WHITEPAPER

Lamport authenticated messaging for blockchains

William D. Doyle*, Pierre-Luc Dallaire-Demers†

Abstract. This paper introduces Lamport Authenticated Messaging on Blockchains (LAMB), a novel scheme for authenticating messages that combines blockchain technology with Lamport signatures. The proposed scheme aims to restore the features of modern asymmetric public-key cryptosystems, while providing increased security and robustness from quantum cryptanalysis. The paper conducts an analysis of the potential applications of LAMB such as securing digital assets and proving identity and also provides a discussion of the potential limitations of this new scheme.

KEY WORDS

1. Lamport Signatures. 2. Post Quantum Cryptography. 3. Authentication.

1. Introduction

The rapid advancement of fault tolerant quantum computers poses a significant threat to the security of traditional public key cryptosystems.[1–3] Despite ongoing efforts to develop new signature schemes to address this challenge,[4] only hash-based systems have been widely adopted and trusted due to their decades of use. However, these single-use key systems are not suitable for situations where public key infrastructure is required.

One possible solution is the use of Merkle trees, but this approach comes with limitations. The capacity of the tree must be known in advance, and although it is possible to sign the root of a new Merkle tree to extend its capacity, this makes the complexity of verification dependent on the history of the tree. Furthermore, key maintenance on the part of the signer becomes increasingly difficult with each message they sign, and there is no feedback mechanism to inform a signer if they have signed two messages with the same secret.

This paper presents a new solution for secure message authentication, called Lamport Authenticated Messaging on Blockchains (LAMB), that combines the resistance of Lamport signatures[5] with the flexibility of smart contracts to provide a more robust and secure method for authenticating messages. We examine the implementation of LAMB in Section 2, discuss the properties of this new system in Section 3, and evaluate its potential use case for securing digital assets in Section 4 and for proving identity in Section 5. We also discuss extensions of this scheme in Section 6.

* william.doyle@pauli.group (williamdoyle.eth), Pauli Group, Canada
† pierre-luc@pauli.group (pldd.eth), Pauli Group, Canada

2. Implementation

LAMB consists of a smart contract used to store Lamport public key hashes (PKH) and verify signatures against those PKHs. From there, the smart contract can execute arbitrary logic. Our implementation requires a blockchain compatible with the Ethereum Virtual Machine (EVM), however, the same technique can be applied on other blockchains supporting Turing complete logic. It should be noted however, that implementing LAMB on Unspent Transaction Output (UTXO) blockchains such as Cardano or Ergo would require some amount of optimization to take full advantage of the UTXO accounting model.

As Lamport keys can only be used to securely sign one message, it is important to keep a record of the state of use of each key. To ensure the permanence of the record, the state of the keys is stored directly on the blockchain. In LAMB, each Lamport key is assigned one of three states: `UNUSED`, `POSTED`, or `REDEEMED`, via its PKH. The default state for all possible PKHs is `UNUSED`, which signifies that the signer does not recognize the PKH. If the state is `POSTED`, it indicates that the signer acknowledges the PKH as one of theirs, but has not yet used it. When a PKH is assigned the state is `REDEEMED`, it signifies that the signer has used the corresponding private key to sign a message.

Additional PKHs can be added by signing them with an existing key. When posting additional PKHs, the smart contract verifies that each new key is currently in the `UNUSED` state before moving it to the `POSTED` state. This prevents key reuse under normal circumstances (i.e the user isn't actively trying to reuse keys by shuffling the components of the secret key or by using the same keys on a second contract).

The smart contract that manages the state of the Lamport keys is described in what follows.

2.1. *`LamportBase`*—is an abstract contract. Its state consists of a mapping from `bytes32` to `PublicKeyHashUsageStatus`, as well as an unsigned 256-bit integer used to track the number of keys with a status of `POSTED`. The main attributes of the smart contracts are shown in Table 1. There are two variants of the `LamportBase` contract, namely `LamportBaseA` and `LamportBaseB`. The primary difference lies in the implementation of `redeemLamport`. Where `LamportBaseA ::redeemLamport` accepts a blob of binary, `LamportBaseB::redeemLamport` expects this data to already be hashed.

While key creation and signatures are done off chain, the verification of the signature is done by the network, via the LAMB contract, to guarantee consensus on the proper use of the Lamport signatures.

2.2. *`verify_u256`* —is a function for verifying Lamport signatures. It accepts an unsigned 256-bit integer which stores the hash of a message, a 256-element long array of `bytes` to store the signature, and a 2 by 256 2D array of `bytes32` to store the full public key. This function verifies the signature and, upon success returns true. This function is pure, meaning it neither modifies the state of the contract nor is its behavior influenced by the state of the contract. The implementation of the function is shown in Listing 1 in Appendix 1.

These are the key component that defines the LAMB system. From those specifications, several properties can be derived.

Name	Type	Description
`AddedPublicKeyHashes`	Event	Emitted when user posts more public key hashes
`RemovedPublicKeyHashes`	Event	Emitted when public key hashes are given the `REDEEMED` status
`publicKeyHashes`	Mapping	A map between all possible public key hashes and their statuses
`liveKeyCount`	Uint256	Indicates how many public key hashes have a status of `POSTED`
`isRedeemable`	Function	Used to check if a public key hash has the status of `POSTED`
`redeemLamport`	Modifier	Takes a public key, a signature, and a blob of binary data which has been signed. Verifies the signed data, moves the corresponding public key hash to a state of `REDEEMED`, and resumes execution.
`addPublicKeyHashes`	Function	Allows the owner to post additional Lamport public key hashes
`removePublicKeyHashes`	Function	Allows the owner to remove `POSTED` public key hashes by moving them to the `REDEEMED` state

Table 1. Functions of the abstract smart contract.

3. Properties of LAMB

The main properties of the LAMB system which distinguish it from other EVM smart contracts are described in this section.

3.1. Secure Against Quantum Attacks—Quantum computers hold a distinct advantage when it comes to solving highly structured mathematical problems, particularly those that are central to public key cryptography. Signature schemes such as the Elliptic Curve Digital Signature Algorithm (ECDSA) and Rivest-Shamir-Adleman (RSA) rely on problems that are challenging to solve, but easy to verify. However, because the underlying problems are highly structured in these systems, quantum computers can exploit their unique capabilities to effectively ”unwind” the problem, resulting in a significant advantage. By making these ”hard to solve, easy to verify” problems ”easy to solve, easy to verify”, quantum computing effectively renders them useless for cryptography.

Hashing is all about reducing structure, internally a hash function contains very little of the kind of structure that quantum computers can ”unwind”. They do contain some of this structure, so there is (theoretically) a quantum advantage, but its not nearly the devastating speedup gained over the problems behind schemes like RSA and ECDSA. Because the authentication logic in LAMB is entirely hash based, it inherits the same quantum security as the hash function used.

Most EVM blockchains use ECDSA with the secp256k1 elliptic curve for their public key cryptography. For LAMB, the security of the Lamport signatures rests on the difficulty of executing a pre-image attack on Keccak256. Classically, solving the Elliptic Curve Discrete Logarithm Problem (ECDLP) underlying ECDSA with the Pollard rho algorithm has a classical cost[6] of $2^{127.8}$. A pre-image attack on Keccak256 has a security factor of 2^{256}. Both ECDSA and Lamport signatures are very secure against classical computers. With a quantum computer, the security factor[7–11] of secp256k1 drops to 2^{33} while Keccak256 still has a complexity factor[12] of $2^{166.5}$. The precise runtime on quantum computers depend on several factors like the fault tolerance clock cycle, the number of physical qubits and the noise level of the individual operations. While 2^{33} operations in one day on a quantum computer are realistic from the 2030s, $2^{166.5}$ quantum operations in one day would require exquisite control over matter far more energetic than what can be produced at the LHC. While practical attacks on ECDSA will be made practical and cheap in the coming decades, attacks on hash-based signatures will remain within the realm of science fiction for the foreseeable future.

3.2. Public state management—LAMB inherently provides protection against replay attacks by marking each key after it has been used. The only exception to this is in the event of a hard fork, which is a situation where the blockchain splits into two separate chains. However, LAMB is designed to handle this scenario gracefully, and we provide a detailed explanation of how it does so later in Appendix 2.

The blockchain’s time stamping and immutability features provide an additional, and rather surprising, benefit: it is almost completely safe to reveal your used Lamport private keys if enough blocks have been mined since you signed a message with it. Since the state of the key is managed publicly on the blockchain, it becomes effectively impossible to reuse within the LAMB smart contract and sign more than one message. It is important to note that this is not a recommended practice, and users are encouraged to securely dispose of their `REDEEMED` keys.

Name	Type	Description
`execute`	Function	Used to execute arbitrary logic such as transferring non native tokens
`sendEther`	Function	Used to transfer the native token of the blockchain
`receive`	Function	Allows the contract to receive the native token

Table 2. Functions of a wallet contract inheriting from `LamportBase`. The functions outlined are all that is required to use LAMB as a contract wallet. Their execution, except for `receive`, is made conditional on the successful redemption of a Lamport key to achieve quantum resistance.

3.3. Consistent Identity—In Ethereum and other EVM-based blockchains, smart contracts have a unique and consistent address, which can serve as a form of permanent digital identity as no other contract or user can claim that same address on the same blockchain. Furthermore, decentralized name services such as the Ethereum Name Service (ENS), can be utilized to assign a user-friendly name to a LAMB contract, making it easier to locate and interact with.

3.4. Fees—One drawback of LAMB is the cost of writing to the blockchain. This cost is in place to incentivize block production.[13] On a blockchain data is expensive as it needs to be stored indefinitely and by all participants in the system. Computation is also expensive as it is required to be re-executed by all participants.

While an exact comparison between LAMB and Externally Owned Account (EOA) transactions is difficult, a useful metric can be found by looking at the combined size of their public keys and signatures. A signature + public key from an EOA is 1,024 bits. Whereas a LAMB signature + public key is 196,608 bits, 192 times larger than is the case for EOAs.

Due to the large size of Lamport signatures and their public keys, LAMB transactions are significantly more expensive compared to typical Ethereum transactions.

This cost can be reduced by implementing various techniques that fit the user's specific requirements, however as transaction fees underpin the security of blockchain they cannot be entirely removed from LAMB. The best technique for reducing transaction costs is to reduce on chain activity. As an example in section 5 we discuss how LAMB can be used to sign keys from other schemes.

3.5. Composability—LAMB can be extended in a number of ways to mix in additional features and functionality. Possible extensions are detailed in Section 6.

With the properties enumerated in this section, it is possible to construct a quantum resistant smart contract wallet from a LAMB contract.

4. Securing Digital Assets

Smart contracts on blockchains can securely hold and transfer funds. When a contract is used primarily for this task, we refer to it as a contract wallet.[14] Contract wallets are often used in cases where the conditions for transferring funds are more complex than what can be achieved with a single ECDSA signature. Post-quantum security for digital assets can be achieved by incorporating the ability to receive and transfer funds in a LAMB contract. The required attributes are shown in Table 2. Quantum resistance is obtained by requiring the successful verification of a Lamport signature for the execution of functions. In addition to reserves of the native token, a wallet derived from LAMB can also secure digital assets from existing token contracts. The

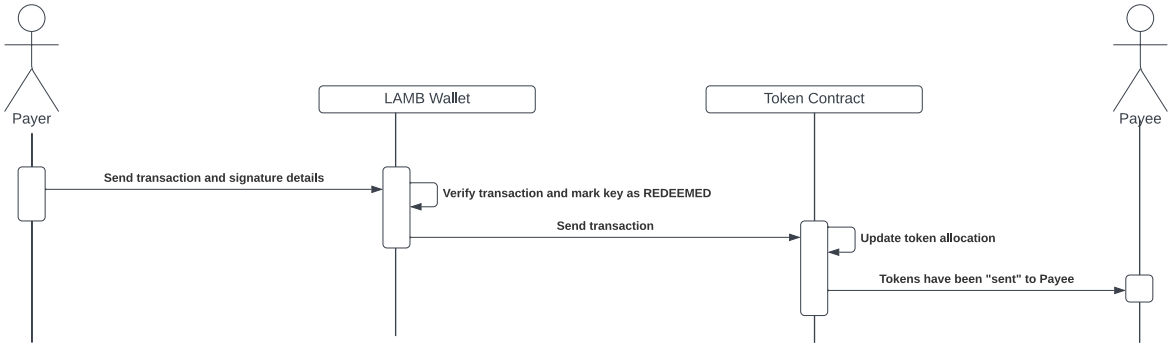

Fig. 1. LAMB can be used by contract wallets to secure assets for the long term from quantum computers.

Name	Type	Description
`signedMessages`	Mapping	A map from all possible 256-bit values to a boolean indicating whether or not the message whose hash is the 256-bit value should be considered to be endorsed
`signMessage`	Function	Accepts a message hash and signature details. If the signature details are valid the message hash is marked as read in the `signedMessages` mapping
`verifyMessage`	Function	Allows anyone to check the hash of a message to see if it has been endorsed or not

Table 3. Attributes of a contract for endorsing messages with LAMB.

process of executing a quantum resistant transaction of a token is shown in Fig. 1. The transaction message to be signed must include the following details:

(1) The address of the token contract or the null address if the token being sent is the native token.
(2) The function signature of the token contract and the arguments to be passed, encoded as binary.
(3) The amount of native tokens to send.
(4) The amount of gas to provide for the transaction.

These details are encoded as they should appear in `msg.data` as received by the token contract. It is a hash of this data which should be signed.

5. Proving identity and enabling crypto-agility

This can be useful for situations where you want to allow an individual to verify some messages but you don't need the general public to be able to verify your message. For example, if Alice and Bob want to speak to each other and be sure the other is who they claim to be, they can each commit to a Merkle root on chain, at the beginning of their conversation. They can then go back and forth sharing signed messages while incurring no additional transaction fees. A simplified exchange, where a signer commits directly to a message, is illustrated in Fig. 2 and the attributes that enable arbitrary messages to be committed to and verified on chain are listed in Table 3.

This also means that LAMB can be used to endorse and revoke keys from other schemes,

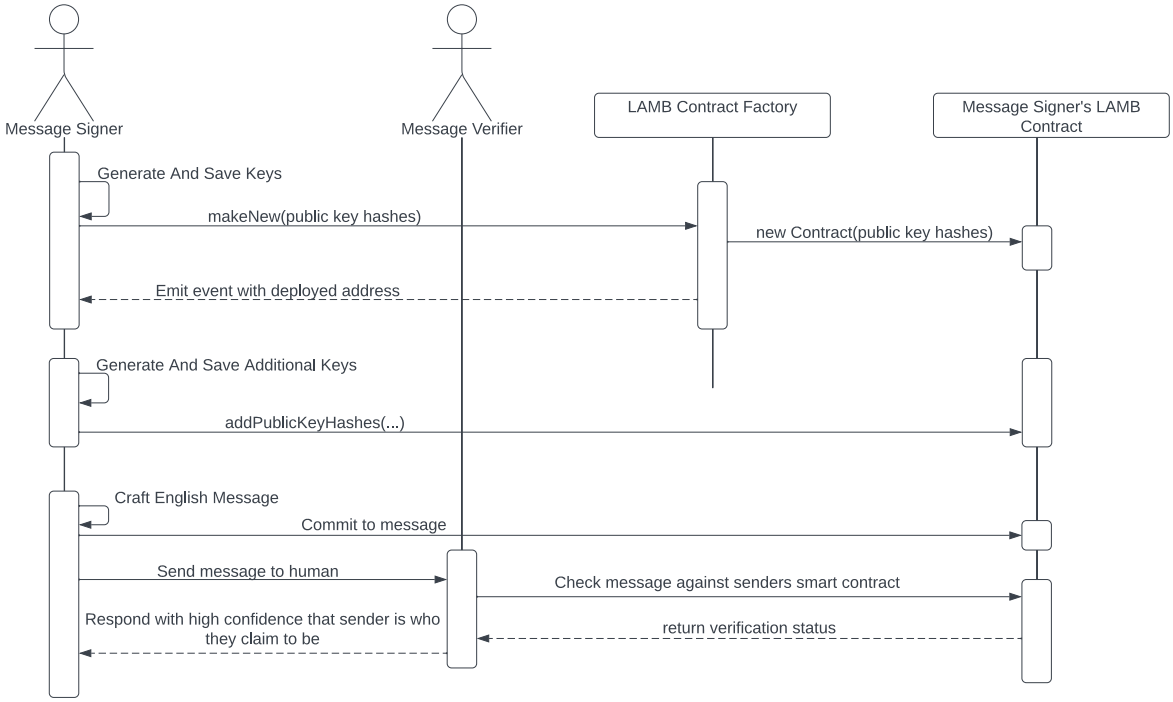

Fig. 2. LAMB can be used to authenticate signed messages.

such as NIST post-quantum methods or ECDSA, by committing to them with Lamport signatures. This affords the use of emerging cryptographic technology while providing an easy means of rescinding the use of these technologies publicly. This crypto-agility makes LAMB particularly compelling as there is much uncertainty about which emerging cryptographic schemes will be adopted and will stand the test of time. Hence, LAMB smart contracts effectively act as a digital bedrock for proving identity and enabling crypto-agility using blockchain technology in an age where quantum computers will become increasingly powerful.

6. Extending LAMB

LAMB can be extended in a number of ways. These extensions enable additional optional features which are not considered to be part of the core LAMB implementation.

6.1. Revocable Signatures—LAMB can be extended to enable revocable signatures by allowing an additional state for keys: `REVOKED`, which can only be reached from a `POSTED` state. Being in a `REVOKED` state would indicate that the key was used to endorse a message, but that the signer no longer wishes to endorse that message.

6.2. Multisig contracts—- A popular use of contract wallets is to require signatures from multiple signatories in order to send assets. LAMB can be adapted to support this threshold governance.

6.3. Temporary ECDSA endorsements—Due to the higher transaction cost of using Lamport Signatures a LAMB Wallet user may desire the ability to endorse an ECDSA key. This ECDSA key would have reduced privileges and could be revoked using the Lamport keys at any time. Additionally an expiration block number could be specified after which the ECDSA key would no longer be considered valid.

6.4. LAMB Native Names—A factory contract can be used to deploy multiple LAMB contracts, but it can also be used to ensure each of the deployed contracts has a unique user selected name. A mapping can be provided to relate each deployed wallet to its unique name.

6.5. Condensed Private Key Generation And Storage—Instead of directly generating 512 256-bit secrets for our private key, we can instead generate a single 256-bit 'seed'. With this seed we can grow the rest of our secret key. We do so by creating 512 different hashing functions, and separately applying each hash function to the seed. The result is 512 values with no useful relationship between them. This is important to note because later, when we reveal half the private key, an adversary must not be able to use one of these elements to find another.

The 512 different hashing functions can be defined in terms of our original hashing function and an index between 0 and 511. By concatenating the index to the input before feeding the input to the original hashing function, we have defined a new, unique, one way hashing function.

6.6. Successive Key Generation—The same trick used to create 512 distinct hashing functions can be extended to allow unlimited secret keys to be grown from a single 256-bit seed. The extension works by keeping a count of the number of keys generated. This nonce is then combined with the original seed and hashed, the result is a new seed that can be used to generate an entire Lamport secret key.

6.7. LAMB as an Updateable Diamond—EIP-2535 outlines a standard for upgradeable modular smart contracts.[15] The message authenticating functionality of LAMB can be used to ensure all upgrades to the diamond are approved by the Lamport signer. This may be useful for creating specific functions to make various activities more explicit. For example, the ability to endorse an ECDSA signer may be easier to use if a specific event is emitted upon the endorsement. Similar extensions may be desirable as new post quantum schemes come in and out of fashion.

6.8. Low `POSTED` key count warning—Should a LAMB contract have no keys in a `POSTED` state it would be unusable. The owner would have no way to commit to more key hashes. The implementations referenced in this paper assume some off chain mechanism to encourage users to maintain a healthy number of key hashes in a `POSTED` state. Such an off chain mechanism would look at the public variable `liveKeyCount` of the user's LAMB contract to know how many keys are left. When seen to be below some threshold, this mechanism would encourage the user to commit to new keys. Such an off chain mechanism is desirable because it does not increase on chain computation.

7. Conclusion

This paper introduced LAMB, a flexible scheme for authenticating messages well suited for use in a post quantum world. The scheme is constructed using smart contracts and Lamport signatures, and doubles as a post quantum blockchain wallet. LAMB works well as a bedrock of trust; it can be used to endorse and revoke keys from other cryptographic schemes. While the consensus layer of EVM blockchains still has to be upgraded to post-quantum public keys such as Crystals-dilithium[16] or Falcon,[17] LAMB can be used preemptively to secure the ownership of individual assets and accounts from quantum computers.

Author Contributions

WDD developed the contracts described in this paper and PLDD provided direction for the project. Both contributed equally to manuscript preparation.

Notes and References

[1] Shor, P. W. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." *SIAM review* **41.2** 303–332 (1999) http://dx.doi.org/10.1137/S0036144598347011.

[2] Proos, J., Zalka, C. "Shor's discrete logarithm quantum algorithm for elliptic curves." *arXiv preprint quant-ph/0301141* https://doi.org/10.48550/arXiv.quant-ph/0301141.

[3] Aggarwal, D., Brennen, G. K., Lee, T., Santha, M., Tomamichel, M. "Quantum attacks on Bitcoin, and how to protect against them." *Ledger* **3** https://doi.org/10.5195/ledger.2018.127 1710.10377.

[4] Alagic, G., *et al.* "Status report on the third round of the NIST post-quantum cryptography standardization process." *US Department of Commerce, NIST* https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=934458.

[5] Lamport, L. *Constructing Digital Signatures from a One Way Function. Technical Report CSL-98* SRI International (1979) this paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. URL https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[6] Bernstein, D. J., Lange, T. "SafeCurves." (2013) URL https://safecurves.cr.yp.to/rho.html.

[7] Roetteler, M., Naehrig, M., Svore, K. M., Lauter, K. "Quantum resource estimates for computing elliptic curve discrete logarithms." In *International Conference on the Theory and Application of Cryptology and Information Security* Springer 241–270 (2017) https://doi.org/10.1007/978-3-319-70697-9_9 1706.06752.

[8] Gheorghiu, V., Mosca, M. "Benchmarking the quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes." *arXiv preprint arXiv:1902.02332* https://doi.org/10.48550/arXiv.1902.02332.

[9] Häner, T., Jaques, S., Naehrig, M., Roetteler, M., Soeken, M. "Improved quantum circuits for elliptic curve discrete logarithms." In *International Conference on Post-Quantum Cryptography* Springer 425–444 (2020) https://doi.org/10.1007/978-3-030-44223-1_23 2001.09580.

[10] Gidney, C., Ekerå, M. "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits." *Quantum* **5** 433 (2021) https://doi.org/10.22331/q-2021-04-15-433 1905.09749v3.

[11] Webber, M., Elfving, V., Weidt, S., Hensinger, W. K. "The impact of hardware specifications on reaching quantum advantage in the fault tolerant regime." *AVS Quantum Science* **4.1** 013801 (2022) https://doi.org/10.1116/5.0073075.

[12] Amy, M., Di Matteo, O., Gheorghiu, V., Mosca, M., Parent, A., Schanck, J. "Estimating the Cost of Generic Quantum Pre-image Attacks on SHA-2 and SHA-3." In R. Avanzi, H. Heys (Eds.), *Selected Areas in Cryptography – SAC 2016* Cham: Springer International Publishing 317–337 (2017) https://doi.org/10.1007/978-3-319-69453-5_18.

[13] Nakamoto, S. "Bitcoin: A Peer-to-Peer Electronic Cash System." (2008) (accessed XX October 20XX) https://bitcoin.org/bitcoin.pdf.

[14] Di Angelo, M., Slazer, G. "Wallet contracts on Ethereum." In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* IEEE 1–2 (2020) https://doi.org/10.1109/ICBC48266.2020.9169467.

[15] Mudge, N. "EIP-2535: Diamonds, Multi-Facet Proxy." *Ethereum Improvement Proposals, no. 2535,* https://eips.ethereum.org/EIPS/eip-2535.

[16] Ducas, L., *et al.* "Crystals-dilithium: A lattice-based digital signature scheme." *IACR Transactions on Cryptographic Hardware and Embedded Systems* 238–268 https://doi.org/10.13154/tches.v2018.i1.238-268.

[17] Fouque, P.-A., *et al.* "Falcon: Fast-Fourier lattice-based compact signatures over NTRU." *Submission to the NIST's post-quantum cryptography standardization process* **36.5** URL https://falcon-sign.info/falcon.pdf.

Appendix 1: Code and Descriptions

```
function verify_u256(
  uint256 bits,
  bytes[256] calldata sig,
  bytes32[2][256] calldata pub
) pure returns (bool) {
  unchecked {
    for (uint256 i; i < 256; i++) {
      if (pub[i][((bits & (1 << (255 - i))) > 0) ? 1 : 0] != keccak256(
         sig[i])) {
        return false;
      }
    }

    return true;
  }
}
```

Listing 1. Verifiy Lamport Signed Message In Solidity

```
enum PublicKeyHashUsageStatus {
  UNUSED,
  POSTED,
  REDEEMED
}
```

Listing 2. Valid States For A Public Key Hash

```
mapping(bytes32 => PublicKeyHashUsageStatus) public publicKeyHashes;
uint256 public liveKeyCount = 0;
```

Listing 3. State Variables Relevent To Tracking Key Useage

```
modifier redeemLamport(
    bytes32[2][256] calldata publicKey,
    bytes[256] calldata signature,
    bytes memory prepackedMessageData
  ) {
    require(liveKeyCount > 0, "LamportBaseA: not initialized or no
       remaining public key hashes");
    bytes32 publicKeyHash = keccak256(abi.encodePacked(publicKey));

    require(isRedeemable(publicKeyHash), "LamportBaseA: currentpub does
        not have a redeemable public key hash");
    require(verify_u256(uint256(keccak256(prepackedMessageData)),
       signature, publicKey), "LamportBaseA: Signature not valid");

    _removeSinglePublicKeyHash(publicKeyHash);
    emit RemovedPublicKeyHashes(liveKeyCount);

    _;
  }
```

Listing 4. Solidity Modifier To Verify And Redeem Lamport Signatures and Keys

Appendix 2: Handling A Hard-fork

If a hard fork occurs and both chains are supported by different communities, LAMB can still remain secure. A LAMB user has two options when dealing with a hardfork: the first option is to destroy the contract on one of the chains. Doing so would require one of the public keys be used on the chain where the contract is destroyed. Meanwhile, on the other chain that public key hash will remain in a `POSTED` state. Left unresolved this could lead to accidentally signing two messages with one secret. On the chain not destroyed, this public key hash must be moved to a `REDEEMED` state. This is the recommended way of handling a hard fork as it minimizes ambiguity.

The alternative option is to keep both versions of the LAMB contract. In this scenario, a user would move half of their keys to a `REDEEMED` state on the first chain, and the other half would be moved to a `REDEEMED` state on the other chain. The key used to expire the keys on one chain should be a member of the set of keys removed on the other chain. This option is not recommended under usual circumstances because it means that off chain verifiers have to check both blockchains in order to authenticate a message. Doing this also increases the likely hood of user error on the part of the signer as it complicates key management.