Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security

Rosario Gennaro[1], Steven Goldfeder[2], and Arvind Narayanan[2]

[1] City College, City University of New York
rosario@cs.ccny.cuny.edu
[2] Princeton University
{stevenag,arvindn}@cs.princeton.edu

Abstract. While threshold signature schemes have been presented before, there has never been an optimal threshold signature algorithm for DSA. Due to the properties of DSA, it is far more difficult to create a threshold scheme for it than for other signature algorithms. In this paper, we present a breakthrough scheme that provides a threshold DSA algorithm that is efficient and optimal. We also present a compelling application to use our scheme: securing Bitcoin wallets. Bitcoin thefts are on the rise, and threshold DSA is necessary to secure Bitcoin wallets. Our scheme is the first general threshold DSA scheme that does not require an honest majority and is useful for securing Bitcoin wallets.

1 Introduction

Threshold signature schemes enable sharing signing power amongst n parties such that any subset of $t+1$ can jointly sign, but any smaller subset cannot. This problem has received much attention in the cryptographic literature, and many such schemes have been designed. Some of these schemes produce signatures that are compatible with standard digital signature schemes. They replace only the signing algorithm and key generation algorithm, but the verification is compatible with the centralized signature schemes.

The Digital Signature Algorithm (DSA) is a very popular signature scheme, and a considerable amount of work has been done to build a threshold signing algorithm to produce a standard DSA signature. However, for reasons that we will elaborate in Section 3.2, building a threshold variant of DSA proved to be significantly difficult. While such schemes have been presented (e.g. [22, 23, 31]), they have serious drawbacks that make them unusable in practice: in particular no general scheme with an optimal number of servers is known. For the past 15 years, the problem has been mostly abandoned. The reason is twofold: there was neither a pressing problem nor a clear solution:

- As we discuss in Section 3.2 the technical difficulties in building a threshold-optimal variant of distributed DSA made this a challenging problem and it was not clear how to proceed from the solutions in [22, 23, 31].

– There was never a great motivation to devise a solution for threshold DSA, in particular one that is optimal in the number of servers. Since there are plenty of optimized threshold signature schemes for other algorithms, one who wanted to use a threshold scheme would generally just choose a different signature scheme that was well suited for the problem at hand. To the best of our knowledge, there has never previously been an application for which DSA was the only option.

In recent years, as it turns out, a major application for threshold DSA signatures has arisen in the world of Bitcoin. Without an ECDSA threshold scheme, bitcoins are subject to a single point of failure and the risks of holding bitcoins are catastrophic. [3] Motivated by this application, we tackle the technical challenges of threshold DSA, and present an efficient and optimal scheme realizing it.

The motivation: Bitcoin's security conundrum

Bitcoin is a cryptographic e-cash system, by far the most widely used today. Unlike traditional banking transactions, Bitcoin transactions of any size can be fully automated – authorized only with a ECDSA signature. One's bitcoins are only as secure as the ECDSA key that can authorize their transfer; if this key is compromised, the Bitcoins will be stolen. Unlike traditional banking transactions, once a Bitcoin transaction is enacted it is irreversible. Even if the coins are known to have been stolen, there is simply no way to reverse the offending transaction.

Indeed, the Bitcoin ecosystem is plagued by constant thefts. The statistics on Bitcoin hacks, thefts, and losses are extraordinary — there have been ten thefts of over 10,000 BTC each since mid-2011, and at least another thirty-four of over 1,000 BTC[4] [5] [4]. Kaspersky labs report detecting about a million infections per month of malware designed to search for and steal bitcoins from machines they infect [30].

The pervasiveness and regularity of these vulnerabilities highlight how Bitcoin is inherently theft-prone. For Bitcoin and cryptocurrencies to gain mainstream adoption, a breakthrough in security is needed — the current situation where a single rogue employee or a piece of malware can empty an organization's funds in hot storage instantly, irreversibly, and anonymously is simply untenable. Securing Bitcoin is equivalent to securing the keys that can authorize transactions. Instead of storing keys in a single location, keys should be split and signing should be authorized by a threshold set of computers. A breach of any one of these machines – or any number of machines less than the threshold

[3] Bitcoin actually uses ECDSA, the elliptic curve variant of DSA, but all of our results in this paper as well as the previous literature on the subject are equally applicable to DSA and ECDSA. When we present the scheme, we explicitly state how to implement it for both DSA and ECDSA.

[4] As of this writing, a bitcoin trades for around USD 250.

[5] The majority, but not all, of these losses have been due to theft of keys.

will not allow the attacker to steal any money or glean any information about the key.

Since Bitcoin transactions use ECDSA keys, the only way to achieve this joint control is with an ECDSA threshold signature algorithm. While Bitcoin does have a built in "multi-signature" function for splitting control, using this severely compromises the confidentiality and anonymity of the participants, and thus it is not advisable to use as we explain fully in Section 6.3. Only an ECDSA threshold signature algorithm can provide the security we need without compromising on privacy.

Our contributions

With a strong motivation for threshold DSA, we still lacked a scheme that was usable to secure Bitcoin keys. The best threshold signature scheme presented was that by Gennaro *et al.* [22]. Their scheme, however, has a considerable setback. The key is distributed amongst n players such that a group of $t+1$ players can jointly reconstruct the key. Yet, in order to produce a signature using their algorithm (without reconstructing the key), the participation of $2t+1$ players is required.

This property of the scheme in [22] has various implications. First, requiring $n \geq 2t+1$ is very limiting in practice: for example it rules out an n-of-n sharing. Furthermore, the implications for a Bitcoin company that wants to distribute its signing power are severe. If the company chooses a threshold of t, then an attacker who compromises $t+1$ servers can steal all of the company's money. Yet, in order for the company to sign a transaction, they must set up $2t+1$ servers. In effect, they must double the number of servers, which makes the job of the attacker easier (as there are more servers for them to target).

In an attempt to get to an optimal number of servers, Mackenzie and Reiter built a specialized scheme for the 2-of-2 signature case [31], a case that was unrealizable using Gennaro *et al.*'s scheme. Yet no general DSA threshold scheme existed that did not suffer from these setbacks. In Appendix A, we sketch how to extend Mackenzie and Reiter to the multiparty case. While the extension does allow $t+1$ players to sign, it is quite inefficient. In particular, it requires $3t-1$ rounds of interaction, and the computation time as well as the storage grows exponentially with the number of players.

In this paper, we present a scheme that is both threshold optimal and efficient. In particular:

1. It requires only $n \geq t+1$ servers to protect against an adversary who compromises up to t servers.
2. The protocol requires only a constant number of rounds
3. The computation time for each player is constant[6]
4. Players require only a constant amount of storage

[6] That is to compute the players share, the computation time does not grow with the number of players. Players do however need to verify proofs from all players.

Our scheme is practical and efficient. We have implemented it and evaluated it, and it is the only scheme that is fully compatible with Bitcoin as well as efficient enough to now be a true candidate for any use case where a threshold signature scheme is desired. We have also spoken with various Bitcoin companies who confirmed that they are eager to incorporate our protocol to secure their systems.

Malicious Faults. If we consider an honest-but-curious adversary, i.e. an adversary that learns all the secret data of the compromised server but does not change their code, then our protocol produces signatures with $n = t + 1$ players in the network (since all players will behave honestly, even the corrupted ones). But in the presence of a malicious adversary, who can force corrupted players to shut down or to send incorrect messages, one needs at least $n = 2t + 1$ players in total to guarantee *robustness*, i.e. the ability to generate signatures even in the presence of malicious faults. In that sense our protocol improves over [22, 23] where $n = 3t + 1$ players are required to guarantee robustness.

But as we already discussed above, we want to minimize the number of servers, and keep it at $n = t + 1$ even in the presence of malicious faults. In this case we give up on robustness, meaning that we cannot guarantee anymore that signatures will be provided. But we can still prove that our scheme is unforgeable. In other words, the adversary can only create a denial of service attack, but not learn any information that would allow him to forge even if there is a single honest player left in the network. This is another contribution of our paper, since it is not clear how to provide such "dishonest majority" analysis in the case of [22, 23].

2 Model, Definitions and Tools

In this section we introduce our communication model and provide definitions of secure threshold signature schemes.

Communication Model. We assume that our computation model is composed of a set of n *players* $P_1, \ldots, P_n$ connected by a complete network of point-to-point channels and a broadcast channel.

The Adversary. We assume that an adversary, $\mathcal{A}$, can corrupt up to t of the n players in the network. $\mathcal{A}$ learns all the information stored at the corrupted nodes, and hears all the broadcasted messages. We consider two type of adversaries:

- *honest-but-curious:* the corrupted players follow the protocol but try to learn information about secret values;
- *malicious:* corrupted players to divert from the specified protocol in *any* (possibly malicious) way.

We assume that the network is "partially synchronous", meaning that the adversary speaks last in every communication round (also known as a *rushing*

adversary.) The adversary is modeled by a probabilistic polynomial time Turing machine.

Adversaries can also be categorized as *static* or *adaptive*. A static adversary chooses the corrupted players at the beginning of the protocol, while an adaptive one chooses them during the computation. In the following, for simplicity, we assume the adversary to be static, though the techniques from [13, 28] can be used to extend our result to the adaptive adversary case.

Given a protocol $\mathcal{P}$ the *view* of the adversary, denoted by $\mathcal{VIEW}_{\mathcal{A}}(\mathcal{P})$, is defined as the probability distribution (induced by the random coins of the players) on the knowledge of the adversary, namely, the computational and memory history of all the corrupted players, and the public communications and output of the protocol.

Signature Scheme. A signature scheme $\mathcal{S}$ is a triple of efficient randomized algorithms (Key-Gen, Sig, Ver). Key-Gen is the *key generator* algorithm: on input the security parameter 1^λ, it outputs a pair (y, x), such that y is the *public key* and x is the *secret key* of the signature scheme. Sig is the *signing* algorithm: on input a message m and the secret key x, it outputs *sig*, a signature of the message m. Since Sig can be a randomized algorithm there might be several valid signatures *sig* of a message m under the key x; with $\mathsf{Sig}(m, x)$ we will denote the set of such signatures. Ver is the *verification* algorithm. On input a message m, the public key y, and a string *sig*, it checks whether *sig* is a proper signature of m, i.e. if $sig \in \mathsf{Sig}(m, x)$.

The notion of security for signature schemes was formally defined in [25] in various flavors. The following definition captures the strongest of these notions: existential unforgeability against adaptively chosen message attack.

Definition 1. *We say that a signature scheme* $\mathcal{S}$ =(Key-Gen,Sig,Ver) *is* unforgeable *if no adversary who is given the public key* y *generated by* Key-Gen*, and the signatures of* k *messages* $m_1, \ldots, m_k$ *adaptively chosen, can produce the signature on a new message* m *(i.e.,* $m \notin \{m_1, \ldots, m_k\}$*) with non-negligible (in* λ*) probability.*

Threshold secret sharing. Given a secret value x we say that the values $(x_1, \ldots, x_n)$ constitute a (t, n)-threshold secret sharing of x if t (or less) of these values reveal no information about x, and if there is an efficient algorithm that outputs x having $t + 1$ of the values x_i as inputs.

Threshold signature schemes. Let $\mathcal{S}$=(Key-Gen, Sig, Ver) be a signature scheme. A (t, n)-threshold signature scheme $\mathcal{TS}$ for $\mathcal{S}$ is a pair of protocols (Thresh-Key-Gen, Thresh-Sig) for the set of players $P_1, \ldots, P_n$.

Thresh-Key-Gen is a distributed key generation protocol used by the players to jointly generate a pair (y, x) of public/private keys on input a security parameter 1^λ. At the end of the protocol, the private output of player P_i is a value x_i such that the values $(x_1, \ldots, x_n)$ form a (t, n)-threshold secret sharing of x. The public output of the protocol contains the public key y. Public/private key pairs (y, x) are produced by Thresh-Key-Gen with the same probability distribution as if

they were generated by the Key-Gen protocol of the regular signature scheme $\mathcal{S}$. In some cases it is acceptable to have a *centralized* key generation protocol, in which a trusted dealer runs Key-Gen to obtain (x, y) and the shares x among the n players.

Thresh-Sig is the distributed signature protocol. The private input of P_i is the value x_i. The public inputs consist of a message m and the public key y. The output of the protocol is a value $sig \in \mathsf{Sig}(m, x)$.

The verification algorithm for a threshold signature scheme is, therefore, the same as in the regular centralized signature scheme $\mathcal{S}$.

Secure Threshold Signature Schemes.

Definition 2. *We say that a (t, n)-threshold signature scheme $\mathcal{TS}$ =(*Thresh-Key-Gen,Thresh-Sig*) is* unforgeable*, if no malicious adversary who corrupts at most t players can produce, with non-negligible (in λ) probability, the signature on any new (i.e., previously unsigned) message m, given the view of the protocol* Thresh-Key-Gen *and of the protocol* Thresh-Sig *on input messages $m_1, \ldots, m_k$ which the adversary adaptively chose.*

This is analogous to the notion of existential unforgeability under chosen message attack as defined by Goldwasser, Micali, and Rivest [25]. Notice that now the adversary does not just see the signatures of k messages adaptively chosen, but also the internal state of the corrupted players and the public communication of the protocols. Following [25] one can also define weaker notions of unforgeability.

In order to prove unforgeability, we use the concept of *simulatable adversary view* [12, 26]. Intuitively, this means that the adversary who sees all the information of the corrupted players and the signature of m, could generate by itself all the other public information produced by the protocol Thresh-Sig. This ensures that the run of the protocol provides no useful information to the adversary other than the final signature on m.

Definition 3. *A threshold signature scheme $\mathcal{TS}$ =(*Thresh-Key-Gen,Thresh-Sig*) is* simulatable *if the following properties hold:*

1. *The protocol* Thresh-Key-Gen *is simulatable. That is, there exists a simulator SIM_1 that, on input a public key y, can simulate the view of the adversary on an execution of* Thresh-Key-Gen *that results in y as the public output.*
2. *The protocol* Thresh-Sig *is simulatable. That is, there exists a simulator SIM_2 that, on input the public input of* Thresh-Sig *(in particular the public key y and the message m), t shares $x_{i_1}, \ldots, x_{i_t}$, and a signature sig of m, can simulate the view of the adversary on an execution of* Thresh-Sig *that generates sig as an output.*

Threshold Optimality. Given a (t, n)-threshold signature scheme, obviously $t + 1$ honest players are necessary to generate signatures. We say that a scheme is *threshold-optimal* if $t + 1$ honest players also suffice.

The main contribution of our work is to present a threshold-optimal DSA scheme for general t. The only known optimal scheme was in [31] for the case

of $(1,2)$-threshold (i.e. 2-out-of-2) threshold DSA. The protocol in [22, 23] is not threshold-optimal as it requires $2t+1$ honest players to compute a signature.

We point out that if we consider an honest-but-curious adversary, then it will suffice to have $n = t+1$ players in the network to generate signatures (since all players will behave honestly, even the corrupted ones). But in the presence of a malicious adversary one needs at least $n = 2t+1$ players in total to guarantee *robustness*, i.e. the ability to generate signatures even in the presence of malicious faults. In that sense our protocol improves over [22, 23] where $n = 3t+1$ players are required to guarantee robustness.

But as we already discussed in the introduction, we want to minimize the number of servers, and keep it at $n = t+1$ even in the presence of malicious faults. In this case we give up on robustness, meaning that we cannot guarantee anymore that signatures will be provided. But we can still prove that our scheme is unforgeable. In other words an adversary that corrupts almost all the players in the network can only create a denial of service attack, but not learn any information that would allow him to forge. This is another contribution of our paper, since it is not clear how to provide such "dishonest majority" analysis in the case of [22, 23][7].

2.1 Additively Homomorphic Encryption

We assume the existence of an encryption scheme E which is additively homomorphic modulo a large integer N: i.e. given $\alpha = E(a)$ and $\beta = E(b)$, where $a, b \in Z_N$, there is an efficiently computable operation $+_E$ over the ciphertext space such that

$$\alpha +_E \beta = E(a + b \bmod N)$$

Note that if x is an integer, given $\alpha = E(a)$ we can also compute $E(xa \bmod N)$ efficiently. We refer to this operation as $x \times_E \alpha$. We denote the message space of E by $\mathcal{M}_E$ and the ciphertext space by $\mathcal{C}_E$.

With $\bigoplus_{i=1}^{t+1} \alpha_i$ we denote the summation over the addition operation $+_E$ of the encryption scheme: i.e. $\bigoplus_{i=1}^{t+1} \alpha_i = \alpha_1 +_E \ldots +_E \alpha_{t+1}$.

One instantiation of a scheme with these properties is Paillier's encryption scheme [35]. We recall the details of the scheme here.

- Key Generation: generate two large primes P, Q of equal length. and set $N = PQ$. Let $\lambda(N) = lcm(P-1, Q-1)$ be the Carmichael function of N. Finally choose $\Gamma \in Z^*_{N^2}$ such that its order is a multiple of N. The public key is (N, Γ) and the secret key is $\lambda(N)$.
- Encryption: to encrypt a message $m \in Z_N$, select $x \in_R Z^*_N$ and return $c = \Gamma^m x^N \bmod N^2$.
- Decryption: to decrypt a ciphertext $c \in Z_{N^2}$, let L be a function defined over the set $\{u \in Z_{N^2} : u = 1 \bmod N\}$ computed as $L(u) = (u-1)/N$. Then the decryption of c is computed as $L(c^{\lambda(N)})/L(\Gamma^{\lambda(N)}) \bmod N$.

[7] The protocols of [22, 23] include multiplications of Shamir secret shares, so the $2t+1$ minimum is inherent.

– Homomorphic Properties: Given two ciphertexts $c_1, c_2 \in Z_{N^2}$ define $c_1 +_E c_2 = c_1 c_2 \bmod N^2$. If $c_i = E(m_i)$ then $c_1 +_E c_2 = E(m_1 + m_2 \bmod N)$. Similarly, given a ciphertext $c = E(m) \in Z_{N^2}$ and a number $a \in Z_n$ we have that $a \times_E c = c^a \bmod N^2 = E(am \bmod N)$.

2.2 Threshold Cryptosystems

In a (t, n)-threshold cryptosystem, there is a public key pk with a matching secret key sk which is shared among n players with a (t, n)-secret sharing. When a message m is encrypted under pk, $t+1$ players can decrypt it via a communication protocol that does not expose the secret key.

More formally, a public key cryptosystem $\mathcal{E}$ is defined by three efficient algorithms:

– key generation Enc-Key-Gen that takes as input a security parameter λ, and outputs a public key pk and a secret key sk.
– An encryption algorithm Enc that takes as input the public key pk and a message m, and outputs a ciphertext c. Since Enc is a randomized algorithm, there will be several valid encryptions of a message m under the key pk; with $\mathsf{Enc}(m, pk)$ we will denote the set of such ciphertexts.
– and a decryption algorithm Dec which is run on input c, sk and outputs m, such that $c \in \mathsf{Enc}(m, pk)$.

We say that $\mathcal{E}$ is semantically secure if for any two messages m_0, m_1 we have that the probability distributions $\mathsf{Enc}(m_0)$ and $\mathsf{Enc}(m_1)$ are computationally indistinguishable.

A (t, n) threshold cryptosystem $\mathcal{TE}$, consists of the following protocols for n players $P_1, \ldots, P_n$.

– A key generation protocol TEnc-Key-Gen that takes as input a security parameter λ, and the parameter t, n, and it outputs a public key pk and a vector of secret keys $(sk_1, \ldots, sk_n)$ where sk_i is private to player P_i. This protocol could be obtained by having a trusted party run Enc-Key-Gen and sharing sk among the players.
– A threshold decryption protocol TDec, which is run on public input a ciphertext c and private input the share sk_i. The output is m, such that $c \in \mathsf{Enc}(m, pk)$.

We point out that threshold variations of Paillier's scheme have been presented in the literature [2, 15, 16, 27]. In order to instantiate our dealerless protocol, we use the scheme from [27] as it includes a dealerless key generation protocol that does not require $n \geq 2t + 1$.

2.3 Independent Trapdoor Commitments

A trapdoor commitment scheme allows a sender to commit to a message with information-theoretic privacy. i.e., given the transcript of the commitment phase

the receiver, even with infinite computing power, cannot guess the committed message better than at random. On the other hand when it comes to opening the message, the sender is only computationally bound to the committed message. Indeed the scheme admits a *trapdoor* whose knowledge allows to open a commitment in any possible way (we will refer to this also as *equivocate* the commitment). This trapdoor should be hard to compute efficiently.

Formally a (non-interactive) trapdoor commitment scheme consists of four algorithms KG, Com, Ver, Equiv with the following properties:

- KG is the key generation algorithm, on input the security parameter it outputs a pair pk, tk where pk is the public key associated with the commitment scheme, and tk is called the *trapdoor*.
- Com is the commitment algorithm. On input pk and a message M it outputs $[C(M), D(M)] = \mathsf{Com}(\mathsf{pk}, M, R)$ where r are the coin tosses. $C(M)$ is the commitment string, while $D(M)$ is the decommitment string which is kept secret until opening time.
- Ver is the verification algorithm. On input C, D and pk it either outputs a message M or $\perp$.
- Equiv is the algorithm that opens a commitment in any possible way given the trapdoor information. It takes as input pk, strings M, R with $[C(M), D(M)] = \mathsf{Com}(\mathsf{pk}, M, R)$, a message $M' \neq M$ and a string T. If $T = \mathsf{tk}$ then Equiv outputs D' such that $\mathsf{Ver}(\mathsf{pk}, C(M), D') = M'$.

We note that if the sender refuses to open a commitment we can set $D = \perp$ and $\mathsf{Ver}(\mathsf{pk}, C, \perp) = \perp$. Trapdoor commitments must satisfy the following properties

Correctness If $[C(M), D(M)] = \mathsf{Com}(\mathsf{pk}, M, R)$ then $\mathsf{Ver}(\mathsf{pk}, C(M), D(M)) = M$.

Information Theoretic Security For every message pair M, M' the distributions $C(M)$ and $C(M')$ are statistically close.

Secure Binding We say that an adversary $\mathcal{A}$ wins if it outputs C, D, D' such that $\mathsf{Ver}(\mathsf{pk}, C, D) = M$, $\mathsf{Ver}(\mathsf{pk}, C, D') = M'$ and $M \neq M'$. We require that for all efficient algorithms $\mathcal{A}$, the probability that $\mathcal{A}$ wins is negligible in the security parameter.

Such a commitment is *non-malleable* [19] if no adversary $\mathcal{A}$, given a commitment C to a messages m, is able to produce another commitment C' such that after seeing the opening of C to m, $\mathcal{A}$ can successfully decommit to a related message m' (this is actually the notion of non-malleability with respect to opening introduced in [17]). We are going to use a related property called *independence* and introduced in [24].

Consider the following scenario: an honest party produces a commitment C and the adversary, after seeing C, will produce another commitment C' (which we to require to be different from C in order to prevent the adversary from simply copying the behavior of the honest party and outputing an identical committed value). At this point the value committed by the adversary should be *fixed*, i.e. no matter how the honest party open his commitment the adversary will always open in a unique way.

The following definition takes into account that the adversary may see and output many commitments ([14]).

Independence For any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ the following probability is negligible in k:

$$Prob\left[\begin{array}{c} \mathsf{pk}, \mathsf{tk} \leftarrow \mathsf{KG}(1^k)\ ;\ m_1, \ldots, m_t \leftarrow \mathcal{M} \\ r_1, \ldots, r_t \leftarrow \{0,1\}^k\ ;\ [c_i, d_i] \leftarrow \mathsf{Com}(\mathsf{pk}, m_i, r_i) \\ (\omega, \hat{c}_1, \ldots, \hat{c}_u) \leftarrow \mathcal{A}_1(\mathsf{pk}, c_1, \ldots, c_t) \text{ with } \hat{c}_j \neq c_i \forall i, j \\ m'_1, \ldots, m'_t \leftarrow \mathcal{M}\ ;\ d'_i \leftarrow \mathsf{Equiv}(\mathsf{pk}, \mathsf{tk}, m_i, r_i, m'_i) \\ (\hat{d_1}, \ldots, hatd_u) \leftarrow \mathcal{A}_2(\mathsf{pk}, \omega, d_1, \ldots, d_t) \\ (\hat{d'_1}, \ldots, \hat{d'_u}) \leftarrow \mathcal{A}_2(\mathsf{pk}, \omega, d'_1, \ldots, d'_t) \\ \exists i : \perp \neq \hat{m}_i = \mathsf{Ver}(\mathsf{pk}, \hat{m}_i, \hat{c}_i, \hat{d_i}) \neq \mathsf{Ver}(\mathsf{pk}, \hat{m}'_i, \hat{c}_i, \hat{d}'_i) = \hat{m}'_i \neq \perp \end{array}\right]$$

In other words even if the honest parties open their commitments in different ways using the trapdoor, the adversary cannot change the way he opens his commitments $\hat{C}_j$ based on the honest parties' opening.

It is possible to prove

Candidate Independent Trapdoor Commitments As shown in [24] independence implies non-malleability. We point out that *all* non-malleable commitments in the literature are also independent one.

The non-malleable commitment schemes in [17, 18] are not suitable for our purpose because they are not "concurrently" secure, in the sense that the security definition holds only for $t = 1$ (i.e. the adversary sees only 1 commitment).

The stronger concurrent security notion of non-malleability for $t > 1$ is achieved by the schemes presented in [14, 21, 32]), and all these schemes can also be proven independent according to the definition presented above. Therefore for the purpose of our threshold DSA scheme, we can use any of the schemes in [14, 21, 32]).

3 The Digital Signature Standard

We define a generic G-DSA signature algorithm as follows. The public parameters include a cyclic group $\mathcal{G}$ of prime order q generated by an element g, a hash function H defined from arbitrary strings into Z_q, and another hash function H' defined from $\mathcal{G}$ to Z_q.

- Secret Key x chosen uniformly at random in Z_q.
- Public Key $y = g^x$ computed in $\mathcal{G}$.
- Signing Algorithm on input an arbitrary message M, we compute $m = H(M) \in Z_q$. Then the signer chooses k uniformly at random in Z_q and computes $R = g^k$ in $\mathcal{G}$ and $r = H'(R) \in Z_q$. Then she computes $s = k^{-1}(m + xr) \bmod q$. The signature on M is the pair (r, s).

– Verification Algorithm On input $M, (r, s)$ and y, the receiver checks that $r, s \in Z_q$ and computes

$$R' = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q} \text{ in } \mathcal{G}$$

and accepts if $H'(r') = r$.

The traditional DSA algorithm is obtained by choosing large primes p, q such that $q|(p-1)$ and setting $\mathcal{G}$ to be the subgroup of Z_p^* of order q. In this case the multiplication operation in $\mathcal{G}$ is multiplication modulo p. The function H' is defined as $H'(R) = R \bmod q$.

The EC-DSA scheme is obtained by choosing $\mathcal{G}$ as a group of points on an elliptic curve of cardinality q. In this case the multiplication operation in $\mathcal{G}$ is the group operation over the curve. The function H' is defined as $H'(R) = R_x \bmod q$ where R_x is the x-coordinate of the point R.

3.1 Threshold DSA

As discussed in Section 2, in a (t, n)-threshold signature scheme the secret key is shared among n servers, in such a way that any t of them has no information about the secret key, while n players can sign a message using a communication protocol that does not require the secret key to be reconstructed at a single server. A scheme is threshold-optimal if exactly $n = t + 1$ honest players can sign.

For the case of DSA, in [22, 23] Gennaro *et al.* present such a non-optimal scheme that requires $n = 2t + 1$ honest players to participate in a signature. In particular this prevents the classical "2-out-of-2" case where the key is split among 2 servers so that both have to cooperate to sign, while 1 has no information about the secret key (in [22, 23] if 1 server has no information about the key, then one would need at least 3 servers to sign). The 2-out-of-2 case is handled by [31] which is the basis of our protocol.

Both schemes are described for the specific case of the DSA scheme, but it is not hard to see that they both work for the generic G-DSA scheme, and therefore also for ECDSA. In the rest of the paper we will use the G-DSA notation.

3.2 The technical issues

The main technical issue in constructing threshold DSA signatures is dealing with the fact that *both* the secret key x and the nonce k have to remain secret. This means that in a threshold scheme, they must be shared in some way among the servers. The protocol in [22] is based on Shamir's secret sharing [39], which means that both x and k are shared using polynomials of degree t. Due to the fact that k and x are multiplied to compute s, the end result is that s will be shared among the players using a polynomial of degree $2t$, which requires $2t + 1$ honest players to be reconstructed.

The protocol in [31] gets around the above problem by using a multiplicative sharing of the secret values in the protocol. This allows an efficient way to multiply k and x without incurring an increase of the number of players required to reconstruct. However it only works for 2 players.

Our first approach was to first extend the techniques in [31] to the case of t-out-of-t players, but that required $O(t)$ round and the use of Paillier's encryption scheme with a modulus $N = O(q^{3t-1})$. Moreover if one wanted to extend that to a t-out-of-n scheme using a combinatorial structure, it would require $O(n^t)$ storage, making it feasible only for small values of n and t. For the sake of comparison, we have included this scheme in Appendix A.

The scheme we present in the next section requires only 6 rounds, constant amount of storage from the players, and uses a Paillier modulus $N > q^8$.

4 Our scheme

In this section, we describe our scheme in three parts. First we describe the initialization phase, in which some common parameters are chosen. Then we describe the key generation protocol, in which the parties jointly generate a DSA key pair $(x, y = g^x)$ with y public and x shared among the players. Finally, we describe the signature generation protocol.

In the following, we assume that if any player does not perform according to the protocol (e.g. by failing a ZK proof, or refusing to open a committed value), then the protocol aborts and stops.

4.1 Initialization phase

In this phase, a common reference string containing the public information pk for an independent trapdoor commitment KG, Com, Ver, Equiv is selected and published. This could be accomplished by a trusted third party, who can be assumed to erase any secret information (i.e. the trapdoor of the commitment) after selection[8].

The common parameters $\mathcal{G}, g, q$ for the DSA scheme are assumed to be known.

4.2 Key generation protocol

Here we describe how the players can jointly generate a DSA key pair $(x, y = g^x)$ with y public and x shared among the players. The idea is to generate a public key E for an additively (mod N) homomorphic encryption scheme E, together with the secret key D in shared form among the players. The value N

[8] Another option is to use a publicly verifiable method that generates the public information, without the trapdoor being known. For example the public parameters in [18] could be generated by using a "random oracle" over some public information (e.g. the hash of the NY Times of a specific day) without anybody knowing the trapdoor (i.e. the discrete log of some group element with respect to a generator).

is chosen to be larger than q^8. Then a value x is generated, and encrypted with E, with the value $\alpha = E(x)$ made public. Note that this is an implicit (t, n) secret sharing of x, since the decryption key of E is shared among the players. We use independent trapdoor commitments KG, Com, Ver, Equiv to enforce the independence of the values contributed by each player to the selection of x (in the following for simplicity we may drop the public key pk and the randomness input when describing the computation of a commitment and write $[C, D] = \mathsf{Com}(m)$)

More specifically, the scheme is described below. We assume that if any commitment opens to $\perp$ or if any of the ZK proofs fails, the protocol terminates without an output.

- The parties run the key generation protocol $\mathsf{TEnc\text{-}Key\text{-}Gen}$ for an additively homomorphic encryption scheme E. If using Paillier's encryption scheme, we can use the threshold version from [27]. The parties run this protocol with $N > q^8$.
- Each player P_i selects a random value $x_i \in Z_q$, computes $y_i = g^{x_i} \in \mathcal{G}$ and $[C_i, D_i] = \mathsf{Com}(y_i)$;
- Each player P_i broadcasts C_i
  - D_i which allows everybody to compute $y_i = \mathsf{Ver}(C_i, D_i)$.
  - $\alpha_i = E(x_i)$;
  - a ZK argument Π_i that states
    - $\exists\ \eta \in [-q^3, q^3]$ such that
    - $g^\eta = y_i$
    - $D(\alpha_i) = \eta$

    If any of the ZK arguments fails, the protocol terminates.
- The players compute $\alpha = \bigoplus_{i=1}^{t+1} \alpha_i$ and $y = \prod_{i=1}^{t+1} y_i$.

The public key for the DSA is set to y. We note that $y = g^x$ and that $\alpha = E(x')$ with $x' = x \bmod q$ since $x' = \sum_{i=1}^{t+1} x_i$ is computed modulo N, but since $N > q^8$, we have that x' is computed actually over the integers.

4.3 Signature Generation

We now describe the signature generation protocol, which is run on input m (the hash of the message M being signed) and the output of the key generation protocol described above. Here too, we assume that if any commitment opens to $\perp$ or if any of the ZK proofs fails, the protocol terminates without an output.

- Round 1

  Each player P_i
  - chooses $\rho_i \in_R Z_q$
  - computes $u_i = E(\rho_i)$ and $v_i = \rho_i \times_E \alpha = E(\rho_i x)$
  - computes $[C_{1,i}, D_{1,i}] = \mathsf{Com}([u_i, v_i])$ and broadcasts $C_{1,i}$
- Round 2

  Each player P_i broadcasts
  - $D_{1,i}$. This allows everybody to compute $[u_i, v_i] = \mathsf{Ver}(C_{1,i}, D_{1,i})$
  - a zero-knowledge argument $\Pi_{(1,i)}$ which states

* $\exists\ \eta \in [-q^3, q^3]$ such that
* $D(u_i) = \eta$
* $D(v_i) = \eta D(E(x))$

Players compute $u = \bigoplus_{i=1}^{t+1} u_i = E(\rho)$ and $v = \bigoplus_{i=1}^{t+1} v_i = E(\rho x)$, where $\rho = \sum_{i=1}^{t+1} \rho_i$ (over the integers)

– Round 3

Each player P_i

- chooses $k_i \in_R Z_q$ and $c_i \in_R [-q^6, q^6]$
- computes $r_i = g^{k_i}$ and $w_i = (k_i \times_E u) +_E E(c_i q) = E(k_i \rho + c_i q)$
- computes $[C_{2,i}, D_{2,i}] = \mathsf{Com}(r_i, w_i)$ and broadcasts $C_{2,i}$

– Round 4

Each player P_i broadcasts

- $D_{2,i}$ which allows everybody to compute $[r_i, w_i] = \mathsf{Ver}(C_{2,i}, D_{2,i})$
- a zero-knowledge argument $\Pi_{(2,i)}$ which states
  * $\exists\ \eta \in [-q^3, q^3]$ such that
  * $g^\eta = r_i$
  * $D(w_i) = \eta D(u) \bmod q$

Players compute $w = \bigoplus_1^{t+1} w_i = E(k\rho + cq)$ where $k = \sum_{i=1}^{t+1} k_i$ and $c = \sum_{i=1}^{t+1} c_i$(over the integers). Players also compute $R = \Pi_1^{t+1} r_i = g^k$ and $r = H'(R) \in Z_q$

– Round 5

- players jointly decrypt w using TDec to learn the value $\eta \in [-q^7, q^7]$ such that $\eta = k\rho \bmod q$ and $\psi = \eta^{-1} \bmod q$
- Each player computes

$$\begin{aligned}\sigma &= \psi \times_E [(m \times_E u) +_E (r \times_E v)] \\ &= \psi \times_E [E(m\rho) +_E E(r\rho x)] \\ &= (k^{-1}\rho^{-1}) \times_E [E(\rho(m + xr))] \\ &= E(k^{-1}(m + xr)) \\ &= E(s)\end{aligned}$$

– Round 6

The players invoke distributed decryption protocol TDec over the ciphertext σ. Let $s = D(\sigma) \bmod q$. The players output (r, s) as the signature for m.

Remark: The size of the modulus N. We note that in order for the protocol to be correct, all the homomorphic operations over the ciphertexts (which are modulo N), must not "conflict" with the operations modulo q of the DSA algorithms. We note that the values encrypted under E are $\sim q^7$. Indeed the ZK proofs guarantee that the values $k, \rho < q^3$. Moreover the "masking" value cq in the decryption of η is at most q^7, so the encrypted values in w_i are never larger than q^8. By choosing $N > q^8$ we guarantee that when we manipulate ciphertexts, all the operations on the plaintexts happen basically over the integers, without taking any modular reduction mod N.

4.4 Zero-knowledge arguments

We now present instantiations for the zero knowledge arguments needed in our protocol, when the underlying encryption scheme being used is Paillier's scheme. While there has been work done systemizing zero knowledge proofs based on the Strong RSA assumption [10, **?**], those works mostly focus on proofs of knowledge. As we do not require our proofs to be proofs of knowledge, we present the design of these proofs ourselves. These argument systems are basically identical to the ones in [31] (more precisely we need to prove simpler statements than the proofs used in [31]).

As in [31] we make use of an auxiliary RSA modulus $\tilde{N}$ which is the product of two safe primes $\tilde{N} = \tilde{P}\tilde{Q}$ and two elements $h_1, h_2 \in Z^*_{\tilde{N}}$ used to construct range commitments via [20].

We refer the reader to [31] for a proof that the protocols described below are (i) statistical zero-knowledge and (ii) sound under the strong RSA assumption on the modulus $\tilde{N}$, which we recall below.

As in [31], the proof that we give is non-interactive. It relies on using a hash function to compute the challenge, e, and it is secure in the Random Oracle Model.

The Proof $\Pi_{1,i}$ For public values c_1, c_2, c_3, we construct a ZK proof $\Pi_{1,i}$ which states

$\exists\ \eta \in [-q^3, q^3]$ such that

- $D(c_1) = \eta D(c_2)$
- $D(c_3) = \eta$

The protocol is as follows. We assume the Prover knows the value $r \in Z^*_N$ used to encrypt η such that $c_3 = (\Gamma)^\eta (r)^N \bmod N^2$.

The prover chooses uniformly at random:

$\alpha \in Z_{q^3}$ $\quad$ $\beta, \in Z^*_N$

$\rho \in Z_{q\tilde{N}}$ $\quad$ $\gamma \in Z_{q^3\tilde{N}}$

The prover computes

$u_1 = (h_1)^\eta (h_2)^\rho \bmod \tilde{N}$ $\quad$ $u_2 = (h_1)^\alpha (h_2)^\gamma \bmod \tilde{N}$

$z = (\Gamma)^\alpha (\beta)^N \bmod N^2$ $\quad$ $v = (c_2)^\alpha \bmod N^2$

$e = \mathtt{hash}(c_1, c_2, c_3, z, u_1, u_2, v)$

$s_1 = e\eta + \alpha$ $\quad$ $s_3 = e\rho + \gamma$

$s_2 = (r)^e \beta \bmod N$

The prover sends all of these values to the Verifier. The Verifier checks that all the values are in the correct range and moreover that the following equations hold

$z = (\Gamma)^{s_1}(s_2)^N(c_3)^{-e} \bmod N^2$ $\quad$ $v = (c_2)^{s_1}(c_1)^{-e} \bmod N^2$
$u_2 = (h_1)^{s_1}(h_2)^{s_3}(u_1)^{-e} \bmod \tilde{N}$ $\quad$ $e = \texttt{hash}(c_1, c_2, c_3, z, u_1, u_2, v)$

The Proof $\Pi_{2,i}$ For public values g, r, w, u we construct a ZK proof $\Pi_{(2,i)}$ which states

$\exists\ \eta_1 \in [-q^3, q^3]$, $\eta_2 \in [-q^8, q^8]$ such that
- $g^{\eta_1} = r$
- $D(w) = \eta_1 D(u) + q\eta_2$

The protocol is as follows. We assume the Prover knows the randomness $r_c \in Z_N^*$ used to encrypt $q\eta_2$ such that $w = u^{\eta_1}\Gamma^{q\eta_2}r_c^N \bmod N^2$. The prover chooses uniformly at random:

$\alpha \in Z_{q^3}$ $\quad$ $\nu \in Z_{q^3\tilde{N}}$
$\beta \in Z_N^*$ $\quad$ $\theta \in Z_{q^8}$
$\gamma \in Z_{q^3\tilde{N}}$ $\quad$ $\tau \in Z_{q^8\tilde{N}}$
$\delta \in Z_{q^3}$ $\quad$ $\rho_1 \in Z_{q\tilde{N}}$
$\mu \in Z_N^*$ $\quad$ $\rho_2 \in Z_{q^6\tilde{N}}$

The prover computes

$z_1 = (h_1)^{\eta_1}(h_2)^{\rho_1} \bmod \tilde{N}$ $\quad$ $u_3 = (h_1)^{\alpha}(h_2)^{\gamma} \bmod \tilde{N}$
$z_2 = (h_1)^{\eta_2}(h_2)^{\rho_2} \bmod \tilde{N}$ $\quad$ $v_1 = (u)^{\alpha}(\Gamma)^{q\theta}(\mu)^N \bmod N^2$
$u_1 = g^{\alpha}$ in $\mathcal{G}$ $\quad$ $v_2 = (h_1)^{\delta}(h_2)^{\nu} \bmod \tilde{N}$
$u_2 = (\Gamma)^{\alpha}(\beta)^N \bmod N^2$ $\quad$ $v_3 = (h_1)^{\theta}(h_2)^{\tau} \bmod \tilde{N}$

$e = \texttt{hash}(g, w, u, z_1, z_2, u_1, u_2, u_3, v_1, v_2, v_3)$

$s_1 = e\eta_1 + \alpha$ $\quad$ $t_1 = (r_c)^e\mu \bmod N$ $\quad$ $t_3 = e\rho_2 + \tau$
$s_2 = e\rho_1 + \gamma$ $\quad$ $t_2 = e\eta_2 + \theta$

The prover sends all of these values to the Verifier. The Verifier checks that all the values are in the correct range and moreover that the following equations hold

$u_1 = (c)^{s_1}(r)^{-e}$ in $\mathcal{G}$ $\quad$ $v_3 = (h_1)^{t_2}(h_2)^{t_3}(z_2)^{-e} \bmod \tilde{N}$
$u_3 = (h_1)^{s_1}(h_2)^{s_2}(z_1)^{-e} \bmod \tilde{N}$ $\quad$ $e = \texttt{hash}(g, w, u, z_1, z_2, u_1, u_2, u_3, v_1, v_2, v_3)$
$v_1 = (u)^{s_1}(\Gamma)^{qt_2}(t_1)^N(w)^{-e} \bmod N^2$

The Proof Π_i For public values g, y, w we construct a ZK proof Π_i which states

$\exists\, \eta \in [-q^3, q^3]$ such that

- $g^\eta = y$
- $D(w) = \eta$

The protocol is as follows. We assume the Prover knows the randomness $r \in Z_N^*$ used to encrypt η such that $w = (\Gamma)^\eta (r)^N \bmod N^2$.

The prover chooses uniformly at random:

$\alpha \in Z_{q^3}$ $\qquad$ $\rho \in Z_{q\tilde{N}}$
$\beta \in Z_N^*$ $\qquad$ $\gamma \in Z_{q^3\tilde{N}}$

The prover computes

$z = h_1^\eta h_2^\rho \bmod \tilde{N}$ $\qquad$ $u_2 = \Gamma^\alpha \beta^N \bmod N^2$
$u_1 = g^\alpha$ in $\mathcal{G}$ $\qquad$ $u_3 = h_1^\alpha h_2^\gamma \bmod \tilde{N}$

$e = \texttt{hash}(g, y, w, z, u_1, u_2, u_3)$

$s_1 = e\eta + \alpha$ $\qquad$ $s_3 = e\rho + \gamma$
$s_2 = (r)^e \beta \bmod N$

The prover sends all of these values to the Verifier. The Verifier checks that all the values are in the correct range and moreover that the following equations hold

$u_1 = (g)^{s_1}(y)^{-e}$ in $\mathcal{G}$ $\qquad$ $u_3 = (h_1)^{s_1}(h_2)^{s_3}(z)^{-e} \bmod \tilde{N}$
$u_2 = (\Gamma)^{s_1}(s_2)^N (w)^{-e} \bmod N^2$ $\qquad$ $e = \texttt{hash}(g, y, w, z, u_1, u_2, u_3)$

The Strong RSA Assumption. Let N be the product of two safe primes, $N = pq$, with $p = 2p' + 1$ and $q = 2q' + 1$ with p', q' primes. With $\phi(N)$ we denote the Euler function of N, i.e. $\phi(N) = (p-1)(q-1) = p'q'$. With Z_N^* we denote the set of integers between 0 and $N-1$ and relatively prime to N.

Let e be an integer relatively prime to $\phi(N)$. The RSA Assumption [38] states that it is infeasible to compute e-roots in Z_N^*. That is, given a random element $s \in_R Z_N^*$ it is hard to find x such that $x^e = s \bmod N$.

The Strong RSA Assumption (introduced in [3]) states that given a random element s in Z_N^* it is hard to find $x, e \neq 1$ such that $x^e = s \bmod N$. The assumption differs from the traditional RSA assumption in that we allow the adversary to freely choose the exponent e for which she will be able to compute e-roots.

We now give formal definitions. Let $SRSA(n)$ be the set of integers N, such that N is the product of two $n/2$-bit safe primes.

Assumption 1 *We say that the Strong RSA Assumption holds, if for all probabilistic polynomial time adversaries $\mathcal{A}$ the following probability*

$$Prob[\ N \leftarrow SRSA(n)\ ;\ s \leftarrow Z_N^*\ :\ \mathcal{A}(N,s) = (x,e)\ \ s.t.\ \ x^e = s \bmod N\ ]$$

is negligible in n.

5 Security Proof

In this section we prove the following

Theorem 1. *Assuming that*

- *The DSA signature scheme is unforgeable;*
- *E is a semantically secure, additively homomorphic encryption scheme;*
- KG, Com, Ver, Equiv *is a independent trapdoor commitment;*
- *the Strong RSA Assunption holds;*

then our threshold DSA scheme in the previous section is unforgeable.

The proof of this theorem will proceed by a traditional simulation argument, in which we show that if there is an adversary $\mathcal{A}$ that forges in the threshold scheme with a significant probability, then we can build a forger $\mathcal{F}$ that forges in the centralized DSA scheme also with a significant probability.

So let's assume that there is an adversary $\mathcal{A}$ that forges in the threshold scheme with probability larger than $\epsilon > \frac{1}{k^c}$ for some constant c.

We assume that the adversary controls players $P_2, \ldots, P_{t+1}$ and that P_1 is the honest player. We point out that because we use concurrently independent commitments (where the adversary can see many commitments from the honest players) the proof also holds if the adversary controls less than t players and we have more than 1 honest player. So the above assumption is without loss of generality.

Because we are assuming a rushing adversary, P_1 always speak first at each round. Our simulator will act on behalf of P_1 and interact with the adversary controlling $P_2, \ldots, P_{t+1}$. Recall how $\mathcal{A}$ works: it first participates in the key generation protocol to generate a public key y for the threshold scheme. Then it requests the group of players to sign several messages $m_1, \ldots, m_\ell$, and the group engages in the signing protocol on those messages. At the end with probability at least ϵ the adversary outputs a message $m \neq m_i$ and a valid signature (r,s) for it under the DSA key y. This probability is taken over the random tape $\tau_{\mathcal{A}}$ of $\mathcal{A}$ and the random tape τ_1 of P_1. If we denote with $\mathcal{A}(\tau_{\mathcal{A}})_{P_1(\tau_1)}$ the output of $\mathcal{A}$ at the end of the experiment described above, we can write

$$Prob_{\tau_1,\tau_{\mathcal{A}}}[\ \mathcal{A}(\tau_{\mathcal{A}})_{P_1(\tau_1)} \text{ is a forgery }\ ] \geq \epsilon$$

We say that an adversary random tape $\tau_{\mathcal{A}}$ is *good* if

$$Prob_{\tau_1}[\ \mathcal{A}(\tau_{\mathcal{A}})_{P_1(\tau_1)} \text{ is a forgery }\ ] \geq \frac{\epsilon}{2}$$

By a standard application of Markov's inequality we know that if $\tau_{\mathcal{A}}$ is chosen uniformly at random, the probability of choosing a good one is at least $\frac{\epsilon}{2}$.

We now turn to building the adversary $\mathcal{F}$ that forges in the centralized scheme. This forger will use $\mathcal{A}$ as a subroutine in a "simulated" version of the threshold scheme: $\mathcal{F}$ will play the role of P_1 while $\mathcal{A}$ will control the other players.

We assume that $\mathcal{F}$ runs the initialization phase in which the public parameters are set. In particular this means that $\mathcal{F}$ knows the trapdoor information tk for the independent trapdoor scheme used in the protocol. This will allow $\mathcal{F}$ to change the opening of its commitments in different ways if necessary. The independence property will guarantee that the adversary $\mathcal{A}$ cannot do that. Also $\mathcal{F}$ will choose a random tape $\tau_{\mathcal{A}}$ for $\mathcal{A}$: we know that with probability at least $\frac{\epsilon}{2}$ it will be a good tape. From now on we assume that $\mathcal{A}$ runs on a good random tape.

$\mathcal{F}$ runs on input a public key y for the centralized DSA scheme, which is chosen according to the uniform distribution in $\mathcal{G}$. The first task for $\mathcal{F}$ is set up an indistinguishable simulation of the key generation protocol to result in the same public key y.

Similarly every time $\mathcal{A}$ requests the signature of a message m_i, the forger $\mathcal{F}$ will receive the real signature (r_i, s_i) from its signature oracle. It will then simulate, in an indistinguishable fashion, an execution of the threshold signature protocol that on input m_i results in the signature (r_i, s_i).

Because these simulations are indistinguishable from the real protocol for $\mathcal{A}$, the adversary will output a forgery with the same probability as in real life. Such a forgery m, r, s is a signature on a message that was never queried by $\mathcal{F}$ to its signature oracle and therefore a valid forgery for $\mathcal{F}$ as well. We now turn to the details of the simulations.

5.1 Simulating the key generation protocol

Recall that in the key generation protocol P_1 first sends C_1, then $\mathcal{A}$ sends the commitments C_i for $i > 1$. Then P_1 decommits y_1 together with the ZK proof α_1, Π_1. Similarly $\mathcal{A}$ decommits y_i together with the ZK proof α_i, Π_i. We denote with $\mathsf{Key\text{-}Gen}(C_i, y_i, \Pi_i)$ the output of the protocol (which can be $\perp$ if the protocol does not terminate successfully).

The simulation $\mathsf{Sim\text{-}Key\text{-}Gen}$ is described below. On input a public key $y = g^x$ for DSA the forger $\mathcal{F}$ plays the role of P_1 as follows

1. The parties run the key generation protocol $\mathsf{TEnc\text{-}Key\text{-}Gen}$ for an additively homomorphic encryption scheme E.
2. Repeat the following steps (by rewinding $\mathcal{A}$) until $\mathcal{A}$ sends valid messages (i.e. a correct decommitment and a correct ZK proof) for $P_2, \ldots, P_{t+1}$
   - $\mathcal{F}$ (as P_1) selects a random value $x_1 \in Z_q$, computes $y_1 = g^{x_1} \in \mathcal{G}$ and $[C_1, D_1] = \mathsf{Com}(y_1)$ and broadcasts C_1. $\mathcal{A}$ broadcast commitments C_i for $i > 1$;
   - Each player P_i broadcasts D_i and α_i, Π_i ($\mathcal{F}$ will follow the protocol instructions).

3. Let y_i the revealed commitment values of each party. $\mathcal{F}$ *rewinds* the adversary to the decommitment step and
   - changes the opening of P_1 to $\hat{D}_1$ so that the committed value revealed is now $\hat{y_1} = y \cdot \prod_{i=2}^{t+1} y_i^{-1}$.
   - broadcasts $\hat{\alpha}_1 = E(0)$ and simulates the ZK proof $\hat{\Pi}_1$
4. The adversary $\mathcal{A}$ will broadcasts $\hat{D}_i, \hat{\alpha}_i, \hat{\Pi}_i$. Let $\hat{y_i}$ be the committed value revealed by $\mathcal{A}$ at this point (this could be $\perp$ if the adversary refused to decommit or the ZK proof fails).
5. The players compute $\alpha = \bigoplus_{i=1}^{t+1} \hat{\alpha}_i$ and $\hat{y} = \prod_{i=1}^{t+1} \hat{y}_i$ (these values are set to $\perp$ if the any of the $\hat{y_i}$ are set to $\perp$ in the previous step).

We now prove a few lemmas about this simulation.

Lemma 1. *The simulation terminates in expected polynomial time and is indistinguishable from the real protocol.*

Proof (of Lemma 1). Since $\mathcal{A}$ is running on a good random tape, we know that the probability over the random choices of $\mathcal{F}$, that $\mathcal{A}$ will correctly decommit is at least $\frac{\epsilon}{2} > \frac{1}{2n^c}$. Therefore we will need to repeat the loop in step (2) only a polynomial number of times in expectation.

The only differences between the real and the simulated views are

- in the simulated one the ciphertext α_1 does not decrypt to the discrete logarithm of y_1 (which in turn implies that α does not decrypt to the discrete logarithm of y).
- P_1 runs a simulated ZK proof instead of a real one.

Since the simulation of the ZK proofs is statistically indistinguishable from the real proof, it is not hard to see that in order to distinguish between the two views one must be able to break the semantic security of the Paillier encryption scheme.

Lemma 2. *For a polynomially large fraction of inputs y, the simulation terminates with output y except with negligible probability.*

Proof (of Lemma 2). First we prove that if the simulation terminates on an output which is not $\perp$, then it terminates with output y except with negligible probability. This is a consequence of the independence property of the commitment scheme. Indeed because the commitment is independent, if $\mathcal{A}$ correctly decommits C_i twice it must do so with the same string, no matter what P_1 decommits too (except with negligible probability). Therefore $\hat{y}_i = y_i$ for $i > 1$ and therefore $\hat{y} = y$.

Then we prove that this happens for a polynomially large fractions of input y. Let $y_{\mathcal{A}} = \prod_{i=2}^{t+1} y_i$, i.e.the contribution of the adversary to the output of the protocol. Note that because of the independence property this value is determined and known to $\mathcal{F}$ by step (3). At that point $\mathcal{F}$ rewinds the adversary and chooses $\hat{y}_1 = y y_{\mathcal{A}}^{-1}$. Since y is uniformly distributed, we have that $\hat{y}_1$ is also uniformly distributed. Because $\mathcal{A}$ is running on a good random tape we know that at this

point there is an $\frac{\epsilon}{2} > \frac{1}{2n^c}$ fraction of $\hat{y}_1$ for which $\mathcal{A}$ will correctly decommit. Since there is a 1-to-1 correspondence between y and $\hat{y}_1$ we can conclude that for a $\frac{\epsilon}{2} > \frac{1}{2n^c}$ of the input y the protocol will successfully terminate.

5.2 Signature generation simulation

After the key generation is over, $\mathcal{F}$ must handle the signature queries issued by the adversary $\mathcal{A}$. When $\mathcal{A}$ requests to sign a message m, our forger $\mathcal{F}$ will engage in a simulation of the threshold signature protocol. During this simulation $\mathcal{F}$ will have access to a signing oracle that produces DSA signatures under the public key y issued earlier to $\mathcal{F}$.

1. Repeat the following steps (by rewinding $\mathcal{A}$) until $\mathcal{A}$ sends valid messages (i.e. a correct decommitment and a correct ZK proof) for $P_2, \ldots, P_{t+1}$. Here $\mathcal{F}$ simply follows the protocol instructions for P_1.
   - Round 1. Each Player P_i computes $[C_{1,i}, D_{1,i}] = \mathsf{Com}(u_i, v_i)$ and broadcasts $C_{1,i}$
   - Round 2. Each Player P_i broadcasts $D_{1,i}$ (which allows to calculate u_i, v_i) and $\Pi_{1,i}$
2. Let $\bar{u} = \bigoplus_{i=2}^{t+1}[(-1) \times_E u_i]$ and $\bar{v} = \bigoplus_{i=2}^{t+1}[(-1) \times_E v_i]$. Repeat the following step until $\mathcal{A}$ sends valid messages (i.e. a correct decommitment and a correct ZK proof) for $P_2, \ldots, P_{t+1}$.
   - $\mathcal{F}$ chooses random values $\rho, \tau \in Z_q$. It rewinds the adversary and changes the opening of P_1 to $\hat{u}_1 = E(\rho) +_E \bar{u}$ and $\hat{v}_1 = E(\tau) +_E \bar{v}$. Let $\hat{u}_i$ and $\hat{v}_i$ be the opening of $\mathcal{A}$.
   - $\mathcal{F}$ simulates the ZK proof $\Pi_{1,i}$.

   Let $\hat{u} = \bigoplus_{i=1}^{t+1} \hat{u}_i$ and $\hat{v} = \bigoplus_{i=1}^{t+1} \hat{v}_i$.
3. Repeat the following steps (by rewinding $\mathcal{A}$) until $\mathcal{A}$ sends valid messages (i.e. a correct decommitment and a correct ZK proof) for $P_2, \ldots, P_{t+1}$. Here $\mathcal{F}$ simply follows the protocol instructions for P_1.
   - Round 3. Each Player P_i computes $[C_{2,i}, D_{2,i}] = \mathsf{Com}(r_i, w_i)$ and broadcasts $C_{2,i}$
   - Round 4. Each Player P_i broadcasts $D_{2,i}$ (which allows to calculate r_i, w_i) and $\Pi_{2,i}$
4. Let $\bar{r} = \prod_{i=2}^{t+1} r_i^{-1}$ and $\bar{w} = \bigoplus_{i=2}^{t+1}[(-1) \times_E w_i]$. Repeat the following step until $\mathcal{A}$ sends valid messages (i.e. a correct decommitment and a correct ZK proof) for $P_2, \ldots, P_{t+1}$.
   - $\mathcal{F}$ queries its signature oracle and receives a signature (r, s) on m. It computes $R = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q} \in \mathcal{G}$ (note that $H'(R) = r \in Z_q$). It finally chooses $\eta \in_R [-q^7, q^7]$ such that $\eta^{-1}(m\rho + r\tau) = s \bmod q$
   - It rewinds the adversary to the previous decommitment step and changes the opening of P_1 to $\hat{r}_1 = R \cdot \bar{r}$ and $\hat{w}_1 = E(\eta) +_E \bar{w}$. Let $\hat{r}_i$ and $\hat{w}_i$ be the openings of $\mathcal{A}$.
   - $\mathcal{F}$ simulates the ZK proof $\Pi_{2,i}$.

   Let $\hat{w} = \bigoplus_{1}^{t+1} \hat{w}_i = E(\eta)$ and $\hat{R} = \Pi_1^{t+1} \hat{r}_i \in \mathcal{G}$ and $\hat{r} = H'(\hat{R}) \in Z_q$.

5. Round 5.
   - players jointly decrypt $\hat{w}$ using TDec to learn the value $\hat{\eta}$ and $\hat{\psi} = \hat{\eta}^{-1} \bmod q$
   - Each player computes

$$\hat{\sigma} = \hat{\psi} \times_E [(m \times_E \hat{u}) +_E (\hat{r} \times_E \hat{v})]$$

6. Round 6. The players invoke distributed decryption protocol TDec over the ciphertext $\hat{\sigma}$ which will result in $\hat{s}$. The players output $(\hat{r}, \hat{s})$ as the signature for m.

Here too, we prove a few lemmas about the simulation.

Lemma 3. *On input m the simulation terminates with output (r, s) a valid signature for m, except with negligible probability.*

Proof (of Lemma 3). Let (r, s) be the signature that $\mathcal{F}$ receives by its signature racle in the last iteration of Step (3) (when the adversary decommits successfully). This is a valid signature for m. We prove that the protocol terminates with output (r, s).

This is a consequence of the independence property of the commitment scheme. Indeed because the commitment is independent, if $\mathcal{A}$ correctly decommits in Step (2) (resp. in Step (4)), its opening must be the same as the opening it presented in Step (1) (resp. in Step (3)) – except with negligible probability. Therefore we have that

$$\hat{u} = E(\rho) \quad \hat{w} = E(\tau) \quad \hat{w} = E(\eta) \quad \hat{\psi} = \eta^{-1} \bmod q \quad \hat{r} = r$$

and

$$\begin{aligned} \hat{\sigma} = \hat{\psi} \times_E [(m \times_E \hat{u}) +_E (\hat{r} \times_E \hat{v})] & \\ &= (\eta^{-1} \bmod q) \times_E [E(m\rho) +_E E(r\tau)] \\ &= E(\eta^{-1}(m\rho + r\tau)) \\ &= E(s) \end{aligned}$$

except with negligible probability.

Therefore when the players jointly decrypt $\hat{\sigma}$ they will recover s.

Lemma 4. *The simulation terminates in expected polynomial time and is indistinguishable from the real protocol.*

Proof (of Lemma 4). Since $\mathcal{A}$ is running on a good random tape, we know that the probability over the random choices of $\mathcal{F}$, that $\mathcal{A}$ will correctly decommit is at least $\frac{\epsilon}{2} > \frac{1}{2k^c}$. Therefore we will need to repeat the loop in steps (1), (2), (3) and (4) only a polynomial number of times in expectation.

The only differences between the real and the simulated views are

– in the simulated view, the plaintexts encrypted in the ciphertexts published by $\mathcal{F}$ do not satisfy the same properties that they would do in the protocol when they were produced by a real player P_1. It is not hard to see that in order to distinguish between the two views one must be able to break the semantic security of the encryption scheme.
– $\mathcal{F}$ runs simulated ZK proofs instead of real ones that would prove those properties. But the simulations are statistically indistinguishable from the real proofs.
– The output of the protocol. In our simulation the output is *always* a correct signature (see Lemma 3) while in the real protocol it might happen that the output is a pair (r, s) which is not a valid signature. This only happens if the adversary is able to fool one of the ZK proofs, but due to the soundness property of the ZK Proofs (which holds under the Strong RSA Assumption) this event happens only with negligible probability, and therefore cannot contribute significantly to distinguish between the two views.
– The distribution of the value η. In the real protocol, η is a fixed value $k\rho$ (which we know is bounded by q^6 at most because of the ZK proofs), masked by a random value in the range of q^7. In our protocol, η is a random value in the range of q^7. It is not hard to see that the two distributions are statistically indistinguishable.

Before we conclude the proof let us point out a major difference in the simulation of the key generation, versus the simulation of the signature generation. In the former we have to accept that it is not possible to generate some public keys y. We can only prove that a sufficiently large fraction of the possible keys can be generated. That's because we have seen that the adversary *can* skew the distribution of the public keys, but not to a sufficiently large extent.

When it comes to signatures, instead, we can always sign messages that the adversary queries. Indeed here too it's true that the adversary can skew the distribution of the signatures (similarly to the way it can skew the distribution of the public keys), but here we are not required to "hit" a specific public key y. Here we are simply required to hit any valid signature for m. By querying the signature oracle several times on the same message (and getting independent signatures on it), the forger is able to hit the specific distribution that the adversary induces on the signatures (that's because there is a sufficiently large fraction of signatures that will be generated by the protocol).

In other words, our simulator perfectly simulates the keys and the signatures that the protocol generates. When it comes to use this simulator to prove unforgeability, the latter is not a problem. The former simply restricts the success of the forger to the keys that are generated by the protocol (not a problem since there is a large fraction of them).

5.3 Finishing up the proof

Proof (of Theorem 1). The forger $\mathcal{F}$ described above produces an indistinguishable view for the adversary $\mathcal{A}$, and therefore, $\mathcal{A}$ will produce a forgery with the

same probability as in real life. The success probability of $\mathcal{F}$ is at least $\frac{\epsilon^3}{8}$. That's because $\mathcal{F}$ has to succeed in choosing a good random tape for $\mathcal{A}$ (this happens with probability larger than $\frac{\epsilon}{2}$) and has to hit a good public key y (this also happens with probability larger than $\frac{\epsilon}{2}$) and finally under those conditions, the adversary $\mathcal{A}$ will output a forgery with probability $\frac{\epsilon}{2}$.

Under the security of the DSA signature scheme, the probability of success of $\mathcal{F}$ must be negligible, which implies that ϵ must also be negligible, contradicting the assumption that $\mathcal{A}$ has a non-negligible probability of forging.

6 Threshold Security for Bitcoin wallets

In this section, we give an overview of Bitcoin, discuss the threat model, and show that deploying our threshold signatures is the best solution to address these threats.

6.1 Bitcoin

Bitcoin is a decentralized digital currency [34]. Bitcoins are owned by *addresses*; an address is simply the hash of a public key. To transfer bitcoins from one address to another, a *transaction* is constructed that specifies one or more input addresses from which the funds are to be debited, and one or more output addresses to which the funds are to be credited. For each input address, the transaction contains a reference to a previous transaction which contained this address as an output address. In order for the transaction to be valid, it must be signed by the private key associated with each input address, and the funds in the referenced transactions must not have already been spent [34, 6].

Each output of a transaction may only be referenced as the input to a single subsequent transaction. It is thus necessary to spend the entire output at once. It is often the case that one only wishes to spend a part of the output that was received in a previous transaction. This is accomplished by means of a *change address* where one lists their own address as an output of the transaction. So, for example, if Alice received 5 bitcoins in a transaction and wants to transfer 3 of them to Bob, she constructs a transaction in which she transfers 3 to Bob's address and the remaining 2 to her own change address.

While it is possible for the sender to include their input address as the change address in the output, the best and recommended practice is to send the change to a newly generated addresses. The motivation for generating new addresses is increased anonymity since it makes it harder to track which addresses are owned by which individuals.

A Bitcoin *wallet* is a software abstraction which seamlessly manages multiple addresses on behalf of a user. Users do not deal with the low level details of their addresses. They just see their total balance, and when they want to transfer bitcoins to another address, they specify the amount to be transferred. The wallet software chooses the input addresses and change addresses and constructs the transaction. New addresses can be generated at any point, and individual

Bitcoin users typically have many addresses. The standard Bitcoin wallet implementation generates a new change address for every transaction.

Separate from change addresses, businesses may wish to maintain multiple addresses in their wallet for other reasons. A common practice is to provide a fresh address every time someone wishes to send bitcoins. This serves two purposes: it allows the business to easily disambiguate between multiple payers (e.g. if Alice and Bob are each paying 1 BTC, by giving a different address to each payer, the business can now track whom it received payment from) and it also increases unlinkability between the business's various transactions.

Signed transactions are broadcast to the Bitcoin peer-to-peer network. They are validated by *miners* who group transactions together into *blocks*. Miners participate in a distributed consensus protocol that collects these blocks into an append-only global log called the *block chain*.

Our treatment of transactions thus far has described what a *typical* Bitcoin transaction looks like. However, Bitcoin allows for far more complex transactions. Every transaction contains a *script* that specifies how the transferred funds may be redeemed. For a typical transaction, the script specifies that one who wants to spend the bitcoins must present a public key that when hashed yields the output address, and they must sign the new transaction with the corresponding private key. A transaction can include a script that specifies complex series of rules that need to be enforced in order for the bitcoins to be spent.

While the original Bitcoin paper does not specify the signature algorithm to be used, the current implementation uses the Elliptic Curve Digital Signature Algorithm (ECDSA) over the secp256k1 curve [6–8].

6.2 Threat model

To classify the problems, we distinguish between internal and external threats as well as between hot and cold wallets. While the term wallet is generally used loosely to refer to a software abstraction (as described in the previous section), we will use the term in the rest of the paper in a more precise sense.

Definition 4 (wallet). *A collection of addresses with the same security policy together with a software program or protocol that allows spending from those addresses in accordance with that policy.*

"Security policy" encompasses the ownership or access-control list and the conditions under which bitcoins in the wallet may be spent.

The terms *hot wallet* and *cold wallet* derive from the more general terms *hot storage*, meaning online storage, and *cold storage*, meaning offline storage. A hot wallet is a Bitcoin wallet for which the private keys are stored on a network-connected machine (i.e. in hot storage). By contrast, for a cold wallet the private keys are stored offline.

Definition 5 (Hot wallet/Cold wallet). *A* hot wallet *is a wallet from which bitcoins can be spent without accessing cold storage. Conversely, a* cold wallet *is a wallet from which bitcoins cannot be spent without accessing cold storage.*

Note that these new definitions refer to the desired effect, not the method of achieving it. The desired effect of a business that maintains a hot wallet is the ability to spend bitcoins online without having to access cold storage.

Adversary	Hot wallet	Cold wallet
Insider	Vulnerable by default; our methods are necessary	Reduces to physical security by default; our methods can help
External (network)	Reduces to network security by default; our methods can help	Safe

Table 1. Taxonomy of threats

Table 1 shows four types of possible threats to Bitcoin wallets. Securing a cold wallet is a physical security problem. While a network adversary is unable to get to a cold wallet, traditional physical security measures can be used to protect it from insiders — for example, private keys printed on paper and stored in a locked safe with video surveillance.

In addition, our methods may be used to supplement physical security measures. Instead of storing the key in a single location, the business can store shares of the key in different locations. The adversary will thus have to compromise security in multiple locations in order to recover the key. Indeed, this is one use case where Bitcoin companies are eager to implement our threshold signature scheme. In private discussions with one of the most prominent Bitcoin exchanges, we were told that they use Shamir secret sharing to secure their cold storage. Of course, this requires them to reconstruct their key in order to access their cold wallet, and they expressed interest in moving over to our scheme instead.

Protecting hot wallets from external attackers is a network security problem; if the network were completely secure, then this would not be an issue. We can use threshold signatures to reduce our reliance on network security. Protecting hot wallets from internal attackers is the most pressing problem. Our central claim is that the level of insecurity of this threat category has no parallels in traditional finance or network security, necessitating Bitcoin-specific solutions.

6.3 Comparison with multisignature approach

While most Bitcoin transactions are spent with a single signature, as we mentioned, Bitcoin in fact specifies a script written in a stack-based programming language which defines the conditions under which a transaction may be redeemed. This scripting language includes support for *multisignature* scripts [1] which require at least t of n specified ECDSA public keys to provide a signature on the redeeming transaction. By default, multisignature transactions are cur-

rently only relayed with $n \leq 3$ keys, but may specify up to an absolute limit of $n = 20$.

Another feature of Bitcoin, *pay-to-script-hash*, enables payment to an address that is the hash of a script. When this is used, senders specify a script hash, and the exact script is provided by the recipient when funds are redeemed. This enables multisignature transactions without the sender knowing the access control policy at the time of sending. A quirk of pay-to-script hash is that the $n \leq 3$ restriction is removed from t-out-of-n multisignature transactions. However, due to a hardcoded limit on the overall size of a hashed script, the recipients are still limited to $n \leq 15$.

Advantages of multisignatures. Multisignature transactions have one clear benefit over using threshold signatures in that they can be signed independently by each participant in a non-interactive manner, whereas the ECDSA threshold signature protocol requires multiple rounds of interaction. Another potential benefit is that the redeeming transaction provides a public record of exactly which t of n keys were used to redeem the transaction, which can help the company keep records for who authorized a given transaction (though this information is also leaked publicly).

Advantages of threshold signatures. We argue that threshold signatures offer fundamental advantages stemming from the fact that in the multisignature approach, the access-control policy is encoded in the transaction and eventually publicly revealed:

Flexibility. Threshold signatures are more flexible than multisignatures in the access policies that they permit as well as in the ability to modify the access policies.

The policies realizable with multisignatures are very limited in practice since only transactions with $n \leq 3$ are relayed by default. With threshold signatures, you can use t and n that are effectively unbounded.

Threshold signatures also allow more flexibility for making changes to the access control policy. If a business using multisignature transactions wants to make any modification to its access control policy, such as adding or removing an employee from those with transaction approval power, this requires a new script and thus a new address. This prevents businesses wishing to transact in Bitcoin from using a long-term static address as it requires moving funds to a new address with each policy update. For some business practices, the ability to have a static address is fundamental. As an example, consider an organization that prints promotional materials with a donation address on it. Multisignatures would not allow them to change the access control policy while keeping that address.

With threshold signatures, the policy is encoded not in the address but in the shares. In our scheme, the share is the encrypted DSA key together with the key share of the underlying homomorphic encryption scheme. To change the policy, the business would just need to re-deal the shares according to the new

policy. Businesses can still use a static address for a receivable account and can maintain the address even if the access control policy changes.

More generally, it is impossible to add multisignature security to an existing address since the two types of addresses are syntactically distinct. The only way to attain multi-factor security is to create a new multisignature address, and transfer the bitcoins to this new address. Threshold signatures, on the other hand, allow one to split up the key of an existing address.

Anonymity. While Bitcoin allows users to be pseudonymous, it does not provide any anonymity guarantees. Indeed, it has been shown that it is not difficult to link various addresses belonging to a single user [33]. Moreover, because the entire transaction log is public, once an address has been associated with a real world identity, one can immediately view every other transaction associated with that address.

Because of Bitcoin's inherent lack of anonymity, various techniques have been developed to provide additional anonymity for Bitcoin users. Three of the most prominent techniques are Mixcoin [9], CoinJoin [5], and the use of change addresses. We show now that none of these techniques are compatible with multisignatures, while they all work as intended with threshold signatures.

As we mentioned in Section 6.1, for purposes of increasing anonymity, the general practice is to use newly generated change addresses which cannot easily be linked to the input addresses [33]. With multisignature transactions, unlinkable change addresses are much harder to achieve. Suppose Alice uses multisignature-based security and makes a purchase. Then the spending address(es) and change address will all have the same t-of-n access control structure, whereas the destination address most likely will not. This allows easily linking Alice's input and output addresses. With threshold signatures, on the other hand, change addresses will be unlinkable when sending funds to any regular (single-key) address or other threshold address (though not when interacting with multisignature addresses or other script hash addresses). In particular, change addresses will provide the exact same benefits with threshold signatures as they do with a single-key address.

Mixcoin and CoinJoin are both based on the technique of mixing, or shuffling the inputs amongst multiple users. Both protocols proceed in independent rounds. During a single Mixcoin round, each user sends a fixed amount of coins to a mixing party which sends the same amount of coins back to a fresh address provided by that user. CoinJoin is also based on the mixing idea but instead of having a centralized mixing party, users combine their inputs and outputs into a single joint transaction that they all sign. Once coins have been mixed with either protocol, it becomes nearly impossible to identify the mapping between input and output addresses.

Consider what happens, however, when one tries to use either Mixcoin or CoinJoin with multisignature addresses. Both of these protocols rely on the fact that all of the input and output addresses are structurally identical and that there is an abundance of such addresses. In order to maintain multisignature security, both the input and output addresses will have to be multisignature

addresses. Moreover, they will have to have the same access structure (i.e. the same t and n). Multisignature addresses cannot be mixed together with regular addresses as it is trivial to link an input address with an output address by just examining the access structure. Moreover, it is highly unlikely that there will be a sufficient number of addresses with a given access structure that are interested in mixing to facilitate mixing each type of address on its own.[9]

Multisignatures also cause a loss of anonymity since the access structure is published on the block chain. When a business presents its script to spend a transaction, its internal access control policy is exposed to the world. Many companies will want confidentiality as to the internal controls that they enforce. Threshold-signed transactions are completely indistinguishable from regular transactions. Not only do they not leak the details of the access-control policy, they do not reveal that access control is being used at all.

Scalability. With multisignature transactions, the size of transactions grows linearly with the access policy as all of the valid signing keys are included in the redeeming script (as well as the sending script, for non-script hash transactions). In addition to the hard limits which Bitcoin enforces ($n \leq 15$ for script hash transactions and $n \leq 20$ in general), this means that more complex access control policies will lead to larger transactions. The implications of this are twofold: firstly these transactions will be subject to increased transaction fees, and secondly they will lead to additional bloat on the block chain.

As threshold signature transactions are indistinguishable from ordinary transactions, they will be no larger than ordinary transactions no matter how complex the underlying access policy is. Thus, they will not require increased fees or generate additional data which must be globally broadcast.

7 Implementation and evaluation

In this section, we describe the design, creation, and evaluation of our implementaton. We describe two different parts of our implementation. Firstly, we implemented our threshold DSA protocol, and we have included the code with this submission. Secondly, we also implemented a 2-factor wallet that can be used by individuals to store their Bitcoins. For the two factor wallet, our scheme was not necessary as Mackenzie and Reiter's scheme already covered the 2-of-2 case. Nevertheless, the system design of our app was non trivial and we describe it here.

[9] One might be tempted to suggest that funds be temporarily transferred to a single signature address for mixing. This is problematic for two reasons, however. Firstly, transferring to a single signature address introduces a single point of failure as the bitcoins can be stolen during this period if the key is compromised. Moreover, one can do second-order analysis and still link the input and output addresses by examining the access structure of the multisignature addresses that transferred bitcoins to the input address and received bitcoins from the output address.

7.1 Our Protocol

We implemented and included the code for a prototype implementation of our protocol. We implemented it in Java, and began with the Java implementation of threshold Paillier in [36]. This implementation is based on the threshold Paillier scheme in [15], and does not support dealerless Paillier.

Our implementations was straightforward; we didn't include any user interface as the idea for the implementation is to be included in Bitcoin companies server code.

Out implementation was extremely efficient. Without verifying other player's zero knowledge proofs, the protocol took under 2 milliseconds to complete. The zero knowledge proofs added an additional 10 seconds per player to verify. The result is that the protocol is extremely efficient.

7.2 Two Factor Wallet

We can extend the principles of dual control to the security of an individual's wallet — here we split control between different the user's personal devices. The private key is not stored on any machine nor is it ever reconstructed during signature generation.

To protect against theft, Alice distributes 2-out-of-2 shares of her private key among two devices that she owns, say her computer and smartphone. When Alice initiates a Bitcoin transaction from her computer, a prompt containing the transaction details will appear on her smartphone via her wallet app. If she confirms, the two devices will sign the transaction using the threshold scheme and broadcast it. We stress that at no point was the key reconstructed on either device; on its own, neither device contains enough information to create a signature. An attacker will have to compromise both her computer and her smartphone to steal her bitcoins.

A Bitcoin wallet with two-factor security is arguably more secure than cash, especially with appropriate backup and recovery options. We can further improve security by generalizing to multi-factor security, but given the usability drawbacks it is not clear if this will be useful in practice.

7.3 Design decisions

Our goal was to create pair of applications, one for a desktop computer and one for a smartphone, which would together form a easy to use wallet. We chose Java because of its cross-platform nature and the availability of many useful libraries. We wanted our code to be easily incorporated into other wallets, so we decided to make our code part of BitcoinJ, the most commonly used Java Bitcoin Library. As a result, we used BouncyCastle, the crypto library used by BitcoinJ, to implement our crypto code.

On the desktop, we created a modified version of the MultiBit wallet software since it is Java based and open source. On the phone (Android) we wrote a simple application from scratch since we required very little user interface.

The options for communication between the two devices are Bluetooth, WiFi, or a centralized server. We ruled out the latter since direct communication is faster, simpler, and more privacy-preserving. Our experience with Bluetooth on Android taught us that it it fairly unreliable, so we settled on WiFi communication. For device discovery (to initiate the communication) we used DNS Service Discovery, (DNS-SD), a system that uses DNS messaging to advertise services on a network. Once the phone and desktop had discovered each other, they used TLS in order to establish secure communication.

To initiate a secure connection we need an out-of-band exchange of key material (since there is no PKI); the method with the best usability-security trade-off seems to be using the phone's camera to capture a 2D-barcode on the desktop. We used the ZXIng barcode library. It can both create and read bar codes so we were able to use it on both devices.

7.4 Security Model

The desktop acts as a trusted dealer when distributing the phone's key share. Although there is some risk in using a trusted dealer, it can be alleviated by booting off a live disk image. Since the initialization phase requires no internet connection, this eliminates the danger of malware as long as the disk is trusted. The desktop transfers the key share and public key to the phone which completes the initialization. The desktop then deletes all record of the phone's key share.

After this point all future communication occurs over TLS (with self-signed client and server certificates) ensuring a completely secure and trusted connection. Although only authenticated messages are required by the threshold signature protocol (since it leaks no confidential information), using TLS prevents any denial of service attacks against the desktop.

7.5 Two-factor application protocol

- Initialization
  - Desktop: Create wallet and display QR code with Public Certificate and one-time-password
  - Phone: Scan QR code and initiate TLS connection using Public Certificate.
  - Phone: Authenticate using one-time-password
  - Phone: Send over public certificate and receive key share
- Transaction
  - Desktop: Create transaction
  - Desktop: Create TLS server socket and wait for phone to connect
  - Phone: Connect to desktop using TLS with client side authentication
  - Phone: Give user choice to approve transaction. Continue if the user approves
  - Desktop: Initiate threshold protocol
  - Phone: Participate in threshold protocol
  - Desktop: Complete transaction with produced signature and add to blockchain

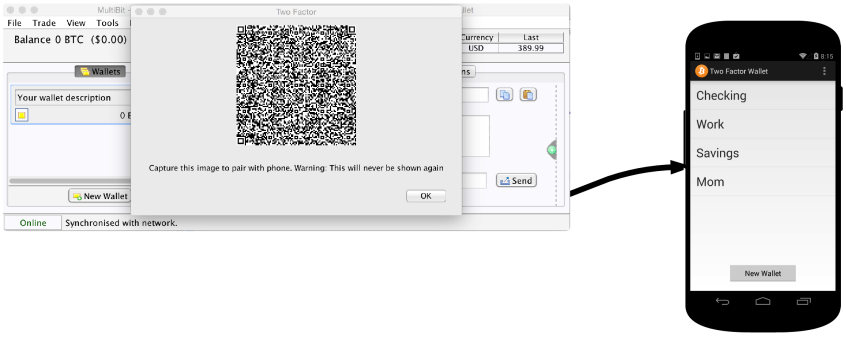

Fig. 1. Initialization Protocol

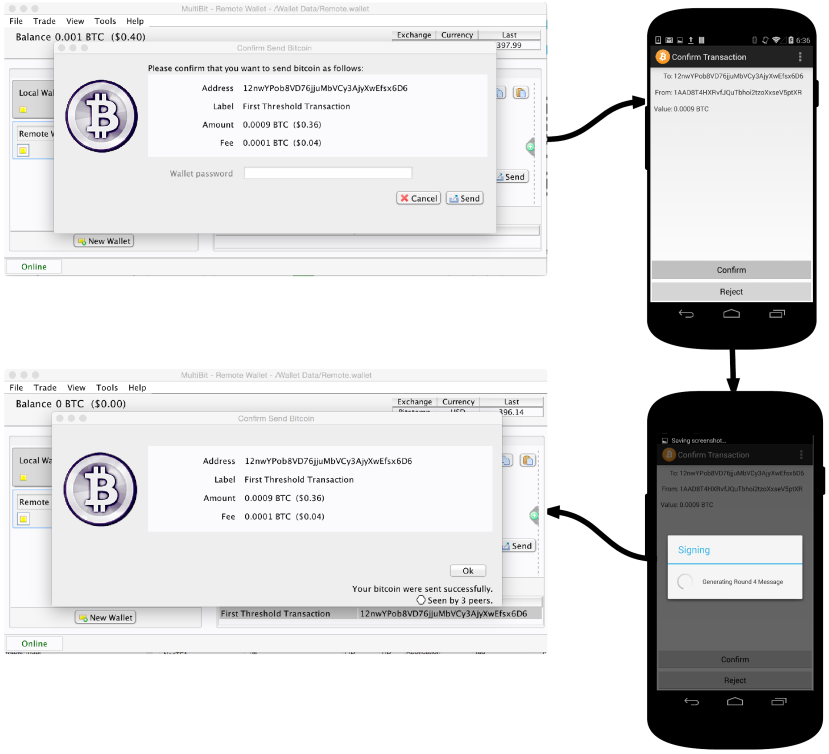

Fig. 2. Transaction Protocol

	Time (Seconds)
Round 1 (Computer)	0.26
Round 2 (Phone)	0.36
Round 3 (Computer)	0.58
Round 4 (Phone)	11.04
Total	13.26

Table 2. This table demonstrates the per round running time of the threshold wallet app as recorded directly on the devices. The majority of time is spent in round 4. During this round 89% is spent creating and verifying zero knowledge proofs. The discrepancy between total time and the sum of the rounds is due to the computer verifying the phones final zero knowledge proof.

7.6 Usage

When a new wallet is created in MultiBit, a QR code is displayed. The Android application scans the QR code which contains a self-signed certificate for the desktop and a one-time-password. The phone then initiates a TLS connection with the desktop using the certificate. The phone authenticates itself using the one-time-password and then sends its own self-signed certificate so that TLS client authentication can be enabled on future connections. The desktop then sends the phone's keyshare and deletes it from memory.

When MultiBit tries to sign a transaction, a server is started and a DNS-SD service is registered to advertise the server. While the phone application runs, it looks for this service and tries to initiate a TLS connection with the server. If it succeeds, the desktop sends the transaction information along with the wallet public key to the phone. If the phone has a keyshare for the public key, it presents the user with the transaction information along with the ability to allow or cancel the transaction. If the user chooses to allow it, the threshold scheme is run to produce a signature on the desktop. Finally the desktop broadcasts the signed transaction to the Bitcoin peer-to-peer network.

We transferred a small amount of bitcoin to our specially created wallet and then spent it by threshold-signing a transaction. Our threshold-signed transaction can be viewed in the block chain.[10]

8 Conclusion

In this paper, we presented the first threshold-optimal signature scheme for DSA. We proved its security, implemented it, and evaluated it. Our scheme is quite efficient, and our implementation confirm that this scheme is ready to be used. Indeed, many Bitcoin companies have expressed great interest in our scheme as

[10] Full details of this transaction can be viewed online at

https://blockchain.info/tx/cf5344b625fe87efa351aadf0
bd542ec437c327b7c29e52245d3b41cea3e205b

it provides a much needed solution to Bitcoin's security problem. We have open sourced our two-factor app, and are open-sourcing our general (t, n) signature code as well in order that companies can actually benefit from our results and begin to use them immediately.

References

1. G. Andresen, "Github: Shared Wallets Design," https://gist.github.com/gavinandresen/4039433, accessed: 2014-03-20.
2. O. Baudron, P.-A. Fouque, D. Pointcheval, G. Poupard and J. Stern. *Practical Multi-Candidate Election System.* PODC'01
3. N. Barić, and B. Pfitzmann. *Collision-free accumulators and Fail-stop signature schemes without trees.* Proc. of EUROCRYPT'97 (LNCS 1233), pp.480–494, Springer 1997.
4. Bitcoin Forum member dree12, "List of Bitcoin Heists," https://bitcointalk.org/index.php?topic=83794.0, 2013.
5. Bitcoin Forum member gmaxwell, "List of Bitcoin Heists," https://bitcointalk.org/index.php?topic=279249.0, 2013.
6. "Bitcoin wiki: Transactions," https://en.bitcoin.it/wiki/Transactions, accessed: 2014-02-11.
7. "Bitcoin wiki: Elliptic Curve Digital Signature Algorithm," https://en.bitcoin.it/wiki/Elliptic_Curve_Digital_Signature_Algorithm, accessed: 2014-02-11.
8. "Bitcoin wiki: Elliptic Curve Digital Signature Algorithm," https://en.bitcoin.it/w/index.php?title=Secp256k1&oldid=51490, accessed: 2014-02-11.
9. J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten :Mixcoin: Anonymity for bitcoin with accountable mixes,' in *Financial Cryptography and Data Security.* Springer, 2014, pp. 486–504.
10. J. Camenisch, A. Kiayias, and M. Yung: On the portability of generalized schnorr proofs. In: Advances in Cryptology-EUROCRYPT 2009, pp. 425–442. Springer (2009)
11. J. Camenisch, S. Krenn, and V. Shoup: A framework for practical universally composable zero-knowledge protocols. In: Advances in Cryptology–ASIACRYPT 2011, pp. 449–467. Springer (2011)
12. R. Canetti. *Universally Composable Security: A new paradigm for cryptographic protocols.* Proc. of 42^{nd} IEEE Symp. on Foundations of Computer Science (FOCS'01), pp.136–145, 2001.
13. R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk and T. Rabin: Adaptive Security for Threshold Cryptosystems. CRYPTO 1999, LNCS Vol.1666, pp 98-115
14. I. Damgård, J. Groth. *Non-interactive and reusable non-malleable commitment schemes.* Proc. of 35^{th} ACM Symp. on Theory of Computing (STOC'03), pp.426-437, 2003.
15. I. Damgård and M. Jurik. *A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System.* PKC'01, LNCS Vol.1992, pp.119–136
16. I. Damgård, M. Koprowski: Practical Threshold RSA Signatures without a Trusted Dealer. EUROCRYPT 2001: LNCS Vol.2045, pp. 152-165

17. G. Di Crescenzo, Y. Ishai, R. Ostrovsky. Non-Interactive and Non-Malleable Commitment. Proc. of 30^{th} ACM Symp. on Theory of Computing (STOC'98), pp.141–150, 1998.
18. G.Di Crescenzo, J. Katz, R. Ostrovsky, A. Smith. *Efficient and Non-interactive Non-malleable Commitment.* Proc. of EUROCRYPT 2001, Springer LNCS 2045, pp.40-59.
19. D. Dolev, C. Dwork and M. Naor. *Non-malleable Cryptography.* SIAM J. Comp. 30(2):391–437, 200.
20. E. Fujisaki, T. Okamoto: Statistical Zero Knowledge Protocols to Prove Modular Polynomial Relations. CRYPTO 1997: LNCS Vol.1294, pp.16-30
21. R. Gennaro. *Multi-trapdoor Commitments and Their Applications to Proofs of Knowledge Secure Under Concurrent Man-in-the-Middle Attacks.* Proc. of CRYPTO'04, Springer LNCS 3152, pp.220–236.
22. R. Gennaro, S. Jarecki, H. Krawczyk and T. Rabin. *Threshold DSS Signatures.* EUROCRYPT'96, LNCS Vol.1070, pp. 354–371.
23. R. Gennaro, S. Jarecki, H. Krawczyk and T. Rabin. *Secure Distributed Key Generation For Discrete Log Based Cryptosystems.* EUROCRYPT'99, LNCS Vol.1592, pp. 295–310.
24. R. Gennaro and S. Micali. *Independent Zero-Knowledge Sets.* ICALP 2006, LNCS vol.4052, pp. 34–45.
25. S. Goldwasser, S. Micali, and R.L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, April 1988.
26. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *SIAM. J. Computing*, 18(1):186–208, February 1989.
27. C. Hazay, G.L. Mikkelsen, T. Rabin, T. Toft. and A.A. Nicolosi: Efficient RSA key generation and threshold Paillier in the two-party setting.
28. S. Jarecki, A. Lysyanskaya. Adaptively Secure Threshold Cryptography: Introducing Concurrency, Removing Erasures. EUROCRYPT 2000: LNCS Vol.1807, pp.221-242
29. D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.
30. Kaspersky Labs, "Financial cyber threats in 2013. Part 2: malware," http://securelist.com/analysis/kaspersky-security-bulletin/59414/financial-cyber-threats-in-2013-part-2-malware/, 2013.
31. P. MacKenzie and M. Reiter. *Two-party Generation of DSA Signatures.* Int. J. Inf. Secur. (2004)
32. P. MacKenzie and K. Yang. *On Simulation-Sound Commitments.* Proc. of EUROCRYPT'04, Springer LNCS 3027, pp.382-400.
33. S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of bitcoins: characterizing payments among men with no names," in *Proceedings of the 2013 conference on Internet measurement conference.* ACM, 2013, pp. 127–140.
34. S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Consulted*, vol. 1, p. 2012, 2008.
35. P. Paillier. *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes.* EUROCRYPT'99, LNCS Vol.1592, pp. 223-238
36. *Paillier Threshold Encryption Toolbox* http://cs.utdallas.edu/dspl/cgi-bin/pailliertoolbox/manual.pdf

37. T. Pedersen. Distributed provers with applications to undeniable signatures. In D. Davies, editor, *Advances in Cryptology–EUROCRYPT'91*, Lecture Notes in Computer Science Vol. 547, Springer-Verlag, 1991.
38. R. Rivest, A. Shamir and L. Adelman. A Method for Obtaining Digital Signature and Public Key Cryptosystems. *Comm. of ACM*, 21 (1978), pp. 120–126
39. A. Shamir. How to Share a Secret. *Communications of the ACM*, 22:612–613, 1979.

A Naive extention of [31]

We began this work by looking at the two party scheme of Mackenzie and Reiter [31] and seeing if it could be extended to the multiparty case. We extended their scheme, and even implemented it, but the results were quite inefficient. Both the computation required as well as the storage required for this (t, n) threshold signature scheme grew exponentially with the number of players. We present the scheme here for reference and comparison.

A.1 Our naive scheme

Mackenzie and Reiter use multiplicative secret sharing which makes multiplication and inversion of secrets easy. Addition of secrets is now more difficult, and to get around this, they use an additively homomorphic encryption scheme. Their scheme is specifically for the two party case, and one of the two parties has a decryption key for the additively homomorphic scheme. This party uses its secret share to compute a partial signature, encrypts the partial signature, and sends the resulting encrypted values to the second party. The second party cannot learn the value of the encrypted partial signature, but it uses its share of the key to contribute its portion of the signature to the ciphertext (as the scheme is additively homomorphic), and then sends the resulting ciphertext back to the first party. The first party then decrypts the ciphertext to reveal the signature. The scheme also incorporates zero knowledge proofs to prove that each party is following the protocol and that the encrypted values that they produce are consistent with well-formed shares (i.e. it is secure against malicious parties).

We generalize Mackenzie and Reiter's scheme to the t-of-t case. The intuitive idea is that t parties pass around the ciphertext and do computations on it with their share, and also construct zero knowledge proofs that their values are consistent. As in Mackenzie and Reiter's scheme, the end result is a ciphertext which is an encryption of the signature. However, whereas in the two party case, one of the parties held the decryption key and can single-handedly decrypt the final signature, in the t party case, the homomorphic decryption key is itself distributed among the parties such that all of them have to cooperate to decrypt the key.

Our protocol proceeds in $3t - 2$ rounds, after which the parties have an encrypted signature which they can then jointly decrypt.

Our protocol works for t-out-of-t case, and we use standard combinatorial structures we show how to use the t-out-of-t scheme to build a t-out-of-n scheme.

Assumptions and Setup. We assume that there are t participants $P_1, \ldots, P_t$ initialized as follows:

- Participant P_i holds a value $x_i \in Z_q$ chosen uniformly at random. The secret key is $x = \Pi_i x_i \bmod q$ and the public key is $y = G^x$ in $\mathcal{G}$. We assume the values $y_i = G^{x_i}$ are public.
- There is a separate public key additively homomorphic encryption scheme E, whose secret key D is shared in a t-out-of-t fashion among the participants. The encryption scheme is homomorphic modulo a large integer N: i.e. given $\alpha = E(a)$ and $\beta = E(b)$, where $a, b \in Z_N$, there is an efficiently computable operation $+_E$ over the ciphertext space such that

  $$\alpha +_E \beta = E(a + b \bmod N)$$

  Note that if x is an integer, given $\alpha = E(a)$ we can also compute $E(xa \bmod N)$ efficiently. We refer to this operation as $x \times_E \alpha$.
  We denote the message space of E by $\mathcal{M}_E$ and the ciphertext space by $\mathcal{C}_E$. We will choose N large enough so that operations modulo N will not "wrap around" and will be consistent to doing them over the integers (that's because we are interested in really doing the operations modulo q, the order of the group). This requires $N > q^{3t+3}$. [11]
- The participants are associated to signature public keys. We assume that they sign every message. In the following the signature is implicitly contained in the messages and verified by each participant upon receipt of a signed message.[12]

Mackenzie Reiter extension Threshold signature protocol. The protocol proceeds in rounds, where each player receives some input, performs some computation, and then passes along the output of this computation. There are n players, $P_1, \ldots, P_n$. In our protocol, players $P_2, \ldots, P_{n-1}$ have completely symmetric roles. That is, they all receive inputs of identical form from the previous player, run the same algorithm, and pass along the message to the next player. However, the computation done by P_1 and P_n is not identical.

We stress, however, that while from a computational perspective not all players have the same role, from a security perspective, all players are identically secure in the same threat model. No player is privileged or trusted in any manner. Consequently, from a security perspective it makes absolutely no difference how the players are numbered and which players are given the roles of P_1 and P_n.

- Round 1
  On input the message M, participant P_1
  - chooses $k_1 \in_R Z_q$ and computes $z_1 = k_1^{-1} \bmod q$

[11] Contrast this with our main scheme in which N is $O(q^8)$ regardless of the number of players.

[12] In our protocol participant P_i will forward to P_j something he received from P_ℓ. By verifying P_ℓ's signature on the forwarded message, P_j is guaranteed of its authenticity.

- computes $\alpha_1 = E(z_1)$ and $\beta_1 = E(x_1 z_1 \bmod q)$
- sets $\hat{\alpha_1} = \hat{\beta}_1 = \perp$
- sends $M, \alpha_1, \beta_1, \hat{\alpha_1}, \hat{\beta}_1$ to P_2

– Rounds 2 to $t-1$
At round $i = 2, \ldots, t-1$, on input the message $M, \alpha_1, \ldots, \alpha_{i-1}, \beta_1, \ldots, \beta_{i-1}$, $\hat{\alpha_1}, \ldots, \hat{\alpha_{i-1}}, \hat{\beta}_1, \ldots, \hat{\beta_{i-1}}$, participant P_i
- abort if $\alpha_1, \ldots, \alpha_{i-1}, \beta_1, \ldots, \beta_{i-1}, \hat{\alpha}_1, \ldots, \hat{\alpha_{i-1}}, \hat{\beta}_1, \ldots, \hat{\beta_{i-1}} \notin \mathcal{C}_E$
- chooses $k_i \in_R Z_q$ and computes $z_i = k_i^{-1} \bmod q$
- computes $\alpha_i = z_i \times_E \alpha_{i-1}$ and $\beta_i = (x_i z_i \bmod q) \times_E \beta_{i-1}$
- computes $\hat{\alpha_i} = E(z_i)$ and $\hat{\beta}_i = E(x_i z_i \bmod q)$
- sends $M, \alpha_1, \ldots, \alpha_i, \beta_1, \ldots, \beta_i, \hat{\alpha_1}, \ldots, \hat{\alpha_i}, \hat{\beta}_1, \ldots, \hat{\beta}_i$ to P_{i+1}

– Round t
On input the message $M, \alpha_1, \ldots, \alpha_{t-1}, \beta_1, \ldots, \beta_{t-1}, \hat{\alpha_1}, \ldots, \hat{\alpha_{t-1}}, \hat{\beta}_1, \ldots, \hat{\beta_{t-1}}$, participant P_t
- abort if $\alpha_1, \ldots, \alpha_{t-1}, \beta_1, \ldots, \beta_{t-1}, \hat{\alpha_1}, \ldots, \hat{\alpha_{t-1}}, \hat{\beta}_1, \ldots, \hat{\beta_{t-1}} \notin \mathcal{C}_E$
- chooses $k_t \in_R Z_q$ and computes $z_t = k_t^{-1} \bmod q$
- computes $R_t = G^{k_t}$ in $\mathcal{G}$
- sends R_t to P_{t-1}

– Rounds $t+1$ to $2t-2$
At round $t+i$ for $i = 1, \ldots, t-2$, on input the message $R_t, \ldots, R_{t-i+1}$, participant P_{t-i}
- computes $R_{t-i} = R_{t-i+1}^{k_{t-i}}$ in $\mathcal{G}$
- sends $R_t, \ldots, R_{t-i}$ to P_{t-i-1}

– Round $2t-1$
On input the message $R_t, \ldots, R_2$, participant P_1
- computes $R_1 = R_2^{k_1}$ in G.
- computes the ZK proof Π_1 which states
  - $\exists\ \eta_1, \eta_2 \in [-q^3, q^3]$ such that
  - $R_1^{\eta_1} = R_2$ and $G^{\eta_2/\eta_1} = y_1$
  - $D(\alpha_1) = \eta_1$ and $D(\beta_1) = \eta_2$
- sends R_1, Π_1 to P_2

– Round $2t+i-2$ for $i = 2, \ldots, t-1$
On input $R_1, \ldots, R_{i-1}, \Pi_1, \ldots, \Pi_{i-1}$, participant P_i
- computes the ZK proof Π_i which states
  - $\exists\ \eta_1, \eta_2 \in [-q^3, q^3]$ such that
  - $R_i^{\eta_1} = R_{i+1}$ and $G^{\eta_2/\eta_1} = y_i$
  - $D(\alpha_i) = \eta_1 D(\alpha_{i-1})$ and $D(\beta_i) = \eta_2 D(\beta_{i-1})$
  - $D(\hat{\alpha_i}) = \eta_1$ and $D(\hat{\beta}_i) = \eta_2$
- sends $R_1, \ldots, R_i, \Pi_1, \ldots, \Pi_i$ to P_{i+1}

– Round $3t-2$
On input $R_1, \ldots, R_{t-1}, \Pi_1, \ldots, \Pi_{t-1}$, participant P_t
- choose $c \in_R Z_{q^{3t-1}}$
- computes $m = H(M)$ and $r = H'(R_1) \in Z_q$
- computes $\hat{\mu} = E(z_t)$
- computes $\mu = [(m z_3 \bmod q) \times_E \alpha_{t-1}] +_E [(r x_3 z_3 \times_E \beta_{t-1}] +_E E(cq)$
- computes the ZK proof Π_t which states

* $\exists\ \eta_1, \eta_2 \in [-q^3, q^3]$ such that
* $R_t^{\eta_1} = G$ and $G^{\eta_2/\eta_1} = y_t$
* $D(\mu) = m\eta_1 D(\alpha_{t-1}) + r\eta_2 D(\beta_{t-1})$
* $D(\hat{\mu}) = \eta_1$

- sends $\mu, \hat{\mu}, \Pi_i, \ldots, \Pi_t$ to all the other participants

– Final Decryption Rounds
At the end of the protocol, each player should have a proof from every other player. They must verify these proofs and abort if the verification fails[13]. The participants invoke the distributed decryption protocol for D over the ciphertext μ. Let $s = D(\mu) \bmod q$. The participants output (r, s) as the signature for M.

Encryption Scheme As in [31] and in our scheme above, we instantiate E with Paillier's encryption scheme [35]. We recall the scheme here.

– Key Generation: generate two large primes P, Q of equal length. and set $N = PQ$. Let $\lambda(N) = lcm(P-1, Q-1)$ be the Carmichael function of N. Finally choose $g \in Z_{N^2}^*$ such that its order is a multiple of N. The public key is (N, g) and the secret key is $\lambda(N)$.
– Encryption: to encrypt a message $m \in Z_N$, select $x \in_R Z_N^*$ and return $c = g^m x^N \bmod N^2$.
– Decryption: to decrypt a ciphertext $c \in Z_{N^2}$, let L be a function defined over the set $\{u \in Z_{N^2} : u = 1 \bmod N\}$ computed as $L(u) = (u-1)/N$. Then the decryption of c is computed as $L(c^{\lambda(N)})/L(g^{\lambda(N)}) \bmod N$.
– Homomorphic Properties: Given two ciphertexts $c_1, c_2 \in Z_{N^2}$ it is easy to see that $c_1 +_E c_2 = c_1 c_2 \bmod N^2$ (If $c_i = E(m_i)$ then $c_1 +_E c_2 = E(m_1 + m_2 \bmod N)$. Similarly, given a ciphertext $c = E(m) \in Z_{N^2}$ and a number $a \in Z_n$ we have that $a \times_E c = c^a \bmod N^2 = E(am \bmod N)$.

We point out that threshold variations of Paillier's scheme have been presented in the literature [2, 15, 16, 27]. In order to instantiate our dealerless protocol, we use the scheme from [27] as it includes a dealerless key generation protocol that does not require $n \geq 2t + 1$.

Zero-knowledge proofs. The ZK proof Π_1 is already described in [31] (as ZK proof Π in their paper). Similarly the ZK proof Π_t is described as Π' in [31].

We now describe the ZK proof Π_i used by the intermediate participants. This is always the same proof $\hat{\Pi}$ called on different inputs. As in [31] we make use of an auxiliary RSA modulus $\tilde{N}$ which is the product of two safe primes $\tilde{N} = \tilde{P}\tilde{Q}$ and two elements $h_1, h_2 \in Z_{\tilde{N}}^*$ used to construct range commitments.

For public values $c, d, w_1, w_2, m_1, m_2, m_3, m_4, m_5, m_6$ we construct a ZK proof $\hat{\Pi}$ that proves

[13] We aimed to simplify the communication channel, but if there is a broadcast channel, each player can directly broadcast its proof to all other players.

– $\exists\ x_1, x_2 \in [-q^3, q^3]$ such that
– $c^{x_1} = w_1$ and $d^{x_2/x_1} = w_2$
– $D(m_1) = x_1 D(m_3)$ and $D(m_2) = x_2 D(m_4)$
– $D(m_5) = x_1$ and $D(m_6) = x_2$

The protocol is as follows. We assume the Prover knows the values $r_5, r_6 \in Z_N^*$ such that $m_5 = g^{x_1} r_5^N \bmod N^2$ and $m_6 = g^{x_2} r_6^N \bmod N^2$. Moreover, the proof that we give is non-interactive. It relies on using a hash function to compute the challenge, e, and it is secure in the Random Oracle Model.

The prover chooses uniformly at random:

$\alpha, \delta \in Z_{q^3}$ $\quad$ $\beta_1, \beta_2 \in Z_N^*$ $\quad$ $\rho_3, \epsilon \in Z_q$
$\rho_1, \rho_2 \in Z_{q\tilde{N}}$ $\quad$ $\gamma, \nu \in Z_{q^3\tilde{N}}$

The prover computes

$z_1 = h_1^{x_1} h_2^{\rho_1} \bmod \tilde{N}$ $\quad$ $v_1 = d^{\delta+\epsilon}$ in $\mathcal{G}$
$u_1 = c^{\alpha}$ in $\mathcal{G}$ $\quad$ $v_2 = w_2^{\alpha} d^{\epsilon}$ in $\mathcal{G}$
$u_2 = g^{\alpha}\beta_1^N \bmod N^2$ $\quad$ $v_3 = m_3^{\alpha} \bmod N^2$
$u_3 = h_1^{\alpha} h_2^{\gamma} \bmod \tilde{N}$ $\quad$ $v_4 = m_4^{\delta} \bmod N^2$
$z_2 = h_1^{x_2} h_2^{\rho_2} \bmod \tilde{N}$ $\quad$ $v_5 = g^{\delta}\beta_2^N \bmod N^2$
$y = d^{x_2+\rho_3}$ in $\mathcal{G}$ $\quad$ $v_6 = h_1^{\delta} h_2^{\nu} \bmod \tilde{N}$

$e = \mathtt{hash}(c, d, w_1, w_2, m_1, m_2, m_3, m_4, m_5, m_6, z_1, u_1, u_2, u_3, z_2, y, v_1, v_2, v_3, v_4, v_5, v_6)$[14]

$s_1 = ex_1 + \alpha$ $\quad$ $t_2 = e\rho_3 + \epsilon$
$s_2 = (r_5)^e \beta_1 \bmod N$ $\quad$ $t_3 = (r_6)^e \beta_2 \bmod N$
$s_3 = e\rho_1 + \gamma$ $\quad$ $t_4 = e\rho_2 + \nu$
$t_1 = ex_2 + \delta$

The prover sends all of these values to the Verifier.

The Verifier checks that all the values are in the correct range and moreover that the following equations hold

$u_1 = c^{s_1} w_1^{-e}$ in $\mathcal{G}$ $\quad$ $v_4 = m_4^{t_1} m_2^{-e} \bmod N^2$
$u_2 = g^{s_1} s_2^N m_5^{-e} \bmod N^2$ $\quad$ $v_5 = g^{t_1} t_3^N m_6^{-e} \bmod N^2$
$u_3 = h_1^{s_1} h_2^{s_3} z_1^{-e} \bmod \tilde{N}$ $\quad$ $v_6 = h_1^{t_1} h_2^{t_4} z_2^{-e} \bmod \tilde{N}$
$v_1 = d^{t_1+t_2} y^{-e}$ in $\mathcal{G}$ $\quad$ $e = \mathtt{hash}(c, d, w_1, w_2, m_1, m_2, m_3, m_4, m_5, m_6,$
$v_2 = w_2^{s_1} d^{t_2} y^{-e}$ in $\mathcal{G}$ $\quad$ $z_1, u_1, u_2, u_3, z_2, y, v_1, v_2, v_3, v_4, v_5, v_6)$
$v_3 = m_3^{s_1} m_1^{-e} \bmod N^2$

***t*-out-of-*n* threshold signature scheme.** A t-out-of-n scheme can be obtained by considering all possible subsets of t participants and instantiating the

[14] This is the step of the proof that relies on the Random Oracle Model. We can construct the proof without random oracles using an interactive proof. In the interactive version of the proof, the Prover sends all of the values computed until this point. The Verifier then issues a challenge e, and the proof proceeds exactly as in the non-interactive version.

above protocol for each subset. We stress that the performance of the t-of-n protocol depends on t and not on n. The only performance overhead of t-of-n over t-of-t is identifying the proper t-of-t share to use.

One possible optimization is that the n participants can use a single encryption key E (rather than one for each subset), where the secret key D is shared in a t-out-of-n fashion among the participants.

A.2 Size of shares

The combinatorial structure to go from t-out-of-t to t-out-of-n requires $O(n^t)$ storage, making it feasible only for small values of n and t. It is an interesting open question to construct threshold DSA signature scheme that does not require storage that is exponential in t.

Interestingly, every application of threshold security to Bitcoin appears to be easily capable of handling the combinatorial structure, for one of two reasons.

1. Many applications require (t, t) sharing and not (t, n) for $t < n$. The (t, t) case does not use the combinatorial structure and thus only requires a single key share stored by each party. Indeed, ours is the first work to propose a (t, t) threshold DSA signature scheme for $t > 2$.
2. Even for our applications that do require a (t, n) signature, the values of t and n are inherently very small due to the nature of security policies used in practice (Section **??**).

A.3 Security Analysis

As we present this protocol for comparison only, we do not provide a full security proof. It is not hard to see, however, that the security proof follows the same lines of the proof in [31], and therefore the security of the entire distributed DSA signature scheme can be reduced to (i) the unforgeability of the DSA signature scheme; (ii) the semantic security of the Paillier encryption scheme (which we recall is equivalent to the N-residuosity assumption modulo N^2) and (iii) to the Strong-RSA Assumption (modulo $\tilde{N}$).

More specifically we prove *existential unforgeability against chosen message attack*, the strongest security notion for signatures. In the distributed case consider an adversary $\mathcal{A}$ controlling $t - 1$ players. Even after the entire set of t parties signs ℓ messages $M^{(1)}, \ldots, M^{(\ell)}$ chosen by $\mathcal{A}$, it should be computationally infeasible for $\mathcal{A}$ to compute a valid signature on a message $M \neq M_i$. We prove that this is the case by a simulation argument which shows that if such an adversary $\mathcal{A}$ exists then there exists a forger $\mathcal{F}$ that can forge a signatures in the underlying "centralized" DSA signature scheme. Since we assume the latter to be unforgeable, then the former cannot happen. We assume a static corruption model, in which $\mathcal{A}$ assumes control of $t - 1$ players at the beginning of the protocol.

So let us assume by contradiction that $\mathcal{A}$ exists and show how to construct $\mathcal{F}$. This forger $\mathcal{F}$ also works in the *chosen message attack model*, i.e. on input

a DSA public key y, it has access to a "signature oracle" which on input $\hat{M}$ returns the signature $\hat{r}, \hat{s}$ under the public key y.

$\mathcal{F}$ runs on input y, the public key of the underlying DSA scheme. It will initiate $\mathcal{A}$ and assume the role of P_i the only honest player not corrupted by $\mathcal{A}$.

Key Generation. Assuming a trusted party initialization of the system, $\mathcal{F}$ will create the public key for the pk for the encryption scheme E and share sk among the players. Note that $\mathcal{F}$ knows sk. Then it will generate random values $x_j \in Z_q$ as the secret share of player P_j; it will compute $\lambda = (\Pi_{j \neq i} x_j)^{-1} \bmod q$ and will set $y_i = y^{\lambda}$.

Signature Generation. When $\mathcal{A}$ requests the signature of a message M, the forger $\mathcal{F}$ will query its signature oracle and get r, s. Let $R = G^{H(M)s^{-1}} y^{rs^{-1}}$. We now show how to simulate a signature protocol so that it results in this signature being output.

Simulating r. The players run the protocol up to Round $2t-1$ with the difference that at Round i the Forger encrypts arbitrary values (e.g. 0) in the $\alpha_i, \hat{\alpha}_i, \beta_i, \hat{\beta}_i$ ciphertexts. Note that at the end of Round t, the Forger knows all the values k_j chosen by $\mathcal{A}$ (since he knows the sk). At round $2t - i$ when $\mathcal{F}$ has to announce R_i it will it will compute $\lambda' = (\Pi_{j \neq i} k_j)^{-1} \bmod q$ and will set $R_i = R^{\lambda'}$.

Simulating s. The players run the protocol from Round $2t - 1$ to the end. The forger $\mathcal{F}$ will simulate the ZK proof Π_i, since it is now proving an incorrect statement. Over the final ciphertext μ, the forger will now simulate the distributed decryption protocol for E so that it results in a value $s' \in Z_{q^{3t}}$ s.t. $s' = s \bmod q$

In order to conclude the proof we must argue that the above simulation is indistinguishable from a real execution of the protocol. Indeed only under this condition we can claim that $\mathcal{A}$ will output a forgery, and therefore $\mathcal{F}$ will succeed.

We note that the above simulation differs from the real execution in three main points

- The final decryption protocol is simulated to "hit" a specific value, instead of the correct decryption of the ciphertext μ. But if the threshold encryption scheme used to do distributed decryption is secure, then this step is indistinguishable from the real-life protocol.
- the ZK proof Π_i is simulated. Due to the zero-knowledge properties, a simulated proof is indistinguishable from the real one.
- The simulated ciphertexts $\alpha_i, \hat{\alpha}_i, \beta_i, \hat{\beta}_i$ sent by $\mathcal{F}$ encrypt values with a different distribution that in the real protocol. But if E is semantically secure then these simulated ciphertexts are computationally indistinguishable from the real ones. Note that this requires another reduction, where we use $\mathcal{A}$ to break the encryption scheme E (in this case the simulation knows the secret key x of the DSA scheme, but does not know sk).