UNBOUND

(MATH OVER MATTER)

Unbound blockchain-crypto-mpc Library

White Paper

Prof. Yehuda Lindell, Chief Scientist
Guy Peer, VP R&D
Dr. Samuel Ranellucci, Cryptographer

December 21, 2018

Table of Contents

1. Problem Statement

In the realm of cryptocurrencies, storing private keys for cryptographic primitives in a secure manner and preventing fraudulent transactions is critical. This is due to the fact that cryptocurrency transactions are irreversible, by definition, and thus stolen funds cannot be recovered. This makes them an extremely attractive target to cybercriminals. In this white paper, we describe Unbound's open-source library for protecting private keys and authorizing transactions via secure two-party computation (MPC). The library can be found at:

https://github.com/unbound-tech/blockchain-crypto-mpc/

2. Overview of the Solution

Unbound's solution is based on no single device holding the private key used to generate signatures and transfer funds. Rather, the private key is shared among two devices (servers/cell phones/laptops) so that no party has any information about the key. Then, in order to generate a signature, the two devices run a secure two-party computation protocol that generates the signature without revealing anything about the parties' key shares to each other. We stress that these devices may or may not be associated with the same person or organization, and these can be any entity. Thus, one could use this to create a wallet, sharing the private key between one's mobile and one's laptop, between one's mobile and a VM in the cloud, and so on. From here on, we use the term *parties* to denote the two devices/entities defined to run the protocols.

The basis of our solution is to use MPC two-party signing protocols in order to carry out all required operations. All our protocols are secure in the presence of malicious adversaries, which means that security is maintained even if one of the devices is breached and running malicious code. This claim is backed by mathematical proofs of security of the protocols.

Our solution supports BIP32/BIP44 key derivation. We assume familiarity with these standards here, and refer the reader to:

- https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
- https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

When using BIP key derivation, there is a BIP master seed or key that is used to derive all future keys. The BIP key derivation is carried out using MPC between the parties and integrated into the MPC signing method, ensuring that the entities only receive their respective shares of the key, and whole key material is never revealed. Our method for BIP key derivation via MPC validates that the derivation is correct, even if some of the parties are corrupted. This is crucial for enabling key recovery of all keys from the BIP master key. We stress that doing BIP derivation in the clear and then running MPC for the signing is not secure, since key material is in the clear.

Backup is a crucial element of any cryptocurrency solution, since the loss of the private key means that funds can never be retrieved. For the purpose of backup, an RSA key pair is generated, and the private RSA key is stored in cold backup. Then, the ECDSA/EdDSA private key is encrypted under the RSA cold backup public key. This encryption is carried out separately by each party on their share (since no party has the entire private key). In addition, each party generates a publicly verifiable zero-

knowledge proof that the correct share of the private key was encrypted. This is needed to prevent a malicious party from rendering the cold backup useless. This method works also with keys derived using BIP or any other key derivation method. However, since the backup is of an actual key, when using BIP derivation, it is carried on the root private key and not the master seed. This suffices for recovering all derived BIP keys, as required. (We do not provide a method for backing up the BIP master seed itself directly, since a more efficient zero-knowledge proof exists for the root node private key.)

Our basic solution consists of MPC protocols running between two parties. However, the same protocols can be used to achieve MPC for the setting of 2-out-of-n parties (where there are n entities, and any pair of them suffice to sign and carry out all other operations). This extension is carried out by having two parties generate the key initially, and reshare their shares to all pairs among the n. We show how to achieve this in our code samples.

Our solution is generic, supporting ECDSA over a secp256k1 curve and supporting EdDSA over an ed25519 curve (or Schnorr). For ECDSA, we implement the protocol that was published in [5]. For EdDSA, we use the threshold signature protocol for Schnorr with additive sharing of the secret key that appears in [10].

Our open-source library contains elements that do not appear in [5]. In particular, the zero-knowledge proofs used in key generation are more efficient, it integrates MPC for BIP key derivation with ECDSA key generation, and it contains key-sharing refresh, which achieves proactive security. We describe these differences in this document.

3. Supported Functionalities

The open source library supports the following operations:

1. *Generate generic secret*: Used to generate an initial secret that is used for BIP derivation.
2. *BIP derivation*: Derive keys in MPC, using the initial secret, according to the BIP32 standard. The result of this step is a key that can be used in ECDSA. Thus, both parties receive the derived public key and hold random shares of the associated private key. Since MPC is used, neither party knows the full private key.
3. *ECDSA/EdDSA key generation*: Uses MPC for two parties to directly generate an ECDSA or EdDSA key. The result is a public key known to both parties, and a random sharing of the associate private key. Since MPC is used, neither party knows the full private key.
4. *Backup*: The backup method receives an RSA backup public key and generates a publicly-verifiable backup of the parties' shares that is guaranteed to be correct. The procedure uses a zero-knowledge proof, guaranteeing that nothing is revealed about the private key shares, even though it is possible to validate correctness. The backup verification function verifies the zero-knowledge proof and can be used by any entity to ensure that the backup is valid (this function receives the public ECDSA/EdDSA key and so validates that the backup is of the private key of the given public key). Finally, the backup restore method takes the backup information and the RSA backup private key and outputs the ECDSA/EdDSA private key.
5. *ECDSA/EdDSA signing*: This method uses a previously generated key to sign in MPC on a message m that is approved by both. Since MPC is used, neither party can generate such a signature alone, or trick the other into signing on a different message than the one it approves.
6. *Key sharing refresh*: This is used by the parties to jointly generate new random shares of an existing shared key so that the old shares become useless. If an attacker steals a share from one device before a refresh and from the other device after a refresh, it learns nothing about the private key. Thus, it must breach and be resident on both parties before any refresh takes place. This security property is called "proactive" in the academic MPC literature.

4. Cryptographic Tools

4.1. ECDSA Signing

The ECDSA signing algorithm is defined as follows. Let G be an elliptic curve group of order q with base point (generator) G. The private key is a random value $x \in Z_q$ and the public key is $Q = x \cdot G$. Signing a message m is as follows

1. We denote $m' = H_q(m)$ as the first $|q|$ bits of $H(m)$ where
   a. $|q|$ is the bitsize of q
   b. H is the hash function SHA-256
2. Choose a random $k \in Z_q^*$
3. Compute $R \leftarrow k \cdot G$ and denote $R = (r_x, r_y)$
4. Compute $r \leftarrow r_x \bmod q,\ s \leftarrow k^{-1} \cdot (m' + r \cdot x) \bmod q$
5. Output (r, s)

4.2. EdDSA Signing

The EdDSA signing algorithm is a version of Schnorr's signature over an Edwards curve. For simplicity, we will describe Schnorr's signatures here. Let G be an Elliptic curve group of order q with base point (generator) G. The private key is a random value $x \in Z_q$ and the public key is $Q = x \cdot G$. Signing a message m is as follows

1. Choose a random $r \in Z_q^*$ (in EdDSA, this is derived from the private key and message using a pseudorandom function)
2. Compute $e \leftarrow H(R, Q, m)$.
3. Compute $s \leftarrow r + x \cdot e \bmod q$.
4. Output (R, s).

4.3. Paillier Encryption

The Paillier encryption scheme is a public-key cryptosystem like RSA that relies on the difficulty of factoring large numbers.

1. The public and private keys for Paillier are generated by choosing two random primes p and q. For a private key (p, q), the public key is $N = pq$.
2. To encrypt a message m, sample a random value r and compute $Enc_{pk}(m; r) := (1 + n)^x \cdot r^n \bmod N^2$.

An important property of Paillier encryption is that it is additively homomorphic. Given ciphertexts c_1, c_2 that are encryptions of plaintexts m_1, m_2, respectively, we

have that $(c_1)^v \ mod \ N^2$ is an encryption of $m \cdot v \ mod \ N$, and $c_1 \cdot c_2 \ mod \ N^2$ is an encryption of the value $m_1 + m_2 \ mod \ N$. Thus, encrypted values can be summed, and multiplied by known scalars, without decrypting.

4.4. Commitment Schemes

A commitment scheme is a cryptographic protocol run between a sender and a receiver that can best be described using a safe. In the commitment phase, the prover puts a message in the safe, locks the safe using a secure combination and gives the safe to the receiver. After this phase, the message is hidden from the receiver (called hiding property), and yet the sender can no longer change what is inside the safe (called the binding property). In the opening or reveal phase, the sender reveals the combination of the safe to the receiver, who can then open it and learn the message. A commitment scheme can be implemented using cryptographic hash functions. In this case, the sender can commit to a message m by choosing a string r (of length say 128 bits) and sending $H(m||r)$ to the receiver. The commitment is then opened by the sender just sending (m, r) and the receiver verifies that the hash of the two values is indeed what was previously sent. This is binding by the collision-resistant property of the hash function, and is hiding under assumptions about the properties of the hash function. In particular, in the random oracle model, where the hash function is modeled as a random function, the value $H(m||r)$ reveals nothing about m.

4.5. Zero-Knowledge Proofs

A zero-knowledge proof is a protocol between a prover and a verifier, in which the prover proves to the verifier that a statement is true without revealing any information. A zero-knowledge proof must fulfill three properties: (soundness) if the statement is false then the prover cannot convince the verifier that the statement is true, (completeness) if the statement is true then the prover can convince the verifier that the statement is true, and (zero-knowledge) the prover should not be able to learn any information from the proof beyond the fact that the statement is indeed true. A zero-knowledge proof of knowledge is a zero-knowledge proof that allows the prover to prove a statement only if it knows an actual NP witness for the given statement. In our protocols, we employ zero-knowledge proofs for the following statements.

1. Proof that a committed value and an encrypted value are the same.
2. Proof that a committed value is in some range of values.

3. Proof of knowledge that a player knows the discrete log of an elliptic-curve point (ZK-DL).
4. Proof that a Paillier public-key was generated correctly (ZK-PPK).
5. Proof that the decryption of a given Paillier ciphertext is the discrete log of a given elliptic curve point (ZK-PDL).
6. Proof that an RSA ciphertext c_i encrypts a value x_i such that $Q_i = x_i \cdot G$, where Q_i is known. This is used for cold backup.

When the zero-knowledge proof is *non-interactive*, it has the property of being publicly verifiable (anyone can read the proof and validate that the statement is indeed correct). This can be used to verify that the backup is valid before transferring funds to an address.

4.6. Oblivious Transfer

Oblivious transfer (typically referred to as OT) is a protocol between two parties, a sender and a receiver. The sender has a pair of inputs x_0, x_1 and the receiver has a choice bit c. The result of the protocol is that the receiver learns x_c (but nothing about the other value x_{1-c}), and the sender learns nothing about c. OT requires the use of public-key primitives, and OT extensions can be used to run a fixed number of OTs based on public-key primitives and then obtain many more actual OTs using hashing only. We use the OT protocol of [7] together with the OT extension of [8]. Overall, this requires 3 rounds of interaction to setup, and 2 rounds for actually carrying out OT executions (although the first round of the actual OT can be sent together with the third round of the setup, and so this requires 4 rounds overall).

4.7. Garbled Circuits

A garbled circuit is a cryptographic primitive that is used in secure two-party computation protocols. At a high level, a garbled circuit is an encrypted version of a function. When the evaluator is given access to the garbled circuit, with an encryption of an input, it learns the output of the function on the given input without learning anything else. Garbled circuits are used for secure two-party computation by having one of the parties create a garbled circuit that is later evaluated by the other party. This result has been known since the late 80's, but many optimizations and improvements from the last few years have made this very efficient in practice.

4.8. Dual Execution

If two players are semi-honest (meaning that it is guaranteed that they run the specified protocol), then they can run secure computation by only using a single garbled circuit. However, if the garbler is malicious, then it can generate an incorrect garbled circuit that can leak information via the output (e.g., it can generate a circuit that gives the key as the output rather than the ciphertext). Achieving security when the parties are malicious is a significant challenge and comes at a cost. One of the methods used to prevent the parties from cheating is called "dual execution". This works by having each party run the computation once as the garbler of the circuit and once as the evaluator of the circuit. After both evaluations, the parties compare the results (without revealing it yet). If both circuits generate the same output, then each one knows that it is correct. This is because each party generates one of the circuits and so, if it is honest, it knows that the circuit and thus the computation is correct. In dual execution, the only information leaked to the attacker is due to there being an abort. That is, if the comparison of the results is "not equal" then this can leak a single bit. For example, a corrupted party could generate a garbled circuit that computes garbage if the first bit of the key is 0 and otherwise computes the correct output. Since the circuit is garbled, this would not be detected. Then, the mere fact of whether or not there was an abort leaks this bit. However, in the usage here for key derivation, one can show that since the attack is detected when the results are not equal the leakage due to such an attack is small (revealing on average 2 bits of the key). As such, if an attack is detected, it is crucial to remove the attacker and then prudent to transfer the funds to a new address. The dual execution methodology was introduced in [6]; the specific version that we use is similar to that of Section 4 of [4] (in particular, the equality test is run on the encoded outputs and not the actual output).

We constructed a new equality test with 3 rounds of communication that is based on the equality test in [9] (using ElGamal in-the-exponent additively homomorphic encryption), but is enhanced to enable a proof of security under the ideal/real model simulation paradigm of secure computation, and to provide (verified) output to both parties.[1]

[1] In order to prove our protocol secure, we defined an ideal functionality for equality that receives either the actual value or the value times the generator point of the group, and compares equality based on this. It is clear that this is equivalent for our purposes. In addition, the adversary can cause the honest party to think that the result was not-equal even if it was equal; this is fine since it only causes the honest party to abort. We stress that the reverse is not possible; if the result was not-equal then the adversary cannot cause the honest party to think that the result was equal.

Using the two-rounds of OT required based on the extension, the entire dual execution has five rounds of communication, including the equality test.

As we have described, our dual execution protocol has the property that if a party cheats, then it can learn a single bit, but at the cost of being caught with probability ½ (informally speaking). Thus, if an attack is detected, use of the key must be stopped. Now, one possible strategy of an adversary is to try to cheat, but to not provide the output of the equality test to the honest party (when it receives this output first), and claim that there was a "crash" or network failure. If this is not dealt with, then the adversary can learn all bits of the secret key. Thus, either executions must always conclude (by storing on disk the messages needed to complete, and continuing after a crash or failure), or some cheating attempt must be assumed. We also note that many executions on the same input key cannot be run in parallel since an attacker can learn a bit from each parallel execution. If this is desired, then it is possible to change the code so that a single equality test is run for all executions, and this will be secure.

5. Cryptographic Protocols Overview

Having described all the cryptographic primitives and tools used in our solution, we now describe the general flow of the protocols. Full details can be found in [5], and we refer the reader to that paper for more information.

5.1. Distributed Key Generation

Parties generate shares of an ECDSA key that can be used for signing. We denote the parties in the protocol by Alice and Bob. We denote the elliptic curve generator by G, and its order by q.

1. The parties generate shares x_1, x_2 of the private key (i.e., the ECDSA private key is $x = x_1 + x_2 \; mod \; q$). In addition, the parties obtain Q, Q_1, Q_2 where $Q_1 = x_1 \cdot G$, $Q_2 = x_2 \cdot G$ and $Q = Q_1 + Q_2$. The value Q is the ECDSA public key.
2. Alice generates a Paillier key and proves to Bob that it was correctly constructed in zero-knowledge. The proof required here is that N is relatively prime to $\phi(N)$; this is sufficient since it guarantees that the homomorphic properties of Paillier are correct. We use the zero-knowledge proof from [3]. Note that the proof in that paper proves that N is square free and that $gcd\big(e, \phi(N)\big) = 1$. Since we are only interested in $gcd\big(N, \phi(N)\big) = 1$, we run their proof with $e = N$ (and we only run the second part with m_2).
3. Alice encrypts x_1 under Paillier and proves to Bob in zero-knowledge that this encrypted value is in the correct range and is the same value that was used for generating Q_1. Denoting the ciphertext by c_{key}, Alice sends c_{key} to Bob.
4. Alice outputs the private key of Paillier as well as (x_1, Q_2, Q).
5. Bob outputs public key of Paillier as well as (x_2, Q_1, Q, c_{key}).

The zero-knowledge proof that we use in step 3 is different to that appearing in [5]. First, we use a non-interactive version of the range proof appearing in [5]; this requires more repetitions of the proof (namely, 128) and thus is more computationally expensive. However, we wished to minimize the number of rounds as much as possible. Second, the actual PDL proof of [5] that we use is essentially the same as the one in the original version of the paper appearing at CRYPTO 2017, and not the one appearing in [5] which is computationally cheaper. Again, we do this since this proof can be non-interactive, and enables us to reduce the number of rounds. Another difference is that instead of requiring Alice to generate x_1 to be less than $q/3$, we simply adjust the values appropriately in the beginning of the proof (by Alice sending how many multiples of $q/3$ should be subtracted; this reveals at most 2 bits of information and so is fine).

We remark that it is possible to use the far more efficient CFT range proof described in Section 1.2.3 of [1]. However, this requires using Pedersen commitments over a group of unknown order, and generation of such parameters in a validated way is very expensive (among other things, it requires generating safe primes which is expensive); see Section 1.2 of [2].

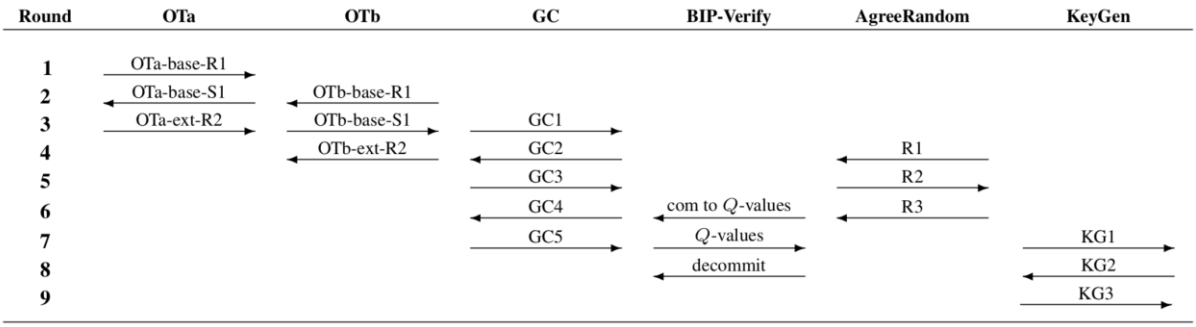
5.2. BIP32/BIP44 Key Derivation

BIP32/BIP44 key derivation is carried out by computing HMAC-SHA512 with the private key in the appropriate node in the BIP hierarchy and a string describing the path to the new node in the tree. We carry out this computation using dual execution (with garbled circuits), with the result being that Alice receives x_1 and Bob receives x_2. The parties then use these results in the key generation described above (i.e., instead of choosing random x_1, x_2, they use the results of the derivation). The BIP derivation run in MPC in our implementation is fully compliant with the standard BIP32/BIP44 standards.

The above description is overly naïve, since it does not prevent parties from cheating and using different values than prescribed. This is important since if we backup only the BIP master key, then if parties use different inputs to the key derivation, keys will be generated that cannot be recovered from the BIP master key. Thus, the MPC protocol includes steps to detect such modifications. This includes additional randomized information in the dual execution, followed by a "BIP verification phase" to detect any cheating. This randomized information is such that many shares of a key are output; some are shares of zero (i.e., both have the same value), some are shares of the previous private key, and some are shares of the new private key. The parties then each compute shares of the public keys, by locally multiplying the generator by each share, and then securely exchange the values. Observe that if they have shares of 0 then they will exchange the same elliptic curve point, if they have shares of the previous private key then they will exchange shares of the previous public key (known to them), and if they have shares of the new private key then they will exchange shares of the new public key. This prevents cheating since if a party input an incorrect value for key derivation, then it must change the shares of the previous public key in order to match the real previous public key. However, it will then be caught since it will be detected if it changes a value based on a share of 0. Thus, if a party changes its input shares, it will be detected with very high probability. This verification step has 3 messages, and is run after the dual execution.

When working with BIP derivation, ECDSA key generation requires first running BIP derivation via dual execution (5 rounds), then verifying the BIP results (3 rounds), and

then running ECDSA key generation on the received shares (3 rounds). In order to reduce the amount of communication, some of these messages can be piggybacked and we have 9 rounds overall. This overall flow is depicted below (note that the AgreeRandom is a secure coin tossing protocol which is used to ensure a fresh session identifier in the actual ECDSA key generation phase), and is helpful to follow the implementation of this more complex protocol combination.



5.3. Backup

Each party holds a share of the private ECDSA key (regular key or BIP derived master key). They can backup this key by simply encrypting each share locally with an externally received RSA public key. We stress that this is a *very sensitive* operation since if a malicious actor can convince both parties to back up the shares using its own public key, it can obtain the private key. Thus, any deployment must ensure that this cannot be done (e.g., allowing backup to be called only from a local call). As with BIP derivation, we must also ensure that each party really encrypts its share of the private key; otherwise the backup can be useless. However, note that if Alice's share of the private key is x_1 then Bob holds $Q_1 = x_1 \cdot G$ and vice versa. Thus, Alice can prove in zero-knowledge that the RSA-encrypted value it provides is indeed that which defines Q_1, and Bob can prove in zero-knowledge that the RSA-encrypted value it provides is indeed that which defines Q_2. These proofs are non-interactive and so can be verified externally by anyone. In addition, by verifying that $Q = Q_1 + Q_2$ is the public key, anyone can verify that the backup is a valid encryption of shares of the private key associated with Q.

5.4. ECDSA Signing

Given a message m that both parties agree to sign, the parties can generate a signature on that message using the protocol of [5] as follows:

1. Alice and Bob generate a random sharing k_1 and k_2 of k, and both learn the value $R = k_1 \cdot k_2 \cdot G$. This generation uses commitments and zero-knowledge proofs in order to ensure that R is (essentially) uniformly distributed in the group.

2. Denote $R = (r_x, r_y)$; each party locally computes $r \leftarrow r_x \, mod \, q$.
3. Bob uses the homomorphic properties of Paillier to compute an encryption of a partial signature on m, using its share of k. Specifically, it generates an encryption of the value $k_2^{-1} \cdot (m' + r \cdot x)$; it can do this since it has an encryption of x_1 and knows all other values. Observe that this is "almost" a signature, in that all that is needed is to multiply it by k_1^{-1}. We remark that Bob also adds a random multiple of q; this is needed in order to hide any difference from the fact that inside Paillier encryption the operations are not computed modulo q. Bob sends the result to Alice.
4. Alice computes the signature on m by decrypting the ciphertext, multiplying the plaintext by k_1^{-1}, and reducing the result modulo q. Alice then checks that the signature is valid. If yes, then she outputs the signature. Otherwise, she aborts.

5.5. EdDSA Signing

Given a message m that both parties agree to sign, the parties can generate a signature on that message using the protocol of [10] as follows:

1. Alice and Bob run two oblivious pseudorandom function evaluations, in order for them to derive pseudo-random shares r_1 and r_2 of r from the message. Both parties also learn the value $R = r_1 \cdot G + r_2 \cdot G$. This generation uses commitments in order to ensure that R is (essentially) uniformly distributed in the group, in the case that one of the parties is corrupted.
2. Each party locally computes $e = H(R, Q, m)$.
3. Bob computes $s_2 = r_2 + x_2 \cdot e \bmod q$ and sends s_2 to Alice.
4. Alice computes $s_1 = r_1 + x_1 \cdot e \bmod q$ and $s = s_1 + s_2 \bmod q$, and outputs (R, s). Observe that $s = s_1 + s_2 = (r_1 + r_2) + (x_1 + x_2) \cdot e = r + x \cdot e \bmod q$, as required.

We remark that the pseudo-random function derivation of r from the message and private key is different to the actual EdDSA standard. Cryptographically, any such method is equivalent, and EdDSA specifies one concrete method. However, the EdDSA method is not "MPC friendly" and would require an expensive circuit evaluation. We could achieve this via dual execution. However, we think that it is not necessary. We stress that by the security of the pseudo-random function used here and in the EdDSA specification, it is not possible to distinguish the use of the method here and the one of the actual EdDSA specification. Thus, this difference can only be detected when running known-input tests.

5.6. Share Refresh

The parties run a secure coin tossing protocol in order to generate a random value r that neither can bias. Then, Alice modifies her share of the private key to be $x_1' = x_1 + r \; mod \; q$, and Bob modifies his share of the private key to be $x_2' = x_2 - r \; mod \; q$. Alice then generates a new Paillier key to encrypt x_1' and proves in zero-knowledge that this is the correct value. This proof involves proving that the same value is encrypted under two different Paillier keys, and is very efficient. (This suffices since Bob can generate an encryption of x_1' by adding the scalar r to the existing encryption of x_1 with the old Paillier key, and then verifying the proof.)

As a result, after the refresh the parties hold a fresh sharing of x, with a new Paillier key. This achieves the property that if an attacker obtains Alice's secret information before a refresh and Bob's secret information after a refresh (or vice versa), it cannot learn *anything* about the private key x. (The Paillier key must be changed as well as the sharing, since otherwise once the Paillier private key is stolen from Alice, it can corrupt Bob at any later time and decrypt c_{key}; this value together with x_2 (both held by Bob) yields the private key x. This level of security is called *proactive* in the academic literature.

We remark that the method described above results in the encryption of x_1' being larger than that of x_1, and in each refresh it can grow by up to q. Since part of the ECDSA signing protocol requires adding random noise to mask the opened value – this is the random multiple of q described in Section 5.4 – it is necessary to increase this multiple to take into account x_1' being larger. We do this in the implementation by adding an additional $2^{80} \cdot q$, which suffices for up to 2^{80} refreshes (something that will never happen).

6. Security Guarantees

All of our protocols are secure in the presence of malicious adversaries. This means that if one of the entities/parties is corrupted (e.g., breached by malware), then it cannot cheat even if it runs specially crafted malicious code. In particular, it cannot obtain a signature on any message that the other (honest) party does not approve. These guarantees are mathematically proven, according to formal cryptographic definitions of security. In addition, by using key-share refresh, as described above, an attacker must reside on both parties within a single refresh period. A strong level of security can be achieved in this way by ensuring strong separation between the devices/parties, making it hard to breach both. We summarize this in the following bullets:

1. The key is protected even when one of the parties has been hacked, even if the attacker can run malicious code.
2. A transaction is only signed if both parties consent to it. If the "two parties" are separate devices of the same person, then the system should be set up for the person to validate the transaction information on both devices.
3. Periodic refreshing of the shares makes it significantly harder for the attacker to break the system.

We remind the reader than any application using this code must take care to treat cheating that takes place in the dual execution, including the case that the last equality message is "dropped". This is crucial for secure usage of the library.

7. References

[1] Fabrice Boudot. Efficient Proofs that a Committed Number Lies in an Interval. In *EUROCRYPT 2000*, Springer (LNCS 1807), pages 431-444, 2000.

[2] Jan Camenisch, Rafik Chaabouni and abhi shelat. Efficient Protocols for Set Membership and Range Proofs. In *ASIACRYPT 2008*, Springer (LNCS 5350), pages 234-252, 2008.

[3] Sharon Goldberg, Leonid Reyzin, Omar Sagga and Foteini Baldimtsi. Certifying RSA Public Keys with an Efficient NIZK. *Cryptology ePrint Archive*: Report 2018/057.

[4] Vladimir Kolesnikov and Payman Mohassel and Ben Riva and Mike Rosulek. Richer Efficiency/Security Trade-offs in 2PC. In *TCC 2015*, Springer (LNCS 9014), pages 229-259, 2015.

[5] Yehuda Lindell. Fast Secure Two-Party ECDSA Signing. In *CRYPTO 2017*, Springer (LNCS 10402), pages 613-644, 2017. (Original version from CRYPTO 2017 here.)

[6] Payman Mohassel and Matthew K. Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Public Key Cryptography 2006*, Springer (LNCS 3958), pages 458-473, 2006.

[7] Tung Chou and Claudio Orlandi. The Simplest Protocol for Oblivious Transfer. In *LATINCRYPT 2015*.

[8] Marcel Keller, Emmanuela Orsini, Peter Scholl. Actively Secure OT Extension with Optimal Overhead. In *CRYPTO 2015*, Springer (LNCS 9215), pages 724-741, 2015.

[9] Yan Huang, Jonathan Katz and David Evans. Quid-Pro-Quo-tocols: Strengthening Semi-honest Protocols with Dual Execution. In *IEEE Symposium on Security and Privacy*, pages 272-284, 2012.

[10] Antonio Nicolosi, Maxwell Krohn, Yevgeniy Dodis, and David Mazières. Proactive Two-Party Signatures for User Authentication. In *Proceedings of the Network and Distributed System Security Symposium* (NDSS), 2003.