke has four interfaces. *Initialization* initiates a KES space $\mathbb{K}^n_{\leq 0}$ received from environment $\mathcal{E}$. *Deploy* creates a new KES instance agreed by two participants involved in this instance within $2\tau_\Delta$. *Setting Timer* allows one of the participants $\mathcal{P}$ to propose a dispute by setting the timer of a KES instance Ke. Whether the other participant $\mathcal{P}'$ respond or not, Ke will be terminated after time $\tau_{Ke.id}$. Let $Commit_{\mathcal{P}}$ be a commitment for a channel state, where $\phi_{ke.id}(Commit_{\mathcal{P}}) = 1$ indicates a valid channel state agreed by both channel parties and vice versa. *TerminateKe* is called to remove a KES instance from $\mathbb{K}$.

Initialization: The functionality is initialized by a message $(ke_1, \ldots, ke_n) \in \mathbb{K}^n_{\leq 0}$ that describes the initial KES instances space from the environment $\mathcal{E}$. The functionality stores this tuple.

Deploy:Upon Receiving a message (Deploy, Ke.id, Ke.keys($\mathcal{P}$)) (for $Ke.id \notin \mathbb{K}$):
Within $2\tau_\Delta$, if received the message (AddOk, Ke.id, Ke.keys($\mathcal{P}'$)) from $\mathcal{P}'$, let $\mathbb{K} := \mathbb{K} \wedge Ke$;
If not, ignore this message and send (Ke-not-deplyed,Ke.id) to $\mathcal{P}$.

Setting Timer: Upon Receiving a message (KeSetTimer, $Ke.id$, $Commit_{\mathcal{P}}$, $\tau_{Ke.id}$) for $Ke \in \mathbb{K}$ and $Ke.timer = \perp$, if $\phi_{ke.id}(Commit_{\mathcal{P}}) = 1$, let $\mathbb{K} := \mathbb{K} \setminus Ke$ and set $Ke.timer = \tau_{Ke.id}$, then $\mathbb{K} := \mathbb{K} \wedge Ke$:
Option 1): If within time $\tau_{Ke.id} + \tau_\Delta$ after step 1, received message (Resp, $Ke.id$, $Commit_{\mathcal{P}'}$) from $\mathcal{P}'$, if $\phi_{ke.id}(Commit_{\mathcal{P}'}) = 1$, terminates Ke and sends message (KeTerminated, $Ke.id$) to $\mathcal{P}$ and $\mathcal{P}'$ respectively;
Option 2): After $\tau_{Ke.id} + \tau_\Delta$, sends (KeyRelease, $Ke.keys$) to $\mathcal{P}$ privately, and (KeTerminated, $Ke.id$) to both $\mathcal{P}$ and $\mathcal{P}'$ respectively.
Otherwise, ignore the setting timer message, and sends a message (KeTimer-not-set, $Ke.id$) to $\mathcal{P}$.

Terminate:Upon Receiving a message (KeTerminate, Ke.id)(for $Ke.id \in \mathbb{K}$) from $\mathcal{P}$, $\mathcal{F}_{kes}$ forwards this message to $\mathcal{P}'$:
1. After the time $\tau_{Ke.id} + \tau_\Delta$, let $\mathbb{K} := \mathbb{K} \setminus Ke$.
2. Within $\tau_{Ke.id} + \tau_\Delta$, $\mathcal{F}_{kes}$ received a message (KeTerminate-confirm, Ke.id) from $\mathcal{P}'$, let $\mathbb{K} := \mathbb{K} \setminus Ke$, and forwards this message to $\mathcal{P}'$.

Fig. 6. Functionality $\mathcal{F}_{kes}$

Money Mechanics of Monero. Monero uses unspent transaction output (UTXO) model to maintain a non-negative set of tuples $\mathbb{R} := \{(r_i, \mathsf{xmr}_i)\}$, where r_i denotes an address and xmr_i is amount of Monero tokens in this addresses. Let $\phi_M(r_i, \mathsf{xmr}_i)$ be the verification function of (r_i, xmr_i), where $\phi_M(r_i, \mathsf{xmr}_i) = 1$ indicates that the corresponding transaction is valid and $\phi_M(r_i, \mathsf{xmr}_i) = 0$ stands for an invalid transaction.

In the verification level, maintaining a Monero ledger is same as maintaining the UTXO set $\mathbb{R}$. We construct the ledger functionality $\mathcal{F}_M$ in figure 7. $\mathcal{F}_M$ maintains $\mathbb{R}$ by processing the message $Remove$ or Add, which remove or add UTXO from or to $\mathbb{R}$ if the corresponding proof is verified. Otherwise, $\mathcal{F}_M$ ignores this message.

Initialization: The functionality is initialized by a message $((r_1, \mathsf{xmr}_1), \ldots, (r_n, \mathsf{xmr}_n)) \in \mathbb{R}^n_{\leq 0}$ that describes the initial coin distribution and come from the environment $\mathcal{E}$. $\mathcal{F}_M$ stores this tuple.
Remove UTXO: Upon Receiving a message $(Remove, (r_i, \mathsf{xmr}_i))$ (for $(r_i, \mathsf{xmr}_i) \in \mathbb{R}$ and $1 = \phi_M((r_i, \mathsf{xmr}_i)))$, let $\mathbb{R} := \mathbb{R} \setminus r_i$.
Adding UTXO: Upon Receiving a message $(Add, (r_j, \mathsf{xmr}_j))$ (for $(r_j, \mathsf{xmr}_j) \notin \mathbb{R}$ and $1 = \phi_M((r_j, \mathsf{xmr}_j)))$, let $\mathbb{R} := \mathbb{R} \wedge (r_j, \mathsf{xmr}_j)$.

Fig. 7. Ledger Functionality $\mathcal{F}_M$

In our model, the operation to Key Escrow Service and Monero ledger will be transferred to the modification of the KES space and UTXO set $\mathbb{R}$. Participants cannot directly access $\mathbb{K}$ and $\mathbb{R}$. Instead the UTXO set $\mathbb{R}$ is maintained via the KES protocol and ledger protocol in the real world or via the functionality $\mathcal{F}_{pay}$ (see Figure 8) in the ideal world.

B. Ideal Functionality

Let Ch_{AB} := ($Ch.id$, $Ch.Alice$, $Ch.Bob$, $Ch.Bal$, $Ch.State$, $Ch.Ke$, $Ch.\tau$) denote a channel with some attributes, channel identifier, channel parties, channel capacity (consisting of Alice and Bob's balances), channel state (a monotonic increasing number), a KES instance associated with $Ch.id$, and a timer. $Ch.Bal$ is defined as a tuple (r, xmr), which has the same format as an element in $\mathbb{R}$, and $Ch.Bal.r = xmr$. To associate a channel with the corresponding Key Escrow Service but not be identified by a third party observer, we require $Ch.ke = Ke.id \neq Ch.id$. The functionality $\mathcal{F}_{pay}$ maintains a *channel space* $\mathbb{C}$, that consists of some registered MoChannel. The ideal functionality $\mathcal{F}_{pay}$ is specified in Figure 8.

The functionality $\mathcal{F}_{pay}$ offers three interfaces: 1) **Channel Establishment** describes an channel establishment between Alice and Bob with a deployed KES instance Ke. This interface adds a channel instance only if both participants confirm the open channel message. Otherwise, $\mathcal{F}_{pay}$ ignores this message. Usually, channel update can process two types of payments: 1) both channel parties transfer coins for the sake of their own needs, which is proceed in one-round; 2) the channel is on the path of a multi-hop payment, which requires a 2-round (lock-then-unlock) interaction among both channel parties and $\mathcal{F}_{pay}$. For the sake of clarify, we divide them into two interfaces, **Channel Update** and **Payment Routing**. Moreover, *Payment Routing* can handle two scenarios, where the locked payment is successful or canceled. 3) **Channel Closure** describes the channel closure procedure, which can be called unilaterally or bilaterally as per different conditions defined in this interface.

We then discuss how these security properties are captured by functionality $\mathcal{F}_{pay}$.

Guaranteed channel closure. According to functionality $\mathcal{F}_{pay}$, as the interface *Closing a Channel* is called collaboratively or unilaterally, an established channel can be closed even if there is a malicious party in the channel.

Channel Establishment:

Upon receiving a message m:=(*mc-open*, Ke, Ch) from $Ch.\mathcal{P}$, where Ke is initiated with KES id $Ke.id$ and $\mathcal{P}$'s secret key $Ke.key(k_{\mathcal{P}})$, and Ch is initiated with channel id $Ch.id$, both channel parties $Ch.\mathcal{P}$ and $Ch.\mathcal{P}'$, $Ch.\mathcal{P}$'s balance $Ch.Bal(r_{Ch.\mathcal{P}}, xmr_{\mathcal{P}})$, initial channel state $Ch.State(0)$ and KES identifier $Ch.Ke(Ke.id)$ at time $\mathcal{T}$, $\mathcal{F}_{pay}$ removes $(r_{Ch.\mathcal{P}}, xmr_{\mathcal{P}})$ from $\mathbb{R}$, and forwards the message m:=(*mc-open*, Ke, Ch) to $Ch.\mathcal{P}'$.
Within time τ_Δ, if $\mathcal{F}_{pay}$ receives a message (*mc-open*, Ke, Ch), which sets $Ke.key(k_{\mathcal{P}'})$ and $Ch.Bal(r_{Ch.\mathcal{P}'}, xmr_{\mathcal{P}'})$, from $Ch.\mathcal{P}'$, then removes $r_{\mathcal{P}'}$ from $\mathbb{R}$ and adds Ke and Ch to $\mathbb{K}$ and $\mathbb{C}$ respectively. Sending a message (*mc-opened*) to parties $Ch.\mathcal{P}$ and $Ch.\mathcal{P}'$ and to the simulator $\mathcal{S}$.
Otherwise, adds $r_{Ch.\mathcal{P}}$ into $\mathbb{R}$ and output (*mc-not-opened*) to $Ch.\mathcal{P}$.

Channel Update:

Upon receiving a message m:= (*mc-update*, $Ch.id, xmr_\Delta$) from a channel party $\mathcal{P}$, where xmr_Δ is the transaction amount. If $Ch \notin \mathbb{C}$ or $Ch.bal(r_{Ch.\mathcal{P}}) - xmr_\Delta < 0$ then ignores this message. Otherwise, set $Ch.bal(r_{Ch.\mathcal{P}}) = Ch.bal(r_{Ch.\mathcal{P}}) - xmr_\Delta$, $Ch.bal(r_{Ch.\mathcal{P}'}) := Ch.bal(r_{Ch.\mathcal{P}'}) + xmr_\Delta$, and $Ch.State = Ch.State + 1$, replace Ch in $\mathbb{C}$ with newly updated Ch, and sends (*mc-updated*, $Ch.id$) to $\mathcal{P}$ and $\mathcal{P}'$ within τ_Δ.

Payment Routing:

Upon receiving a message m:= (*mc-routepay*, $\mathbb{C}', xmr_\Delta, \{\tau_{id}\}$), where the number of elements in $\mathbb{C}$ equals to the number of elements in $\{\tau_{id}\}$ (e.g. there are n elements in $\mathbb{C}$ and $\{\tau_{id}\}$ respectively), if $\mathbb{C}' \not\subset \mathbb{C}$ or $Ch.bal(r_{Ch.\mathcal{P}}) - xmr_\Delta < 0$ for every $Ch \in \mathbb{C}'$, then ignore this message. Otherwise, sends (*mc-routepay*, $Ch, Ch', \tau_{Ch.id}, \tau_{Ch'.id}, xmr_\Delta$) to $Ch'.\mathcal{P}$ within τ_Δ, where $Ch'.\mathcal{P} = Ch.\mathcal{P}'$ (*Setup*). If $\tau_{id'} < \tau_{id}$, $Ch'.\mathcal{P}$ sends a lock payment message m$'$:= (*mc-routepay-lock*, $Ch'.id', xmr_\Delta, \tau_{id'}$) to $\mathcal{F}_{pay}$ (*Lock*). Otherwise, $Ch'.\mathcal{P}$ ignores this message. The unlock phase is proceed as follows:
1. Within $\tau_{Ch.id} + 2\tau_\Delta$, where $\tau_{Ch.id}$ is the largest number in $\{\tau_{id}\}$, upon receives $2n$ messages (each Ch has two messages) m:= (*mc-routepay-unlock*, $Ch.id$), for $Ch \in \mathbb{C}'$, $\mathcal{F}_{pay}$ sets $Ch.bal(r_{Ch.\mathcal{P}}) := Ch.bal(r_{Ch.\mathcal{P}}) - xmr_\Delta$, $Ch.bal(r_{Ch.\mathcal{P}'}) := Ch.bal(r_{Ch.\mathcal{P}'}) + xmr_\Delta$, and $Ch.State = Ch.State + 1$, and replaces the old Ch in $\mathbb{C}$ with the newly updated Ch.
2. For a channel $Ch \in \mathbb{C}'$, after $\tau_{id} + 2\tau_\Delta$, if received messages m:= (*mc-routepay-cancel*, $Ch.id, Ch.State + 2$) from both $Ch.\mathcal{P}$ and $Ch.\mathcal{P}'$, $\mathcal{F}_{pay}$ sets $Ch.State = Ch.State + 2$, and replaces the old Ch in $\mathbb{C}$ with newly updated Ch.
3. Otherwise, $Ch.\mathcal{P}$ sends a message m:= (*mc-close*, $Ch.id, Ch.State, \tau_{id}$) following the interface Channel Closure.

Channel Closure:

Upon receiving a message m := (*mc-close*, $Ch.id$, $Ch.State$, $Commit_{\mathcal{P}}$, τ_{id}) from $Ch.\mathcal{P}$.
1) If $\phi_{ke.id}(Commit_{\mathcal{P}}) = 1$, $\mathcal{F}_{pay}$ calls $\mathcal{F}_{kes}$.Setting Timer with a message (KeSetTimer, $Ch.Ke.id$, $Commit_{\mathcal{P}}$, $\tau_{Ch.id}$).
Within τ_{id}, $\mathcal{F}_{pay}$ receives (*mc-close-Ok*, $Ch.id$, $Ch.State$, $Commit_{\mathcal{P}'}$) from $Ch.\mathcal{P}'$ and $\phi_{ke.id}(Commit_{\mathcal{P}'}) = 1$, removes Ch from $\mathbb{C}$ and Ke from $\mathbb{K}$, adds $r'_{Ch.\mathcal{P}}$ and $r'_{Ch.\mathcal{P}'}$ to $\mathbb{R}$;
Otherwise, $\mathcal{F}_{pay}$ receives (*mc-key*, $Ch.id$, $Ke.id(Ch.Ke)$, $Ke.key(k_{\mathcal{P}'})$) from $\mathcal{F}_{kes}$ and forwards this message to $Ch.\mathcal{P}$, adds r'_A with $Ch.bal(Ch.\mathcal{P}) + Ch.bal(Ch.\mathcal{P}')$ coin to $\mathbb{R}$, and removes Ch from $\mathbb{C}$ and Ke from $\mathbb{K}$.
2) Else if $\phi_{ke.id}(Commit_{\mathcal{P}}) = 0$: ignore this message and send (*mc-not-closed*, $Ch.id$) to $Ch.\mathcal{P}$.

Fig. 8. Ideal Functionality $\mathcal{F}_{pay}$

Guaranteed payout for honest channel parties. According to functionality $\mathcal{F}_{pay}$, the honest party will be paid no less than his latest balance in the channel.

On-chain unidentifiability. The functionality $\mathcal{F}_{pay}$ follows the original money mechanics of Monero, thus the environment $\mathcal{E}$ cannot distinguish whether the operation over $\mathbb{R}$ is proceed

by $\mathcal{F}_{pay}$ in the ideal world or the protocol in the real world.
Atomicity. The intermediate participants $Ch.\mathcal{P}'$ can unlock a *route-payment* only when he received a message from $\mathcal{F}_{pay}$ denoting that he has paid in the next hop. It implies that all the on-path payments are atomic.
Unlockablity. According to the interface Channel Update, it covers the scenario that a locked payment is succeed or failed, which prevent a channel from being locked forever.
Sender/Receiver privacy. According to the interface *Channel Update*, all the intermediate nodes receive messages with the same format from $\mathcal{F}_{pay}$, and neither of them can inform whether the node connected with him is the sender/receiver.
Path privacy. Similar to the *Sender/Receiver privacy* property, an intermediate node will receive the messages containing only two channels he involved, but no additional information about the entire path.

$\mathcal{F}_{pay}$ captures the all the the security properties defined in Section IV-B, if the Theorem 1 holds.

Theorem 1: Assuming that the signature scheme on Monero and employed by Key Escrow Service are existentially unforgeable against adaptive chosen-message attacks and our CAS scheme is secure, our system running in the $(\mathcal{F}_{kes}, \mathcal{F}_M)$-hybrid world emulates an ideal functionality $\mathcal{F}_{pay}$ w.r.t Monero ledger $\mathcal{L}_M$.

Proof: We have already informally argued about the security of our scheme above. We then construct a simulator $\mathcal{S}$ in the ideal world to observe then emulate the behavior of some fixed adversary $\mathcal{A}$ in the real world. $\mathcal{S}$ generates the secret-public key pairs and initial witness-statement pairs for each participants. $\mathcal{S}$ then sends these public keys to each corrupted participant with his secret key, and the initial statements to two corrupted participants who share a channel together with their initial witnesses respectively. We assume that honest party will response a message once he received, while the corrupted party can decide when the message are delivered within a time τ_Δ. $\mathcal{S}$ starts from the channel establishment as an honest party, and watches the instruction of $\mathcal{A}$ to the corrupt parties. To make it impossible to distinguish between the simulated and the real execution, $\mathcal{S}$ forwards the message from $\mathcal{F}_{kes}$, $\mathcal{F}_M$, $\mathcal{F}_{pay}$ and other honest parties to the corrupted parties.
a) **Channel Establishment**. The simulation of this part: $\mathcal{S}$ simulates the ledger functionality and KES functionality, plays the role of a honest party $Ch.\mathcal{P}$ sending message (*mc-open*,Ke, Ch) to the functionality $\mathcal{F}_{pay}$, which removes the UTXO $r_{Ch.\mathcal{P}}$ from $\mathbb{R}$. When received another open channel message from $Ch.\mathcal{P}'$, $\mathcal{F}_{pay}$ adds Ke and Ch to $\mathbb{K}$ and $\mathbb{C}$ respectively (or adds $r_{Ch.\mathcal{P}}$ to $\mathbb{R}$ if not received the message from $Ch.\mathcal{P}'$).
b) **Channel Update**. This part starts when $\mathcal{S}$ receives a message (*mc-update*, $Ch.id, xmr_\Delta$) from $\mathcal{E}$, and forwards the message to party $Ch.\mathcal{P}$. We discuss two cases of $\mathcal{P}$ below:
1. $Ch.\mathcal{P}$ is honest. $Ch.\mathcal{P}$ sends a message meaning that he wants to updates the channel. Within time τ_Δ, $\mathcal{S}$ forwards the message to $\mathcal{F}_{pay}$, who sends a confirmation message to $Ch.\mathcal{P}$ indicating $Ch.\mathcal{P}'$'s agreement on the update request.
2. $Ch.\mathcal{P}$ is honest, but $Ch.\mathcal{P}'$ is corrupted. $Ch.\mathcal{P}$ sends a message meaning that he wants to updates the channel. Within time τ_Δ, $\mathcal{S}$ forwards the message to $\mathcal{F}_{pay}$, and $Ch.\mathcal{P}'$ does not send the message to confirm this update. Whether this update is beneficial to $Ch.\mathcal{P}$ or $Ch.\mathcal{P}'$, if $Ch.Bal(r_{Ch.P'}) - xmr_\Delta > 0$, $\mathcal{S}$ sends the confirmation message in the name of $Ch.\mathcal{P}'$ as he knows the private keys and witnesses of all the parties and the channel updated. If he uses a previous state to close Ch, $\mathcal{S}$ will correct their balances and $Ch.\mathcal{P}$ loses coins. Otherwise, if $Ch.Bal(r'_{Ch.P}) - xmr_\Delta < 0$ and $Ch.\mathcal{P}'$ does not confirm this message, then $\mathcal{S}$ does not either.
c) **Payment Routing**. This part starts when $\mathcal{S}$ receives a message (*mc-routepay*, $Ch.id, xmr_\Delta, \tau_{id}$) from $\mathcal{E}$, and forwards the message to P, we consider the following cases:
1. P is the sender. P sends some messages to setup a multi-hop payment path. $\mathcal{S}$ forwards these messages to the receiver and all the intermediate participants respectively. Notice that $\mathcal{P}'$ is on the next hop of the payment path and shares a channel Ch with $\mathcal{P}$. $\mathcal{P}'$ replies confirm payment message within the time τ_Δ. $\mathcal{F}_{pay}$ receives all lock payment messages within the next time τ_Δ, all the on-path payments are succeed.
2. $Ch.\mathcal{P}$ is an honest participant, but $Ch.\mathcal{P}'$ is a corrupted receiver. $Ch.\mathcal{P}$ and $Ch.\mathcal{P}'$ sends a message to confirm a payment lock within the channel between $Ch.\mathcal{P}$ and $Ch.\mathcal{P}'$ However, $Ch.\mathcal{P}'$ does not unlock this payment within $\tau_{id_{\mathcal{P},\mathcal{P}'}}, +\tau_\Delta$. This will be corrected by $\mathcal{S}$, who sends a unlock payment message to $Ch.\mathcal{P}$ in the name of $Ch.\mathcal{P}'$.
d) **Channel Closure**. This part starts when $\mathcal{S}$ receives a message (*mc-close*, $Ch.id$, $Ch.State$, $Commit_{\mathcal{P}}$, τ_{id}) from $\mathcal{E}$, we consider the following cases:
1. $\mathcal{P}$ proposes a channel closure request. $\mathcal{P}$ sends a message to $\mathcal{F}_{kes}$ for closing the channel, which will be forwarded to $\mathcal{P}'$ within the time τ_Δ. $\mathcal{P}'$ then confirm or update the request. Once the close channel request is updated by $Ch.\mathcal{P}'$, and $\mathcal{S}$ forwards $\mathcal{P}'$ response to $\mathcal{F}_{kes}$.
2. $\mathcal{P}$ proposes a channel closure request, but $\mathcal{P}'$ is corrupted. $\mathcal{P}$ sends a message to $\mathcal{F}_{kes}$ for closing the channel, which will be forwarded to $\mathcal{P}'$ within the time τ_Δ. $\mathcal{P}'$ ignores this close channel request, $\mathcal{S}$ will respond in the name of $\mathcal{P}'$. ∎

VI. PERFORMANCE

We develop MoNet over Monero as a proof of concept implementation to evaluate its efficiency.

A. Evaluation

Implementation. We implement 2P-CLRAS scheme over Monero in golang using the libraries, moneroutil [17], emmy [18], and kyber [19]. The implementation takes about 800 lines of code (LoC) over these libraries. All experiments are run on a macOS with processor 2.6 GHz 6-Core Intel Core i7 and memory 16GB 2400 MHz DDR4.

As per MoNet, the message complexity within a channel in each phase includes the number of *on-chain transactions*, *signatures* and *off-chain messages*.

The number of on-chain transactions. Establishing a channel requires 1 transaction on Monero and Ethereum respectively, and no on-chain transaction is required on both Monero and Ethereum for processing an off-chain payment (updating the channel). Routing a payment in the best case does not require any on-chain transaction on neither Monero nor Ethereum, but requires 1 on-chain transaction on Monero and 2 on-chain transactions on Ethereum for the worst case. It requires 1 transaction on Monero and Ethereum respectively if both channel parties close the channel collaboratively or an additional transaction on Ethereum under a dispute scenario.

The number of signatures and off-chain messages. As per the 2P-CLRAS scheme in Algorithm 2, some algorithms requires both signers' interaction, and a single interaction has two messages exchange between them. For example, 2P-CLRAS.PSign has two interactions with 4 messages between both signers, and each of 2P-CLRAS.JGen, 2P-CLRAS.SWGen and 2P-CLRAS.NewSw requires 1 interaction with 2 messages. Signatures are required for each off-chain message, adaptor signatures, and on-chain transactions. According to Figure 4 and 5: there are 10 off-chain messages, 1 on-chain transactions with two signatures from both channel parties, and 1 adaptor signature at the channel establishment phase, thus it requires 13 signatures; there are 4 off-chain messages and an adaptor signature at the channel update phase, thus it requires 5 signatures; there are 7 off-chain messages and an adaptor signature when routing a payment, thus it requires 8 signatures (We count the number of messages delivered within a channel but do not include the messages received from the sender in the count); there are 2 off-chain messages with 2 signatures to close a channel (the signature required by the on-chain transaction is created in each channel update phase as an adaptor signature, thus we do not count it in).

Computation time. Our evaluation of 2P-CLRAS scheme shows that the computation time of $SWGen(\cdot)$ is about $3.5ms$, $NewSW(\cdot)$ is about $30ms$, $PSign(\cdot)$ is about $3.5ms$, $Adapt(\cdot)$ is about $198ns$, $PVrfy(\cdot)$ is about $3.4ms$, and $CVrfy(\cdot)$ is about $330ms$.

The computation time of processing an off-chain payment equals to the time of executing the algorithms, including $NewSW(\cdot)$, $PSign(\cdot)$, $PVrfy(\cdot)$, and $CVrfy(\cdot)$, which is about $367ms$. As the general network latency is about $60ms$ for 4G WAN and internet connections[4], it requires about $427ms$ to process a transaction within a channel. MoChannel improves the throughput on Monero from 1000[5] tps to $2.34D$, where D is the number of channels opened on Monero. For example, as of Jan 2022, there are more than $80,000$ channels[6] in the Lightning Network for Bitcoin. If the MoChannel is of the same scale, it has the potential to provide a throughput of over $180,000$ TPS.

Communication Overhead. We measure the communication overhead as the size of messages that two parties need to exchange for each off-chain payment. The to be exchanged messages size, including an adaptor signature, a newly generated statement and the proof on two consecutive statements, is about 18 KB.

Optimization. We can further optimize MoChannel by pre-computing a batch of statement-witness pairs. The optimized channel update requires only the creation and verification of an adaptor signature. We present the comparison in processing an off-chain payment between the original and optimized MoChannel in Table I. The computation time of creating and verifying an off-chain payment is about 6.9ms, thus it requires 66.9ms to process an off-chain payment within a channel under the 4G WAN network latency, which is $6.4\times$ faster than the non-optimized version (441ms). If at the same scale of lightning network, the optimized MoChannel can handle about 14.9 transactions per channel per second, and throughput is more than $1,100,000$ tps, which could reach the same level of lightning network ($1,000,000$ tps [7]).

TABLE I
PERFORMANCE COMPARISON.

	Original MoChannel	Optimization
Creation	33.5ms	3.5ms
Verification	333.4ms	3.4ms
Throughput	$180,000$ tps	$1,100,000$ tps

In the optimized MoChannel, it requires only about 0.03 KB exchanged data in processing an off-chain transaction. We evaluate the optimized MoChannel with 100 transactions by using the pre-computation. The result shows that if two channel parties pre-compute 100 payment sessions, it requires about 0.08ms for each party to create 100 witness-statement pairs and the proof on two consecutive witnesses, and about 3.46s to verify these witness-statement pairs consecutiveness. The size of all proof is about 1.76 MB.

We also evaluate the performance of **routing a payment via multi-hop MoChannel** described in Section IV-C. Our implementation simulates the optimistic scenario, where the receiver unlocks the a payment by himself. Within a channel, locking a payment means creating a locked 2P-CLRAS scheme. The performance of a multi-hop payment in a single channel (with pre-computation) is concluded in Table II. Routing a multi-hop payment by using MoNet can be processed within $68.68ms{\cdot}n_h$ via n_h hops with a 4G WAN latency 60ms.

TABLE II
PERFORMANCE OF A MULTI-HOP MOCHANNEL PAYMENT.

	the Optimized MoChannel
Setup	0.25ms
Lock	4.78ms
Unlock	3.65ms

For the sake of exploring the feasibility of our system, we also provide a proof of concept implementation for Key Escrow Service contract in Ethereum using Truffle [20], a development framework of Ethereum smart contract. The evaluation shows that, it costs 127869 gas to deploy a Key Escrow Service contract, about 49801 gas to retrieve both parties' funds without dispute, and about 123412 gas to process the

---

[4] Data collected from https://www.sas.co.uk/learning/complete-guide-to-4g-wan

[5] Data collected from https://alephzero.org/blog/what-is-the-fastest-blockchain-and-why-analysis-of-43-blockchains/ on 23 Aug, 2021

[6] Data collected from https://txstats.com/dashboard/db/lightning-network?orgId=1.

[7] Data collected from https://medium.com/coinmonks/how-does-bitcoin-get-scalable-with-the-lightning-network-63591040462c

channel dispute (including the scenario that closing a channel with dispute and routing a payment with dispute) on Ethereum.

ACKNOWLEDGEMENT

This work was partially supported by the Australian Research Council (ARC) under project DE210100019 and project DP220101234.

REFERENCES

[1] S. A. K. Thyagarajan, G. Malavolta, F. Schmidt, and D. Schröder, "Paymo: Payment channels for monero," *IACR Cryptol. ePrint Arch.*, p. 1441, 2020.

[2] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sánchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," IEEE Security and Privacy 2022, 2022.

[3] J. Poon and T. Dryja, "The Bitcoin lightning network: scalable off-chain instant payments," 2016.

[4] I. Tsabary, M. Yechieli, and I. Eyal, "MAD-HTLC: because HTLC is crazy-cheap to attack," *CoRR*, vol. abs/2006.12031, 2020.

[5] P. Moreno-Sanchez, A. Blue, D. V. Le, S. Noether, B. Goodell, and A. Kate, "DLSAG: non-interactive refund transactions for interoperable payment channels in monero," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, vol. 12059. Springer, 2020, pp. 325–345.

[6] L. Aumayr, S. A. K. Thyagarajan, G. Malavolta, P. Monero-Sánchez, and M. Maffei, "Sleepy channels: Bitcoin-compatible bi-directional payment channels without watchtowers," *CCS'22*, 2022.

[7] Z. Sui, J. K. Liu, J. Yu, M. H. Au, and J. Liu, "AuxChannel: Enabling efficient bi-directional channel for scriptless blockchains," in *AsiaCCS'22*, 2022.

[8] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[9] A. Erwig, S. Faust, K. Hostáková, M. Maitra, and S. Riahi, "Two-party adaptor signatures from identification schemes," in *Public Key Cryptography (1)*, ser. Lecture Notes in Computer Science, vol. 12710. Springer, 2021, pp. 451–480.

[10] J. K. Liu, V. K. Wei, and D. S. Wong, "Linkable spontaneous anonymous group signature for ad hoc groups," in *Australasian Conference on Information Security and Privacy*. Springer, 2004, pp. 325–335.

[11] J. Camenisch and A. Lysyanskaya, "A formal treatment of onion routing," in *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, ser. Lecture Notes in Computer Science, V. Shoup, Ed., vol. 3621. Springer, 2005, pp. 169–187.

[12] M. Stadler, "Publicly verifiable secret sharing," in *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, ser. Lecture Notes in Computer Science, U. M. Maurer, Ed., vol. 1070. Springer, 1996, pp. 190–199.

[13] B. Schoenmakers, "A simple publicly verifiable secret sharing scheme and its application to electronic," in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 1666. Springer, 1999, pp. 148–164.

[14] M. P. Jhanwar, "A practical (non-interactive) publicly verifiable secret sharing scheme," in *ISPEC*, ser. Lecture Notes in Computer Science, vol. 6672. Springer, 2011, pp. 273–287.

[15] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[16] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 106–123.

[17] Paxos, "moneroutil," 2017.

[18] Xlab, "Library for zero-knowledge proof based applications (like anonymous credentials)," 2020.

[19] dedis, "Advanced crypto library for the go language," https://github.com/dedis/kyber, 2020.

[20] C. S. Inc., "Truffle Suite."

```
$\mathsf{cOneway}_{\mathcal{A},R}(\lambda)$:
    $(Y^0, y^0) \leftarrow \mathsf{SWGen}(\lambda)$;
    $((Y^{i+1}, y^{i+1}), P^{i+1})$
        $\leftarrow \mathsf{NewSW}((Y^i, y^i), pp)$;
        $\mathsf{CVrfy}((Y^i, Y^{i+1}), P^{i+1}) = 1$;
    $(Y^*, y^*) \leftarrow \mathcal{A}(Y^{i+1}, y^{i+1})$;
    If $\mathsf{CVrfy}(Y^{i+1}, Y^*, P^{i+1}) = 1$ :
        return 1;
Else return 0;
```

```
$\mathsf{cVrfy}_{\mathcal{A},R}(\lambda)$:
    $(Y^0, y^0) \leftarrow \mathsf{swGen}(\lambda)$;
    $((Y^{i+1}, y^{i+1}), P^{i+1}) \leftarrow$
        $\mathsf{NewGen}((Y^i, y^i), pp)$;
    $\mathsf{CVrfy}((Y^i, Y^{i+1}), P^{i+1}) = 1$;
    $((Y^*, y^*), P^*) \overset{R}{\leftarrow} \mathcal{A}(Y^i, y^i)$;
    If $\mathsf{CVrfy}(Y, Y^*, P^*) = 1$ :
        return 1;
Else return 0;
```

Fig. 9. Experiments $\mathsf{cOneway}_{\mathcal{A},R}$ and $\mathsf{cVrfy}_{\mathcal{A},R}(\lambda)$

APPENDIX

This section formally defines the three properties, *consecutiveness* and *consecutive verifiability*, *one-wayness* of VCOF.

Definition 3 (Consecutiveness): The two statement-witness pairs (Y^i, y^i) and (Y^{i+1}, y^{i+1}) are consecutive, if

$$(Y^0, y^0) \leftarrow \mathsf{SWGen}(\lambda)$$
$$((Y^{i+1}, y^{i+1}), P^{i+1}) \leftarrow \mathsf{NewSW}((Y^i, y^i), pp)$$

Definition 4 (Consecutive Verifiability): A VCOF is consecutive verifiable, if for every probabilistic polynomial-time adversary $\mathcal{A}$, the probability of running the experiment $\mathsf{CVrfy}_{\mathcal{A},R}$, defined in Fig. 9, is $Pr[\mathsf{CVrfy}_{\mathcal{A},R}(\lambda) = 1] \leq negl(\lambda)$, where the probability of $Y^{i+1} = Y^{i+1*}$ is a negligible value $\frac{1}{|\Delta_Y|}$.

Definition 5 (One-wayness): A VCOF satisfies one-wayness for a PPT adversary $\mathcal{A}$, if the advantage $Adv = Pr[\mathsf{cOneway}_{\mathcal{A},R}(\lambda) = 1] \leq \mathsf{negl}(\lambda)]$, where the experiment $\mathsf{cOneway}_{\mathcal{A},R}$ is defined in Fig 9.

The consecutiveness of the VCOF is straightforward. We skip the proof of consecutiveness and present proofs of *consecutive verifiability* and *one-wayness*.

Lemma A.1: A VCOF is consecutive verifiable, if the embedded proof system (P_c, V_c) is secure.

Proof. Suppose that there is a PPT adversary $\mathcal{A}$, who can break the consecutive verifiability security of VCOF, we construct a simulator $\mathcal{B}$ to break the embedded proof system of VCOF. First, there is a given statement-witness pair (Y, y) to $\mathcal{B}$, and $\mathcal{B}$ forwards (Y, y) to $\mathcal{A}$.

For all the consecutiveness oracle queries of a q_o-tuple $\{(\tilde{Y}, \tilde{y})\}$ from $\mathcal{A}$ on a statement-witness pair $(Y, y) \in \{(\tilde{Y}, \tilde{y})\}$, $\mathcal{B}$ picks a random (Y'^*, y'^*) from their corresponding domain and constructs a P^* by using (Y, y) and (Y'^*, y'^*), then returns $((Y'^*, y'^*), P^*)$ to $\mathcal{A}$. We have $(Y', y') \leftarrow f_c((Y, y), pp)$, and the probability of (Y'^*, y'^*) equals to (Y', y') is negligible.

If $\mathcal{A}$ can output $((Y'^*, y'^*), P^*)$, which satisfies $1 \leftarrow V_c((Y, Y'^*), P^*)$, $\mathcal{B}$ can produce not only a P on (Y, y) and (Y', y'), but also a P^* on (Y, y) and (Y'^*, y'^*), which breaks the soundness requirement of the embedded proof system. ∎

Lemma A.2: A VCOF is one-way, if the embedded consecutive function f_c is one-way.

Proof. Suppose that there is a PPT adversary $\mathcal{A}$, who can break the one-wayness of VCOF, we construct a simulator $\mathcal{C}$ to break the security of the embedded consecutive function f_c. First, given a statement-witness pair (Y', y'), which is generated from an pair (Y, y), where y is unknown to $\mathcal{C}$, who forwards (Y', y') to $\mathcal{A}$. $\mathcal{C}$ forwards (Y', y') to $\mathcal{A}$.

For all the consecutiveness oracle queries of a q_c-tuple $\{(\tilde{Y'}, \tilde{y'})\}$ from $\mathcal{A}$ on a statement-witness pair $(Y', y') \in \{(\tilde{Y'}, \tilde{y'})\}$, $\mathcal{C}$ picks (Y^*, y^*) randomly from their corresponding domain, and return them to $\mathcal{A}$.

If $\mathcal{A}$ outputs a $((Y^*, y^*), P^*)$ on a given $((Y', y'), P, \Delta)$, where $(Y', y') \notin \{(\tilde{Y'}, \tilde{y'})\}$ and $1 \leftarrow \mathsf{CVrfy}((Y^*, Y'), P^*)$. It implies that $\mathcal{C}$ can output a solution (Y^*, y^*) for the given instance (Y', y'), where $f_c((Y^*, y^*), pp) = (Y', y')$. Thus, $\mathcal{C}$ breaks the security of the embedded one-way function f_c. ■