Threshold Signatures in Blockchain

Manan Monga

December 2020

1 Introduction

In a blockchain construction, operations are irreversible. As an adversary or other hostile entity can catastrophically affect a blockchain's contents, security is paramount. Conventional key generation schemes may be attacked or have signatures stolen. Key protection is not sufficient, and secret sharing with numerous parties can be time consuming and expensive. Thus, systems are looking towards threshold signature schemes. Revolving around conducting operations only once a minimum threshold of secret sharers participate or consent, threshold signatures are appearing in more and more asymmetric cryptography schemes [1, 3]. In this survey paper, we primarily focus on the Elliptic Curve Digital Signature Algorithm, henceforth abbreviated as ECDSA.

2 Multiparty Computation

The aim of secure multiparty computation is to enable parties to carry out distributed computing tasks in a secure and efficient. Distributed computing literature often deals with questions of computing under the threat of machine crashes and unavoidable inadvertent faults, secure multiparty computation is concerned with the possibility of deliberately malicious behavior by some adversarial entity which may attack the protocol [2]. The aim of this attack could be to learn private information or to cause the result of the computation to be incorrect. Therefore, two important requirements on any secure computation protocols are privacy and correctness. The privacy requirement states that nothing should be learned beyond what is absolutely necessary; more exactly, parties should learn their output and nothing else. The correctness requirement states that each party should receive its correct output. Therefore, the adversary must not be able to cause the result of the computation to deviate from the function that the parties had set out to compute.

Suppose two or more parties wished to compute the output of a function together. This function would accept a private value from each party. For the purposes of privacy, these parties would have to conduct a zero-knowledge proof, in which none of the sides learn anything outside of the output and their own respective inputs. While eavesdropping from saboteurs to obtain data from the communication line is certainly a factor, there are separate algorithms that consider this concern. Usually, models that account for significant computation are featured [1, 2].

3 Applications of Threshold Cryptography

Threshold Cryptography refers to the practice of sharing a private key between n parties with the stipulation that any subset must require t members to decrypt or sign. In some settings, this requirement can be more complex, needing more than just a minimal number of parties. For every private-key operation, threshold cryptography requires the condition of t parties to be met, which will be referred to as a quorum. Threshold Digital Signature Schemes allow a quorum to delegate their joint authority to sign a message to send to any subcommittee among themselves [2].

As threshold cryptography can be used to provide a high level of key protection, this technique is commonly found in cloud computing, authentication, sensor networks, and electronic voting applications [1].

Currently, ECDSA signing, a standard of threshold cryptography, is used in Bitcoin and other cryptocurrencies. A secure cryptocurrency wallet should enable the user to split their signing key among multiple devices and require a quorum to transfer money; banks and other institutions have interests in offering full cryptocurrency custody solutions to customers. Many startup companies have begun adopting threshold cryptography for key protection. Threshold cryptography provides flexibility, allowing arbitrary and complex access structures; anonymity, such that no parties learn any information about other parties; and scalability.

4 Elliptic Curve Digital Signature Algorithm

A standardized derivative of the earlier DSA devised by David Kravitz. While DSA was based upon arithmetic operaitons modulo over a prime, ECDSA uses elliptic curve operations over finite

fields [3]. Requiring shorter key lengths for an equivalent level of security and being more efficient than its predecessor, ECDSA is perhaps the most widespread of signature schemes. Current applications involving ECDSA include authenticated messaging, binary signing, and remote login - fields requiring high efficiency. Unfortunately, many existing techniques for generating ECDSA signatures require the invocation of heavy, inefficient cryptographic primitives.

Previously, ECDSA encountered issues with constructing efficient protocols for threshold cryptography. Pallier key generation was used to achieve several breakthroughs, but as PKG required immense computing power and was limited to two parties, this was an incomplete solution for multiparty ECDSA [2, 3]. When ECDSA is used as a threshold signature scheme, the stipulations overlap those principles found in multi-party computation protocols.

Namely, no malicious party should be able to subvert protocols to glean information from another party regarding the secret key, and no subset of malicious parties smaller than the quorum size minimum may be able to collude to generate signatures.

5 ECDSA - "Lindell"

As performance concerns and avoidance of certain assumptions often motivate the use of ECDSA, that current implementions with ECDSA themselves suffer from poor performance is unfortunate. However, [3] details the replacing of Pallier additively homomorphic encryption with ElGamal in-the-exponent. This replacement in this "Lindell" protocol enables ECDSA to achieve practical distributed key generation and fast signing.

5.1 ECDSA Signing Algorithm

Let E be an Elliptic curve group of order q with generator G. The private key is a random value $x \leftarrow Z_q$, and the public key is $Q = x * G$. ECDSA signing on a message $m \in \{0,1\}^*$ is defined as follows:

1. Compute m' to be the $|q|$ leftmost bits of $SHA256(m)$, denoted as $H_q(m)$.
2. Choose a random $k \leftarrow Z_p^*$.
3. Compute $R \leftarrow k * G$. Let $R = (r_x, r_y)$.
4. Compute $r = r_x mod q$ and $s \leftarrow k^{-1} * (m' + r * x) mod q$.
5. Output (r, s).

5.2 ElGamel In The Exponent

Let G represent a group of order q where the DDH assumption is assumed to be hard, and let g be a generator of this group. To complete this construction to be consistent with Elliptic curve notation, the group operation is addition; upper-case letters are group elements; and lower-case letters are scalars in Z_q. For this variant of ECDSA, this workaround to provide accessibility over PKG does not act as a valid encrpytion scheme, as decryption requires solving the discrete log problem.

An encryption of a value $m \in Z_q$ with public key $P \in G$ is denoted $\text{EGexpEnc}_p(m)$ and is formally defined by [3]:

$$EGexpEnc_p(m) = (A, B) = (r * g, r * P + m * g)$$

r is a random value taken from Z_q. This encryption is additively homomorphic, meaning that two ciphertexts can be added by calculating the sum of the two ciphertexts.

5.3 Secure Multiplication - $\mathcal{F}_{mult}$

Operating as an extended version of a multiplication functionality from additive shares, $\mathcal{F}_{mult}$ can be considered a helper functionality to ECDSA. The basic functionality is that each party P_i provides a_i and b_i for input and receives c_i. For additional ECDSA needs, each party also receives $a * G$. An additional stipulation required for security is that honest parties are limited to random input shares only, while corrupt parties can choose their own inputs. As part of use with ECDSA, this protocol is private but not always correct. In other words, malicious adversaries cannot learn more than allowed, but correctness is not a guarantee.

This multiplication functionality is used to securely compute a functionality family for ECDSA. One function is the ability to securely multiply values together, allowing parties to choose random x_i, k_i values and obtain $R = k * G$.

5.4 Distributed ECDSA Signing

As mentioned in the previous section, $\mathcal{F}_{mult}$ is heavily used to construct a protocol for securely computing $\mathcal{F}_{ECDSA}$. As part of the ECDSA construction, there are two phases with respectively different algorithms presented: key generation and signing. Within each algorithm, each party has the description of the group (G, q) and the number of parties n.

5.4.1 Key Generation

Each party P_i works as follows:

1. P_i sends $(init, G, q)$ to $\mathcal{F}_{mult}$ to run the initialization phase.
2. P_i sends $(input, sid_{gen})$ to $\mathcal{F}_{mult}$ and receives $(input, sid_{gen}, x_i)$.
3. P_i waits to receive $(input, 0)$ to $\mathcal{F}_{mult}$.
4. P_i sends $(element - out, 0)$ to $\mathcal{F}_{mult}$.
5. P_i receives $(element - out, 0, Q)$ from $\mathcal{F}_{mult}$.
6. Output: P_i locally stores Q as the ECDSA public-key.

5.4.2 Signing

Each party P_i works as follows, upon input Sign(sid, m) where sid is a unique session id sid:

1. P_i sends (input,$sid||1$) and (input,$sid||2$) to $\mathcal{F}_{mult}$ and receives (input,$sid||1, k_i$) and (input,$sid||2, p_i$). Both k_i and p_i denote the current cumulative sum of k and p, respectively.
2. After receiving (input,$sid||1$) and (input,$sid||2$) from $\mathcal{F}_{mult}$, P_i sends (mult, $sid||1, sid||2$) and (element-out, $sid||1$) to $\mathcal{F}_{mult}$.
3. P_i receives (mult-out,$sid||1, sid||2, r$) and (element-out,$sid||1, R$) from $\mathcal{F}_{mult}$, where $r = k * p$ and $R = k * G$.
4. P_i computes $R = (r_x, r_y)$ and $r = r_x mod q$.
5. P_i sends (affine, 0 $sid||3, r, m'$) to $\mathcal{F}_{mult}$, where $sid||3$ will be associated with $m' + x * r mod q$.
6. P_i sends (mult, $sid||2, sid||3$) to $\mathcal{F}_{mult}$.
7. P_i receives (mult-out, $sid||2, sid||3, b$) from $\mathcal{F}_{mult}$, where $b = p * (m' + x * r) mod q$.
8. P_i computes $s' = r^{-1} * b mod q$ and $s = min\{s, q - s\}$.
9. Output: P_i outputs (r, s).

5.5 Security Model

Through models and ideal/real behavior, two security properties for this protocol can be demonstrated. In the presence of malicious adversaries and static corruptions, the protocol follows the standard for instances no honest majority - security with abort [3]. This means that a corrupted or malicious party can learn output but an honest party could not. As in the protocol schematics, the adversary to the ideal model receives the output first and sends either (abort,j) or (continue,j) to the trusted party, where j represents the current party's index. Each party with indexes $j \in [n]$ will either receive the output or receive abort, depending on what that trusted party receives. This means that as different honest parties can either receive output or receive aborts, the information passed to each honest party is not consistent. Termed non-unanimous abort, referring to the inconsistent outputs, the protocol can be easily transformed such that all honest parties will receive the output if at least one honest party received output [3].

In other related work, the security of this protocol in a hybrid model with ideal functionalities that securely act as helper functions has already been proven [3]. Specifically, as long as subprotocols are used, indistinguishability of the output from honest and malicious parties using real protocols from calling a trusted party that just directly computes the ideal functionalities is guaranteed.

5.6 Efficiency and Results

There were two different "Lindel" instantiations considered for comparison: one based on oblivious transfer and one based on Paillier. However, only the Paillier-based protocol was developed in C++ and tested on AWS with Intel machines with 1 GB RAM and a 1Gb/s network card [3]. It was discovered that the OT-based variant required significantly fewer computations and exponentiations, as once the base OTs are computed in the KeyGen phase, OT extensions used in the actual signing have negligible, near-irrelevant cost. Additionally, the Pallier-based protocol requires additional expensive large-integer exponentiation calculations.

However, the variant with Paillier-based private multiplication features significantly lower communication costs, which makes this option more attractive for most real-world scenarios. Fortunately, the costs are low enough to be practical, especially for cryptocurrency applications. On a single-threaded context, the signing time goes from 304ms with 2 parties to about 3 seconds with 10 parties to 5 seconds with 20 parties [3]. While these speeds can be significantly increased through multi-threading, they are already practical. However, the process of KeyGen requires much more time: 11 seconds for 2 parties, 17 seconds for 10 parties, and 28 seconds for 20 parties. As key generation must only be run once, even this lengthy process can be considered practical.

Ultimately, the goal of the "Lindell" protocol was to demonstrate that practical key generation, signing, and distribution could be achieved for multiparty threshold ECDSA signature schemes. The results with the provided algorithms and implemented subroutines demonstrate that, owing to the communication times with Pallier-based private multiplication, those goals are feasible. This protocol, by demonstrating the mechanics of practical distributed key generation, paves the way to practically implementing solutions for applications demanding threshold ECDSA signing. However, as the "Lindell" protocol was akin to a pioneer, it lacks features and efficiency that other, newer protocols contain.

6 ECDSA - "GG20"

The "Lindell" protocol was one of the first to provide efficient threshold ECDSA with additional distributed key generation and has opened the door to more advanced ECDSA protocols. Named after its researchers and the year that it was published in, "GG20" is one such protocol that seeks to alleviate two drawbacks found in prior threshold signature schemes. Specifically, "GG20" is a protocol for threshold ECDSA that improves efficiency and enables identification of misbehaving parties [4].

Identifying misbehaving parties can be crucial for some applications. In most applications, being able to identify rogue servers is a convenience, allowing masters to avoid restarting all servers. However, in an application that involves several distinct parties controlling key shares, the inability to identify aborts can be catastrophic. In the latter case, to identify the rogue agent is to enable the service.

6.1 ECDSA Scheme Constructions

Players receive input $\{G,g\}$, the cyclic group used by the ECDSA signature scheme. Each player P_i is associated with a public key E_i for an additively homomorphic encryption scheme ϵ. Much of the construction matches that found in the previous protocol "GG18", albeit with augmentations that are later exploited to enable identifying aborts [4].

6.1.1 KeyGen Protocol

- Phase 1: Each Player P_i selects $u_i \in_R Z_q$; computes $[KGC_i, KGD_i] = \text{Com}(g^{u_i})$. Each Player P_i broadcasts E_i, the public key for Paillier's cryptosystem.
- Phase 2: Each Player P_i broadcasts KGD_i. Let y_i be the value decommitted by P_i. The player P_i performs a (t, n) Feldman-VSS, a verifiable secret sharing scheme extension, of the value u_i, with y_i as the free term. The public key is then set to the product of all y_i. Each player adds the private shares received during the n Feldman VSS protocols, and the resulting values x_i are a (t, n) Shamir's secret sharing of the secret key.
- Phase 3: Let $N_i = p_i * q_i$ be the RSA modulus associated with E_i. Each player P_i proves in zero knowledge that they know x_i using Schnorr's protocol.

6.1.2 Signing Protocol

This protocol is run on input m, the hash of the message M being signed, and the output of the previous KeyGen protocol. Let $S \subseteq [1..n]$ be the set of all players participating in the signature. The size of S should equal $t + 1$. Thus, ephemeral secrets can be shared using a $(t, t + 1)$ secret sharing scheme and do not require the general (t, n) structure.

- Phase 1: Each Player P_i selects $k_i, r_i \in_R Z_q$; computes $[C_i, D_i] = \text{Com}(g^{r_i})$ and broadcast C_i. k and r are thus defined as the sum of all k_i and r_i, respectively.

- Phase 2: Every pair of players P_i, P_j engages in two multiplicative-to-additive share conversion subprotocols. However, as the first message for these protocols is the same, it is only sent once.
- Phase 3: Every player P_i broadcasts δ_i and computes $\delta^{-1} mod q$, and $\delta = \sum_{i \in S} \delta_i = kr$. Additionally, every player P_i also broadcasts T_i and proves in zero knowledge that they know σ_i, γ_i.
- Phase 4: Each Player P_i broadcasts D_i, and R_i are the values decommitted by P_i. The players compute the sum of all R_i, and $R = g^{k^{-1}}$, and $r = H'(R)$.
- Phase 5: Each Player P_i broadcasts $\bar{R}_i = R^{k_i}$, as well as a zero knowledge proof of consistency between R_i and $E_i(k_i)$, acquired from Phase 2. If $g \neq$ the product of all $\bar{R}_i$, the protocol aborts.
- Phase 6: Each player P_i broadcasts $S_i = R^{\sigma_i}$ and a zero knowledge proof of consistency between S_i and T_i, acquired from Phase 3.
- Phase 7: Each player P_i broadcasts $s_i = mk_i + r\sigma_i$ and set $s =$ the sum of all s_i. If the signature (r, s) is correct for m, the players accept, otherwise they abort.

6.2 Noninteractive Online Phases

"GG20" is capable of being split into an offline preprocessing stage. In this preprocessing stage, most of the computation and commnuication is completed. Subsequently comes the online stage, in which the message is known, consisting of a single communication round. In this one round, each player performs a scalar multiplication, which results in a later significantly lowered computation cost [4].

During the preprocessing phase, additive shares of s are created from additive sharings of x, k. At some point, a new distributed verification check is performed on those shares of s to ensure that they combine to construct a correct signature. As this check is performed on the additive shares before the message is known, this check can be moved to the preprocessing phase. Such a move reduces complexity and eliminates the requirement for online interactivity. Within the online phase, because the message would already be known, players would only require only scalar multiplication and one communication round.

6.3 Identifying Aborts

Identifying misbehaving parties efficiently is a key contribution of "GG20" [4]. If an abort happens during the preprocessing stage, then the full signature has not been revealed yet and the players can reveal the random choices they made during the protocol so far so that their behavior can be verified. Bad players would thus be identified. If an abort occurs during the online stage, then the shares of the signatures that each player reveals can easily be checked to be correct against public information procured by the offline stage.

In this protocol, an abort will happen if any player deviates from the protocol in a clearly identifiable way by not complying with instructions. Assume that all messages transferred between players are signed, so identifying the messages' origins is possible. In the case of such an abort, the guilty party would clearly be identified and removed. These identifications are of simpler difficulty. In the case of an abort, identifying guilty parties even when they had not deviated from the protocol significantly is a much larger contribution of "GG20".

Within "GG20"'s KeyGen protocol, there are two possible places an abort can occur. In Phase 2, a player complaining about receiving an inconsistent Feldman share will cause an abort. In Phase 3, a player failing to prove knowledge of x_i or the correctness of their Paillier key will cause an abort [4]. In the latter case, failing a zero knowledge proof will immediately reveal the guilty party. However, in Phase 2, a player P_i is complaining about receiving a faulty Feldman VSS share from player P_j, meaning that the suspect between those two players is ambiguous. However, a simple identification protocol requires the complaining player to publish their received share for all other players to authenticate. Regardless of where the abort occurs, once the misbehaving player is removed, the key generation protocol must be rerun with fresh randomness to ensure a secure key.

Within "GG20"'s Sign protocol, there are eight instances in which an abort can occur. Five of these eight instances involve a player failing a zero knowledge proof or a commitment opening and are easily concluded. However, in the instance that the final output, that the signature (r, s) must be correct for m, causes a fail, identification is slightly more complicated. Each player P_i would thus be asked to confirm $R^{s_i} = \bar{R}_i^m * S_i^r$. Whichever player fails is then identified as the malicious party. In the last two cases, a quorum-wide value fails to authenticate, meaning that potential fault is spread too far to disentangle from similar distributed verification checks [4].

In the last two cases, to prove that the players ran the protocol correctly to confirm that fault is with a malicious party, players are asked to reconfirm various values inputted at previous stages. Once the protocol is vindicated, each player P_i will then reveal their individual values $k_i, \gamma_i, \alpha_i, \beta_i$, which all players will confirm. From this, the corrupting player can be identified.

6.4 Efficiency and Results

Unlike other contemporary protocols, the "GG20" protocol does not require distributed verification of the signature's validity. The removal of this check causes the protocol to be more round count efficient and removes any and all dependencies on the multi-round interactive parts. Additionally, the implementation of parallelization during the protocol participation phase becomes possible. Furthermore, the signature may be computed in as little as a single round, greatly increasing efficiency and reducing the total round count required [4].

There was a second protocol developed that continued to further simplify "GG20". This second protocol was designed for applications in which identifiable abort is not necessary, allowing this protocol to have a preprocessing phase that is reduced in round and computational complexity. Specifically, players in Phase 3 do not commit to the value σ_i using T_i. In the main "GG20" scheme, this step was to ensure players were committed to the correct partial signature s_i and enabled the accountability of corrupt parties. As the commit and correctness check are removed, complexity is also reduced.

For testing purposes, "GG20" was implemented in Go [4]. For additional fairness purposes, this protocol was ran using a single core when benchmarking. However, much of the code is highly parallelizable and could significantly reduce the reported runtimes. The results indicated that the protocol was practical even with a non-interactive online phase. While other protocols required online interactivity, "GG20" only requires one elliptic curve multiplication and one addition to complete computation for the online phase. Thus, "GG20" allows players to asynchronously participate without the need to be online simultaneously. In terms of other goals motivating the construction of "GG20", this protocol also reduces computations of contemporary protocols and adds significant functionality: noninteractive online phases and identifiable aborts.

7 ECDSA - "DJN20"

Ivan Damgard et al. provide a ECDSA protocol for threshold signatures that is simple and efficient in terms of both computation and bandwidth usage. The protocol is in the honest majority model with abort and is secure against active adversary and works for n parties with a security threshold t defined as $n \geq 2t + 1$.

The full proof in the Universal Composability model perfectly realizes the standard ECDSA functionality with no additional assumptions. This protocol is also extremely efficient in the online model because it suits a pre-processing based data pipeline. Parties, on receiving the message to be signed, can compute a sharing of the signature using only local operations and do not need to interact with each other.

7.1 Protocol Construction

In this construction, each party generates a private key $[x]$ using joint random secret sharing. Then, a public key y is calculated as $y = g^x$. For secret sharing, plain Shamir secret sharing is used along with a simpler protocol to reveal g^x. The protocol for revealing g^x should work despite multiple malicious parties and should abort if $[x]$ is not a consistent sharing between the parties.

To sign a message M, the parties have to generate a sharing of the nonce $[k]$ and then reveal g^k similar to how the public key is revealed. Beaver's inversion trick is the used to compute $[k^{-1}]$ as follows:

1. Generate a random sharing $[a]$
2. Multiply and open $w = [a][k]$

This is a degree of $2t$ sharing of ak so ak can be recovered as long as all parties participate honestly, but with malicious parties, there needs to be an abort toleration to correctly open an authenticator $W = g^{ak}$ that lets parties verify the correctness of W.

7.2 Point Power Calculation

An integral part of this protocol is the sub-protocol $y < -POWOPEN(g, [x])$ that given a sharing $[x]$ and a generator g, reveals the value $y = g^x$. $POWOPEN$ works as:

1. Every party P_i broadcasts $y_i = g^{x_i}$ to all the other parties.
2. f is the unique degree t polynomial defined by the $t + 1$ honest parties' shares such that $f(0) = x$
3. After P_i has received all the g^{x_j} for each $y_j < -\{y_{t+2}, y_{t+3}.....y_n\}$ it verifies that y_j is consistent with the degree t polynomial defined by the first $t + 1$ values $y_1, y_2.....y_{t+1}$. This verification is done by doing a Lagrange interpolation "in the exponent"
4. If the verification is passed, P_i again uses Lagrange interpolation on $y_1, y_2.....y_{t+1}$ to compute $y = g^x = g^{f(0)}$

$POWOPEN$ is an important part of the key generation step and also adds a security feature because for $n \geq 2t+1$ all honest parties will abort if the input sharing defined by the honest parties is inconsistent, i.e., if the shares done are not points on a degree t polynomial, irrespective of what other actions the corrupted parties perform.

7.3 Key Generation

For key generation in this protocol, the aim is to generate a sharing $[x]$ of a uniformly random value $x < -Z_q$ and reveal to each party $y = g^x$. The protocol makes the parties use plain Shamir secret sharing for obtaining x and lets them run $POWOPEN$ to obtain $y = g^x$.

For correctness, the use of Shamir sharing instead of Verifiable secret sharing means that a single malicious party P_i may cause $[x]$ to become an inconsistent sharing and this will cause $POWOPEN$ to abort.

An important part of the protocol is that no honest party P_i reveals their value g^{x_i} they have received the shares x_j from all other parties P_j which forces the corrupt parties P_j to "commit" to their values x_j prior to having seen y.

The protocol also ensure if two parties output a public key, it is the same public key y. In addition, all subsets of $t+1$ honest parties that receive output, will receive shares of the same private key x.

7.4 Signature Generation

To generate a signature, the parties generate $[a]$ using Shamir sharing and then compute $[w] = [a][k]$ and open $[w]$ using $POWOPEN$.The multiplication is done as:

1. Each party computes their share $w_i = a_i k_i$
2. Results in shares on a polynomial f_w of degree $2t$ with $f_w(0) = w$.
3. Since $n \geq 2t+1$, there are enough parties to interpolate w if they all reveal their shares.

To avoid that their shares leak unintended information, parties compute a random degree $2t$ zero sharing as $[b] \text{¸} ZSS(2t)$ and then reveal instead shares $a_i k_i + b_i$. This protocol is denoted as $w < -WMULOPEN([a],[k];[b])$. The "$w$" is for "weak", since a single malicious party can cause the protocol to output anything and the only guarantee this is provided by the protocol is that no information about a and k, except of the product ak, is revealed.

Since $[a]$ and $[b]$ are being generated using secret sharing, they are not known to be consistent sharings at this point. But, each share b_i is known to be a random value that blinds the share $a_i k_i$, which is not necessarily random. To deal with corrupt parties leading to bad shares w_i which can cause parties to end up with the wrong value of w we use a trick in order to compute an authenticator $W = g^{ak}$. This allows each party to check that $g^w = W$ and abort if not. Since correctness of $POWOPEN$ is ensured as explained above, even if $[a]$ is not a consistent sharing, all honest parties will abort at this point unless $w = ak$.

Finally, given $[x]$, $[k^{-1}]$, $r = F(g^k)$ and the message m to sign, the parties compute the value:

$$[s] = [k^{-1}](m + r[x])$$

7.5 Fairness in Online Phase

This protocol has no fairness or termination guarantee. The adversary gets to see the signature (r, s) and may then abort the protocol before any honest party receives the signature. In practice, parties will retry on abort and the adversary may therefore end up with several valid signatures $(r_1, s_1), ...(r_L, s_L)$ on message m without any of the honest parties knowing any of these signatures. This is not a forgery because a forgery can only happen with messages that the honest parties actually intended to sign.

Since we assume an honest majority it is indeed possible to achieve fairness. The basic protocol can be extended with just two additional pre-processing rounds in order to achieve fairness. The main idea is that in addition to R and $[k^{-1}]$ the parties also prepare a sharing of $[xk^{-1}]$ in the pre-processing. Doing so, $[s]$ can be computed as $[s] = m[k^{-1}] + r[xk^{-1}]$, using only linear operations. Taking this one step further, by reducing the degree of $[xk^{-1}]$ to $t$and turning both $[k^{-1}]and[xk^1]$ into suitable verifiable secret sharings , this protocol achieves the property that online, when Mm is known, the signature can be computed given only $t+1$ correct shares and the correctness of each share can be validated.

7.6 Performance

The protocol requires four rounds of interaction between the servers to generate a signature. The first three rounds can be processed before the message that needs to be signed known and can be computed as a presignature. The presignature denotes the value R and the sharings $[k-1]$, $[e]$, $[d]$ produced during the first three rounds. Each party can save a unique presignature id (such as R). When the message M is revealed, the parties need only then agree on a presignature id and can

complete the signature protocol d just one r The$<-\text{rot}_o\text{col}_i\text{s}$ designed_to run $n \geq 2t+1$ parties. A random element $r <- Z_q$ can be represented using $log_2 q$ bits and an element in $G <- Z_p \times Z_p$ using $log_2 p$ bits (roughly) using point compression.

The protocol is constant-round and the communication complexity is $O(\kappa n^2)$ under the assumption that both $logp$ and $logq$ are proportional to a security parameter κ. In the case of small number of parties, the computational bottleneck is likely to be the "long" curve exponentiations, i.e., computing g^r for random values $r <- Z_q$. For large n, since the protocol is constant round, the $O(n^2)$ amount of arithmetics in Z_q that each party needs to perform will eventually become the bottleneck.

8 Conclusion

Threshold cryptography enables a set of two or more parties to carry out cryptographic operations, without any single party holding the secret key. This is extremely useful in a Blockchain construction when it is essential to obtain consensus from multiple parties before the chain can be propagated further. We have seen how ECDSA signing can be used to ensure multiple signatories on a transaction. Threshold cryptography also supports quorum approvals involving many parties (for example, requiring (t + 1)-out-of-n parties to sign, and maintaining security for any subset of t corrupted parties) which makes it the right choice for distributed signing in a Blockchain.

9 References

1. Yao, A. C. (n.d.). Protocols for Secure Computations (extended abstract) [Scholarly project]. Retrieved from https://research.cs.wisc.edu/areas/sec/yao1982-ocr.pdf

2. Lindell, Y. (n.d.). Fast Secure Two-Party ECDSA Signing [Scholarly project]. Retrieved from https://eprint.iacr.org/2017/552.pdf

3. Lindell, Y., Nof, A., amp; Ranellucci, S. (2018, October 14). Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody [Scholarly project]. Retrieved from https://eprint.iacr.org/2018/987.pdf

4. Rosario Gennaro and Steven Goldfeder. One round threshold ECDSA with identifiable abort. Cryptology ePrint Archive, Report 2020/540, 2020. https://eprint.iacr.org/2020/540.

5. Ivan Damgard, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergard. Fast threshold ecdsa with honest majority. Cryptology ePrint Archive, Report 2020/501, 2020. https://eprint.iacr.org/2020/501.