DLSAG: Non-Interactive Refund Transactions For Interoperable Payment Channels in Monero

Pedro Moreno-Sanchez[1], Arthur Blue[2], Duc V. Le[3], Sarang Noether[4], Brandon Goodell[4], Aniket Kate[3]

[1] TU Wien pedro.sanchez@tuwien.ac.at
[2] Independent Researcher
[3] Purdue University {le52,aniket}@purdue.edu
[4] Monero Research Lab {sarang,surae}@getmonero.org

Abstract. Monero has emerged as one of the leading cryptocurrencies with privacy by design. However, this comes at the price of reduced expressiveness and interoperability as well as severe scalability issues. First, Monero is restricted to coin exchanges among individual addresses and no further functionality is supported. Second, transactions are authorized by linkable ring signatures, a digital signature scheme used in Monero, hindering thereby the interoperability with virtually all the rest of cryptocurrencies that support different digital signature schemes. Third, Monero transactions require an on-chain footprint larger than other cryptocurrencies, leading to a rapid ledger growth and thus scalability issues. This work extends Monero expressiveness and interoperability while mitigating its scalability issues. We present *Dual Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups (DLSAG)*, a linkable ring signature scheme that enables for the first time *non-interactive refund transactions* natively in Monero: DLSAG can seamlessly be implemented along with other cryptographic tools already available in Monero such as commitments and range proofs. We formally prove that DLSAG provides the same security and privacy notions introduced in the original linkable ring signature [31] namely, unforgeability, signer ambiguity, and linkability. We have evaluated DLSAG and showed that it imposes even slightly lower computation and similar communication overhead than the current digital signature scheme in Monero, demonstrating its practicality. We further show how to leverage DLSAG to enable off-chain scalability solutions in Monero such as payment channels and payment-channel networks as well as atomic swaps and interoperable payments with virtually all cryptocurrencies available today. DLSAG is currently being discussed within the Monero community as an option for adoption as a key building block for expressiveness, interoperability, and scalability.

1 Introduction

Bitcoin fails to provide meaningful privacy guarantees as largely demonstrated in the literature [10,11,27,35,43,47]. In this state of affairs, Monero appeared in the cryptocurrency landscape with the distinguishing factor of adopting privacy by

a design principle, combining for the first time *stealth address* [45], *linkable ring signatures* [31], *cryptographic commitments* [39] and *range proofs* [16]. As of the time of writing, Monero has been regularly among the top 15 cryptocurrencies in market capitalization, has catered more than 6 million transactions since its creation [8], and is the most popular CryptoNote-style cryptocurrency [1]. Currently, the Monero blockchain processes around 4000 daily transactions and Monero coins are part of a daily trade volume of more than 76M USD [2]. Monero has, however, significant room for improvement. First, Monero suffers from *reduced expressiveness*: While cryptocurrencies like Bitcoin or Ethereum enable somewhat complex policies to spend coins (e.g., a coin can be governed by script-based rules), Monero only supports coins governed with (mostly a single) private key, reducing the functionality to simple transfer of coins with no policy associated with it.

Cryptocurrencies such as Bitcoin and Ethereum overcome this lack of expressiveness by adding a script language at the cost of fungibility [9] (i.e., transaction inputs/outputs can be easily distinguished by their script) and interoperability as those script languages are not compatible with each other. Thus, it is interesting to include new policies to spend Monero coins *cryptographically*, instead of including a scripting language that hampers fungibility and interoperability.

Second, Monero suffers from similar *scalability issues* as Bitcoin [18]: The permissionless nature of the Monero consensus algorithm limits the block rate to one block every two minutes on average. In fact, the scalability problem in Monero is more pressing. The crucial privacy goal in Monero relies on well-established cryptographic constructions to homogenize transactions: linkable ring signatures are used to obfuscate what public key corresponds to the signer of a transaction while commitment schemes and range proofs are leveraged to hide the exchanged amounts, ensure transaction validity and the expected coin supply. These key design choices make Monero transactions require higher on-chain footprint than transactions in other cryptocurrencies. Although used only for less than five years, the Monero blockchain has currently a size of 63.75 GB and grows at around 500MB per month [4].

Given this trend, it would be interesting to enable payment channels and payment channel networks [6, 32, 42] in Monero, a scalability solution already adopted in Bitcoin and Ethereum where the transaction rate is no longer limited by the global consensus but rather by the latency among the two users involved in a given payment. However, this is far from trivial as current payment-channel networks are built upon script languages (e.g., hash-time lock contract) or digital signatures schemes such as ECDSA or Schnorr that are not available in Monero. Leveraging these techniques in Monero would hamper its fungibility.

In summary, the current state of affairs in Monero with respect to the reduced expressiveness, lack of interoperability, and severe scalability issues calls for a solution. Adopting solutions provided in other cryptocurrencies like Bitcoin and Ethereum is not seamlessly possible as they are not backwards compatible with Monero. Moreover, as aforementioned, the inclusion of a scripting language would hamper the fungibility and interoperability of Monero.

Our contributions. In this work, we present *Dual Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups (DLSAG)*, the linkable ring signature scheme for Monero that improves upon the lack expressiveness, interoperability, and scalability guarantees in Monero. In particular:

– **Expressiveness**. We formalize DLSAG (Section 3), a new linkable ring signature scheme that relies only on cryptographic tools already available in Monero and improves its expressiveness. In a bit more detail, DLSAG enables for the first time that Monero coins can be spent with one of two signing keys, depending on the relation between a time flag and the height of the current block in the Monero blockchain.

– **Scalability**. We describe how to leverage the DLSAG signatures to encode for the first time non-interactive refund transactions in Monero, where Alice can pay to Bob a certain amount of coins redeemable by Bob before a certain time in the future. After such time expires, the coins can be refunded to Alice. Refund transactions are the building block that opens the door for the first time to scalability solutions based on payment channels for Monero (Section 6). In particular, we describe how to build uni-directional payment channels, payment-channel networks, off-chain conditional payments and atomic swaps.

– **Interoperability**. We further show that it is possible to combine the aforementioned payment channels protocols with the corresponding ones in other cryptocurrencies, making thereby Monero interoperable (Section 6).

– **Formal analysis**. We formally prove that DLSAG achieves the security and privacy goals of interest for linkable ring signatures, namely, unforgeability, signer ambiguity, and linkability as introduced in [31] (Section 3).

– **Implementation and adoption**. We have implemented DLSAG and evaluated its performance (Section 4) showing that it imposes a single bit more of communication overhead and smaller computation overhead as the current digital signature scheme in Monero, demonstrating thus its practicality. In fact, DLSAG is a new result that paves the way in practice towards an expressiveness and scalability solution urgently needed in Monero to improve its integration in the cryptocurrency landscape. DLSAG is actively being discussed within the Monero community as an option for adoption [7, 37] and it is compatible with other CryptoNote-style cryptocurrencies [1].

Comparison with related work. Poelstra introduced the notion of *Scriptless Scripts* [41] as a means of encoding somewhat limited smart contracts that no longer require the Bitcoin scripting language. Malavolta et al. [33] formalized this notion and extended it to support Schnorr and ECDSA digital signatures. In this work, we instantiate the notion of Scriptless Scripts to realize conditional payments compatible with DLSAG and the current Monero protocol. Bitcoin payment channels [5, 19, 42] have been presented in the literature as a scalability solution for the Bitcoin blockchain. Bitcoin payment channels have been then leveraged to build payment-channel networks in academia [24, 26, 32] and in industry [6, 40, 42]. However, none of these solutions are compatible with the current Monero. They rely on either Bitcoin script [6, 32, 42], ZCash script [24], Ethereum contracts [26] or Schnorr signature scheme [40], none of which are

available in Monero. Similarly, Bitcoin scripts have been leveraged to construct an atomic swap protocol [15]. We, instead, present a payment-channel network and atomic swap protocols that no longer require scripting language, and it is compatible with Monero. Goodell and Noether have proposed threshold signatures [23] for Monero whereas Libert et al. [30] proposed a logarithmic-size ring signature from the DDH assumption; although interesting, they do not address the expressiveness, interoperability and scalability issues considered in this work.

2 Background

Notation. We denote by λ the security parameter. We denote by $\mathsf{poly}(\lambda)$ a polynomial function in λ and $\mathsf{negl}(\lambda)$ a negligible function in λ. We denote by $\mathbb{G}$ a cyclic group of prime order q and by g we denote a fixed generator of such group. We denote by $(\mathsf{pk}, \mathsf{sk})$ a pair of public and secret keys. We denote by $\mathbf{pk}$ an array of public keys. We use letters A to Z to identify users in a protocol. We denote by $\texttt{XMR}$ the Monero coins. Finally, we consider two hash functions: (i) $\mathsf{H_s}$ takes as input a bitstring and outputs a scalar (i.e., $\mathsf{H_s} : \{0,1\}^* \to \mathbb{Z}_q$); (ii) $\mathsf{H_p}$ takes as input a bitstring and outputs an element of $\mathbb{G}$ (i.e., $\mathsf{H_p} : \{0,1\}^* \to \mathbb{G}$).

Transactions. A Monero transaction [45] is divided in *inputs* and *outputs*. They are defined in terms of tuples of the form $(\mathsf{pk}, \textsc{Com}(\gamma), \Pi\text{-}amt)$ where pk denotes a fresh public key, $\textsc{Com}(\gamma)$ denotes a *cryptographic commitment* [39] to the amount γ and $\Pi\text{-}amt$ denotes a *range proof* [16] that certifies that the committed amount is within a range $[0, 2^k]$ where k is a system parameter. In particular, each input consists of a set of such tuples while each output consists of a single tuple. The set of public keys included in an input is called a *ring*. Finally, the transaction includes a digital signature σ for each input.

In the illustrative example shown in Fig. 1, we assume that Alice has previously received 5 $\texttt{XMR}$ in the public key $\mathsf{pk_A}$. We also assume that she wants to pay Bob 4 $\texttt{XMR}$. For that, Alice first should get Bob's public key ($\mathsf{pk_B}$) and a fresh public key for herself ($\mathsf{pk'_A}$) to keep the change amount. Second, Alice should choose a set of $n-1$ output tuples $\{(\mathsf{pk}_i, \textsc{Com}(v_i), \Pi\text{-}amt_i)\}$ already available in the Monero blockchain to complete the input. Finally, Alice should create a valid signature of the transaction content using the ring $(\mathsf{pk}_1, \ldots, \mathsf{pk}_{n-1}, \mathsf{pk_A})$ and her private key $\mathsf{sk_A}$. For that, she uses a linkable ring signature scheme.

Inputs:
[0] $\{(\mathsf{pk}_1, \textsc{Com}(v_1), \Pi\text{-}amt_1), \ldots, (\mathsf{pk}_{n-1}, \textsc{Com}(v_{n-1}), \Pi\text{-}amt_{n-1}), (\mathsf{pk_A}, \textsc{Com}(5), \Pi\text{-}amt_{\mathsf{A}})\}$
Outputs:
[0] $\mathsf{pk_B}$, $\textsc{Com}(4)$, $\Pi\text{-}amt_{\mathsf{B}}$; [1] $\mathsf{pk'_A}$, $\textsc{Com}(1)$, $\Pi\text{-}amt'_{\mathsf{A}}$
Authorizations:
[0] σ

Fig. 1: Illustrative example of a (simplified) Monero transaction. Alice ($\mathsf{pk_A}$) contributes 5 $\texttt{XMR}$ to pay 4 $\texttt{XMR}$ to Bob ($\mathsf{pk_B}$) and get 1 $\texttt{XMR}$ back ($\mathsf{pk'_A}$). Finally, the transaction is authorized with a ring signature σ from the input ring.

Linkable ring signatures. The signature scheme used in Monero is an instantiation of the *Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups* (LSAG)[5] signature scheme [31]. We recall the definition of LSAG in Definition 1. Here, we explicitly add a generic definition of the linking algorithm which was briefly mentioned in [31].

Definition 1 (LSAG [31]). *An LSAG signature scheme is a tuple of algorithms (*KeyGen*,* Sign*,* Vrfy*,* Link*) defined as follows:*

- $\mathsf{sk}, \mathsf{pk} \leftarrow \textsc{KeyGen}(\lambda)$*: The* KeyGen *algorithm takes as input the security parameter* λ *and outputs a pair of private key* sk *and public key* pk*.*
- $\sigma \leftarrow \textsc{Sign}(\mathsf{sk}, \mathbf{pk}, m)$*: The* Sign *algorithm takes as input a private key* sk*, a list* $\mathbf{pk}$ *of* n *public keys which includes the one corresponding to* sk*, a message* m *and outputs a signature* σ*.*
- $b \leftarrow \textsc{Vrfy}(\mathbf{pk}, m, \sigma)$*: The* Vrfy *algorithm takes as a public key list* $\mathbf{pk}$*, a message* m *and a signature* σ*, and returns* 1 *if* $\exists \mathsf{sk}, \mathsf{pk} \leftarrow \textsc{KeyGen}(\lambda)$ *s.t.* $\mathsf{pk} \in \mathbf{pk}$ *and* $\sigma := \textsc{Sign}(\mathsf{sk}, \mathbf{pk}, m)$*. Otherwise, it returns* 0*.*
- $b \leftarrow \textsc{Link}((\mathbf{pk}_1, m_1, \sigma_1), (\mathbf{pk}_2, m_2, \sigma_2))$*: The* Link *algorithm takes as input two triples* $(\mathbf{pk}_1, m_1, \sigma_1)$ *and* $(\mathbf{pk}_2, m_2, \sigma_2)$*. The algorithm outputs 1 if* $\exists(\mathsf{sk}, \mathsf{pk}) \leftarrow \textsc{KeyGen}(\lambda)$ *s.t.* $\mathsf{pk} \in \mathbf{pk}_1$*,* $\mathsf{pk} \in \mathbf{pk}_2$*,* $\sigma_1 := \textsc{Sign}(\mathsf{sk}, \mathbf{pk}_1, m_1)$ *and* $\sigma_2 := \textsc{Sign}(\mathsf{sk}, \mathbf{pk}_2, m_2)$*. Otherwise, the algorithm outputs* 0*.*

Apart from the straightforward correctness definition, Liu et al. [31] define three security and privacy goals for a LSAG signature scheme. We present them here informally and defer their formal description to Section 3.2.

– **Unforgeability**: The adversary without access to the secret key should not be able to compute a valid signature σ on a message $\mathtt{m}$.

– **Signer ambiguity**: Given a valid signature σ on a message $\mathtt{m}$, the adversary should not be able to determine better than guessing what public key within the ring corresponds to the secret key used to create the signature.

– **Linkability**: Given two rings $\mathbf{pk}_1$, $\mathbf{pk}_2$, two valid signatures σ_1, σ_2 in two messages $\mathtt{m}_1$, $\mathtt{m}_2$, there should exist an efficient algorithm that faithfully determines if the same secret key has been used to create both signatures.

Due to the lack of space, we defer to Appendix A the detailed construction of LSAG used in Monero, and refer to [38] for its security and privacy analysis.

The current LSAG in Monero only supports transfer of coins authorized by a signature, reducing the expressiveness to payments. Adding a script language (as done in Bitcoin or Ethereum) would harm fungibility (i.e., transaction inputs/outputs can be easily distinguished by their script) and interoperability as those languages are not compatible with each other. Instead, in this work we aim to propose a signature scheme for Monero that cryptographically supports more expressive transaction authorization policies, without hampering the security and privacy guarantees of the current digital signature scheme.

[5] Monero in fact uses a matrix version of LSAG (MLSAG) [38] to prove balance without revealing spent ring members. We describe here the simplest LSAG version but our constructions can be trivially extended to support matrix version.

3 Dual-Key LSAG (DLSAG)

3.1 Key ideas and construction of DLSAG

Our approach builds upon a *tuple format* defined as $((\mathsf{pk}_{\mathsf{A},0}, \mathsf{pk}_{\mathsf{B},1}), \textsc{Com}(\gamma), \Pi\text{-}amt, t)$ and that enables to spend it to two different public keys (and potentially two different users) depending on a flag t. A dual-key tuple deviates from the current Monero tuple in two main points (highlighted in blue): (i) it contains two public keys instead of one to identify the two users that can possibly spend the output; and (ii) it includes an additional element t that denotes a switch (e.g., $\mathsf{pk}_{\mathsf{A},0}$ is used if t is smaller than the current block height in the Monero blockchain) between the public keys.

Dual-key tuple format enables the encoding of the logic for a refund transaction. In the sample tuple shown above, assume that t signals that $\mathsf{pk}_{\mathsf{A},0}$ must be used. Then Alice must choose a ring of the form $(\mathbf{pk}_0, \mathbf{pk}_1)$, containing $(\mathsf{pk}_{\mathsf{A},0}, \mathsf{pk}_{\mathsf{B},1})$ at some position, and sign with the secret key sk_A, that is, the secret key corresponding to the public key $\mathsf{pk}_{\mathsf{A},0}$. Conversely, if t signals that $\mathsf{pk}_{\mathsf{B},1}$ must be used, Bob can then sign with sk_B instead. Note that if a single user knows both sk_A and sk_B, such an user can always use a dual-key tuple independently of the value t.

The remaining step is to design a linkable ring signature scheme that supports this new tuple format. This, however, requires to address the following challenges.

Key-image mechanism. The ring signature scheme currently used in Monero achieves linkability by publishing the key-image constructed from the single public key. For instance, Alice produces a signature with sk_A; the signature will contain the key-image $\mathcal{I} = \mathsf{H}_\mathsf{p}(\mathsf{pk}_\mathsf{A})^{\mathsf{sk}_\mathsf{A}}$. If Alice signs again with sk_A, the same key-image would be computed and this can be detected. To mimic this behavior while handling the dual-key tuple format, the challenge is to define a single key-image that uniquely identifies a pair of public keys $(\mathsf{pk}_0, \mathsf{pk}_1)$ and yet can be computed knowing only one of the signing keys sk_b. Similar to the Diffie-Hellman key exchange mechanism [20], our approach redefines the key-image as $\mathcal{J} = g^{\mathsf{sk}_0 \cdot \mathsf{sk}_1}$. This intuitively fulfills the expected requirements: (i) the knowledge of sk_b suffices to compute $\mathcal{J} := \mathsf{pk}_{1-b}^{\mathsf{sk}_b}$; (ii) it uniquely identifies $(\mathsf{pk}_0, \mathsf{pk}_1)$ since $\mathsf{pk}_{1-b}^{\mathsf{sk}_b} = \mathsf{pk}_b^{\mathsf{sk}_{1-b}}$.

Hardening key-image linkability. The aforementioned key-image definition allows to link the pair of public keys $(\mathsf{pk}_0, \mathsf{pk}_1)$. However, it is crucial to make the key-image unique not only to the pair of public keys but also to the output that contains them itself. Otherwise, one of the users could create another dual-key tuple with the same pair of public keys, create a signature with it (and thus a key-image), and effectively make the funds in the original tuple unspendable since in Monero every key-image is only allowed to appear once. That can be mitigated by introducing a random unique identifier, m, to each output, and this identifier can be included in the computation of the key-image without violating the security and privacy guarantees of the signature scheme. In Monero, such an unique identifier can be constructed by hashing the transaction that included the output and the output's position in the transaction.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \textsc{KeyGen}(\lambda)$: Choose $\mathsf{sk}_0, \mathsf{sk}_1$ uniformly at random from $\mathbb{Z}_q$, m as a bitstring chosen uniformly at random from $\{0,1\}^\lambda$. Set both $\mathsf{pk}_b := g^{\mathsf{sk}_b}$ for $b \in \{0,1\}$. Output $\mathsf{sk} = (\mathsf{sk}_0, \mathsf{sk}_1), \mathsf{pk} = (\mathsf{pk}_0, \mathsf{pk}_1, m)$.
- $\sigma \leftarrow \textsc{Sign}(\mathsf{sk}_b, \mathbf{pk}, \mathtt{tx})$: Parse: $((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, m_1), \ldots, (\mathsf{pk}_{n,0}, \mathsf{pk}_{n,1}, m_n)) \leftarrow \mathbf{pk}$. Sample $s'_0, s_1, \ldots, s_{n-1}$ from $\mathbb{Z}_q$. Compute:

$$\mathcal{J} := \mathsf{pk}_{n,1-b}^{m_n \cdot \mathsf{sk}_b};\ L_0 := g^{s'_0};\ R_0 := \mathsf{pk}_{n,1-b}^{s'_0 \cdot m_n};\ h_0 := \mathsf{H_s}(\mathtt{tx}||L_0||R_0);$$

  Then, for $i \in \{1, \ldots, n-1\}$, compute the following sequences:

$$L_i := g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_{i-1}};\ R_i := \mathsf{pk}_{i,1-b}^{s_i \cdot m_i} \cdot \mathcal{J}^{h_{i-1}};\ h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Now, solve for s_0 such that $\mathsf{H_s}(\mathtt{tx}||g^{s_0} \cdot \mathsf{pk}_{n,b}^{h_{n-1}}||\mathsf{pk}_{n,1-b}^{s_0 \cdot m_n} \cdot \mathcal{J}^{h_{n-1}}) = h_0$. For that, we get $s_0 = s'_0 - h_{n-1} \cdot \mathsf{sk}_b$. Return: $\sigma = (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{J}, b)$.
- $b' \leftarrow \textsc{Vrfy}(\mathbf{pk}, \mathtt{tx}, \sigma)$: Parse
  $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{J}, b) \leftarrow \sigma;\ ((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, m_1), \ldots, (\mathsf{pk}_{n,0}, \mathsf{pk}_{n,1}, m_n)) \leftarrow \mathbf{pk}$

  For $i \in \{1, \ldots, n\}$, compute the sequences:

$$L_i := g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_{i-1}};\ R_i := \mathsf{pk}_{i,1-b}^{s_i \cdot m_i} \cdot \mathcal{J}^{h_{i-1}};\ h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Return 1 if $h_0 = h_n$. Otherwise, return 0.
- $b \leftarrow \textsc{Link}((\mathbf{pk}_1, \mathtt{tx}_1, \sigma_1), (\mathbf{pk}_2, \mathtt{tx}_2, \sigma_2))$: If $(\textsc{Vrfy}(\mathbf{pk}_1, \mathtt{tx}_1, \sigma_1) \wedge \textsc{Vrfy}(\mathbf{pk}_2, \mathtt{tx}_2, \sigma_2)) = 0$: return 0. Else, parse: $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{J}_1, b_1) \leftarrow \sigma_1$ and $(s'_0, s'_1, \ldots, s'_{n-1}, h'_0, \mathcal{J}_2, b_2) \leftarrow \sigma_2$. Return 1 if $\mathcal{J}_1 = \mathcal{J}_2$, and 0 otherwise.

Fig. 2: Construction of DLSAG. For ease of exposition, we assume that the secret key sk_b corresponds with the public key $pk_{n,b}$. As noted before, the position of the true signer's public key is chosen uniformly random.

Thus, we may view the rings used in DLSAGs as consisting of unique triples, $(\mathsf{pk}_0, \mathsf{pk}_1, m)_{[1,n]}$, and we define the *dual key-image* to be $\mathcal{J} := g^{m_j \cdot \mathsf{sk}_{j,0} \cdot \mathsf{sk}_{j,1}}$, for some $j \in [1, n]$ corresponding to the position of the true signer in the ring.

The rest is to follow the idea of the Monero LSAG modified to support the new linkability tag. Fig. 2 introduces the details of the DLSAG construction.

3.2 Security analysis

In this part, we state the theorems for the security of DLSAG. Due to the lack of space, we defer their formal definitions and the security proofs to Appendix B.

Theorem 1 (DLSAG unforgeability). *DLSAG signature scheme is existentially unforgeable against adaptive chosen-plaintext attack according to Definition 3 provided that the One-More Discrete Logarithm (OMDL) problem* [6] *is hard, under the random oracle model.*

[6] The One-More Discrete Logarithm hardness assumption is defined in [12].

Theorem 2 (DLSAG signer ambiguity). *DLSAG achieves signer ambiguity according to Definition 4 provided that the Decisional Diffie-Hellman assumption (DDH) is hard, under the random oracle model.*

Theorem 3 (DLSAG linkability). *DLSAG achieves linkability as defined in Definition 5 provided that the OMDL problem is hard, under the random oracle model.*

Further security and privacy analysis. We have analyzed the security and privacy of the digital signature scheme. Recent privacy studies on Monero [28, 36] show that composition of several transactions (and thus signatures) can lead to new threats and leakages. In particular, we observe that DLSAG allows an observer to track when the receiver spends his coin if the sender use the stealth address mechanism used in Monero to generate the one time address for the receiver. Such linkability issue can be mitigated if the receiver spends his coins as soon as he receives it. We defer to Appendix E the discussion on this and others venues for future work in security and privacy.

4 Implementation and performance analysis

Implementation. We developed a prototypical C++ implementation of DLSAG to demonstrate the feasibility of our DLSAG construction in comparison with the Monero LSAG. We have implemented DLSAG and LSAG using the same cryptographic library, `libsodium` [3], and cryptographic parameters (i.e. the ed25519 curve) as currently used in Monero.

Testbed. We conducted our experiments on a commodity desktop machine, which is equipped with Intel(R) Core(TM) i5-7400 CPU @ 3.00 GHz CPU, 12GB RAM. In these experiments, we focus on evaluating the overhead of DLSAG over LSAG in terms of computation time and signature size.

Computation time. The results depicted in Table 1 show that the running time of DLSAG is practically the same as the running time of LSAG in both signing and verifying algorithms. Thus, DLSAG could be included in Monero without incurring computation overhead. We estimate that the computation time for DLSAG is systematically a 7% smaller than that of LSAG. One of the main reasons is that in DLSAG, we eliminate the use of hash-to-point evaluations (e.g., as required in the old key-image mechanism). More specifically, for ring of

	LSAG		**DLSAG**	
Ring Size	Sign	Vrfy	Sign	Vrfy
5	1.929	1.634	1.771	1.499
10	3.863	3.569	3.665	3.398
15	5.873	5.577	5.625	5.352
20	8.045	7.750	7.516	7.248

Table 1: Running time (in milliseconds) of DLSAG and LSAG for different ring sizes

size n, both DLSAG signing and verifying algorithms incur approximately $\approx 4n$ group operations and n hash-to-scalar evaluations while in LSAG, signing and verifying algorithms require additional n hash-to-point evaluations, which we see as the main factor for the differences in running time. Therefore, our evaluation shows that DLSAG does not impose any computation overhead in comparison to current LSAG. In fact, if adopted, DLSAG might even slightly improve the signature creation and verification times.

Signature size. Here, we studied the overhead in terms of signature size, and thus indirectly the communication overhead imposed by DLSAG. We observed that in comparison to the LSAG signature, the signature of DLSAG has just one extra parity bit to indicate the position of the public key needed for verification (i.e., either pk_0 or pk_1). This short signature size can be achieved at the cost of higher tuple footprint. However, DLSAG enables off-chain payments and thus reducing the number of on-chain tuples required overall. In summary, this evaluation shows that DLSAG can be deployed in practice with almost no communication overhead and yet improves the scalability of Monero since it enables off-chain operations as we discuss later in this paper.

5 DLSAG in Monero

Bootstrapping DLSAG in Monero. DLSAG can be seamlessly added into Monero. First, Monero regularly performs network upgrades for consensus rules and protocol improvements that allows for the integration of new functionality such as DLSAG. Second, it is possible to have transactions that mix LSAG with DLSAG. A mixed transaction will contain a LSAG signature for each single-key input and a DLSAG signature for each input in the dual-key format. In fact, both formats only differ in the number of public keys and the inclusion of an extra field (i.e. flag t). Thus, Monero operations and verifications on the commitment and range proofs remain compatible.

Fungibility. Different tuple formats coexisting on the blockchain may be detrimental to fungibility. For instance, miners might decide to stop mining certain transactions depending on the tuple format chosen. In order to mitigate that, we note that direct transfers using single-key tuples can easily be simulated by setting the two public keys of the dual-key tuples to belong to a single user. Thus, the fungibility of Monero may not be hampered with dual-key tuples only.

Backwards compatible timelock processing. Dual-key tuples contain a flag t in the clear. We envision that this flag is implemented in Monero as a block height, so that given a pair $(\mathsf{pk}_0, \mathsf{pk}_1)$, pk_0 can be used before block t is mined and pk_1 is used afterwards. Although it is unclear and an interesting future research work, it could be possible that the different t values leak enough information for an adversary to break privacy, in the spirit of Monero attacks shown in the recent literature [28, 36]. Given that, in this work we proactively propose an alternative timelock processing scheme that allows to have indistinguishable timeouts. This scheme, added as an extension to the dual-key tuple format and

Inputs:
[0] $((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}), \textsc{Com}(v_1), \Pi\text{-}amt_1, \textsc{Com}(t_1), \Pi\text{-}time_1), \ldots,$ $(\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}), \textsc{Com}(v_{n-1}), \Pi\text{-}amt_{n-1}, \textsc{Com}(t_{n-1}), \Pi\text{-}time_{n-1}),$ $((\mathsf{pk}_\mathsf{A}, \mathsf{pk}'_\mathsf{A}), \textsc{Com}(10), \Pi\text{-}amt_\mathsf{A}, \textsc{Com}(t_\mathsf{A}), \Pi\text{-}time_\mathsf{A})$
Outputs:
[0] $(\mathsf{pk}_\mathsf{B}, \mathsf{pk}''_\mathsf{A})$, $\textsc{Com}(10)$, $\Pi\text{-}amt'_\mathsf{A}$, $\textsc{Com}(t_\mathsf{B})$, $\Pi\text{-}time_\mathsf{B}$
Authorizations:
[0] σ^0

Fig. 3: A simplified Monero transaction using dual-key tuples and hidden timelocks.

DLSAG signature scheme helps to maintain the fungibility of Monero. We note that this timelock processing could be of individual interest as timelocks are part of virtually all cryptocurrencies.

The core idea of the timelock processing scheme is as follows. Instead of including t in the clear, each output contains a Pedersen commitment to that value $\textsc{Com}(t, r_1)$, where r_1 is the mask value which is included along with a proof (Π-*time*) that t is in the range $[0, 2^k]$. Now, one can prove that t has expired as follows: pick t' such that $t < t'$. If T is a block height such that $t' < T$, that would tell the miner that indeed $t < T$, and such a transaction will be mined only if the appropriate key is being used. In order to convince the miner that the relation $t < t'$ holds, the signer picks a random mask r_2 and forms the Pedersen commitment $\textsc{Com}(t' - t, r_2)$, and includes this commitment along with the value t', a range proof Π-*time* to prove that $t' - t$ is in range $[0, 2^k]$ and other ring member information.

5.1 Putting all together

In this section, we use the illustrative example in Fig. 3 to revisit the processes of spending and verifying a transaction assuming that Monero includes dual-key tuples, supports DLSAG signature scheme and the timelock processing scheme.

Assume that Alice has previously received 10 `XMR` in the public key $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}'_\mathsf{A})$ (i.e., input [0]). Assume that she wants to pay Bob for a service worth 10 `XMR` with a certain timeout t_B. Thus, either Bob claims the 10 `XMR` before t_B or Alice gets them refunded at the address pk''_A. For this, Alice can create the transaction shown in Fig. 3. After this transaction is added to the Monero blockchain, Bob can get his coins by spending the output [0]. In the following, we describe the generation of this transaction and how it can be verified by the interested party (e.g., miners).

Transaction generation. Assume that Alice wants to spend coins held in $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}'_\mathsf{A})$. First, Alice invokes the Sign algorithm for DLSAG on input $(\mathsf{sk}_\mathsf{A}, ((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}), \ \ldots, (\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}), (\mathsf{pk}_\mathsf{A}, \mathsf{pk}'_\mathsf{A}), \mathtt{tx})$, obtaining thereby a signature σ. Second, she has to use the timelock processing mechanism to prove that t_A has not expired. For that, she creates the tuple $(\textsc{Com}(t_\mathsf{A}), t'_\mathsf{A}, \textsc{Com}(t'_\mathsf{A} - t_\mathsf{A}), \Pi\text{-}time_\mathsf{A})$ as mentioned above. Similar to the problem of publishing commitment of amounts, publishing $\textsc{Com}(t_\mathsf{A})$ would reveal what public key within the

ring is being used, hindering thus signer ambiguity. Fortunately, we can adapt the approach in Monero to handle value commitments for $\textsc{Com}(t_{\mathsf{A}})$ (Appendix C).

Transaction validation. Every miner can validate the inclusion of Alice's transaction in a block at height T by checking whether $t'_{\mathsf{A}} < T$. If so, he proceeds to verify the range proofs for the commitment values. Next, he verifies that the DLSAG signature is correct using the corresponding VRFY algorithm. Finally, the miner checks that the dual ring signature is also correct using the VRFY algorithm as defined in DLSAG. We remind that using the extension of DLSAG as defined in the appendix, the miner would have to verify only one dual signature, using the DLSAG verification algorithm.

6 Applications in Monero enabled by DLSAG

6.1 Building blocks

Zero-knowledge proofs (ZKP). A ZKP system allows a prover to prove to a verifier the validity of a statement without revealing more information than the pure validity of the statement itself. In particular, a ZKP is composed by two algorithms (ZKPROVE, ZKVERIFY) defined as follows. First, the prove algorithm $\Pi \leftarrow \text{ZKPROVE}(st, w)$ takes as input a statement st and a witness w and returns a proof Π. The verification algorithm $\top, \bot \leftarrow \text{ZKVERIFY}(st, \Pi)$ takes as input a statement st and returns $\top$ if Π is a valid proof for st. Otherwise, it returns $\bot$. We require a ZKP that fulfills the zero-knowledge, soundness and completeness properties [22].

In our constructions, we instantiate it with the sigma protocol [46], using the Fiat-Shamir heuristic to make it non-interactive [21]. For simplicity of notation, we denote by $\Pi(\{x\}, (X, g))$ a proof of the fact that $X = g^x$ where X and g are public and x is maintained private from the verifier. Moreover, we denote by $\Pi(\{x\}, (X, g) \wedge (X', g'))$ a proof of the fact that $X = g^x$ and $X' = g'^x$, where x is maintained private from the verifier and the rest of values are public.

2-of-2 DLSAG signatures. Assume that Alice and Bob want to jointly pay a receiver R for a service. We require that Alice and Bob jointly create a ring signature that spends γ $\mathtt{XMR}$ from a dual-key $(\mathsf{pk}_{\mathsf{AB},0}, \mathsf{pk}_{\mathsf{AB},1})$, distributing them as γ' to $(\mathsf{pk}_{\mathsf{R},0}, \mathsf{pk}_{\mathsf{R},1})$ and the remaining $\gamma - \gamma'$ back to themselves. For that, Alice and Bob execute $\text{2OF2RSSIGN}(\mathsf{pk}_{\mathsf{AB},b}, [\mathsf{sk}_{\mathsf{AB},b}]_{\mathsf{A}}, [\mathsf{sk}_{\mathsf{AB},b}]_{\mathsf{B}}, \mathtt{tx})$ protocol, as shown in Fig. 4. The 2OF2RSSIGN protocol largely resembles the SIGN algorithm from the DLSAG scheme. The main difference comes in the computation of $h_0 = \mathsf{H_s}(\mathtt{tx}||g^r||\mathsf{pk}^{rm}_{\mathsf{AB},1-b})$ where the targets g^r and $\mathsf{pk}^{rm}_{\mathsf{AB},1-b}$, as well as their shared key-image $\mathcal{J}_{AB}$, have to be jointly constructed by Alice and Bob.

This protocol results in Alice and Bob obtaining their share of the signature $[\sigma]_{\mathsf{A}}$ and $[\sigma]_{\mathsf{B}}$ that they must combine to complete the final ring signature $\sigma := ([s_0]_{\mathsf{A}} + [s_0]_{\mathsf{B}}, s_1, \ldots s_{n-1}, h_0, (\mathcal{J}_{\mathsf{A}} \cdot \mathcal{J}_{\mathsf{B}}))$. Interestingly, Alice (and similarly Bob) can verify that $[\sigma]_{\mathsf{B}}$ is indeed a share of a valid signature σ by computing

$$g^{([s_0]_{\mathsf{A}}+[s_0]_{\mathsf{B}})} \stackrel{?}{=} \frac{(R_{\mathsf{A}} \cdot R_{\mathsf{B}})}{\mathsf{pk}^{h_{n-1}}_{\mathsf{AB},b}}, \text{ where } R_{\mathsf{A}} = g^{[s'_0]_{\mathsf{A}}} \text{ and } R_{\mathsf{B}} = g^{[s'_0]_{\mathsf{B}}}$$

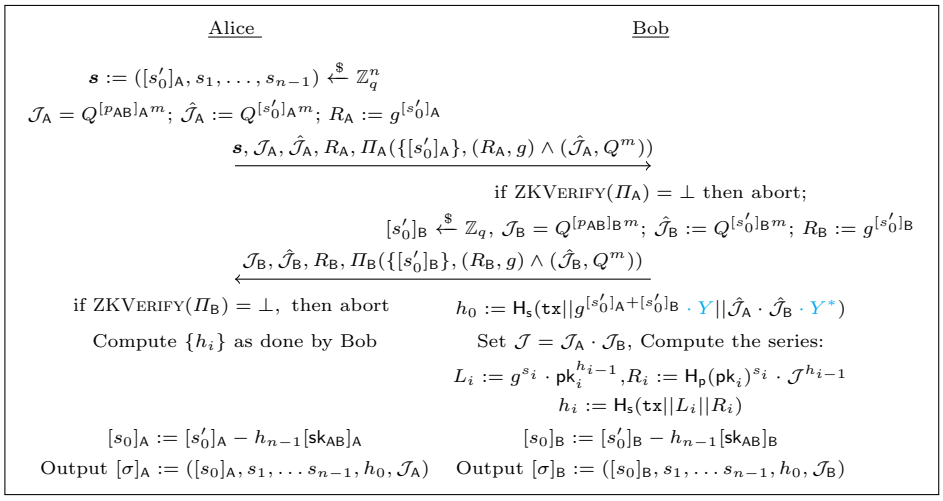

Fig. 4: Description of the protocol 2of2RSSign($\mathsf{pk}_{\mathsf{AB}}, [\mathsf{sk}_{\mathsf{AB}}]_\mathsf{A}, [\mathsf{sk}_{\mathsf{AB}}]_\mathsf{B}, \mathtt{tx}$), where $\mathsf{pk}_{\mathsf{AB}}$ denotes a one-time address shared between Alice and Bob, $[\mathsf{sk}_{\mathsf{AB}}]_\mathsf{A}$, $[\mathsf{sk}_{\mathsf{AB}}]_\mathsf{B}$ denote the Alice and Bob shares of the private key for P_{AB}, and $\mathtt{tx}$ denotes the transaction to be signed. The ring used was: $((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}), \ldots, (\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}), (\mathsf{pk}_{\mathsf{AB},0}, \mathsf{pk}_{\mathsf{AB},1}))$ and omitted for readability. The pseudocode in light blue denotes the changes required to implement the 2of2RSSignCond($\mathsf{pk}_{\mathsf{AB}}, [\mathsf{sk}_{\mathsf{AB}}]_\mathsf{A}, [\mathsf{sk}_{\mathsf{AB}}]_\mathsf{B}, \mathtt{tx}, Y, Y^*$) protocol, that additionally takes as input two group elements of the form $Y := g^y$ and $Y^* := \mathsf{pk}_{\mathsf{AB},1}^{ym}$.

6.2 Payment channels in Monero

Background. A *payment channel* enables several payments between two users without committing every single one of them to the blockchain. For this reason, *payment channels* are being widely developed as a scalability solution in cryptocurrencies such as Bitcoin [42]. However, the conceptual differences between Monero and Bitcoin hinder a seamless adoption of Bitcoin payment channels in Monero. We instead leverage the refund transactions described in this work.

The lifecycle of a payment channel between Alice and Bob consists of three steps. First, Alice and Bob must *open* a payment channel by including an on-chain transaction that transfers XMR from Alice into a public key $\mathsf{pk}_{\mathsf{AB}}$ whose private key $\mathsf{sk}_{\mathsf{AB}}$ is shared by Alice and Bob, that is, Alice holds $[\mathsf{sk}_{\mathsf{AB}}]_\mathsf{A}$ and Bob holds $[\mathsf{sk}_{\mathsf{AB}}]_\mathsf{B}$ such that $[\mathsf{sk}_{\mathsf{AB}}]_\mathsf{A} + [\mathsf{sk}_{\mathsf{AB}}]_\mathsf{B} = \mathsf{sk}_{\mathsf{AB}}$. Second, they perform *off-chain payments* by locally adjusting how many XMR each of them gets from the shared address. Finally, they must *close* the payment channel by submitting a second on-chain transaction that distributes the XMR from the shared address to Alice and Bob as defined by the last balance agreed off-chain. Thus, payment channels require only two on-chain transactions (open and close) but allow for many off-chain payments to take place during its life time. In the following, we show our design of payments channel using the building blocks explained in Section 6.1.

Open a payment channel. Assume that Alice holds γ XMR in a dual key $(\mathsf{pk}_{\mathsf{A},0}, \mathsf{pk}_{\mathsf{A},1})$ and she wants to create a payment channel with Bob. First, she transfers γ XMR to a dual key of the form $(\mathsf{pk}_{\mathsf{AB}}, \mathsf{pk}'_\mathsf{A})$ and sets the timeout to a

desired block height t. This way, if Bob never manages to coordinate with Alice to spend from $\mathsf{pk}_{\mathsf{AB}}$, she will automatically regain control of her funds after that height, eliminating the need for a separate refund transaction. On the other hand, if Bob has received any off-chain transfers from $\mathsf{pk}_{\mathsf{AB}}$, he needs to be sure to put the final balance on chain before the height t is reached.

Off-chain payments. Assume that Alice wants to pay $\gamma' < \gamma$ `XMR` to Bob using the aforementioned payment channel. For that, Alice transfers γ' `XMR` from $(\mathsf{pk}_{\mathsf{AB}}, \mathsf{pk}_{\mathsf{A}})$ to a Bob's dual address $(\mathsf{pk}_{\mathsf{B},0}, \mathsf{pk}_{\mathsf{B},1})$ and the change $\gamma - \gamma'$ `XMR` back to an Alice's dual address $(\mathsf{pk}_{\mathsf{A}.0}, \mathsf{pk}_{\mathsf{A},1})$. As the `XMR` are being spent from the shared address $\mathsf{pk}_{\mathsf{AB}}$, the transaction must be signed by both users to be valid. The cornerstone of payment channels, however, is that only Alice signs `otx` and gives her share of the signature $[\sigma]_{\mathsf{A}}$ to Bob, who can in turn verify it. At this point, Bob publish the transaction and get the γ' `XMR` before the timelock expires. Instead, Bob locally stores `otx` and the corresponding signature $[\sigma]_{\mathsf{A}}$ until either Bob receives another off-chain payment for a value higher than γ' `XMR` or the channel is about to expire.

Close channel. The channel between Alice and Bob can be closed for two reasons. First, Bob does not wish to receive more off-chain payments from Alice. Then, assume that Bob got a pair $(\mathtt{tx}, [\sigma]_{\mathsf{A}})$, where `tx` is the last agreed balance. He can simply complete σ' with his own share $[\sigma']_{\mathsf{B}}$ and publish the transaction. Second, if the timelock included in the deposit transaction expires, and Alice regains control of the original γ `XMR` deposited.

6.3 Conditional payments in Monero

A *conditional payment* only becomes valid if the receiver can give the solution to a cryptographic problem such as finding the preimage of a hash value or solving an instance of the discrete logarithm problem. Conditional payments open many new applications such as payment-channel networks as well as atomic swaps and therefore we consider them of independent interest.

We aim to simulate the following *Discrete-log Timelock Contract (DTLC)* contract defined on a group element $Y = g^y$, an amount γ of `XMR` and a timeout t. **DTLC (Alice, Bob, Y, γ, t):** (i) If Bob produces a value y such that $g^y = Y$ before t days, Alice pays Bob γ `XMR`; (ii) If t elapses, Alice gets the γ `XMR` back.

Here, we describe our implementation of the **DTLC** contract by means of an example. Assume that Alice and Bob got γ `XMR` in a dual address $(\mathsf{pk}_{\mathsf{AB}}, \mathsf{pk}_{\mathsf{A}})$ created, for instance, in the opening of a payment channel between Alice and Bob. Further assume that Alice wants to perform a conditional payment (`ctx`) for $\gamma' < \gamma$ `XMR` to Bob conditioned on him knowing the discrete logarithm of Y.

Alice and Bob sign `ctx` using the 2OF2RSSIGNCOND protocol (Fig. 4, light blue pseudocode) on the condition Y. The cornerstone of this protocol is to imagine that there are three users instead of two that jointly execute the protocol: Alice, who contributes $([s'_0]_A, [\mathsf{sk}_{\mathsf{AB}}]_{\mathsf{A}})$, Bob, who contributes $([s'_0]_B, [\mathsf{sk}_{\mathsf{AB}}]_{\mathsf{B}})$, and a "third user" who contributes (y, y). After running the protocol, Alice and Bob obtain $[\sigma]_{\mathsf{A}}$ and $[\sigma]_{\mathsf{B}}$, but they also require y to complete the signature.

Therefore, after running the 2OF2RSSIGNCOND protocol, Bob gives his signature share $[\sigma]_{\mathsf{B}}$ to Alice who in turn can verify its validity and reply with her signature share $[\sigma]_{\mathsf{A}}$. This exchange, in this order, ensures that `ctx` is only published if value y is revealed and if the height lock ℓ has not been reached.

Now we note that whenever Bob claims his `XMR` at the `ctx`, he should provide the signature σ that contains $[s'_0]_A + [s'_0]_B + y$, and Bob can do this only if he knows the value y. But as soon as that signature is published, Alice trivially learns y from σ as she already knows $[\sigma]_{\mathsf{A}}$ and $[\sigma]_{\mathsf{B}}$. Additionally, we note that the values y and Y remain invisible, and therefore outside observers cannot use them to link this transaction with any other transactions using the same condition values (e.g. the counterpart transaction in an atomic swap). In fact, this transaction is indistinguishable from non-conditional Monero transactions, contributing thereby to the fungibility of the Monero cryptocurrency.

6.4 Payment-Channel network in Monero

Assume that Alice wants to perform an off-chain payment to Dave using a path of opened payment channels of the form Alice, Bob, Carol, Dave. Such a payment is performed in three phases. First, Dave creates a condition ($Y := g^y, Y^* := \mathsf{pk}^{ym}_{\mathsf{CD},1}$) and communicates the conditions (Y, Y^*) to Alice. Second, Alice creates a conditional payment to Bob under condition (Y, Y^*), who in turn creates a conditional payment to Carol under the same condition, and finally Carol creates the last conditional payment to Dave under condition (Y, Y^*). Finally, in the third phase, Dave reveals y to Carol to pull the coins from her, who in turn, reveals y to Bob and finally Bob to Alice.

We have to overcome a subtle but crucial challenge to make such construction fully compatible with Monero. The problem consists on that the same condition (Y, Y^*) cannot be used by every pair of users in the path: While g is the same for every user, each Y_i^* requires the value y (only known by Dave before the payment is settled) and the dual address $(\mathsf{pk}_{P_iP_{i+1}}, \mathsf{pk}_{P_i})$ that defines each of the payment channels (and therefore only known by the two users sharing the channel). To overcome that, we add an extra round of communication where each pair of users forward to the receiver of the payment their shared address' refund address multiplied by their output identifier (i.e., $\mathsf{pk}_{\mathsf{A}}^{m_{AB}}$ where pk_{A} is the refund address of the pair $(\mathsf{pk}_{\mathsf{AB}}, \mathsf{pk}_{\mathsf{A}})$). Upon reception of these values, the receiver computes the pair (Y, Y_i^*) for each user along with a zero-knowledge proof of the fact that both condition values are constructed as expected. Finally, the receiver sends these conditions and proofs back in the payment path.

Now, before setting the conditional payment, each user must validate the zero-knowledge proof produced by the receiver to ensure that the condition for the incoming payment is built upon the same value y as the condition for the outgoing payment. It is important to note that soundness of the zero-knowledge scheme does not allow Dave to cheat on the proof and still be correctly validated by other users. Otherwise, it could be the situation that an intermediate user loses coins because his outgoing payment goes through but cannot use the same value y for unlocking the incoming payment.

6.5 Atomic swaps

Monero does not support *Hash Timelock Contract* **HTLC** [44], the building block for atomic swaps in other cryptocurrencies. Instead, we leverage DTLC-based conditional payments (Section 6.3) to enable atomic swaps between Monero and other cryptocurrencies. We describe our approach with an example.

Assume that Alice has 1 bitcoin and wants to exchange it by 1 XMR from Bob. For that, Alice first creates a value y and sets $h := \mathsf{H}(y)$, $Y := g^y$, $Y^* := \mathsf{pk}_{\mathsf{AB},1}^{ym}$ She then creates a zero-knowledge proof Π of the fact that the discrete logarithm of Y w.r.t. g and Y^* w.r.t. $\mathsf{pk}_{\mathsf{AB},1}^{m}$ are the same as the pre-image of h. Second, Alice creates a Bitcoin transaction that transfers her 1 bitcoin to Bob using the **HTLC**(Alice, Bob, h, 1, 1 day). Finally, Alice gives h, Y, Y^* and Π to Bob.

The idea now is that Bob creates a Monero conditional payment conditioned on (Y, Y^*), as described in Section 6.3, that transfers his 1 XMR to Alice. However, Bob must first check that indeed the discrete-log of Y and Y^* is also the pre-image of h so that the swap is indeed atomic. Otherwise, Alice could simply claim the 1 XMR from Bob but Bob could not claim the bitcoin from Alice. Bob ensures the atomicity of the swap by checking the validity of the proof Π.

We note that the above protocol requires a zero-knowledge proof protocol such as ZK-Boo [17] or Bulletproofs [16] to prove knowledge of the pre-image of a hash value. We also note that if Schnorr signatures are available in both cryptocurrencies or **HTLC** is substituted by discrete-log based constructions [33], zero-knowledge proofs may not be needed.

7 Concluding remarks and outlook

We present DLSAG, a linkable ring signature scheme that serves as a building block to improve expressiveness, interoperability and scalability in Monero. We have formally proven that DLSAG provides unforgeability, sender ambiguity and linkability. We also evaluate the performance of DLSAG showing that DLSAG provides a single bit of communication overhead while slightly reducing the computation overhead when compared to current LSAG. Moreover, we contribute additional cryptographic schemes (e.g., timelock processing) to help to maintain the fungibility of Monero. DLSAG enables payment channels, payment channel networks, and atomic swaps for the first time in Monero. DLSAG is currently under consideration by Monero researchers as an option for adoption and it is also compatible with other CryptoNote-style cryptocurrencies [1].

In the future, we plan to extend the payment channel construction to support bi-directional payments, to devise new cryptographic primitives to enlarge the set of policies available with DLSAG to spend coins (e.g., merging DLSAG with threshold signatures) and to study whether the principles of DLSAG can be carried over to other linkable ring signatures [25, 29, 30].

Acknowledgments. This work has been partially supported by the Austrian Science Fund (FWF) through the Lisa Meitner program and by the National Science Foundation under grant CNS-1846316.

References

1. Cryptonote currencies, https://cryptonote.org/coins
2. https://coinmarketcap.com/, https://coinmarketcap.com/
3. Libsodium documentation, https://libsodium.gitbook.io/doc/
4. Monero monthly blockchain growth, https://moneroblocks.info/stats/blockchain-growth
5. Payment channels, https://en.bitcoin.it/wiki/Payment_channels
6. Raiden network, https://raiden.network/
7. Research meeting: 18 March 2019, 17:00 UTC, https://github.com/monero-project/meta/issues/319
8. Understanding the structure of Monero's LMDB and how explore its contents using mdb_stat, https://monero.stackexchange.com/questions/10919/understanding-the-structure-of-moneros-lmdb-and-how-explore-its-contents-using
9. What is Fungibility?, https://www.investopedia.com/terms/f/fungibility.asp
10. Androulaki, E., Karame, G.O., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating User Privacy in Bitcoin. In: FC. pp. 34–51 (2013)
11. Barber, S., Boyen, X., Shi, E., Uzun, E.: Bitter to Better — How to Make Bitcoin a Better Currency. In: FC. pp. 399–414 (2012)
12. Bellare, Namprempre, Pointcheval, Semanko: The One-More-RSA-Inversion Problems and the Security of Chaum's Blind Signature Scheme. Journal of Cryptology **16**(3), 185–215 (Jun 2003)
13. Bellare, M., Neven, G.: Multi-signatures in the plain public-key model and a general forking lemma. In: CCS. pp. 390–399 (2006)
14. Bender, A., Katz, J., Morselli, R.: Ring signatures: Stronger definitions, and constructions without random oracles. In: Theory of Cryptography Conference. pp. 60–79. Springer (2006)
15. Bowe, S., Hopwood, D.: Hashed time-locked contract transactions (2017), https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki
16. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short Proofs for Confidential Transactions and More. In: S&P. pp. 315–334 (2018)
17. Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives (2017), https://eprint.iacr.org/2017/279
18. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün Sirer, E., Song, D., Wattenhofer, R.: On Scaling Decentralized Blockchains. In: FC. pp. 106–125 (2016)
19. Decker, C., Wattenhofer, R.: A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In: Stabilization, Safety, and Security of Distributed Systems SSS. pp. 3–18 (2015)
20. Diffie, W., Hellman, M.: New Directions in Cryptography. IEEE Trans. Inf. Theor. **22**(6), 644–654 (Sep 2006)
21. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO. pp. 186–194 (1987)
22. Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. jacm **38**(3), 691–729 (1991)
23. Goodell, B., Noether, S.: Thring Signatures and their Applications to Spender-Ambiguous Digital Currencies. Cryptology ePrint Archive, Report 2018/774 (2018), https://eprint.iacr.org/2018/774

24. Green, M., Miers, I.: Bolt: Anonymous Payment Channels for Decentralized Currencies. In: CCS. pp. 473–489 (2017)
25. Jivanyan, A.: Lelantus: Towards confidentiality and anonymity of blockchain transactions from standard assumptions. Cryptology ePrint Archive, Report 2019/373 (2019), https://eprint.iacr.org/2019/373
26. Khalil, R., Gervais, A.: Revive: Rebalancing off-blockchain payment networks. In: CCS. pp. 439–453 (2017)
27. Koshy, P., Koshy, D., McDaniel, P.: An Analysis of Anonymity in Bitcoin Using P2P Network Traffic. In: FC. pp. 469–485 (2014)
28. Kumar, A., Fischer, C., Tople, S., Saxena, P.: A Traceability Analysis of Monero's Blockchain. In: ESORICS. pp. 153–173 (2017)
29. Lai, R.W.F., Ronge, V., Ruffing, T., Schrder, D., Thyagarajan, S.A.K., Wang, J.: Omniring: Scaling up private payments without trusted setup - formal foundations and constructions of ring confidential transactions with log-size proofs. Cryptology ePrint Archive, Report 2019/580 (2019), https://eprint.iacr.org/2019/580
30. Libert, B., Peters, T., Qian, C.: Logarithmic-size ring signatures with tight security from the ddh assumption. In: Lopez, J., Zhou, J., Soriano, M. (eds.) Computer Security. pp. 288–308. Springer International Publishing, Cham (2018)
31. Liu, J.K., Wei, V.K., Wong, D.S.: Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups. In: Information Security and Privacy. pp. 325–335 (2004)
32. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: CCS. pp. 455–471 (2017)
33. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In: NDSS (Jan 2019)
34. Maxwell, G., Poelstra, A., Seurin, Y., Wuille, P.: Simple Schnorr Multi-Signatures with Applications to Bitcoin. Cryptology ePrint Archive, Report 2018/068 (2018), https://eprint.iacr.org/2018/068
35. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A Fistful of Bitcoins: Characterizing Payments Among Men with No Names. In: IMC. pp. 127–140. IMC '13 (2013)
36. Möser, M., Soska, K., Heilman, E., Lee, K., Heffan, H., Srivastava, S., Hogan, K., Hennessey, J., Miller, A., Narayanan, A., Christin, N.: An Empirical Analysis of Traceability in the Monero Blockchain. PETS **2018**(3), 143 – 163 (2018)
37. Noether, S., Goodel, B.: Dual linkable ring signatures, https://ww.getmonero.org/resources/research-lab/pubs/MRL-0008.pdf
38. Noether, S., Mackenzie, A.: Ring Confidential Transactions. Ledger **1**(0), 1–18 (Dec 2016)
39. Pedersen, T.P.: Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In: CRYPTO. pp. 129–140 (1991)
40. Poelstra, A.: Lightning in scriptless scripts (2017), https://lists.launchpad.net/mimblewimble/msg00086.html
41. Poelstra, A.: Scriptless scripts (2017), https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-03-mit-bitcoin-expo/slides.pdf
42. Poon, J., Dryja, T.: The Bitcoin Lightning Network. Whitepaper (2016), http://lightning.network/
43. Reid, F., Harrigan, M.: An Analysis of Anonymity in the Bitcoin System. In: Security and Privacy in Social Networks, pp. 197–223. New York, NY (2013)
44. Rusty: Lightning Networks Part II: Hashed Timelock Contracts (HTLCs) (2015), https://rusty.ozlabs.org/?p=462

45. van Saberhagen, N.: Cryptonote v 2.0. Whitepaper (2013), `https://cryptonote.org/whitepaper.pdf`
46. Schnorr, C.P.: Efficient signature generation by smart cards. Journal of Cryptology **4**(3), 161–174 (1991)
47. Spagnuolo, M., Maggi, F., Zanero, S.: BitIodine: Extracting Intelligence from the Bitcoin Network. In: FC. pp. 457–468 (2014)

A Linkable ring signature in Monero

Fig. 5 shows the construction of LSAG originally used in the current Monero cryptocurrency.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \textsc{KeyGen}(\lambda)$: Choose sk uniformly at random and set $\mathsf{pk} := g^{\mathsf{sk}}$. Output sk, pk.
- $\sigma \leftarrow \textsc{Sign}(\mathsf{sk}, \mathbf{pk}, \mathtt{tx})$: Parse: $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \leftarrow \mathbf{pk}$. Sample $s'_0, s_1, \ldots, s_{n-1}$ from $\mathbb{Z}_q$. Compute:

$$\mathcal{I} := \mathsf{H_p}(\mathsf{pk}_n)^{\mathsf{sk}}; L_0 := g^{s'_0};\ R_0 = \mathsf{H_p}(\mathsf{pk}_n)^{s'_0};$$
$$h_0 := \mathsf{H_s}(\mathtt{tx}||L_0||R_0).$$

  For $i \in \{1, \ldots, n-1\}$ compute the following series:

$$L_i := g^{s_i} \cdot \mathsf{pk}_i^{h_{i-1}};\ R_i := \mathsf{H_p}(\mathsf{pk}_i)^{s_i} \cdot \mathcal{I}^{h_{i-1}}$$
$$h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Solve for s_0 such that: $\mathsf{H_s}(\mathtt{tx}||g^{s_0} \cdot \mathsf{pk}_n^{h_{n-1}}||\mathsf{H_p}(\mathsf{pk}_n)^{s_0} \cdot \mathcal{I}^{h_{n-1}}) = h_0$. For that, we get that $s_0 = s'_0 - h_{n-1} \cdot \mathsf{sk}$. Return $\sigma = (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$
- $b \leftarrow \textsc{Vrfy}(\mathbf{pk}, \mathtt{tx}, \sigma)$: Parse:

$$(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I}) \leftarrow \sigma,\ (\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \leftarrow \mathbf{pk}$$

  For $i \in \{1, \ldots, n\}$, compute the sequences:

$$L_i := g^{s_i} \cdot P_i^{h_{i-1}};\ R_i := \mathsf{H_p}(P_i)^{s_i} \cdot \mathcal{I}^{h_{i-1}}$$
$$h_i := \mathsf{H_s}(\mathtt{tx}||L_i||R_i)$$

  Return 1 if $h_0 = h_n$. Otherwise, return 0.
- $b \leftarrow \textsc{Link}((\mathbf{pk}_1, \mathtt{tx}_1, \sigma_1), (\mathbf{pk}_2, \mathtt{tx}_2, \sigma_2))$: If $(\textsc{Vrfy}(\mathbf{pk}_1, \mathtt{tx}_1, \sigma_1) \wedge \textsc{Vrfy}(\mathbf{pk}_2, \mathtt{tx}_2, \sigma_2)) = 0$, return 0. Else: parse $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I}_1) \leftarrow \sigma_1$ and $(s'_0, s'_1, \ldots, s'_{n-1}, h'_0, \mathcal{I}_2) \leftarrow \sigma_2$. Return 1 if $\mathcal{I}_1 = \mathcal{I}_2$. Otherwise, return 0.

Fig. 5: Construction of LSAG in Monero [38]. For ease of exposition, in the signing algorithm we assume that the secret key sk corresponds with the n-th public key pk_n. In practice, the position of true signer's public key is chosen uniformly random.

B DLSAG security properties

In this section, we state the definitions of DLSAG unforgeability, signer ambiguity, and linkability. Signer ambiguity and linkability properties are similar to those in LSAG [31], adapted to DLSAG syntax for readability. For the unforgeability property, we used the existential unforgeability of ring signatures with respect to insider corruption introduced in [14]. Then, to prove the security of the DLSAG signature scheme, we outline computational hardness assumptions as well as the general forking lemma [13] used in our proofs in Appendix B.2. Then, we later provide our proofs for stated theorems in Appendix B.3.

B.1 Security properties of DLSAG

Definition 2 (Existential unforgeability of ring signature with respect to insider corruption). *Let λ be a security parameter, let N, q_H, q_S, q_C be natural numbers such that $q_C \leq N \leq \mathsf{poly}(\lambda)$, $1 \leq q_H \leq \mathsf{poly}(\lambda)$, $1 \leq q_S \leq \mathsf{poly}(\lambda)$. Let $(\mathbb{G}, q, g)$ be some group parameters from a Dual LSAG signature scheme* (KeyGen, Sign, Verify, Link)*. Let $\mathcal{O}^C$ be a corruption oracle that can be queried up to q_C times which acts as a discrete logarithm oracle. Let $\mathcal{O}^S$ be a signature oracle that can be queried up to q_S times. Presume $\mathcal{O}^S$ takes as input some ring of public keys $\mathbf{pk}$, message m, signing index ℓ, and parity bit b, and produces as output a valid signature. Let $\mathcal{O}^H$ be a random oracle that can be queried up to q_H times.*

The Dual LSAG signature scheme is said to be existentially unforgeable with respect to insider corruption if any PPT algorithm $\mathcal{A}$ has at most a negligible probability of success in the following game.

1. *The challenger selects a set of N public keys from the Dual LSAG signature scheme key space $\boldsymbol{PK} \leftarrow \{(\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, m_0), \cdots, (\mathsf{pk}_{N,0}, \mathsf{pk}_{N,1}, m_N)\}$ and sends this set to the player $\mathcal{A}$.*
2. *The player is granted access to oracles $\mathcal{O}^C$, $\mathcal{O}^S$, and $\mathcal{O}^H$.*
3. *The player outputs a message m, a ring of public keys $\mathbf{pk} = \{(Y_{1,0}, Y_{1,1}, m'_1), (Y_{2,0}, Y_{2,1}, m'_2), \cdots, (Y_{R,0}, Y_{R,1}, m'_R)\} \subseteq \boldsymbol{PK}$ where $R \geq 1$ and a purported forgery (σ, b).*

The player $\mathcal{A}$ wins if Verify$(\boldsymbol{pk}, m, \sigma) = 1$ *and the following additional success constraints are satisfied:*

- *The keys in $\mathbf{pk}$ are distinct and every key $(Y_{i,0}, Y_{i,1}, m'_i) \in \mathbf{pk}$ satisfies $(Y_{i,0}, Y_{i,1}, m'_i) = (\mathsf{pk}_{j(i),0}, \mathsf{pk}_{j(i),1}, m_{j(i)}) \in \boldsymbol{PK}$ for some $j(i)$;*
- *$\mathcal{O}^C$ has not been queried with any $Y_{i,b}$ for any i;*
- *The purported forgery is not a complete copy of a query to $\mathcal{O}^S$ with its corresponding response.*

Definition 3 (Existential unforgeability with respect to insider corruption [14]). *For a fixed N, q_H, q_S, and q_C, if $\mathcal{A}$ is an algorithm that operates in the game defined Definition 2 in time at most t and succeeds at the above game*

with probability at least ϵ, we say $\mathcal{A}$ is a $(t, \epsilon, N, q_H, q_S, q_C)$-forger where ϵ is measured over the joint distribution of the random coins of $\mathcal{A}$ and the challenge set $\boldsymbol{PK}$.

Definition 4 (DLSAG signer ambiguity [31]). *A DLSAG signature scheme with security parameter λ is signer ambiguous if for any PPT algorithm $\mathcal{A}$, on inputs any message $\mathtt{m}$, any list $\mathbf{pk}$ of n public key pairs, any valid signature σ on $\mathbf{pk}$ and $\mathtt{m}$ generated by user π, such that $\mathsf{sk}_\pi \notin \mathcal{D}_t$ and any set of t private keys $\mathcal{D}_t := \{\mathsf{sk}_1, \ldots, \mathsf{sk}_t\}$ where $\{g^{\mathsf{sk}_1}, \ldots, g^{\mathsf{sk}_t}\} \subset \mathbf{pk_b}$, $n - t \geq 2$ and b is extracted from σ. There exists a negligible function $\mathsf{negl}(\cdot)$ such that:*

$$\left| \Pr[\mathcal{A}(\mathtt{m}, \mathbf{pk}, \mathcal{D}_t, \sigma) = \pi] - \frac{1}{n-t} \right| \leq \mathsf{negl}(\lambda)$$

Definition 5 (DLSAG linkability). *A DLSAG signature scheme is linkable if there exists a PPT algorithm* Link *that takes as input two rings $\mathbf{pk_1}, \mathbf{pk_2}$, two messages $\mathtt{tx}_1, \mathtt{tx}_2$, their corresponding DLSAG signatures σ_1, σ_2 (with respective true signing indices π_1 and π_2 not provided to* Link*), and outputs either 0 or 1, such that there exists a negligible function $\mathsf{negl}(\cdot)$ with the property that:*

$$\begin{aligned} &\Pr[\textsc{Link}((\mathbf{pk_1}, \mathtt{tx}_1, \sigma_1), (\mathbf{pk_2}, \mathtt{tx}_2, \sigma_2)) = 1 | (\mathsf{pk}_{\pi_1}, m_{\pi_1}) \neq (\mathsf{pk}_{\pi_2}, m_{\pi_2})] + \\ &\Pr[\textsc{Link}(\mathbf{pk_1}, \mathtt{tx}_1, \sigma_1), (\mathbf{pk_2}, \mathtt{tx}_2, \sigma_2)) = 0 | (\mathsf{pk}_{\pi_1}, m_{\pi_1}) = (\mathsf{pk}_{\pi_2}, m_{\pi_2})] \leq \mathsf{negl}(\lambda) \end{aligned}$$

B.2 Preliminaries

In order to prove the security of the proposed scheme, we first need to introduce the following definitions and results.

Definition 6 (Forking algorithm [13]). *Let $\mathcal{A}$ be a PPT algorithm that takes as input some inp. Assume $\mathcal{A}$ has access to a random oracle $\mathcal{O}^{H_s}$ that outputs random element from $\mathbb{Z}_q$ and the query responses are temporally ordered by index $e_0, e_1, \ldots, e_{q_H - 1}$. Define the forking algorithm associated with $\mathcal{A}$, denoted $\mathtt{F}_\mathcal{A}$, as the following algorithm:*

1. *Take as input some inp, select random coins ρ for $\mathcal{A}$, and select q_H oracle query responses, $e_0, e_1, \ldots, e_{q_H - 1} \xleftarrow{\$} \mathbb{Z}_q$.*
2. *Execute $\alpha \leftarrow \mathcal{A}(inp; \rho)$, responding to the i^{th} query to $\mathcal{O}^{H_s}$ made by $\mathcal{A}$ with the response e_i.*
3. *If $\alpha = \perp$ return $\perp$ and terminate. Otherwise, parse $(j, out) \leftarrow \alpha$.*
4. *Select new oracle query responses $e'_j, e'_{j+1}, \ldots, e'_{q_H - 1} \xleftarrow{\$} \mathbb{Z}_q$.*
5. *Execute $\alpha' \leftarrow \mathcal{A}(inp; \rho)$, responding to the i^{th} query to $\mathcal{O}^{H_s}$ made by $\mathcal{A}$ with the response e_i when $i < j$ and e'_i otherwise.*
6. *If $\alpha' = \perp$, return $\perp$ and terminate. Otherwise, parse $(j', out') \leftarrow \alpha'$.*
7. *If $j = j'$ and $e_j \neq e'_j$, return (j, out, out'). Otherwise, return $\perp$.*

Lemma 1 (Generalized forking lemma [13]). *Let q_H be an integer, $\mathcal{A}$ be a randomized algorithm which takes as input some main input inp and $h_0, h_1, \ldots$*

$, h_{q_H-1} \in \mathbb{Z}_q$ *and returns either a distinguished failure symbol* $\perp$ *or a pair* (j, out)*, where* $0 \leq j < q$ *and out is some side output. The accepting probability of* $\mathcal{A}$*, denoted* $acc(\mathcal{A})$*, is defined as the probability that* $\mathcal{A}$ *does not output* $\perp$ *(where this probability is measured over the random selection of inp,* $\{e_i\}_{i=0}^{q_H-1}$*, and* $\{e'_i\}_{i=j}^{q_H-1}$*). Let* $\mathcal{B}$ *be the forking algorithm associated with* $\mathcal{A}$ *from Definition 6. Let* $acc(\mathcal{B})$ *be the probability (over the draw of inp and the random coins of* $\mathcal{B}$*) that* $\mathcal{B}$ *returns a non-*$\perp$ *output. Then*

$$acc(\mathcal{B}) \geq acc(\mathcal{A}) \left(\frac{acc(\mathcal{A})}{q_H} - \frac{1}{q} \right).$$

In particular, if $\mathcal{A}$ *has non-negligible acceptance probability, then so does* $\mathcal{B}$*.*

Definition 7 (One-More discrete logarithm hardness [12]). *Let* λ *be a security parameter. Let* N *be natural number such that* $1 \leq N < \mathsf{poly}(\lambda)$*. Let* $(\mathbb{G}, q, g) \leftarrow \textsc{Setup}(1^\lambda)$ *be some group parameters. Let* $\mathcal{O}^C$ *be a corruption oracle. For any fixed* N*, these group parameters are said to satisfy the one-more discrete logarithm hardness (OMDL) assumption for* N *if any PPT algorithm* $\mathcal{A}$ *has at most a negligible probability of success in the following game.*

1. *A sequence of* $N+1$ *independent and identically distributed observations of a uniform random variable on* $\mathbb{G}$ *are made,* $S = \{\mathcal{H}_0, \ldots, \mathcal{H}_N\} \subseteq \mathbb{G}$*. The group parameters* $(\mathbb{G}, q, g)$ *and the set* S *are sent to* $\mathcal{A}$*.*
2. $\mathcal{A}$ *is granted oracle access to* $\mathcal{O}^C$*.*
3. $\mathcal{A}$ *outputs an index* $0 \leq i \leq N$ *and a scalar* $x \in \mathbb{Z}_q$*.*

$\mathcal{A}$ *succeeds if* $g^x = \mathcal{H}_i$*, the corruption oracle* $\mathcal{O}^C$ *is not queried with* $\mathcal{H}_i$*, and the corruption oracle* $\mathcal{O}^C$ *is queried at most* N *times.*

Definition 8. *If* $\mathcal{A}$ *is an algorithm that runs in time at most* t *and succeeds at the one-more discrete logarithm game for some* N *with probability at least* ϵ*, then we say* $\mathcal{A}$ *is a* (t, ϵ, N)*-OMDL solver where* ϵ *is measured over the joint distribution of the random coins of* $\mathcal{A}$ *and the challenge group elements* $\mathcal{H}_i$*.*

Definition 9 (Decisional Diffie-Hellman assumption). *Let* $(\mathbb{G}, q, g)$ *be the group parameters. We say the Decisional Diffie-Helman Problem is hard relative to* $\mathbb{G}$ *if for all probabilistic polynomial time algorithms* $\mathcal{M}$ *there exist a negligible function* $\epsilon(\cdot)$ *such that*

$$\begin{aligned} \Pr[\mathcal{M}(\mathbb{G}, g, q, A, B, C) = b : (A, B, C) = (A_b, B_b, C_b) \\ where\ (A_0, B_0, C_0) = (g^{a_0}, g^{b_0}, g^{c_0}); \\ (A_1, B_1, C_1) = (g^{a_1}, g^{b_1}, g^{a_1 b_1})] \\ \leq \frac{1}{2} + \epsilon(\lambda) \end{aligned}$$

where a_i, b_i, c_i *for* $i \in \{0, 1\}$ *are uniformly chosen from* $\mathbb{Z}_q$*.*

B.3 Proofs of stated theorems

In this subsection, we provide our proofs for our stated theorems.

Proof of Theorem 1

Proof. We construct (t', ϵ', N')-OMDL solver $\mathcal{B}$ from a $(t, \epsilon, N, q_H, q_S, q_C)$-forger $\mathcal{A}$. $\mathcal{A}$ takes as input a set of N public keys from the signature scheme, has q_S oracle queries available to a signing oracle $\mathcal{O}^S$, has q_H oracle queries available to a random oracle $\mathcal{O}^{H_s}$, and has q_C oracle queries available to a corruption oracle $\mathcal{O}^C$. We wrap $\mathcal{A}$ in an algorithm $\mathcal{A}'$ with the same oracle access that is appropriate for use in the forking algorithm.

$\mathcal{B}$ takes as input a set of $N' + 1 = 2N$ group elements (the challenge points) and has up to N' queries available to a corruption oracle $\mathcal{O}^C$. $\mathcal{B}$ executes a forking algorithm $\mathtt{F}_{\mathcal{A}'}$ as a black box, passing the challenge points onto $\mathtt{F}_{\mathcal{A}'}$ as input, which in turn forks a black box execution of $\mathcal{A}'$ (the simple wrapper of $\mathcal{A}$) using the challenge points as input.

$\mathcal{B}$ answers corruption oracle queries made by $\mathtt{F}_{\mathcal{A}'}$ by querying $\mathcal{O}^C$ directly and passing along the result. $\mathtt{F}_{\mathcal{A}'}$ answers corruption oracle queries for $\mathcal{A}'$ by passing them along to $\mathcal{B}$. $\mathtt{F}_{\mathcal{A}'}$ simulates responses to random oracle queries to $\mathcal{O}^{H_s}$ (or signing oracle queries to $\mathcal{O}^S$, respectively) made by $\mathcal{A}'$ by flipping coins (or by flipping coins and backpatching, respectively).

In a transcript resulting in a successful forgery, $\mathcal{A}'$ queries the random oracle during verification with all queries of the form

$$h_{i+1} \leftarrow \mathcal{O}^{H_s}(\mathtt{tx} \,||\, g^{s_i} \cdot Y_{i,b}^{h_i} \,||\, Y_{1,(1-b)}^{s_i} \cdot \mathcal{J}^{h_i}).$$

That is to say, $\mathcal{A}'$ does not guess h_{i+1} but actually queries the random oracle at least once in each transcript (except in transcripts that occur with negligible probability). To see why, note that if $\mathcal{A}$ does not make one of these queries, then $\mathcal{A}$ is selecting h_{i+1} at random by flipping coins and later discovering that h_{i+1} is precisely the image of some $(\mathtt{tx} \,||\, g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_i} \,||\, \mathsf{pk}_{i,1-b}^{s_i} \cdot \mathcal{J}^{h_i})$ through the random oracle. This occurs with probability at most $1/q$ which is negligible.

In the transcript of $\mathcal{A}'$, queries made to the random oracle occur in linear order; denote the responses received by $\mathcal{A}$ as $e_0, e_1, e_2, \ldots$. Define the distinguished pair (j, i) to be the index of the oracle response e_j such that the oracle query $e_j = h_{i+1} = \mathcal{O}^{H_s}(\mathtt{tx} \,||\, g^{s_i} \cdot Y_{i,b}^{h_i} \,||\, Y_{1,(1-b)}^{s_i} \cdot \mathcal{J}^{h_i})$ corresponds to the first verification query made to the random oracle. *We refer to such a transcript as a (j, i)-forgery.*

Note that any algorithm executing $\mathcal{A}$ in a black box can inspect the transcript of $\mathcal{A}$ and extract the pair (j, i) in $O(q_H)$ time. Hence, if $\mathcal{A}$ takes time t, then the simple wrapper $\mathcal{A}'$ takes time $t + O(q_H)$. Note that the acceptance probabilities $acc(\mathcal{A}) = acc(\mathcal{A}')$, and $\mathcal{A}'$ can be used in the forking lemma. The algorithm $\mathtt{F}_{\mathcal{A}'}$ runs $\mathcal{A}'$ as a black box, selecting its random tape. $\mathtt{F}_{\mathcal{A}'}$ rewinds the transcript of $\mathcal{A}'$ while preserving the random tape and the oracle responses preceding the rewind point $e_0, e_1, \ldots, e_{j-1}$. The algorithm $\mathtt{F}_{\mathcal{A}'}$ responds with new random values $e'_j, e'_{j+1}, \ldots$ from that point forward. By the forking lemma, if $\mathcal{A}'$ has success

probability $acc(\mathcal{A}) > \epsilon$, then $\mathtt{F}_{\mathcal{A}'}$ has success probability $acc(\mathcal{B}) > \epsilon\left(\frac{\epsilon}{q_h} - \frac{1}{q}\right)$. In particular, if ϵ is non-negligible, then so is $acc(\mathtt{F}_{\mathcal{A}'})$.

For timing, the forking algorithm associated with $\mathcal{A}'$ runs in twice the time of $\mathcal{A}'$ in addition to whatever additional time is required to simulate the oracle queries made by $\mathcal{A}'$. In particular, since $\mathcal{A}'$ runs in time $t + O(q_H)$, $\mathtt{F}_{\mathcal{A}}$ runs in time at most $2t + O(4q_H + 2q_S)$.

Now in both transcripts produced by $\mathtt{F}_{\mathcal{A}'}$, the first random oracle query relevant to the forgery is the j^{th} query, and in both transcripts, the inputs to this query are identical. However, in each transcript, the query responses are different. In the first transcript we have

$$e_j = \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid L \mid\mid R)$$

and in the second transcript we have

$$e'_j = \mathcal{O}^{H_s}(\mathtt{tx} \mid\mid L \mid\mid R)$$

for some $e_j \neq e'_j$, and where the inputs to these queries are identical.

Since **pk** is included in $\mathtt{tx}$, the ring of public keys in the forgery is the same in each transcript. At this point in the transcript, the forger may not have decided which ring member this assignment is made to, i.e. may not have decided upon an index i or value s_i such that $L = g^{s_i} \cdot Y_{i,b}^{h_i}$, and $R = Y_{1,(1-b)}^{s_i} \cdot \mathcal{J}^{h_i}$. Certainly the forger cannot know the values of h_i except with negligible probability, either, since the index j was selected to be the first oracle query used in verification of the forgery.

In fact, since $e_j \neq e'_j$ and this is the first oracle query made, the probability that the subsequent signature challenges $\{h_i\}_i$ are identical in each transcript is negligible. Yet the forger has produced from the first transcript some s_i, h_i and from the second transcript some s'_i, h'_i such that $L = g^{s_i} \cdot Y_{i,b}^{h_i} = g^{s'_i} \cdot Y_{i,b}^{h'_i}$.

Any algorithm running the forking algorithm $\mathtt{F}_{\mathcal{A}'}$ as a black box learns the index i common to both transcripts, the signing data from each transcript s_i and s'_i and the challenges h_i, h'_i from those transcripts, and can compute the discrete logarithm

$$Y_{i,b} = g^{\frac{s'_i - s_i}{h_i - h'_i}}$$

in time that is $O(1)$ related to inverting scalars.

Hence, $\mathcal{B}$ takes $2N$ group elements as input, runs in time at most $2t + O(4q_H + 2q_S + 1)$, has acceptance probability at least $\epsilon\left(\frac{\epsilon}{q_h} - \frac{1}{q}\right)$, makes at most $q_C \leq 2N - 1 = N'$ corruption oracle queries, and yet successfully produces the discrete logarithm of at least one challenge point.

Proof of Theorem 2

Proof. We will consider WLOG that the DLSAG is signed by the first public key of the key pair, i.e. the before-key. The case for the after key is completely analogous.

Let m be a message, and $0 \leq t \leq n-2$. Let $\boldsymbol{pk}$ be a ring of n public keys pairs, of which t private keys are known that corresponding to some of the before-keys. Let σ be a DLSAG of the message m, with the ring $\boldsymbol{pk}$ by a random public key of index π whose private key is not among the revealed ones.

Assume that there exists a non-negligible function $\epsilon(\cdot)$ and PPT $\mathcal{A}$ such that:

$$\Pr[\mathcal{A}(\mathtt{m}, \boldsymbol{pk}, \sigma) = \pi] \geq \frac{1}{n-t} + \epsilon(\lambda)$$

We will use $\mathcal{A}$ to construct a PPT $\mathcal{M}$ that violates the DDH assumption with non-negligible advantage.

Indeed, without loss of generality, we provide our proof for $t = 0$. The proof for $t \neq 0$ can be carried out in the same manner.

Upon receiving the DDH triple (A, B, C), $\mathcal{A}$ picks $n-1$ public key pairs of which $\mathcal{A}$ knows corresponding private before-keys. Append (A, B) at the end to obtain an n-sized ring, $[(A_i, B_i)]_{i=1}^{n}$ Pick a random index π and swap the pair in that entry with (A, B), let that be $\boldsymbol{pk}$.

In order to generate a purported signature σ by that ring with the index π on the given message. We will toss coins to set the random oracle query responses and feed those results back to $\mathcal{A}$ when it queries the oracle for verification.

Specifically, we pick random values $s_1, \ldots, s_n, h_1, \ldots, h_n$, and define, for all $i \in \mathbb{Z}_n$ the oracle query responses as:

$$h_{i+1} := \mathsf{H_s}(\mathtt{m} || g^{s_i} \cdot A_i^{h_i} || B_i^{s_i} \cdot C^{h_i}).$$

If $C = g^{ab}$, then the above will be a proper DLSAG signature with the given oracle; if not, then C is just a random point and shouldn't be more likely to be linked to A, B than any other pair in the ring by $\mathcal{A}$.

Since $\mathcal{A}$ is able to extract the true signer from the given key image with non-negligible advantage, we feed $\sigma = (h_1, s_1, \ldots, s_n, C)$ to it. We set $\mathcal{M}(A, B, C)$ to return 1 if $\mathcal{A}(m, \boldsymbol{pk}, \sigma) = \pi$, and return a coin toss otherwise. Computing $\mathcal{M}$'s advantage:

$$\begin{aligned}
&\Pr[\mathcal{M}(A, B, C) = b | b = 1] = \\
&\Pr[(\mathcal{M}(A, B, C) = b | b = 1) \wedge (\mathcal{A}(m, \boldsymbol{pk}, \sigma) = \pi)] \\
&+ \Pr[(\mathcal{M}(A, B, C) = b | b = 1) \wedge (\mathcal{A}(m, \boldsymbol{pk}, \sigma) \neq \pi)] \\
&\geq 1 \cdot \left(\frac{1}{n} + \epsilon(\lambda)\right) + \frac{1}{2}\left(1 - \frac{1}{n} - \epsilon(\lambda)\right) \\
&= \frac{1}{2} + \frac{1}{2n} + \frac{\epsilon(\lambda)}{2}
\end{aligned}$$

And

$$\begin{aligned}
&\Pr[\mathcal{M}(A,B,C)=b|b=0]= \\
&\Pr[(\mathcal{M}(A,B,C)=b|b=0)\wedge(\mathcal{A}=\pi)] \\
&+\Pr[\mathcal{M}(A,B,C)=b|b=0\wedge\mathcal{A}\neq\pi] \\
&=0\left(\frac{1}{n}\right)-\frac{1}{2}\left(1-\frac{1}{n}\right) \\
&=\frac{1}{2}-\frac{1}{2n}
\end{aligned}$$

Combining the two equations, we get:

$$\begin{aligned}
&\Pr[\mathcal{M}(A,B,C)=b]= \\
&\quad\Pr[b=1]\Pr[\mathcal{M}(A,B,C)=b|b=1]\ +\Pr[b=0]\Pr[\mathcal{M}(A,B,C)=b|b=0] \\
&\quad\geq\frac{1}{2}\left(\frac{1}{2}+\frac{1}{2n}+\frac{\epsilon(\lambda)}{2}\right)+\frac{1}{2}\left(\frac{1}{2}-\frac{1}{2n}\right)=\frac{1}{2}+\frac{\epsilon(\lambda)}{4}
\end{aligned}$$

Since $\epsilon(\lambda)$ was non-negligible, so is $\epsilon(\lambda)/4$, which shows that $\mathcal{M}$ breaks the DDH assumption with non-negligible probability, as we wanted to show.

Proof of Theorem 3

Proof. We will use the notation introduced in the previous proof. Notice that in the unforgeability proof, the discrete logarithm of $Y_{i,b}$ was extracted by comparing the two representations of the same point L. At that point, one could have also extracted the discrete logarithm of $\mathcal{J}$ with respect to the point $Y_{1,(1-b)}^{m_i}$ by comparing the two representations of the point R:

$$\mathcal{J}=Y_{1,(1-b)}^{\left(\frac{s_i'-s_i}{h_i-h_i'}\right)m_i}.$$

Moreover, those discrete logarithms are the same.

Now, if there existed a PPT adversary $\mathcal{A}$, having no prior knowledge of private keys in the b-bit component other than the private key of a certain $(\mathcal{Z}^{(0)},\mathcal{Z}^{(1)},m)$, that could produce a signature σ with a purported dual key image $\bar{\mathcal{J}}$, distinct from the honest key image $\mathcal{J}$.

Then we could fork $\mathcal{A}$ and extract a second signature σ' whose first verification query is the same as that of σ.

$$e_j=\mathcal{O}^{H_s}(\mathtt{tx}\ ||\ L\ ||\ R)$$

and in the second transcript we have

$$e_j'=\mathcal{O}^{H_s}(\mathtt{tx}\ ||\ L\ ||\ R)$$

for some $e_j \neq e'_j$. Writing the representations of those two points we get:

$$g^{s_i} \cdot Y_{i,b}^{h_i} = L = g^{s'_i} \cdot Y_{i,b}^{h'_i}, \text{ and}$$

$$Y_{1,(1-b)}^{s_i m_i} \cdot \bar{\mathcal{J}}^{h_i} = R = Y_{1,(1-b)}^{s'_i m_i} \cdot \bar{\mathcal{J}}^{h'_i}$$

There are two cases to consider: If $Y_{i,b} = \mathcal{Z}^{(b)}$, then, as observed at the beginning of this proof, we extract the discrete logarithm of $\bar{\mathcal{J}}$ and conclude that $\bar{\mathcal{J}} = \mathcal{J}$, a contradiction.

Otherwise, if $Y_{i,b} \neq \mathcal{Z}^{(b)}$, then, again as observed at the beginning, we extract the discrete logarithm of $Y_{i,b}$, thus solving the DLP for that point.

By the above corollary, all we are left to show is that $\Pr[\textsc{Link}(\mathtt{tx}_1, \sigma_1, \mathtt{tx}_2, \sigma_2) = 1 | (\mathsf{pk}_{\pi_1}, m_{\pi_1}) \neq (\mathsf{pk}_{\pi_2}, m_{\pi_2})]$ is negligible.

Since our Link algorithm just compares the dual key images, this would require a PPT algorithm $\mathcal{A}$ to obtain two tuples of the form (A, B, m_1) and (C, D, m_2) such that they both have the same point as dual key image, $\mathcal{J} = g^{abm_1} = g^{cdm_2}$.

However, if the output containing the dual address (A, B) is created at the i_1 position of the output vector of transaction $\mathtt{tx}_1$, then $m_1 := \mathsf{H_s}(\mathtt{tx}_1, i_1)$. This means, by the ROM, that a and b have to be fixed before the value of m_1. Similarly, $m_2 := \mathsf{H_s}(\mathtt{tx}_2, i_2)$ can only be known after c and d are fixed.

Each side of the equation $g^{abm_1} = g^{cdm_2}$ therefore behaves as a random oracle, so the chance of they matching is negligible. This shows that our DLSAG scheme is linkable.

C Transaction example

In this section, we will work out an example of how a concrete Monero transaction using dual outputs and hidden time locks could work.

Assume that Alice wants to spend coins held in an output of the form:

$$((\mathsf{pk}_0, \mathsf{pk}_1, m), \mathcal{A}, \Pi_{\mathcal{A}}, \mathcal{T}, \Pi_{\mathcal{T}})$$

where $\mathcal{A} = \textsc{Com}(\gamma, r)$ and $\mathcal{T} = \textsc{Com}(t, k)$, for some amount γ and some timelock t known to Alice, and $\Pi_{\mathcal{A}}$ and $\Pi_{\mathcal{T}}$ are the range proofs of those commitments. And let's say she would like to create the new outputs of the form:

$$((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}), \mathcal{A}_1, \Pi_{\mathcal{A}_1}, \mathcal{T}_1, \Pi_{\mathcal{T}^1})$$

$$((\mathsf{pk}_{2,0}, \mathsf{pk}_{2,1}), \mathcal{A}_2, \Pi_{\mathcal{A}_2}, \mathcal{T}_2, \Pi_{\mathcal{T}^2}).$$

First she decides on amounts γ_i such that $\gamma_1 + \gamma_2 + fee = \gamma$, and the corresponding timelocks t_i and their respective masks r_i and k_i. Then, for $i = 1, 2$, she sets $\mathcal{A}_i := \textsc{Com}(\gamma_i, r_i)$ and $\mathcal{T}_i := \textsc{Com}(t_i, k_i)$, and computes the range proofs for those commitments, $\Pi_{\mathcal{A}_i} := \text{RProve}(\gamma_i, r_i)$ and $\Pi_{i,\mathcal{T}} := \text{RProve}(t_i, k_i)$.

Next, she needs to prove that she can spend her output according to the timelock t. That means proving that the timelock t has or has not expired, and

then signing with the appropriate bit-key. WLOG, let's assume that she controls pk_1 so that t must already have expired. For that, she picks t' such that $t < t'$, but also $t' < T$, where T is a block height for which she wishes her transaction to be mined. She picks a random mask k', and computes $\mathcal{T}_{dif} := \textsc{Com}(t' - t, k')$ and $\mathit{\Pi}_{\mathcal{T}_{dif}} := \textsc{RProve}(t' - t, k')$.

Now, she picks $n - 1$ decoy outputs from the blockchain, computes their output identifiers m_j, and forms the ring:

$$((\mathsf{pk}_{j,0}, \mathsf{pk}_{j,1}, m_j), \mathcal{A}_j, \mathit{\Pi}_{\mathcal{A}_j}, \mathcal{T}_j, \mathit{\Pi}_{\mathcal{T}_j})_{[1,n]}.$$

As before, for ease of exposition, we assume that her output is the last one in the ring, but in practice its position must be selected uniformly at random to preserve signer ambiguity. She won't need the range proofs for her signing, ignoring them, she is left with:

$$((\mathsf{pk}_{j,0}, \mathsf{pk}_{j,1}, m_j), \mathcal{A}_j, \mathcal{T}_j)_{[1,n]}.$$

Before continuing, observe that the following are commitments to zero:

$$\frac{\mathcal{T}_n}{\mathcal{T}_{dif} \cdot h^{t'}} = \frac{\textsc{Com}(t, k)}{\textsc{Com}(t' - t, k') \cdot h^{t'}} = g^{(k-k')}$$

$$\frac{\mathcal{A}_n}{\mathcal{A}_1 \cdot \mathcal{A}_2 \cdot h^{fee}} = \frac{\textsc{Com}(\gamma, r)}{\textsc{Com}(\gamma_1, r_1) \cdot \textsc{Com}(\gamma_2, r_2) \cdot h^{fee}} = g^{(r-r_1-r_2)}.$$

So that if we define:

$$\mathcal{T}_{j,zero} := \frac{\mathcal{T}_j}{\mathcal{T}_{dif} \cdot h^{t'}}, \mathcal{A}_{j,zero} := \frac{\mathcal{A}_j}{\mathcal{A}_1 \cdot \mathcal{A}_2 \cdot h^{fee}},$$

then we should get commitments to zero for $j = n$, which can in turn be viewed as signing keys.

Here, the most straight forward approach is to extend the format of current MLSAGs and concatenate another component to the signature in the following way:

Alice picks random values $s'_0, s_1, \cdots, s_{n-1}, r'_0, r_1, \cdots, r_{n-1}$ and $q'_0, q_1, \cdots, q_{n-1}$ and computes:

$$L_0 := g^{s'_0}, R_0 := \mathsf{pk}_0^{s'_0 \cdot m_n}, \mathcal{A}_0 := g^{r'_0}, \mathcal{T}_0 := g^{q'_0}$$

and $h_0 := \mathsf{H_s}(\mathtt{tx}||L_0||R_0||A_0||T_0)$. Next, for $j \in [1, n-1]$, she computes:

$$\begin{aligned} L_j &:= g^{s_j} \cdot \mathsf{pk}_{j,1}^{h_{j-1}} \\ R_j &:= \mathsf{pk}_{j,0}^{s_j \cdot m_j} \cdot \mathcal{J}^{h_{j-1}} \\ A_j &:= g^{r_j} \cdot \mathcal{A}_j^{h_{j-1}} \\ T_j &:= g^{q_j} \cdot \mathcal{T}_j^{h_{j-1}} \\ h_j &:= \mathsf{H_s}(\mathtt{tx}||L_j||R_j||A_j||T_j) \end{aligned}$$

Finally, she computes:

$$s_0 := s'_0 - h_{n-1} \cdot \mathsf{sk}_1$$
$$r_0 := r'_0 - h_{n-1} \cdot (r - r_1 - r_2)$$
$$q_0 := q'_0 - h_{n-1} \cdot (k - k').$$

Therefore, the signature is:

$$\sigma := (s_0, \cdots, s_{n-1}, r_0, \cdots, r_{n-1}, q_0, \cdots, q_{n-1}, h_0, \mathcal{J}, 1).$$

Transaction validation. A miner that receives Alice's transaction and is considering including it in a block at height T will start by checking whether $t' < T$. If so, he proceeds to verify the range proofs for the commitment values. Finally, he validates the signature by computing, for $j \in [0, n]$:

$$L_j := g^{s_j} \cdot \mathsf{pk}_{j,1}^{h_{j-1}}$$
$$R_j := \mathsf{pk}_{j,0}^{s_j \cdot m_j} \cdot \mathcal{J}^{h_{j-1}}$$
$$A_j := g^{r_j} \cdot \mathcal{A}_j^{h_{j-1}}$$
$$T_j := g^{q_j} \cdot \mathcal{T}_j^{h_{j-1}}$$
$$h_j := \mathsf{H_s}(\mathtt{tx}||L_j||R_j||A_j||T_j).$$

If $h_n = h_0$, then the transaction is valid, and can be mined.

D Stealth address in Monero

In this section, we present how stealth addresses are used in Monero. Intuitively, one use stealth addresses to generate an one time address known only to the receiver and sender.

A stealth address is composed of two group elements $A := g^a, B := g^b$ and represents a Monero account where a user can receive multiple payments without interacting with the potential senders. The sender creates a fresh public key $\mathsf{pk} := g^{\mathsf{H_s}(A^r)} \cdot B$ where r is chosen uniformly at random and pays the receiver by sending γ $\mathtt{XMR}$ in a transaction that includes, among other information, pk and the value $R := g^r$. The receiver verifies the reception of a payment by computing $\mathsf{pk}' := g^{\mathsf{H_s}(R^a)} \cdot B$ and checking whether $\mathsf{pk}' = \mathsf{pk}$. Moreover, the receiver can spend the γ $\mathtt{XMR}$ by setting $\mathsf{sk} := \mathsf{H_s}(R^a) + b$.

We will later discuss what the privacy implication when combining DLSAG and the current stealth address generation in Appendix E.

E Future directions

In this section, we identify the following future research directions:

– **Bi-directional payment channels:** In this work, we present a construction for uni-directional payment channels. An extension is thus the design and implementation of bi-directional payment channels. In particular, we find interesting to investigate if techniques in Lightning Network are compatible with our payment channels or what are the challenges otherwise.

– **Further expressiveness:** We envision that expressiveness of DLSAG could be expanded with threshold signatures similar to those of Thring [23] and key aggregation similar to that of [34]. A thorough investigation of these approaches constitutes a venue for future research.

– **Extend security and privacy models:** So far, security and privacy definitions for Monero focus on individual signatures. However, recent studies [28,36] show that an adversary that considers several transactions (and thus several signatures) at a time, can create profiling information about the users. Thus, new security and privacy models are required to further characterized the security and privacy notions provided by the complete Monero cryptocurrency. Moreover, we plan to study the privacy guarantees provided by suggested extensions such as the timelock processing scheme.

– **Timelock offset analysis and mitigations:** To prove to the network that a certain timelock t has or has not expired, the signer publishes the timelock offset value t', which leaks information about the position of the real timelock t, which in turn leaks information about whether a certain ring is likely to represent the spend of an output that was controlled by two different parties, or just one. Coming up with heuristics to separate those two cases, on one hand; and, on the other hand, figuring out the correct timelock distributions to draw t from for transactions where it is not meaningfully being used should become interesting areas of research.

– **New privacy implications:** With the use of DLSAG and the new key image mechanism, we introduce a new privacy implication in the Monero blockchain. In particular, given two rings and their corresponding signatures, the sender can determine whether the two truly spent public keys belong to the same user (i.e., the two public keys where derived from the same stealth address with randomness provided by the sender herself). We briefly explain how the traceability method works as follow:
Let $(B_1, B_2) = (b_1 g, b_2 g)$ be the stealth address of Bob. Let assume that Alice needs to pay to Bob twice, Alice generates 2 dual addresses as follow:

$$(pk_{B,0}, pk_{B,1}) = (H_s(B_1^{r_1}) \cdot B_2, H_s(B_1^{r_2}) \cdot B_2)$$
$$(pk'_{B,0}, pk'_{B,1}) = (H_s(B_1^{r_3}) \cdot B_2, H_s(B_1^{r_4}) \cdot B_2)$$

where r_1, r_2, r_3, r_4 are chosen uniformly at random from $\mathbb{Z}_q$. Here, we note that Alice knows the value of r_1, r_2, r_3, r_4.
As we discussed in Appendix D, Bob will use the following corresponding secret keys to spend the money in future transaction:

$$(sk_{B,0}, sk_{B,1}) = (H_s(R_1^{b_1}) + b_2, H_s(R_2^{b_1}) + b_2)$$
$$(sk'_{B,0}, sk'_{B,1}) = (H_s(R_3^{b_1}) + b_2, H_s(R_4^{b_1}) + b_2)$$

However, with the new key image mechanism, when Bob spends those outputs, he will need to publish two transaction with the following two key images:

$$\mathcal{J}_1 = g^{m \cdot sk_{B,0} \cdot sk_{B,1}}$$
$$\mathcal{J}_2 = g^{m' \cdot sk'_{B,0} \cdot sk'_{B,1}}$$

where m, m' is defined to be the hash of the transaction and the output, so Alice computes both m, m'. Thus, given two different key images, Alice can determine if Bob created those two transaction by computing:

$$\begin{aligned}
\frac{\mathcal{J}_1^{m^{-1}}}{\mathcal{J}_2^{m'^{-1}}} &= \frac{g^{(H_s(R_1^{b1})+b_2)\cdot(H_s(R_2^{b1})+b_2)}}{g^{(H_s(R_3^{b1})+b_2)\cdot(H_s(R_4^{b1})+b_2)}} \\
&= \frac{g^{H_s(R_1^{b1})H_s(R_2^{b1})+H_s(R_2^{b1})b_2+H_s(R_2^{b1})b_2+b_2^2}}{g^{H_s(R_3^{b1})H_s(R_4^{b1})+H_s(R_3^{b1})b_2+H_s(R_4^{b1})b_2+b_2^2}} \\
&= \frac{g^{H_s(B_1^{r1})H_s(B_1^{r2})} \cdot B_2^{H_s(B_1^{r1})} \cdot B_2^{H_s(B_1^{r2})}}{g^{H_s(B_1^{r3})H_s(B_1^{r4})} \cdot B_2^{H_s(B_1^{r3})} \cdot B_2^{H_s(B_1^{r4})}}
\end{aligned} \tag{1}$$

The final step of Eq. (1) contains all information known to Alice. Therefore, she precomputes that value to determine when Bob starts spending those coins paid by her. However, Alice can only determine when Bob starts spending one step in the future, and after that she should not know when Bob will spend again. Thus, one way to mitigate that problem is to require Bob to generate another dual address and move all money received to the new address. This adds the need for another transaction, but it enables payment channel and off-chain payments, thus paving the way to reduce the overall number of on-chain transactions. It would be an interesting future work to determine what other privacy implications are there when combining DLSAG with Monero, and whether different stealth address schemes and key image definitions exist that would avoid this issue.