

A Generic Framework for Cross-Chain Atomic Swaps of Digital Tokens

János Tapolcai*, Bence Ladóczki*, Lajos Rónyai†

* Dept. of Telecommunications and Artificial Intelligence, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, and with the HUNREN-BME Information Systems Research Group.

† HUN-REN Institute for Computer Science and Control (SZTAKI), and with the Dept. of Algebra and Geometry, Institute of Mathematics, Budapest University of Technology and Economics.

Abstract—Decentralisation is one of the most important structural principles of blockchains and it should be present in all aspects of these systems. There is a growing scientific effort to develop protocols that enable the exchange of digital assets between two heterogeneous blockchains without a third-party arbitrator. We investigate the scenario where two parties wish to exchange digital tokens and run a swap protocol without the intervention of a third party. This is a complex and cooperative process for which – in a trust-less environment – atomicity and fairness must be ensured. While smart contracts provide a convenient way to achieve this, we want to ensure that the two parties leave no traceable data on the respective chains. The challenge here is that the two digital assets can reside on different chains with different signature schemes. Tackling these issues, we present a protocol description for the exchange of two arbitrary cryptocurrencies. A generic standard for atomic swap implementations in terms of the required cryptographic primitives is given. Additionally, we derive a protocol for the necessary interactions between two arbitrary chains using these primitives. To demonstrate applicability, we illustrate how this protocol can be instantiated with Schnorr and BLS signatures.

I. INTRODUCTION

The multifaceted nature of blockchain implementations has motivated active research into cross-chain interoperability [1], [2] in recent years. Centralised cryptocurrency exchanges (CEXs) provide a fairly convenient solution, as long as the users accept the fact that the exchange keeps track of all of their transactions and that they are no longer the sole owner of their digital assets. The largest cryptocurrency exchanges (Coinbase, Binance, OKX, etc.) list Bitcoin [3] along with hundreds, if not thousands, of alternative digital assets. These tokens are traded [4] with virtually no restrictions.

In this study we scrutinise and formalise the exchange process of two parties who wish to trade directly. We assume that the parties somehow agree on an exchange rate, for example using an off-chain order book [5]. Note that wrapped tokens can be exchanged 1 : 1. Once this is done the tokens shall be exchanged without the intervention of a third party. To that end, we define a protocol for two-party atomic swaps based on abstract primitives so that the swap process can be analysed

without an in-depth understanding of the implementations of signature schemes and timelocks. The framework is atomic in the sense that it either completes in its entirety or not at all.

Here, we present protocols for the exchange of two arbitrary tokens. The tokens can reside on different chains implementing completely different cryptographic primitives. Block times and signature schemes can be different too. Existing research has typically focused on specific cryptocurrency pairs [6] and according to Wikipedia, as of June 2023, there were more than 25,000 cryptocurrencies listed on online marketplaces. Clearly, designing an atomic swap protocol for each pair without a higher level of abstraction is an intractable task. Here the necessary formalism for this abstraction is presented. Our results – being an important step in the space of blockchain interoperability – can later be extended [7] to more complex exchange protocols, whereby multiple parties can atomically exchange multiple tokens [8]. Even though our primitives can be easily implemented using smart contracts, we wish to deal with the algebraic manipulation of the formulas of digital signatures, and therefore we omit discussions [9], [10] on smart contracts.

We first review existing solutions designed for specific cryptocurrency pairs and synthesise the key building blocks to define a general protocol that could serve as a framework for a broad range of digital asset exchanges. We break down these solutions into universal steps, proposing an abstraction that is independent of the underlying signature algorithm. Such a well-formulated abstraction aids the analysis of the atomic swap protocol and its security properties can be proven without relying on equations of cryptography in the argumentation. Our aim is to achieve this, while ensuring that the cryptographic primitives align well with existing ones and can be easily adapted to a wide range of blockchains. Essentially, we formalise widely used cryptographic primitives – such as signature generation, offsets, adaptor signatures, multisig addresses, etc. – and our primary interest here lies in identifying what properties each primitive must satisfy. While many of these properties are close to trivial, we describe them for the sake of completeness, and we note that some properties are used slightly differently across the different cryptographic algorithms.

Our discussion involves examining numerous atomic swap protocols designed for specific blockchain pairs – for example,

JT and BL were partly supported by Project no. K23 146347 of National Research, Development and Innovation Fund of Hungary. LR is partially supported by a European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory, Hungary, and by the Program of Excellence TKP2021-NVA-02 at the Budapest University of Technology and Economics.

- 1) \mathcal{A} with address X_{A_1} in Chain₁, \mathcal{B} with address X_{B_2} in Chain₂. \mathcal{A} and \mathcal{B} agree on the exchange rate e_1 , e_2 and addresses X_{A_2} , X_{B_1} , X_{AB_1} , X_{AB_2} .
- 2) \mathcal{A} locks e_1 amount of tokens in Chain₁ to X_{AB_1} (these are multi-sig address of \mathcal{A} and \mathcal{B}).
- 3) \mathcal{B} learns that e_1 was locked to X_{AB_1} in Chain₁ and locks e_2 in Chain₂ to X_{AB_2} (the multi-sig address of X_{A_2} and X_{B_2}).
- 4) \mathcal{A} learns that e_2 was also locked to X_{AB_2} in Chain₂, they exchange data so that \mathcal{A} can unlock the token from Chain₂ and \mathcal{B} has almost sufficient information to unlock the tokens from Chain₁. It is still insufficient, as \mathcal{B} still needs a so-called *signature offset*, which can be computed from the signature, unlocking the tokens in Chain₁.
- 5) \mathcal{A} unlocks e_2 tokens in Chain₂ and reveals the signature offset
- 6) Leaning the signature offset, \mathcal{B} can now unlock e_1 tokens in Chain₁

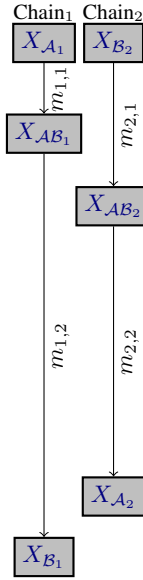


Fig. 1: The basic steps of a cross-chain atomic swap between parties \mathcal{A} and \mathcal{B} .

the protocol between Bitcoin (BTC) and Monero (XMR) described in [6], [11] – and separating the arguments that are specific to the cryptographic algorithms from those that are generally applicable to atomic swap protocols between arbitrary chains [12]. The cryptography-specific components are encapsulated in Primitives and Protocols. Our application provides an opportunity to formalise the abstract layer of cryptography that has always existed behind the formulas. The basic mechanism to exchange digital assets between two parties and two chains is shown in Fig. 1.

For the part where we discuss instantiation (see the Appendix), we confine our attention to the most widespread signature schemes, namely the one proposed by Schnorr [13], and the BLS [14] algorithms. Note that the formulas for ECDSA [15], [16] are similar to those of the Schnorr signature algorithm and we derive the relevant framework in the full version of this work. Let us emphasise that an atomic swap implementation between two chains can be delivered as long as our cryptographic primitives are implementable using the underlying signature scheme of the chain. For example, with Schnorr (and ECDSA) signatures our protocol can be used over arbitrary elliptic curves. With this work, we make the following contributions:

- A general protocol description for cross-chain atomic swaps between arbitrary pairs of blockchains without relying on a trusted third party.
- The protocol is defined at an abstraction level that is independent of the signature scheme. We demonstrate the fairness [17] of the protocol at this level of abstraction.
- We show how the proposed abstraction can be instantiated using Schnorr, and BLS signatures
- We extend the primitives for the BLS signature scheme so that at the end we extract the statement and not the witness for a hard relation.

The work is organised as follows. After discussing the related work in Sec. II, in Sec. III a brief introduction on timeouts is given. Then we motivate our work and present a general protocol for atomic exchanges. In Sec. IV the protocol is constructed in its entirety and in Sec. V compatibility issues are discussed. We provide a proof of fairness at this level of abstraction in Sec. VI. Instantiation using the Schnorr, and BLS signature schemes is presented in Sec. VII. A conclusion is given in Sec. VIII.

II. RELATED WORK

To connect our results to past achievements, we note that the proposed atomic swap protocol stems from adaptor signatures [18]–[20] that have been introduced to Bitcoin with BIP-340 [21]. Quantum-resistant adaptor signatures have been described in [22]. Going further, we refer the interested reader to scientific works on atomic swaps [9], [10], [23]–[36] investigating the arising issues in cross-chain protocols from various perspectives including security, game theory, graph theory, fairness and incentive compatibility. Among these, the basic idea of an atomic swap implementation based on homomorphic hashing introduced by J. Kirsten et al. in [37] bears substantial theoretical similarity to our work. So does [12], where the authors, by giving a minute introduction on how to instantiate adaptor signature primitives from given identification schemes, attempt to give a taxonomy of signatures to uncover the common characteristics of certain types of signature schemes. Adaptor signatures are defined in [18] as standalone primitives. An ECDSA signature-based scriptless solution for privacy-preserving PCNs is proposed in [38] in addition to a security analysis of prior PCNs. [8] presents a universal protocol for ECDSA and Schnorr. The authors use the term universal to imply that their protocol does not rely on special scripting capabilities that should be supported by the chain, not even hash-time lock contracts. They defeat this problem with timelock puzzles. Note that, unlike our work, none of these proposals deal with BLS signatures. [55] does so, but it is not universal.

Verifiably encrypted signatures (VES) [39] can also facilitate cross-chain swaps between chains using different signature schemes. The suitability of Schnorr and ECDSA signature schemes for one-time VES is presented in [40], and based on these findings, in [6] the author proposes a scheme based on VES for XMR-BTC cross-chain swaps. The main idea here is that it is possible to recover the encrypting key by knowing the plaintext and the ciphertext and this provides enough information to finalise the swap. Witness encryption is presented in [41], witness hiding adaptor signatures are discussed in [42] and the concept of adaptor wallets is presented in [43].

III. PROTOCOL SPECIFICATION

First, let us elaborate on the steps of a cross-chain swap depicted in Figure 1. At the onset of the swap, there are e_1 tokens in Chain₁ assigned to public key X_A and e_2 tokens in Chain₂ assigned to public key X_B . The two chains might use different digital signature algorithms. Let us assume that

\mathcal{A} and \mathcal{B} have already agreed upon the exchange rate $\frac{e_1}{e_2}$, as well as other conditions of the swap, such as timeouts and the sequence of public keys used for the swap.

A. Timeouts

The e_1 tokens in Chain_1 first get transferred to a multi-sig address X_{AB_1} . The address is computed from public key X_{A_1} and some secret key x_{B_1} of \mathcal{B} . This transaction is governed by a timeout mechanism, meaning that the tokens get transferred back to public key X_A at a specified time τ_1 , unless a second transaction appears. Should the atomic swap be successful, a second transaction is submitted later to Chain_1 to transfer the tokens to X_{B_1} , a public address owned by \mathcal{B} . Similar arrangements are made for Chain_2 with a timelock until τ_2 . To ensure fairness, it is crucial that the time lock τ_1 expires later than the time lock τ_2 , i.e., $\tau_1 > \tau_2$. Otherwise, if \mathcal{A} can unlock the tokens before τ_2 , she could potentially run away with both tokens. The crux of the swap happens after Step 3, when both \mathcal{A} and \mathcal{B} have locked their tokens in multi-sig addresses X_{AB_1} and X_{AB_2} , respectively. At this point, until the timeout τ_2 (with $\tau_1 > \tau_2$) expires, neither party can access the tokens without the consent of the other. This mutual collaboration is a crucial aspect of the atomic swap protocol. After this step, \mathcal{B} obtains an adaptor signature (*pre-signature*). The pre-signature can be *adapted* into a valid one when the corresponding secret information is revealed. The secret information consists of a valid signature X_{AB_2} associated with a transaction transferring e_2 tokens to X_{A_2} . Once \mathcal{B} obtains this adaptor signature, he must cooperate with \mathcal{A} to produce a valid signature. The atomic swap concludes when \mathcal{A} submits the signature to Chain_2 to receive e_2 tokens. Using this signature, \mathcal{B} can convert his adaptor signature into a valid one. Upon completion, \mathcal{A} will have e_2 tokens in Chain_2 at address X_{A_2} , and \mathcal{B} will have e_1 tokens in Chain_1 at X_{B_1} .

B. Definition of basic primitives

Here we define 7 primitives and 2 protocols that we prescribe as requirements to implement the full atomic swap protocol introduced above. Most of these protocols are close to trivial. They have been defined in various ways using different notations. We aim to establish a suitable level of generality in the definitions so that the atomic swap protocol can be constructed using them as building blocks, and crucial properties, such as *fairness* can be formally proven. Our definitions include the inputs, outputs, and constraints for each primitive, along with assumptions necessary for the fairness proof. The primitives are cryptographic algorithms executed by a single party. Based on these primitives, we formulate Protocol 11, a generic protocol specification of atomic swaps between two arbitrary chains.

We use the terminology *public information* for data known by the environment, and *private information* that is known by only one of the parties. Downstream, we use red for private information and we use subscripts to identify the corresponding user. Capital letters are used for points on an elliptic curve when elliptic curve cryptography (ECC) is employed. The

terms field elements/integers and group elements/EC points are used interchangeably. For each public key X (an elliptic curve point in ECC) there exists a secret key x (typically an element from a multiplicative group of integers modulo a prime or prime-power), such that with x one can generate a signature (Primitive 1) and the corresponding X is sufficient to verify (Primitive 2) that the corresponding secret key was used during signature generation. The signature is assumed to conceal the secret key x , which is added as Requirement 1.0. Note that this assumption is slightly stronger than existential unforgeability, but at this stage, we wish to set aside the commonly employed probabilistic arguments.

For these requirements we use the word *hard* meaning that effective algorithms can only have negligible advantage over pure guessing in calculating the variables under consideration. For detailed definitions refer to [18]. Note that, we cannot state that the secret key is completely concealed, but when a message is signed, we can assume that only a negligible amount of information about x is revealed, which is denoted by ϵ . In ECC, ϵ is usually a line that passes through point X . On its own, this is meaningless information, but for the security of the protocol, it is important to ensure that these small pieces of information do not accumulate into something that could compromise the signature. Unfortunately, there have been real-life examples [44] where the randomness used in signatures was predictable or biased [45], allowing these small information fragments to accumulate and break the signature [46]–[48].

Primitive 1: $\text{Sign}(m, x) = (\sigma, \epsilon)$

Input: m message (e.g. a transaction) x secret key

Output: σ a digital signature, and ϵ defines the “negligibly small” amount of information revealed about x

Req. 1.0: Calculating x from public information is hard

Primitive 2: $\text{VerifySig}(\sigma, m, X) = 0/1$

Input: σ signature, m message, X public key

Output: 1 if σ was computed from x of X , 0 otherwise

It is important to note that Req. 1.0 is defined as unsolvability, following standard engineering practice (e.g. one should perform an infeasible number of calculations). This provides a convenient basis for formal proofs in the context of atomic swap analysis. Such simplification have some limitations. For instance hash-and-sign signature schemes are believed to be secure under heuristic conditions, namely when the hash function is replaced by a random oracle model [49], [50] in which the oracle gives no information on the preimage [51]. According to Canetti, Goldreich and Halevi [52] random oracles cannot be implemented in a secure way.

The crux of adaptor signatures lies in *pre-signature adaptability* [12]. For this to work, a primitive that can obscure a valid signature in a verifiable way is introduced. Later, the offset can be extracted upon learning the valid signature. The offset, serving as the secret that triggers the final transaction in Step 6 of Fig. 1, should contain information that is valid in both chains. Therefore, it is convenient to have the offset

to be the same type as the secret key. For most signature schemes this can be an integer (a field element). However, in certain types of signatures, such as those using BLS where the signature is simply an elliptic curve point, one has no means to calculate an integer offset. Consequently, we define the offset functions more generally, allowing for both integer values and elliptic curve points. If the two chains utilise different elliptic curves, an offset that is an elliptic curve point on one chain cannot be used on the other because a point that satisfies the equation of one elliptic curve might not satisfy the equation of the other. As a result, atomic swaps with BLS signatures can only be implemented within strict limits. This limitation is confirmed by the impossibility result in [12], where it is demonstrated that unique [53] signature schemes (e.g., the BLS scheme) cannot be transformed into an adaptor signature scheme. Theorem 1 in [12] states that for adaptor schemes of deterministic signatures it is possible to construct efficient adversaries that extract the witness of a hard relation from a statement. To counter this result, we designed our protocol to accommodate offsets that are elliptic curve points. With this modification, our protocol is able to facilitate swaps over BLS signatures, assuming that elliptic curves are the same in both chains. Our framework naturally supports the Schnorr and ECDSA (or any other signature scheme that relies on an integer offset) schemes.

To make our framework as general as possible, we define Primitive 3 to generate offsets. This function returns a private and a corresponding public value (e.g. the *proof*). This public value serves as an input for Primitive `VerifyOffset`. In case the signature contains an integer `GenerateOffset`, it returns a random integer with the corresponding EC point. If the two chains use different elliptic curves, then the proof contains two EC points. In this case, the implementation of `GenerateOffset` also depends on the other curve. For the BLS scheme, this Primitive returns an EC point (T) instead of an integer (t), and the proof is composed of the two messages m_1, m_2 . For the sake of generality, the input of this Primitive comprises two messages m_1 and m_2 corresponding to the two transactions in the swap (see Fig. 1). Note that we do not postulate that the output should be random (i.e. for BLS it is not), instead it is sufficient that computing $x, t/T$ is hard.

Primitive 3: `GenerateOffset`(m_1, m_2, x) = $t/T, proof$

Input: x secret key,
 m_1, m_2 messages describing the two transactions of the swap
Output: t/T the offset, *proof* a public version of the offset
Req. 3.0: Computing $x, t/T$ from public information is hard

Primitive 4 must obscure a valid signature using the offset. The output of this procedure is an offset signature, denoted as Σ . The key idea is that the offset signature Σ cannot be converted into a valid signature as long as the offset value t/T is not revealed. The hardness of this problem has been reduced to the existential unforgeability of the underlying signature scheme for Schnorr and ECDSA adaptor signatures in [18], and we assume that existential unforgeability under chosen

message attack for adaptor signatures holds here as well. As detailed in Req. 4.1, there must exist an inverse offsetting function that allows the recovery of a signature from an offset signature once t/T is known.

Primitive 4: `Offset`($\sigma, t/T$) = Σ

Input: σ signature, t/T offset (either an integer or an EC point)
Output: Σ a digital offset signature that can be transformed into a signature σ once t/T is known
Req. 4.0: Computing t/T or σ from public information is hard
Req. 4.1: `Offset`(`Offset`($\sigma, t/T$), $-t/T$) = σ

The mechanism to verify an offset is defined in Primitive 5. This primitive operates on the message and the public key associated with the signature σ .

Primitive 5: `VerifyOffset`($\Sigma, m, proof, X$) = 0/1

Input: Σ offset signature, m the message, *proof* and X used in `GenerateOffset`.
Output: 1 if Σ corresponds to a valid offset signature for messages m_1, m_2 and public key X ; 0 otherwise.

Once both the offset signature Σ and the corresponding signature σ are available, the offset (t/T) can be extracted using Primitive 6, referred to as `GetOffset`. It is important to note that many signature schemes use a random nonce value to generate signatures. As such, for a given message m and a public key X , multiple valid signatures `Sign`(m, x) exist. For `GetOffset` to operate properly, Σ must be an offset signature derived from σ , meaning that if a random nonce is used during signature generation it must not differ, see Req. 6.1 of `VerifyOffset`.

Primitive 6: `GetOffset`(Σ, σ) = t/T

Input: Σ adaptor signature, σ signature.
Output: t/T the offset, if
`VerifyOffset`($\Sigma, m, proof, X$) = 1
Req. 6.1: For signature schemes with random nonce, Σ and σ are required to conceal the same nonce, i.e.
`VerifyOffset`($\Sigma, m, proof, X$) = 1.

Primitive 7 is a 2-party computation (2PC) protocol, equivalent to a 2-party joint key generation protocols within the universal composability [54] framework. Such constructions were alluded to in [8] and in [55].

Primitive 7: `MultiPubKey`(x_A, X_B) = X_{AB}

Input: x_A secret key X_B public key
Output: X_{AB} a multi-sig public key
Req. 7.1: `MultiPubKey`(x_A, X_B) = `MultiPubKey`(x_B, X_A)

The joint secret key is known neither to \mathcal{A} nor to \mathcal{B} . Should they act cooperatively, a valid signature can be constructed using Protocol 8 without revealing their individual secret keys. The public key X_{AB} is utilised to lock funds in the chain with a timeout, ensuring that neither party can access the funds without the cooperation of the other. Should they fail to cooperate until timeout the tokens get refunded to their original owners.

Protocol 8: $\text{AdaptorMultiSign}(m, x_A, t_A/T_A, \text{proof}, x_B) = \Sigma_{AB}$

Input: m message, x_A, x_B secret keys, t_A/T_A the offset ($\neq 0$) with its public proof , \mathcal{A} is in possession of $x_A, t_A/T_A, X_B$ and \mathcal{B} is in possession of x_B, X_A , finally, $X_{AB} = \text{MultiPubKey}(X_A, X_B)$

Output: \mathcal{B} has Σ_{AB} which is the offset of σ_{AB} . If the signature schemes use random nonce, then the same nonce is used in Σ_{AB} and σ_{AB} .

Req. 8.0: Computing $x_A, t_A/T_A$ from x_B and public information is hard

Req. 8.1: Computing x_B from $x_A, t_A/T_A$ and public information is hard

Req. 8.2: \mathcal{B} has Σ_{AB} such that

$$\text{VerifyOffset}(\Sigma_{AB}, m, \text{proof}, X_{AB}) = 1$$

Req. 8.3: If \mathcal{B} sends the offset signature Σ_{AB} to \mathcal{A} , then she can compute a valid signature σ_{AB} , i.e.

$$\text{VerifySig}(\sigma_{AB}, m, X_{AB}) = 1.$$

Req. 8.4: \mathcal{A} or \mathcal{B} has no means to calculate σ_{AB}

Req. 8.5: The protocol is fair [17]

Finally, Protocol 8 is introduced for the generation of a signature corresponding to a multi-sig public key by two parties, \mathcal{A} and \mathcal{B} , in a way that neither \mathcal{A} nor \mathcal{B} reveals the corresponding secret key. This process requires the active participation of both parties. The proposed protocol defines a combination of multisign, offset, and adaptor signature [18]–[20] protocols known in the literature. If the signature scheme is not deterministic and uses a random nonce (e.g., Schnorr or ECDSA), we must ensure that Req. 6.1 of `getOffset` is satisfied. This can be guaranteed in `AdaptorMultiSign` by generating the nonce in a way that involves both parties. In Req. 8.5, we borrow the definition of fairness from [17]. A fair exchange is a protocol guaranteeing that either both parties get what they want or neither of them does. If the protocol is fully executed and \mathcal{B} sends Σ_{AB} to \mathcal{A} , then he can compute a valid signature σ_{AB} for address X_{AB} . However, \mathcal{B} is still not able to sign the transaction during the execution without t_A/T_A . \mathcal{B} eventually holds Σ_{AB} which is the offset signature of σ_{AB} by t_A/T_A . The offset t_A/T_A is one of the input parameters of the protocol along with its public proof needed for verification.

C. Timelock transactions

We also need to formalise transactions where the ownership of a token is transferred from one cryptographic key to another. Such a transaction must be signed by the original owner of the token. In particular, we assume that the chains support timelocks and that once a transaction is submitted to the chain it is immutable. However, a timelock introduces an additional condition: the new ownership is temporary, awaiting a second transaction. Should the second transaction not appear the original transaction is automatically canceled at a specified time, τ . If $\tau = \infty$, we refer to it as a `Transaction`(m, σ).

In many chains timelock is a basic mechanism that restricts the spending of digital assets until a specified block height or time in the future. This mechanism is currently implemented in almost every blockchain (natively or with smart contracts) [56]. In privacy-sensitive use cases, one may prefer to avoid timelock constructions that are traceable [8]. Timelocks can be implemented with time-lock puzzles as in [8]

Protocol 9: $\text{TimeLockTransaction}(m, \sigma, \tau)$

Input: m message, σ signature, τ time when timelock expires

Req. 9.1: σ is a valid signature by the owner of the token in message m

Req. 9.2: The transaction with message m remains final on the chain until a specified time τ , after which it is reverted unless a second transaction is initiated.

with verifiable timed signatures [57]. This can be also useful for those few cryptocurrencies (e.g., Zcash and Monero [58]) that were designed to avoid the use of timelocked assets, primarily to improve privacy. This way the atomic swaps are made up of transactions that are indistinguishable from standard one-to-one transactions.

D. Heterogenous swaps

Protocol 10 is designed for blockchains with different digital signature algorithm schemes. This protocol verifies that the two public keys generated by the corresponding signature schemes conceal the same secret key. Equality of discrete logarithms in groups of unknown order can be ascertained using the ideas from [59] and [60]. This function has to be implemented for every pair of cryptographic algorithms. For example a verification algorithm for ECC is a zero-knowledge proof based implementation of Protocol 10 between curves `secp256k1` and `curve25519` is presented in [61]. In this case the proof generated by `GenerateOffset` is composed of two elliptic curve points, let us denote them by T_1 and T_2 .

Protocol 10: $\text{VerifyPublicKeyPairs}^{1,2}(t, \text{proof}) = 0/1$

Input: t secret key, $\text{proof} = (T_1, T_2)$ a pair of public keys from digital signature schemes 1 and 2, respectively

Output: 1 if the secret key of T_1 and T_2 are the same (mod $\min(q_1, q_2)$), else 0

Req. 10.0: Computing t from proof is hard

A critical aspect of this protocol is that when a random secret t is generated, it must be valid on both chains; that is, t should be less than or equal to $\min(q_1, q_2)$, where q_1 is the size of the private key space in `Chain1`, and q_2 is for `Chain2`. for chains with the same digital signature scheme the protocol is trivial, in the sense that $T_1 \stackrel{?}{=} T_2$.

IV. THE PROTOCOL

Now, we can now formalise the sought atomic swap protocol between chains with different signature algorithms, see Protocol 11. We denote primitives corresponding to `Chain1` with a superscript 1, and those corresponding to `Chain2` with a superscript 2. For some Protocols we use both superscripts, when they pertains to both chains.

The protocol starts when \mathcal{A} possesses e_1 tokens in `Chain1` associated with the address X_{A_1} , and \mathcal{B} holds e_2 tokens in `Chain2` associated with the address X_{B_2} . Once terminated, \mathcal{A} owns e_2 tokens in `Chain2` with the address X_{A_2} (to ensure transaction unlinkability $X_{A_1} \neq X_{A_2}$), and \mathcal{B} owns e_1 tokens in `Chain1` with the address X_{B_1} .

Protocol 11: AtomicSwap

Input: \mathcal{A} has a wallet in Chain₁ address $X_{\mathcal{A}_1}$, and \mathcal{B} in Chain₂ address $X_{\mathcal{B}_2}$

Timeline:

\mathcal{A} knows $x_{\mathcal{A}_1}, x_{\mathcal{A}_2}, X_{\mathcal{B}_1}, X_{\mathcal{B}_2}$ \mathcal{B} knows $x_{\mathcal{B}_1}, x_{\mathcal{B}_2}, X_{\mathcal{A}_1}, X_{\mathcal{A}_2}$

- 1 \mathcal{A} and \mathcal{B} agree on the exchange rates (e_1/e_2), timing ($\tau_1 > \tau_2$) and public
- 2 addresses ($X_{\mathcal{B}_1}, X_{\mathcal{B}_2}, X_{\mathcal{A}_1}, X_{\mathcal{A}_2}$), prove that they know their private keys.
- 3 $X_{\mathcal{AB}_1} \leftarrow \text{MultiPubKey}^1(x_{\mathcal{A}_1}, X_{\mathcal{B}_1})$ $X_{\mathcal{AB}_1} \leftarrow \text{MultiPubKey}^1(x_{\mathcal{B}_1}, X_{\mathcal{A}_1})$
- 4 $X_{\mathcal{AB}_2} \leftarrow \text{MultiPubKey}^2(x_{\mathcal{A}_2}, X_{\mathcal{B}_2})$ $X_{\mathcal{AB}_2} \leftarrow \text{MultiPubKey}^2(x_{\mathcal{B}_2}, X_{\mathcal{A}_2})$
- 5 $m_{1,1} \leftarrow$ transfer e_1 tokens from $x_{\mathcal{A}_1}$ to $X_{\mathcal{AB}_1}$ in Chain₁
- 6 $\text{TimeLockTransaction}^1(m_{1,1}, \text{Sign}(m_{1,1}, x_{\mathcal{A}_1}), \tau_1)$
- 7 $m_{2,1} \leftarrow$ transfer e_2 tokens from $x_{\mathcal{B}_2}$ to $X_{\mathcal{AB}_2}$ in Chain₂
- 8 $\text{TimeLockTransaction}^2(m_{2,1}, \text{Sign}(m_{2,1}, x_{\mathcal{B}_2}), \tau_2)$
- 9 $m_{2,1} \leftarrow$ transfer e_1 tokens from $X_{\mathcal{AB}_1}$ to $X_{\mathcal{B}_1}$ in Chain₁
- 10 $m_{2,2} \leftarrow$ transfer e_2 tokens from $X_{\mathcal{AB}_2}$ to $X_{\mathcal{A}_2}$ in Chain₂
- 11 $(t_{\mathcal{A}}/T_{\mathcal{A}}, \text{proof}) \leftarrow \text{GenerateOffset}^{1,2}(m_{2,1}, m_{2,2}, x_{\mathcal{A}_2})$
- 12 $\xrightarrow{\text{proof}}$
- 13 For different ECs $\text{VerifyPublicKeyPairs}^{1,2}(t, \text{proof}) \stackrel{?}{=} 1$
- 14 $(\Sigma_1, \sigma_1) \leftarrow \text{AdaptorMultiSign}^1(m_{2,1}, x_{\mathcal{A}_1}, t_{\mathcal{A}}/T_{\mathcal{A}}, \text{proof}, x_{\mathcal{B}_1})$
- 15 $\text{VerifyOffset}^1(\Sigma_1, m_{2,1}, \text{proof}, X_{\mathcal{AB}_1}) \stackrel{?}{=} 1$
- 16 $(\Sigma_2, \sigma_2) \leftarrow \text{AdaptorMultiSign}^2(m_{2,2}, x_{\mathcal{A}_2}, t_{\mathcal{A}}/T_{\mathcal{A}}, \text{proof}, x_{\mathcal{B}_2})$
- 17 $\text{VerifyOffset}^2(\Sigma_2, m_{2,2}, \text{proof}, X_{\mathcal{AB}_2}) \stackrel{?}{=} 1$
- 18 $\xleftarrow{\Sigma_2}$
- 19 $\sigma_2 \leftarrow \text{Offset}^2(\Sigma_2, -t_{\mathcal{A}}/\Theta T_{\mathcal{A}})$
- 20 $\text{VerifySig}^2(\sigma_2, m_{2,2}, X_{\mathcal{AB}_2}) \stackrel{?}{=} 1$
- 21 If time is $< \tau_2$, then $\text{Transaction}^2(m_{2,2}, \sigma_2)$
- 22 $t_{\mathcal{A}}/T_{\mathcal{A}} \leftarrow \text{GetOffset}^2(\Sigma_2, \sigma_2)$, $\sigma_1 \leftarrow \text{Offset}^1(\Sigma_1, -t_{\mathcal{A}}/\Theta T_{\mathcal{A}})$
- 23 $\text{Transaction}^1(m_{2,1}, \sigma_1)$

Output: \mathcal{A} has e_2 token in Chain₂ address $X_{\mathcal{A}_2}$, \mathcal{B} has e_1 in Chain₁ address $X_{\mathcal{B}_1}$

Req. 11.1: \mathcal{B} should be able to Offset^1 in Step 21.

At the onset of the protocol, \mathcal{A} and \mathcal{B} agree on the details of the swap, such as the exchange rates (e_1/e_2), timing ($\tau_1 > \tau_2$), and public addresses $X_{\mathcal{B}_1}, X_{\mathcal{B}_2}, X_{\mathcal{A}_1}, X_{\mathcal{A}_2}$. If $X_{\mathcal{A}_1} \neq X_{\mathcal{A}_2}$, then \mathcal{A} must prove that she knows the corresponding private key $x_{\mathcal{A}_1}$ to avoid rogue key attacks [12]. This can be done by signing a message or using another type of zero-knowledge proof. The situation is similar for \mathcal{B} if $X_{\mathcal{B}_1} \neq X_{\mathcal{B}_2}$. In Steps 3 and 4, both \mathcal{A} and \mathcal{B} generate a multi-sig address for each chain, denoted by $X_{\mathcal{AB}_1}$ in Chain₁ and $X_{\mathcal{AB}_2}$ in Chain₂, respectively. In Step 5-6, \mathcal{A} initiates a transaction in Chain₁, transferring e_1 tokens to the address $X_{\mathcal{AB}_1}$, and sets a timeout τ_1 on it. At this point, the token is jointly owned by both \mathcal{A} and \mathcal{B} until τ_1 . In other words, to generate a signature for this multi-sig address, both parties – with their respective private keys – have to collaborate. The timeout condition is needed to ensure that the assets do not get locked indefinitely if one of the parties goes offline. This timeout mechanism triggers a refund transaction if a second transaction does not occur. Once \mathcal{B} deems the transaction in Chain₁ final, in Steps 7-8, he initiates a similar transaction in Chain₂. This involves transferring e_2 tokens to the multi-sig address $X_{\mathcal{AB}_2}$ with a timeout τ_2 . The address $X_{\mathcal{AB}_2}$ is a public key that combines $X_{\mathcal{A}_2}$ and $X_{\mathcal{B}_2}$. Therefore, once the tokens are transferred to $X_{\mathcal{AB}_2}$ after Step 6, \mathcal{A} has no means to unilaterally transfer them without \mathcal{B} 's consent, and similarly, \mathcal{B} does not have ac-

cess to them without \mathcal{A} 's acknowledgment. Once both tokens are locked, both parties generate two messages to swap the ownership of the assets. Here, $m_{2,1}$ is a transaction message triggering the transfer of e_1 tokens from $X_{\mathcal{AB}_1}$ to $X_{\mathcal{B}_1}$ in Chain₁ (Step 9), and in an analogous fashion $m_{2,2}$ does the same with e_2 tokens from $X_{\mathcal{AB}_2}$ to $X_{\mathcal{A}_2}$ in Chain₂ when valid signatures are provided (Step 10). Once a signed message $m_{2,2}$ is submitted to Chain₂, \mathcal{A} 's balance gets updated. Similarly, when a signed message $m_{2,1}$ is submitted to Chain₁, then e_1 tokens in Chain₁ are transferred to \mathcal{B} . At Step 11, \mathcal{A} generates an offset, denoted as $t_{\mathcal{A}}/T_{\mathcal{A}}$, which is used later to compute two adaptor signatures. These adaptor signatures are calculated using the same offset (Steps 13-17). In Step 11 \mathcal{A} transfers the public *proof* corresponding to the offset. If Chain₁ and Chain₂ implement different signature algorithms, \mathcal{A} must generate the public version of the offset for both chains and convince \mathcal{B} that they are the same, without disclosing the actual offset value. In Step 13, this can be done, with Protocol $\text{VerifyPublicKeyPairs}$ that works only for integer $t_{\mathcal{A}}$. In this case the proof is two elliptic curve points $T_{\mathcal{A}}^1$ and $T_{\mathcal{A}}^2$ on different curves corresponding to the same integer $t_{\mathcal{A}}$. Once \mathcal{B} confirms that $T_{\mathcal{A}}^1$ and $T_{\mathcal{A}}^2$ both conceal the same offset, the transaction is given the green light. Note that, when Chain₁ and Chain₂ implement the same elliptic curve, then Step 13 can be omitted. If Chain₁ uses BLS signatures, the offset must be an elliptic curve point. In this case the swap can only be performed if Chain₂ also implements BLS on the same elliptic curve, thereby alleviating the need for Protocol $\text{VerifyPublicKeyPairs}$.

To execute the atomic swap in its entirety, the same AdaptorMultiSign protocol has to be executed between the parties twice (Steps 14 and 16). Once for $m_{2,1}$ in Steps 14-15, and again for $m_{2,2}$ in Steps 16-17. This process requires active participation from both parties. Should the verification algorithm not return 1 for either of the parties, they immediately abandon the deal.

In Step 18, \mathcal{B} sends the offset signature Σ_2 to \mathcal{A} . Then, in Step 19, with the knowledge of $t_{\mathcal{A}}/T_{\mathcal{A}}$, \mathcal{A} offsets Σ_2 to obtain the valid signature σ_2 . In Step 20-21, the final stage of the protocol, \mathcal{A} submits $m_{2,2}$ to Chain₂ along with σ_2 . This action in Step 21 furnishes \mathcal{B} with the necessary secret nonce $t_{\mathcal{A}}/T_{\mathcal{A}}$, to sign $m_{2,1}$ in Chain₁. Consequently, in Step 23, \mathcal{B} can use this signature to submit a transaction and acquire e_1 tokens in Chain₁, directed to the address $X_{\mathcal{B}_1}$. This step concludes the swap.

V. COMPATIBILITY AND LIMITATIONS

Table I summarises the compatibility of each primitive. We wish to draw attention to the main limitation of the proposed scheme. Note that Protocol $\text{VerifyPublicKeyPairs}$ requires an integer t as an input, and GetOffset is not going to work for BLS with an integer offset. On the other hand, observe that, these Primitives are required to operate only in Chain₂ in Protocol 11. Consequently, we introduce a condition of asymmetry here. Chain₂ must not use the BLS signature scheme unless the other chain uses BLS on the

TABLE I: Primitives by Signature Scheme

offset	Schnorr	ECDSA	BLS	
	integer	integer	point	integer
Sign, VerifySig, MultiPubKey	✓	✓	✓	✓
GenerateOffset	✓	✓	✓	(✓)
Offset, VerifyOffset	✓	✓	✓	✓
GetOffset	✓	✓	✓	
AdaptorMultiSign	✓	✓	✓	✓

same elliptic curve. In other words, swaps between BLS and Schnorr/ECDSA fit into this picture only if Chain₁ is the one with BLS signatures. Note that the role of \mathcal{A} and \mathcal{B} is not completely symmetric. For the stochastic implications of this fact the interested reader is referred to [62] and [26]. Finally, note that when working with the same elliptic curves `VerifyPublicKeyPairs` can be omitted and the protocol can even cope with situations when both chains use BLS signatures.

VI. FAIRNESS

In this subsection, we discuss the fairness aspects of the above-described protocol. We prove Req. 8.5 at the level of abstraction that we have just defined. Our goal was to discuss the security of atomic swaps in such a way that it is independent of the signature scheme. Security and the signature scheme depend on the respective formulas, but these are concealed at the current level of abstraction within Req. 1.0, 3.0, 4.0, 4.1, 6.1, 7.1, 8.0-8.4, 9.0-9.2, 10.0. The specific formulas for Schnorr, and BLS signatures are given below. The proofs of the individual requirements for specific signature schemes are quite complex in each case and are therefore omitted to keep the text as clear as possible. Note that our notion of security is somewhat similar to the one employed for multiparty computation in the UC framework [54], however here we rely on the requirements for each primitive and investigate the fairness aspects of the protocol accordingly. Proving that a specific primitive is secure can be challenging, given that the definition of “public information” is quite broad. Side-channel, nonce reuse and virtual machine rewinding attacks can further weaken the security of these primitives.

Theorem 1. *The Protocol 11. - AtomicSwap is fair.*

Proof. To prove fairness (see definition in [17]), first, we verify correctness (the expected outcome of the protocol); \mathcal{A} with e_2 tokens in Chain₂ at address X_{A_2} , \mathcal{B} with e_1 tokens in Chain₁ at address X_{B_1} . Then, let us suppose that the protocol is aborted at each stage and see whether the parties can learn something that would then enable them to calculate the final output of the protocol while the other party is deprived of this option.

First, observe that – `GetOffset` aside, which reveals t/T – no secret information is leaked during the execution of primitives and protocols, as for every secret value we have a requirement that computing it from any public information is hard (see Req. 1.0, 3.0, 4.0, 8.0, and 8.1). Such strict requirements are necessary for our formal proof, as they allow us to assume that no secret information is leaked.

Step 1-5: According to Req. 7.1, the value X_{AB_1} computed by \mathcal{A} is the same as the one computed by \mathcal{B} . The same holds for X_{AB_2} . According to Req. 1.0, the public addresses can be revealed. To be able to run Protocol `AdaptorMultiSign` later, \mathcal{A} sends a zero-knowledge proofs to \mathcal{B} to demonstrate that he knows x_{A_1} and x_{A_2} , and similarly, \mathcal{A} makes sure that \mathcal{B} knows x_{B_1} and x_{B_2} .

Step 6: According to Req 9.2, \mathcal{A} can recover the tokens at τ_1 as long as \mathcal{B} does not submit m_1 with σ_1 . According to Req. 9.1, the transaction is valid, and Chain₁ should be able to accept it.

Step 7-8: The situation is similar to Step 6, \mathcal{B} can recover the tokens at $\tau_2 < \tau_1$ if \mathcal{A} does not submit m_2 with σ_2 .

Step 9-11: There is no change in ownership.

Step 12: According to Req. 3.0. the *proof* is sent to \mathcal{B} .

Step 13: Due to Req. 10.0, \mathcal{B} has insufficient information to recover t_A/T_A .

Step 14: \mathcal{A} can initiate Protocol `AdaptorMultiSign` to generate a multi-sig for public key X_{AB_1} and message $m_{2,1}$, because the entry conditions are met: $X_{AB} = \text{MultiPubKey}(X_A, X_B)$, and \mathcal{A} is in possession of $x_A, t_A/T_A, X_B$ and \mathcal{B} is in poss. of x_B, X_A . According to Req. 8.0, $x_{A_1}, t_A/T_A$ are not revealed to \mathcal{B} , and to Req. 8.1, x_B is not revealed to \mathcal{A} . According to Req. 8.2, if the protocol is fully executed \mathcal{B} will possess an offset signature Σ_1 for address X_{AB_1} and message $m_{2,1}$. According to Req. 8.4, \mathcal{B} has no means to calculate σ_1 . According to Req. 8.5, the protocol is fair, thus in Step 14 either both parties get what they want or neither of them does.

Step 15: \mathcal{B} verifies multi-sig Σ_1 . At this point, once \mathcal{B} learns t_A/T_A , he can get hold of e_1 tokens. For this reason, he is willing to participate in a second round of Protocol `AdaptorMultiSign` initiated by \mathcal{A} to generate a multi-sig for public key X_{AB_2} and message $m_{2,1}$.

Step 16: Similar to Step 14. \mathcal{A} initiates Protocol `AdaptorMultiSign` again, this time to generate a multi-sig for public key X_{AB_2} and message $m_{2,2}$. According to Req. 8.2 and 8.4. of Protocol `AdaptorMultiSign` only \mathcal{B} knows Σ_2 . This, with Req. 8.1, ensures the impossibility of \mathcal{A} learning σ_2 without \mathcal{B} sharing Σ_2 .

Step 17-18: After the Protocol \mathcal{B} verifies Σ_2 whether it is a valid offset signature with the same t_A/T_A as Σ_1 . If yes, he sends it to \mathcal{A} .

Step 19-21: \mathcal{A} receives Σ_2 , and accordingly to Req. 8.3 and Req 4.1 she should be able to de-offsets using t_A/T_A . If the obtained signature is not valid for m_2 , she can terminate the process and wait till τ_1 to revert transaction $m_{1,1}$. Note that \mathcal{B} still does not have sufficient knowledge to compute a signature of $m_{2,1}$, as he only knows the corresponding offset signature Σ_1 . \mathcal{A} can terminate the swap if the signature is not valid.

Step 21: If there is time until τ_2 then \mathcal{A} can submit σ_2 to Chain₂ with $m_{2,2}$ to transfer e_2 token to his walet X_{A_2} . If the time for a successful transaction is not sufficient until τ_2 then the swap is aborted, and \mathcal{A} waits until τ_1 to revert his tokens in Chain₁.

Step 22: \mathcal{B} has an adaptor signature Σ_1 and a regular

signature σ_1 . If the signature scheme employs a nonce, Req. 6.1 must be satisfied. This is ensured by the output of Protocol `AdaptorMultiSign`, as both Σ_1 and σ_1 must use the same nonce. It will be enough for \mathcal{B} to get his e_1 token in `Chain_1`. \mathcal{B} had to perform these steps before τ_1 , where $\tau_2 < \tau_1$.

Step 23: The protocol terminates as prescribed. \square

VII. INSTANTIATION

Let $f()$ denote a one-way function that maps a field element in the range $[0, q-1]$ to an element of a cyclic group of order q . In the context of ECC, this output is a point on an EC over a finite field \mathbb{F}_p . q and p are typically primes of size $\sim 2^{256}$. The one-way function $f()$ is assumed to be homomorphic (or a.k.a. additive) [63], i.e. there exists an efficient algorithm for operator \oplus , \ominus and \times such that:

$$f(x+y) = f(x) \oplus f(y), f(yx) = y \times f(x). \quad (1)$$

where x and y can be any field element.

A. Primitives for Schnorr signatures

This subsection illustrates how the primitives we described above can be implemented using Schnorr signatures, calculated with to the following congruence modulo the prime q :

$$s \equiv r + h \cdot x \pmod{q}. \quad (2)$$

If we fix the value for any three of the variables $1 \leq s, r, h, x < q$, then there shall be only one possible value for the fourth variable (may be zero) that satisfies the congruence. In a similar vein, they satisfy the following equation as well:

$$f(s) = f(r) \oplus h \times f(x), \quad (3)$$

where $f(s)$, $f(r)$ and $f(x)$ are group elements, while h is an integer.

Schnorr Prm. 1: `Sign(m, x) = σ , $f(s) = R \oplus h \times X$`

$$X \leftarrow f(x), r \xleftarrow{\$} Z_q, R \leftarrow f(r), h \leftarrow H(m|R|X), \\ s \leftarrow r + h \cdot x \pmod{q}, \sigma \leftarrow (R, s)$$

return σ

Schnorr Prm. 2: `VerifySig(σ, m, X) = 0/1`

$$(R, s) \leftarrow \sigma, S \leftarrow f(s), h \leftarrow H(m|R|X)$$

return $S \stackrel{?}{=} R \oplus h \times X$

The Schnorr Primitive 1 demonstrates how a valid signature is generated for a message using the secret key x . The algorithm uses the hash of the message, where the hash is calculated using a function $H()$ that maps an array of characters of arbitrary length to an integer $[1, q-1]$. $H()$ is invoked on a string that is a concatenation of the message, $f(r)$ and $f(x)$ to make the signature ephemeral.

The signature has two fields. The random nonce concealed by the one-way function $R = f(r)$ and s . Note that s does not reveal the secret key x as long as r is hidden. Computing r from $R = f(r)$ is as hard as computing x from $f(x)$ (i.e. an (EC)DLP instance). Negligible information about x is revealed through $f(s) = R \oplus h \times X$, which by itself is useless. The use of the same nonce for two signatures lead to nonce reuse attacks.

For the verification of signatures refer to Schnorr Primitive 2 relying on the homomorphic property of the one-way function $f()$. In other words, instead of verifying Eq. (2), the algorithm verifies whether the equality of Eq. (3) holds.

Schnorr Prm. 3: `GenerateOffset(*) = t, T`

$$t \xleftarrow{\$} Z_q, T \leftarrow f(t) \\ \text{return } t, T$$

Schnorr Prm. 4: `Offset(σ, t) = Σ`

$$(R, s) \leftarrow \sigma, T \leftarrow f(t), \Sigma \leftarrow (R, s + t)$$

return Σ

Schnorr Prm. 5: `VerifyOffset(Σ, m, T, X) = 0/1`

$$(R, s') \leftarrow \Sigma, S' \leftarrow f(s'), h \leftarrow H(m|R|X).$$

return $S' \ominus T \stackrel{?}{=} R \oplus h \times X$

Schnorr Primitives 3, 4, 5, 6 are responsible for concealing valid signatures with an offset. When the two chains in the swap use different ECs, then Schnorr Primitive 3 returns two elliptic curve points, one for each chain. Here a random offset t is added to Eq. (2). Note that, `Offset(Offset(σ, t), $-t$) = σ` , as $R = R \oplus T \ominus T$ and $s = s + t - t$. Once an adaptor signature, a public offset, a message, and a public key are obtained, one can verify that this is a valid adaptor signature concealing a valid signature that would be accepted by the consensus rules of the chain. Validation can be executed using Primitive 5.

With Σ and σ one can generate t using Primitive `GetOffset`. For `GetOffset` to function correctly, Σ has to be an offset signature computed from σ , i.e. using the same random nonce R .

Schnorr Prm. 6: `GetOffset(Σ, σ) = t`

$$(R, s') \leftarrow \Sigma, (R, s) \leftarrow \sigma, t \leftarrow s' - s \\ \text{return } t$$

Schnorr Primitive 6 postulates how an offset signature can be transformed into a valid one. For this maneuver, we have the following two congruences

$$s' \equiv r + t + h \cdot x \pmod{q}, \quad (4)$$

$$s \equiv r + h \cdot x \pmod{q}, \quad (5)$$

where Eq. (4) holds, because `VerifySig(σ, m, x) = 1`, where x is the private key, $h = H(m|R|X)$ and r is the random nonce, and Eq. (5) holds because `VerifyOffset(Σ, m, T, X) = 1`, where t is the offset, and both signatures σ and Σ use the same nonce r .

For two parties to interact on the chain, we devise an additional multi-sign primitive. For various pieces of (secret or public) information, we use a subscript indicating the party who generated it, e.g., we use $X_{\mathcal{A}}$ for the public key of \mathcal{A} and $X_{\mathcal{B}}$ for the public key of \mathcal{B} .

For the implementation of `MultiPubKey` under the Schnorr signature scheme, refer to Schnorr Prm. 7. The signing process for multi-sig addresses is presented in Schnorr Protocol 8. Generating a random nonce by two parties can be

Schnorr Prm. 7: $\text{MultiPubKey}(x_A, X_B) = X_{AB}$

$X_A \leftarrow f(x_A)$
return $X_A \oplus X_B$

Schnorr Prot. 8: $\text{AdaptorMultiSign}(m, x_A, t_A, T_A, x_B) = \Sigma_{AB}, \sigma_{AB}$

\mathcal{A} knows x_A, t_A, X_B \mathcal{B} knows x_B, X_A

- 1 $X_{AB} \leftarrow X_A \oplus X_B = \text{MultiPubKey}(x_A, X_B)$
- 2 $X_{AB} \leftarrow X_A \oplus X_B = \text{MultiPubKey}(x_B, X_A)$
- 3 $r_A \xleftarrow{\$} Z_q, R_A \leftarrow f(r_A)$ $r_B \xleftarrow{\$} Z_q, R_B \leftarrow f(r_B)$
- 4 $\leftarrow R_B, ZKP(r_B) \longrightarrow$
- 5 $T_A \leftarrow f(t_A), R \leftarrow R_A \oplus R_B, h \leftarrow H(m|R|X_{AB})$
- 6 $s_1 \leftarrow r_A + hx_A \pmod{q}, \hat{\sigma}_1 \leftarrow (R_A, s_1)$
- 7 $\Sigma_1 \leftarrow \text{Offset}(\hat{\sigma}_1, t_A)$
- 8 $\leftarrow \Sigma_1, T_A, ZKP(r_A) \longrightarrow$

$(R, s'_1) \leftarrow \Sigma_1, h \leftarrow H(m|R|X_{AB})$
 $s_2 \leftarrow r_B + hx_B \pmod{q}, \Sigma_{AB} \leftarrow (R, s'_1 + s_2)$

$\text{VerifyOffset}(\Sigma_{AB}, m, T_A, X_{AB}) \stackrel{?}{=} 1$

$\leftarrow \Sigma_{AB}$

$\sigma_{AB} \leftarrow \text{Offset}(\Sigma_{AB}, -t_A)$
 $\text{VerifySig}(\sigma_{AB}, m, X_{AB}) \stackrel{?}{=} 1$

done such that \mathcal{B} generates a random nonce r_B and sends $R_B = f(r_B)$ to \mathcal{A} . Next, \mathcal{A} generates a random nonce r_A , computes $R_A = f(r_A)$ and then the parties eventually use $R_A \oplus R_B$ as the common nonce. To avoid rogue key attacks \mathcal{B} sends a zero-knowledge (ZK) proof of r_B to \mathcal{A} besides R_B , and similarly \mathcal{A} sends a ZK proof of r_A to \mathcal{B} , see [64].

In Step 2 and 3, a random nonce r_B is generated by \mathcal{B} , and the corresponding public value (calculated by applying the one-way function) is sent to \mathcal{A} . In Step 4 and 5, \mathcal{A} performs a process similar to $\text{Sign}(m, x_A)$; however, the hash of the message is concatenated with X_{AB} and a random nonce generated by both \mathcal{A} and \mathcal{B} as described by the above process. In Step 6, \mathcal{A} generates the signature using r_A only, which is offset by a random nonce t_A and sent to \mathcal{B} along with R and T_A . Without knowing r_A nothing is revealed about x_A .

Observe a similar situation on \mathcal{B} 's side, he also computes h by concatenating m , R and X_{AB} . Now \mathcal{B} can compute a common $s = s_1 + s_2$. The calculated signature corresponds to the following congruence relation:

$$s_1 + s_2 \equiv r_A + r_B + t + h \cdot x_A + h \cdot x_B \pmod{q} \quad (6)$$

which holds if and only if

$$f(s_1 + s_2) = f(r_A) \oplus f(r_B) \oplus T \oplus h \times (f(x_A) + f(x_B)),$$

where $h = H(m | (f(r_A) \oplus f(r_B)) | (f(x_A) + f(x_B)))$.

Usually a verifiably offset signature that contains a valid signature is called an *adaptor signature* [12], [18], [19], [40]. In Step 8, \mathcal{B} has a valid adaptor signature Σ_{AB} for X_{AB} the Primitive 5 returns with e_2 on this side, attesting the validity of Σ_{AB} . Finally, when Σ_{AB} is sent to \mathcal{A} , she can extract the witness for T_A ($-t_A$) and calculate σ_{AB} .

B. Primitives using Boneh-Lynn-Shacham (BLS) signatures

The BLS signature algorithm operates in gap Diffie-Hellman groups where the decisional Diffie-Hellman (DDH) is easy, while the computational Diffie-Hellman (CDH) is hard. The

scheme makes use of two homomorphic one-way functions, $f_1()$ and $f_2()$, mapping field elements to EC points on two pairing-friendly elliptic curves, forming two distinct cyclic groups. These mappings adhere to the properties defined in Eq. (1). We denote the group operation for the first cyclic group associated with $f_1()$ by \oplus , and $G_1 = f_1(1)$ as its generator. Similarly, for $f_2()$, the group operation is denoted by \oplus and $G_2 = f_2(1)$ as its generator.

The validity of a BLS signature is ascertained by using bilinear pairings (Weil pairing [65]), denoted $e(x, y)$ in this work. $e(x, y)$ takes two EC points and produces an m -th root of unity. The pairing is used to check whether $z \equiv x \cdot y \pmod{q}$ holds without knowing x, y , and z , by testing whether $e(f_1(x), f_2(y)) \stackrel{?}{=} e(f_1(z), G_2)$ holds. This computation furnishes one with a DDH answer. The security of BLS signatures can be provably reduced to the hardness of the Computational Diffie-Hellman (CDH) problem [14].

Another important aspect of BLS signatures is that messages are hashed to elliptic curve points by using the constant time Shallue-Woestijne-Ulas (SWU) map [66]. Public keys are generated as $X = f_2(x)$ where x is the secret key.

BLS Prm. 1: $\text{Sign}(m, x) = \sigma, \quad e(S, G_2) = e(\text{SWU}(H(m)), X)$

$h \leftarrow H(m), \quad H \leftarrow \text{SWU}(h), \quad S \leftarrow xH, \quad \sigma \leftarrow (S)$
return σ

BLS Prm. 2: $\text{VerifySig}(\sigma, m, X) = 0/1$

$(S) \leftarrow \sigma, \quad h \leftarrow H(m), \quad H \leftarrow \text{SWU}(h)$
return $e(S, G_2) \stackrel{?}{=} e(H, X)$

BLS Prm. 1 and 2 outline the implementation of calculating and verifying signatures. The signature itself is a point S in the group associated with $f_1()$. To compute the signature, the message m is first hashed to the curve using the SWU map. The signature is verified as follows. The hash of the message m is calculated and mapped to the curve to obtain point H . Then, with the use of the bilinear pairing one verifies that the signature is valid according to the following identities $e(H, X) = e(H, xG_2) = e(xH, G_2) = e(S, G_2)$.

BLS Prm. 3: $\text{GenerateOffset}(m_1, m_2, x) = T, \text{proof}$

$(S_1) \leftarrow \text{Sign}(m_1, x), \quad (S_2) \leftarrow \text{Sign}(m_2, x)$
return $S_1 \ominus S_2, (m_1, m_2)$

BLS Prm. 4: $\text{Offset}(\sigma, T) = \Sigma$

$(S) \leftarrow \sigma, \quad \Sigma \leftarrow (S \oplus T)$
return Σ

BLS Primitives 3, 4, 5, 6 are responsible for concealing valid signatures using an offset. The offset we use for BLS slightly differs from the offset used for Schnorr and ECDSA. This is because a BLS signature – as opposed to Schnorr and ECDSA signatures – is an EC point and it has no field elements. For the protocol to work with BLS signatures the offset must be an EC point with $f_2()$. In this case, we define

the offset not as a random value but as the difference of signatures for m_1 and m_2 . Such an offset meets Req. 1.0, as x cannot be computed from T . The proof of the offset is going to be the input of the primitive. Offsetting the signature involves adding the offset to the signature. It is easy to verify that such operation meets Req. 4.0–4.1 defined in `Offset`. BLS Prm. 5 `VerifyOffset` verifies the validity of the offset, using the two messages m_1 and m_2 and exploiting BLS aggregation:

$$\begin{aligned} e(H_1 \oplus H_2, X) &= e(H_1 \oplus H_2, xG_2) = e(x(H_1 \oplus H_2), G_2) = \\ e(xH_1 \oplus xH_2, G_2) &= e(S_1 \oplus S_2, G_2) = e(T, G_2). \quad (7) \end{aligned}$$

BLS Prm. 5: `VerifyOffset`(Σ, m_1, m_2, X) = 0/1

$(T) \leftarrow \Sigma$, $h_1 \leftarrow H(m_1)$, $H_1 \leftarrow SWU(h_1)$,
 $h_2 \leftarrow H(m_2)$, $H_2 \leftarrow SWU(h_2)$
return $e(T, G_2) \stackrel{?}{=} e(H_1 \oplus H_2, X)$

BLS Prm. 6: `GetOffset`(Σ, σ) $\leftarrow T$

$(S', T_1) \leftarrow \Sigma$, $(S) \leftarrow \sigma$, $T \leftarrow S' \oplus S$
return T

BLS Prm. 7: `MultiPubKey`(X_A, X_B) = X_{AB}

return $X_A \oplus X_B$

BLS Protocol 8: `AdaptorMultiSign`(m, x_A, T_A, x_B) $\leftarrow \Sigma_{AB}, \sigma_{AB}$

A knows x_A, T_A, X_A B knows x_B, X_B
1 $\Sigma_{AB} = \text{Sign}(m, x_B)$
2 -----
3 $\sigma_{AB} \leftarrow \text{Offset}(\Sigma_{AB}, T_A)$
4 $\text{VerifySig}(\sigma_{AB}, m, X_{AB}) \stackrel{?}{=} 1$

We can implement `GetOffset` for BLS in a slightly restricted way, such that the function returns only an EC point. The `MultiPubKey` primitive can be implemented for BLS using the same logic as for Schnorr. Last, but not least we define Protocol `AdaptorMultiSign` for BLS.

BLS Protocol 11 is designed to accommodate BLS signatures; i.e. Protocol 11 instantiated with BLS, assuming that they use the same elliptic curves. Primitive `GenerateOffset` returns a point T_A , and without the corresponding field element, one has no means to run Protocol `VerifyPublicKeyPairs`. Here we emphasise that the two chains must use the same sets of pairing-friendly curves. Next Primitive `AdaptorMultiSign` generates two valid signatures Σ_1 and Σ_2 . One can use this information and T_A can be verified using Primitive `VerifyOffset`. The execution of the second `VerifyOffset` can be omitted as computing Σ_2 did not involve \mathcal{A} . Then Σ_2 is sent to \mathcal{A} , who can de-offset the signature to obtain a valid signature σ_2 for message $m_{2,2}$. Once \mathcal{B} learns σ_2 he can obtain the offset S_A and with it offset Σ_1 to generate σ_1 .

BLS Protocol 11: AtomicSwap BLS using the same set of elliptic curves

\mathcal{A} knows $x_{A_1}, x_{A_2}, X_{B_1}, X_{B_2}$ \mathcal{B} knows $x_{B_1}, x_{B_2}, X_{A_1}, X_{A_2}$
11 Steps 1-10 are the same as in Protocol 11
12 $(S_A) \leftarrow \text{Sign}(m_{2,1}, x_A)$, $(S_2) \leftarrow \text{Sign}(m_{2,2}, x_A)$
13 $T_A \leftarrow S_A \oplus S_2$ T_A
14 $h_1 \leftarrow H(m_{2,1})$, $H_1 \leftarrow SWU(h_1)$, $\Sigma_1 \leftarrow x_B H_1$
15 $h_2 \leftarrow H(m_{2,2})$, $H_2 \leftarrow SWU(h_2)$, $\Sigma_2 \leftarrow x_B H_2$
16 $e(T_A, G_2) \stackrel{?}{=} e(H_1 \oplus H_2, X_A)$
17 $\sigma_2 \leftarrow (\Sigma_2 \oplus S_2)$ Σ_2
18 $\text{VerifySig}^2(\sigma_2, m_{2,2}, X_{AB_2}) \stackrel{?}{=} 1$
19 If time is $< \tau_2$, then `Transaction`²($m_{2,2}, \sigma_2$)
20 $S_A \leftarrow \sigma_2 \oplus \Sigma_2 \oplus T_A$, $\sigma_1 \leftarrow (\Sigma_1 \oplus S_A)$
21 `Transaction`¹($m_{2,1}, \sigma_1$)

BLS Prm. (integer offset) 4: `Offset`(σ, t) = Σ

$(S) \leftarrow \sigma$, $T \leftarrow f_2(t)$, $\Sigma' \leftarrow \text{Sign}(m, t)$, $\Sigma \leftarrow (\Sigma' \oplus S, T)$
return Σ

C. BLS Primitives with integer offset on `Chain2`

Finally, to be able to perform a swap when `Chain1` uses BLS, while `Chain2` has primitive `GetOffset`, we need a second variant of Primitives `Offset` and `VerifyOffset`. To this end, Protocol `AdaptorMultiSign` for BLS is introduced. Here we use an integer offset (t_A) and exploit the properties of BLS signature aggregation as in [55]. BLS Prm. 5 `VerifyOffset` verifies the validity of the offset using the two messages m_1 and m_2 . `AdaptorMultiSign` with integer offsets is also defined, where we shift randomness in the private key using the offset (t_A).

VIII. CONCLUSION

To promote decentralisation in cryptocurrencies they should be exchanged in a decentralised manner, via atomic exchanges between the parties. To further attract attention to atomic exchanges, standardisation in the space is necessary. So far atomic swaps have been devised for specific pairs of chains. In this work, we synthesized and simplified these approaches by breaking them down into Primitives and Protocols. Some

BLS Prm. (integer offset) 5:

`VerifyOffset`(Σ, m, T, X) = 0/1
return `VerifySig`($\Sigma, m, X \oplus T$)

BLS Protocol (integer offset) 8:

`AdaptorMultiSign`(m, x_A, t_A, x_B) = Σ_{AB}, σ_{AB}

\mathcal{A} knows x_A, t_A, X_A \mathcal{B} knows x_B, X_B
1 $(S_A) \leftarrow \text{Sign}(m, x_A)$, $(\Sigma_A) \leftarrow \text{Sign}(m, t_A)$
2 $T_A \leftarrow f_2(t_A)$, $\Sigma_1 \leftarrow S_A \oplus \Sigma_A$
3 $(\Sigma_{AB}) \leftarrow \text{Sign}(m, x_B)$
4 $\text{VerifySig}(\Sigma_1, m, X_A \oplus T_A) \stackrel{?}{=} 1$

 Σ_{AB}
Once \mathcal{B} performs $t_A \leftarrow \text{GetOffset}^2(\Sigma_2, \sigma_2)$
He computes $(\Sigma_A) \leftarrow \text{Sign}(m, t_A)$ and $S_A \leftarrow \Sigma_1 \oplus X_A$

of these primitives are slightly different from the de facto counterparts; namely, we propose adaptor signatures that are an offset version of a multi-signature, where the offset can be an integer or even an elliptic curve point. Our primary aim was to define a minimum set of Primitives and Protocols necessary to bring forth atomic swaps between chains using different signature schemes. The proposed protocol specification has been synthesised from atomic swap schemes developed for the Schnorr signature scheme, but with our modifications, it is general enough to work with deterministic signatures (BLS).

REFERENCES

- [1] D. Castejon-Molina, et.al., “A cryptographic layer for the interoperability of CBDC and cryptocurrency ledgers,” Cryptology ePrint 2023/116.
- [2] E. Tairi, et.al., “LedgerLocks: A security framework for blockchain protocols based on adaptor signatures,” Cryptology 2023/1315.
- [3] N. Satoshi, “Bitcoin: A peer-to-peer electronic cash system.” 2008.
- [4] R. A. Schwartz and L. Peng, *Market Makers*. Springer US, 2013.
- [5] dYdX, “dydx Info,” <https://dydx.exchange/>, accessed on Dec. 2024.
- [6] J. Guggen, “Bitcoin-monero cross-chain atomic swap,” *Cryptology*, 2020.
- [7] K. Kajita, G. Ohtake, T. Takagi, “Consecutive adaptor signature scheme: From two-party to n -party settings,” Cryptology 2024/241.
- [8] S.A.K. Thyagarajan, et.al., “Universal atomic swaps: Secure exchange of coins across all blockchains,” in *IEEE SP*, 2022.
- [9] R. Van Der Meyden, “On the specification and verification of atomic swap smart contracts,” in *Int. IEEE ICBC*, 2019, pp. 176–179.
- [10] D. Miraz et.al., “Atomic cross-chain swaps: Development, trajectory and potential of non-monetary digital token swap facilities,” *AETIC*, 2019.
- [11] P. Hoensch and L. S. del Pino, “Atomic swaps between Bitcoin and Monero,” *arXiv preprint arXiv:2101.12332*, 2021.
- [12] A. Erwig, S. Faust, K. Hostáková, M. Maitra, and S. Riahi, “Two-party adaptor signatures from identification schemes,” in *IACR PKC*, 2021.
- [13] C.-P. Schnorr, “Efficient identification and signatures for smart cards,” in *Advances in Cryptology – CRYPTO*, 1989, pp. 239–252.
- [14] D. Boneh, B. Lynn, and H. Shacham, “Short signatures from the weil pairing,” in *ASIACRYPT*, Springer, 2001, pp. 514–532.
- [15] N. Kobitz, “Elliptic curve cryptosystems,” *Math. of Computation*, 1987.
- [16] D. Johnson, A. Menezes, and S. Vanstone, “The elliptic curve digital signature algorithm (ECDSA),” *Int. J. Inf. Sec.*, vol. 1, 2001.
- [17] S. Micali, “Simple and fast optimistic protocols for fair electronic exchange,” in *ACM PODC*, 2003, pp. 12–19.
- [18] L. Aumayr, et.al., “Generalized channels from limited blockchain scripts and adaptor signatures,” in *ASIACRYPT*, 2021.
- [19] G. Maxwell, et.al., “Simple schnorr multi-signatures with applications to bitcoin,” *Designs, Codes and Cryptography*, vol. 87, 09 2019.
- [20] B. Tu, M. Zhang, and C. Yu, “Efficient ecdsa-based adaptor signature for batched atomic swaps,” in *ISC*. Springer, 2022, pp. 175–193.
- [21] P. Wuille, J. Nick, and T. Ruffing, “Schnorr signatures for secp256k1,” *Bitcoin Improvement Proposal*, vol. 340, 2020.
- [22] F. Renan and P. Kutas, “SQIAsignHD: SQIAsignHD adaptor signature,” Cryptology ePrint Archive, Paper 2024/561, 2024.
- [23] L. Deng, H. Chen, J. Zeng, and L.-J. Zhang, “Research on cross-chain technology based on sidechain and hash-locking,” in *EDGE*, 2018.
- [24] M. Herlihy, B. Liskov, and L. Shrira, “Cross-chain deals and adversarial commerce,” *Proc. VLDB Endow.*, vol. 13, no. 2, pp. 100–113, oct 2019.
- [25] M. Belotti, S. Moretti, M. Potop-Butucaru, and S. Secci, “Game theoretical analysis of cross-chain swaps,” in *IEEE ICDCS*, 2020.
- [26] J. Xu, D. Ackerer, and A. Dubovitskaya, “A game-theoretic analysis of cross-chain atomic swaps with htcs,” in *IEEE ICDCS*, 2021.
- [27] J. Rueegger and G. S. Machado, “Rational exchange: Incentives in atomic cross chain swaps,” in *IEEE ICBC*, 2020, pp. 1–3.
- [28] Y. Xue and M. Herlihy, “Hedging against sore loser attacks in cross-chain transactions,” in *ACM PODC*, 2021.
- [29] R. Han, H. Lin, and J. Yu, “On the optionality and fairness of atomic swaps,” in *ACM AFT*, 2019.
- [30] J. A. Liu, “Atomic swaptions: Cryptocurrency derivatives,” *ArXiv*, 2018.
- [31] P. Robinson, et.al., “Atomic crosschain transactions for ethereum private sidechains,” *Blockchain: Research and Applications*, 2022.
- [32] M. Borkowski, et.al., “Towards atomic cross-chain token transfers: State of the art and open questions within tast,” TUW Report, 2018.
- [33] A. Dubovitskaya, D. Ackerer, and J. Xu, “A game-theoretic analysis of cross-ledger swaps with packetized payments,” in *FC*, 2021.
- [34] I. Tsabary, M. Yechieli, A. Manuskin, and I. Eyal, “Mad-htlc: Because htlc is crazy-cheap to attack,” in *IEEE SP*, 2021.
- [35] M. Herlihy, “Atomic cross-chain swaps,” in *ACM PODC*, 2018, pp.
- [36] V. Zakhary, D. Agrawal, and A. Abbadi, “Atomic commitment across blockchains,” *arXiv preprint arXiv:1905.02847*, 2019.
- [37] J. Kirsten and H. Davarpanah, “Anonymous atomic swaps using homomorphic hashing,” *Available at SSRN 3235955*, 2018.
- [38] G. Malavolta, et.al., “Anonymous multi-hop locks for blockchain scalability and interoperability,” in *Proc. NDSS Symposium*, 2019.
- [39] D. Boneh, et.al., “Aggregate and verifiably encrypted signatures from bilinear maps,” in *EUROCRYPT*, 2003.
- [40] L. Fournier, “One-time verifiably encrypted signatures a.k.a. adaptor signatures,” 2020, <https://github.com/LLFourn/one-time-VES/>.
- [41] V. Madathil, et.al., “Cryptographic oracle-based conditional payments,” Cryptology ePrint 2022/499.
- [42] X. Liu, et.al., “Adaptor signatures: New security definition and a generic construction for NP relations,” Cryptology ePrint 2024/1051.
- [43] A. Erwig and S. Riahi, “Deterministic wallets for adaptor signatures,” Cryptology ePrint Archive, Paper 2022/1450, 2022.
- [44] N. T. Courtois, et.al., “Private key recovery combination attacks: On extreme fragility of popular bitcoin key management,” Cryptology 2014
- [45] J. Breiter, et.al., “Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies,” Cryptology ePrint 2019/023.
- [46] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow, “Elliptic curve cryptography in practice,” in *FC*, 2014.
- [47] N. Heninger, et.al., “Mining your ps and qs: Detection of widespread weak keys in network devices,” in *USENIX Security Symposium*, 2012.
- [48] P. Q. Nguyen, et.al., “The insecurity of the elliptic curve digital signature algorithm with partially known nonces,” *Designs, Codes and Cryptography*, 2003.
- [49] M. Bellare and P. Rogaway, “Random oracles are practical: a paradigm for designing efficient protocols,” in *ACM CCS*, 1993.
- [50] N. Kobitz and A. J. Menezes, “The random oracle model: a twenty-year retrospective,” *Designs, Codes and Cryptography*, 2015.
- [51] R. Canetti, “Towards realizing random oracles: Hash functions that hide all partial information,” Cryptology ePrint 1997/007, 1997.
- [52] R. Canetti, O. Goldreich, and S. Halevi, “The random oracle methodology, revisited,” Cryptology ePrint 1998/011, 1998.
- [53] S.-T. Shen, A. Rezapour, and W.-G. Tzeng, “Unique signature with short output from cdh assumption,” in *Provable Security*, 2015.
- [54] R. Canetti, “Universally composable security: a new paradigm for cryptographic protocols,” in *Proc. IEEE Symposium on Foundations of Computer Science*, 2001, pp. 136–145.
- [55] H. Gokay, F. Baldimtsi, and G. Ateniese, “Atomic swaps for Boneh–Lynn–Shacham (BLS) based blockchains,” in *ESORICS*, 2024.
- [56] R. Belchior, et.al., “A survey on blockchain interoperability: Past, present, and future trends,” *ACM Computing Surveys (CSUR)*, 2021.
- [57] S.A.K. Thyagarajan, et.al., “Verifiable timed signatures made practical,” *ACM CCS*, 2020
- [58] S.A.K. Thyagarajan, G. Malavolta, F. Schmidt, and D. Schröder, “PayMo: Payment channels for monero,” Cryptology ePrint 2020/1441.
- [59] G. Ateniese, “Efficient verifiable encryption (and fair exchange) of digital signatures,” in *ACM CCS*, 1999.
- [60] J. Camenisch and M. Michels, “Separability and efficiency for generic group signature schemes,” in *Advances in Cryptology – CRYPTO*, 1999.
- [61] S. Noether, “Discrete logarithm equality across groups,” 2018, monero Research Lab, Technical Note MRL-0010.
- [62] B. Lądóczy, J. Bíró, and J. Tapolcai, “Stochastic analysis of the success rate in atomic swaps between blockchains,” in *BCCA*, 2022.
- [63] R. L. Rivest, L. Adleman, M. L. Dertouzos et al., “On data banks and privacy homomorphisms,” *Foundations of secure computation*, 1978.
- [64] D. Boneh, I. Haitner, and Y. Lindell, “Exponent-VRFs and their applications,” Cryptology ePrint Archive, Paper 2024/397, 2024.
- [65] J. H. Silverman, *The Arithmetic of Elliptic Curves*, Springer, 2009.
- [66] R. S. Wahby and D. Boneh, “Fast and simple constant-time hashing to the BLS12-381 elliptic curve,” Cryptology ePrint 2019/403.