Unstoppable Wallets: Chain-assisted Threshold ECDSA and its Applications

Unstoppable Wallets: Chain-assisted Threshold ECDSA and its Applications

Guy Zyskind
guyz@mit.edu
MIT Media Lab
Cambridge, MA, USA

Avishay Yanai
ay.yanay@gmail.com
Tel Aviv, Israel

Alex "Sandy" Pentland
pentland@mit.edu
MIT Media Lab
Cambridge, MA, USA

ABSTRACT

The security and usability of cryptocurrencies and other blockchain-based applications depend on the secure management of cryptographic keys. However, current approaches for managing these keys often rely on third parties, trusted to be available at a minimum, and even serve as custodians in some solutions, creating single points of failure and limiting the ability of users to fully control their own assets. In this work we first revisit the problem of threshold ECDSA by considering the commonly admissible 'server-aided' model, namely, the presence of a semi-honest and non-colluding service provider. Then, we leverage that model and consider cases where that 'server' is distributed, introducing the novel concept of *unstoppable wallets*; hence eliminating any single point of failure. *Unstoppable wallets* are programmable threshold ECDSA wallets that allow users to co-sign transactions with a confidential smart contract, rather than a singular third-party. We construct highly efficient threshold ECDSA protocols that form the basis of unstoppable wallets and prove their security in the server-aided model, achieving the standard notion of fairness and robustness even in case of a dishonest majority among the signers. Our protocols minimize the *write-complexity* for threshold ECDSA key-generation and signing, while reducing communication and computation overhead.

We provide a proof-of-concept implementation of these protocols, written in a smart contract language, deployed on the Secret Network - a blockchain that plays the role of the server. Using that deployment, we showcase the protocols' applicability for two interesting applications, *policy checking* and *wallet exchange*, as well as their efficiency by demonstrating low gas costs and fees.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; *Usability in security and privacy*; **Digital signatures**.

KEYWORDS

Threshold ECDSA, Cryptocurrency Custody, Server-aided MPC, Confidential Smart Contracts

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07…$15.00
https://doi.org/10.1145/3634737.3637657

ACM Reference Format:
Guy Zyskind, Avishay Yanai, and Alex "Sandy" Pentland. 2024. Unstoppable Wallets: Chain-assisted Threshold ECDSA and its Applications. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3634737.3637657

1 INTRODUCTION

Threshold ECDSA (Elliptic Curve Digital Signature Algorithm) is a cryptographic technique that enables multiple parties to jointly sign a message using a shared secret key. It has gained increasing importance in the custody of cryptocurrencies and Web3 due to its ability to improve security and control over private key management. By enabling multiple parties to jointly control a single private key, threshold ECDSA allows for the creation of multisignature (multisig) accounts that require multiple approvals before a transaction can be signed. This enhances security by reducing the risk of funds being stolen or lost due to a single point of failure. Additionally, threshold ECDSA enables the creation of flexible and customizable access policies.

Current deployments of threshold ECDSA (e.g., Fireblocks, Coinbase Wallet, BitGo, Zengo and others) rely on a third party service provider for availability and are often limited by closed-sourced vendors [41]. In practice, relying on a service provider for availability has proven itself inadequate, and in certain cases even disastrous, leading to significant loss of funds (e.g., FTX[1] and Prime Trust[56] incidents, to name a few). Furthermore, even beyond availability, trusting a service provider with *correctness* is difficult, as for example, a compromised service provider may ignore a client's policy and convince the client to sign an unintended transaction (e.g., by changing the user interface). Such attacks are not theoretical, see MyEtherWallet for example [14].

In this paper, we introduce *unstoppable wallets* as a novel concept that addresses the limitations of current threshold ECDSA systems. An unstoppable wallet is a threshold ECDSA wallet where the counterparty co-signing transactions with the user (or a set of users) is not a singular third-party, but rather a blockchain itself , which naturally provides strong availability and correctness. This enables the creation of programmable wallets that are controlled directly by a smart contract, such as those being explored through the concept of *account abstraction* [62]. Unstoppable wallets push this idea further, as they can operate cross-chain and are not limited to Ethereum or EVM chains only.

Since generally speaking, blockchains cannot keep a private state, we require the use of blockchains that support *confidential smart contracts*, which are gaining popularity and are being explored in both research and practice. Confidential smart contracts can be

[1]https://en.wikipedia.org/wiki/FTX

constructed by different underlying techniques and assumptions, such as Trusted Execution Environments (TEEs) [13, 22, 32, 45], Secure Multiparty Computation (MPC) [12, 16, 29, 42, 69], or Homomorphic Encryption (HE) [57, 58]. Similar to how smart contracts eliminate the need for excessive trust in intermediaries, enabling novel applications like Decentralized Finance, unstoppable wallets further enable new use-cases that require similar levels of trust. To illustrate this, consider a decentralized lending protocol such as Compound[2]. In Compound, the smart contract acts as a trusted escrow between lenders and borrowers. In contrast, many centralized service providers offering the same functionality have recently failed, leading to billions in customer funds lost. Similarly, with unstoppable wallets, one can securely implement use-cases such as a *wallet exchange*, which allows users to atomically trade *wallets*. With an unstoppable wallet construct, an underlying smart contract atomically escrows and facilitates the process (just like lending in DeFi). Identifying a wallet as an asset on its own rather than a vehicle to store assets has many advantages. For example, one may transfer all its token portfolio in one transaction, even if part of it is staked, locked or lent; in addition, wallets can accrue reputation according to their activities, which may increase/decrease their value. In some sense, one may see a wallet as a non-fungible token (NFT).

1.1 Practical Model for Cryptographic Protocols

Cryptographic protocols in the server-aided model are common in the literature and real-world deployments, for both generic and application specific functionalities (see [40, 51] and references within). In this section we take this model further and argue that replacing the server with a blockchain serves as a better real-world realization of that model, in particular when availability is necessary.

Designers of cryptographic protocols have been increasingly relying on blockchains as their broadcast channel infrastructure [24, 38, 39, 50], as they may assist in achieving desired properties (e.g., [39]), and work around known impossibility results (e.g., [24]). Such a transition has prompted researchers to explore other benefits that can be derived from blockchains.

At their core, blockchains provide such benefits due to the strong *availability* and *correctness* properties that they provide. One might also wish for a blockchain that entirely handles sensitive information, such as cryptographic keys, and is able to confidentially perform operations (like sign and decrypt) using the keys. Confidential smart contracts-enabled blockchains aim to offer exactly that, by relying on an underlying MPC protocol or TEEs. Whichever assumption is used, it is important to note that while active faults in cryptographic multiparty protocols can be publicly detected and attributed, privacy breaches (in general) are not. Consider, for instance, a publicly auditable MPC protocol (in short, a protocol in which everyone can determine if a party faithfully follows the protocol steps or not). In case an attacker corrupts a sufficient number of parties it can "silently" break privacy; however, even such an attacker is not able to break the correctness of the protocol. .

While we can rely on blockchains for correctness and availability, when it comes to privacy, and especially as it relates to storing signature keys, the above suggests that it is better to split the trust between the users themselves (who owns the signature keys) and the blockchain, even if the latter employs its own underlying privacy-preserving techniques. In our case of threshold signatures, we rely on the blockchain to store a partial secret, which is only a share of the actual underlying signing key. By doing so, breaking the blockchain security layer only reveals that share of the secret, and not the full key.

Equipped with this intuition, we present a Threshold ECDSA protocol for n parties, out of which at most $t < n$ are malicious and colluding, to generate an ECDSA key-pair and sign messages, with the aid of a blockchain as described above. We assume the blockchain supports confidential smart contracts, meaning that it uses privacy-preserving techniques to protect a contract state from outside actors. To keep the model as general as possible, we do not prescribe which technique the blockchain uses to achieve confidentiality. Instead, we capture the properties described above by modeling the blockchain as an additional semi-honest and non-colluding party, referred to as P_c. Such a party can easily play the role of a broadcast channel (by simply relaying a message to all other parties) and hold and operate on secrets.

Another benefit of this model, is that parties do not necessarily need to know each other in advance, or set up complex ad-hoc communication networks with point-to-point channels across each set of parties, or an underspecified broadcast channel, as is common with MPC protocols. Moreover, parties can come and go as they please, even mid-execution of a protocol, since all coordination is done on-chain, which is guaranteed to be robust. Finally, our protocols support detection of cheating parties, which can be immediately translated to a monetary punishment on-chain.

Lifting all communication on-chain is advantageous at a high level because it simplifies protocol implementation in practice, as each node only reads and writes to a single endpoint, regardless of the number of counterparties. Specifically, by relying on a blockchain, one does not need to take care of network synchronization, and 'proofs of silence' (i.e., a proof that a participant did not send a message) are taken for granted[3]. There are several other benefits to this, such as pseudonimity, higher degree of censorship-resistance, public accountability (e.g., in the context of DAO multi-sigs), etc. Finally, recall that in some settings, and in the dishonest majority setting that we address in particular, implementation of a broadcast channel is impossible. Thus, this blockchain assisted model implicitly outsources the broadcast channel operation to an external entity.

In this new communication model every message, either peer-to-peer or broadcast, is translated to a blockchain transaction, which is inherently a broadcast message. On one hand, broadcasting on chain may entail significantly larger latency than a plain broadcast that is implemented among the parties. On the other hand, all messages are available to the participants whenever they are ready to consume them. This enables an easy recovery and auditability by participants that experienced a temporary offline period.

This necessitates the reassessment of the concept of rounds – a crucial performance metric used to evaluate protocols in the

[2]https://compound.finance/

[3]This should not be interpreted as everything being perfect when using a blockchain; rather, we argue that using a blockchain obscures these problems away from the developer. Indeed, a block lacking a message from a user does not necessarily mean the user did not send that message; for example, the recent blockchain block's validator/miner may have censored that message.

standard MPC model (without the aid of blockchains). The number of rounds informally measures the longest sequence of interdependent messages sent between parties. In this modified model, each round consists of one or more parties writing to the blockchain, followed by all parties reading from it. Although one might assume that this would typically involve decomposing each round into two separate rounds, we must recognize that writing to a blockchain is significantly more costly than reading, as it necessitates consensus and updating a replicated state.

As a result, our primary objective in this model is to minimize the total number of messages, with a specific focus on reducing the number of sequential *writes*, simply referred to as 'writes' hereafter. While in previous works on threshold ECDSA the number of writes is equivalent to the number of rounds, this does not hold for the protocols presented in this work, highlighting the importance of identifying a common performance metric.

Figure 1 illustrates the model we described. All parties are connected via a slow *write* channel to the blockchain party P_c, which also acts as a public bulletin board they can read from (specifically, we assume P_c has a public state anyone can read from). Finally, being one of the computing parties, P_c also maintains its own private state.

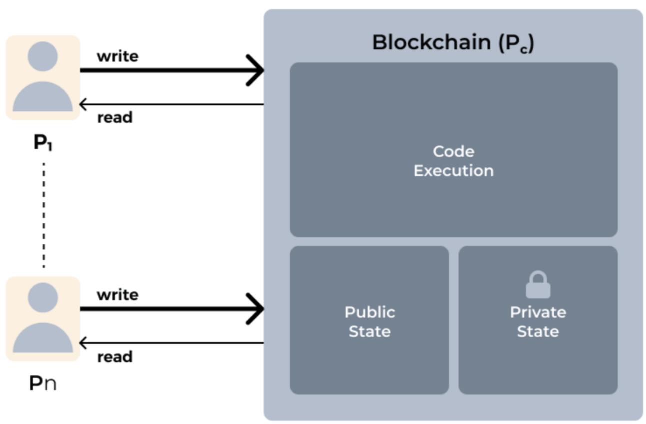


Figure 1: Communication Model Illustrated

1.2 Our contributions

In this paper, we make the following main contributions:

- We revisit the problem of threshold ECDSA by considering a real-world 'server-aided' model, and construct new protocols for threshold ECDSA in that model. Defining messages *to the server* as 'write' and messages *from the server* as 'read', our protocols enjoy the minimal number of 'writes', compared to previous works (see Table 1). In particular, our robust threshold ECDSA incurs only a single write message from the parties, and another one from the signature initiator, compared to previous protocols that incur at least four writes. We also greatly reduce communication and computation, by avoiding the use of expensive cryptographic primitives such as Paillier encryption and costly zero knowledge proofs over Paillier ciphertexts. We provide a full proof of security of our protocols in the server-aided model, treating the server as a semi-honest non-colluding party, and show that the protocols offer both *fairness* and *robustness*. That is, we achieve *robustness* in the sense that if $t+1$ parties agree to sign on a message (and hence participate in the protocol faithfully), then they will obtain the signed message.
- We implement these protocols as smart contracts and deploy them on a functioning blockchain with confidential smart contract capabilities. By doing this, we introduce the concept of *unstoppable wallets* - programmable threshold ECDSA wallets where the counterparty co-signing transactions with the user (or a set of users) is not a singular third-party we need to rely on, but rather a confidential smart contract.
- We ran benchmarks of our protocols, ranging from $n = 2$ to $n = 15$ signers, and prove their real world applicability by reporting on their respective gas costs and fees. To show case the importance of wallet programmability we develop two applications: a *multisignature wallet with policy checks* and a *wallet exchange*.

1.3 Related Work

Our work builds upon the existing body of research on concretely efficient threshold ECDSA protocols in the dishonest majority setting. Previous works in this setting can be grouped into several categories:

- Protocols using Paillier's Homomorphic Encryption (HE) with a small number of rounds but high computational cost [11, 18, 34, 35, 48]. These also require expensive zero-knowledge proofs over Paillier ciphertexts. Optimized variants for the two-party variants also exist (e.g., [47, 65]).
- Replacing HE with class group-based schemes as in [19–21], which improves the efficiency of zero-knowledge proofs but not the number of rounds, while introducing different assumptions on class groups of imaginary quadratic fields.
- Oblivious transfer (OT)-based protocols [30, 31], that reduce cryptographic assumptions and computational overhead but increases round complexity.
- Protocols that are based on generic MPC; in particular in such protocols multiplication triplets are pre-processed [1, 26]. These protocols typically increase the overall number of rounds (and hence, the number of writes) and in some cases introduce newer assumptions such as Learning Parity with Noise (LPN) [1].

In contrast to prior work, our protocols are designed to be chain-friendly, by reducing the number of writes without resorting to heavyweight cryptographic tools like HE and expensive ZKP that are likely too inefficient to run in a constraint blockchain environment.

Our protocol also achieves two often overlooked properties for threshold ECDSA: *fairness* and *robustness*. The current state-of-the-art honest majority threshold ECDSA protocol by Damgard et al., [27] achieves fairness in six writes, as opposed to 1-2 writes in our work, and by well-known impossibility results, dishonest majority protocols (without blockchain assistance) cannot hope to achieve fairness at all [24, 25].

As to robustness, since the original work of Gennaro et al., on threshold (EC)DSA for a super-honest majority ($n \geq 4t + 1$) more than two decades ago [36], most known efficient protocols in the dishonest majority setting (e.g., [18, 34, 35, 48]) sacrifice robustness for additional efficiency gains. These protocols move from threshold to additive secret sharing as soon as pre-signing starts, leaving no room to handle faults mid-execution. Recently, attempts to partially address robustness have been proposed. Gagol et al. [33] suggested a robust scheme which requires all parties to participate honestly in the pre-signature phase, while others proposed schemes with identifiable aborts instead (e.g., [18] [21]). In a concurrent and independent work, Wong et al., [63] achieve a stronger notion of robustness they call 'self-healing robustness', where as long as the signers in the online-phase are a subset of the signers in the pre-signature phase, their scheme is either robust (for an honest majority) or gracefully falls back to identifiable aborts otherwise. In contrast, in this work we achieve the standard plain notion of robustness, where signers in pre-signing and signing can be disjoint.

For a comprehensive comparison of our work with the existing literature, please refer to Table 1. Note that we also describe a scenario unique to our work, where there is a single signing party involved (i.e., $n = 1$). This is an interesting scenario, as it allows a single user to increase their wallet security by having P_c as a co-signer. Similarly, some use cases, like wallet exchange, may make more sense under this setting. However, for this scenario, we describe a modified version of [47] and show that while it is less efficient than our main protocol, it can still run on-chain.

Other works address *generic* MPC with fairness and public verifiability via bulletin boards (that can be implemented with blockchains). Bentov et. al, Kumerasen et. al, and Baum et. al [5, 9, 43, 44] achieve a revised form called 'fairness with penalties' using gradual release mechanisms and deposits using a blockchain. Choudhuri et al. [24] showed how to use blockchains to achieve the standard notion of fairness (without penalties), by leveraging either witness encryption, which is too expensive in practice, or off-chain TEEs. Baum et al. and Rivinius et al. show how to achieve public verifiablity and robustness using a public bulletin board [4, 6, 53]. Similarly, a long line of works of MPC-as-a-service systems inspired by blockchains have emerged in recent years [7, 23, 28, 37–39, 49, 50, 61, 68, 69]. While they address how a blockchain can help with the general MPC problem (or how MPC can add confidentiality to blockchains), our work, as far as we know, showcases the first threshold ECDSA protocol that effectively provides both fairness and robustness, by relying on an external blockchain.

Finally, in contrast to all prior works solving threshold ECDSA, we are the first to consider a model for MPC protocols that leverage a special non-colluding semi-honest party. We show how such a model greatly increases efficiency, simplifies protocol design, and achieves properties of interest – in our scenario, fairness and robustness. We further show how this model is realized in practice through the use of a confidential smart contracts, which have garnered significant interest in recent years [2, 3, 8, 12, 13, 16, 22, 29, 32, 42, 45, 46, 57–59, 64, 66, 67, 69].

2 PRELIMINARIES

We use κ as a computational security parameter. For $x, y \in \{0,1\}^*$ the expression $x||y$ is the concatenation of x and y. Uniformly sampling a random value x from a set X is denoted by $x \leftarrow X$. The result of a probabilistic algorithm A on inputs $x_1, x_2, \ldots$ is written by $x \leftarrow A(x_1, x_2, \ldots)$; or $x = A(x_1, x_2, \ldots ; r)$ for randomness r. by $(\mathbb{G}, G, q)$ we denote the ECDSA elliptic curve group, its generator and its order, respectively. For an element in the group $H \in \mathbb{G}$, we write $H.x$ to denote its x-coordinate.

Protocol	Parties	Writes	Messages	Primitives	Properties
LN18 [48]	n	8	$O(n^2)$	Paillier	
CGGMP20 [18]	n	4	$O(n^2)$	Paillier	IA
DKLS19 [31]	n	log(t) + 6	$O(n^2)$	OT	
BMP22 [10]	n	4	$O(n)$	Paillier	
CCLST20 [20]	n	8	$O(n^2)$	CL-HE	
CGCL+23 [21]	n	7	$O(n^2)$	CL-HE	IA, Fairness (Honest Majority)
WMYC23 [63]	n	5	$O(n^2)$	Paillier	Self-healing
Lindell17 [47]	2	2	$O(1)$	Paillier	
XAXYC21 [65]	2	3	$O(1)$	HE/OT	
CCLST19 [19]	2	3	$O(1)$	CL-HE	
DKLS18 [30]	2	7	$O(1)$	OT	
This work	n	1-2	$O(n)$	Group	Fairness
This work	n	1-2	$O(n^2)$	Group	Robustness
This work	1	1	$O(1)$	Paillier	

Table 1: Comparison with related work. For protocols that support pre-signing – the number of writes consists of both pre-sign and sign phase, ignoring amortization.

2.1 The ECDSA Scheme and Functionality

The ECDSA scheme is defined by the following algorithms (the group $\mathbb{G}, G, q$ is an implicit parameter in the algorithms):

- Gen(). Choose $x \leftarrow \mathbb{Z}_q^*$ and compute $X = x \cdot G$. Output x as the signing key and X as the verification key.
- Sign(x, M). For a message $M \in \{0,1\}^*$, choose $k \leftarrow \mathbb{Z}_q^*$ and compute $r = (k \cdot G).x \mod q$ and $s = k^{-1}(m + rx) \mod q$, where $m = H_q(M)$ and $H_q : \{0,1\}^* \to \mathbb{Z}_q$ is modeled as a random oracle. Output the signature (r, s).
- Verify$(X, M, (r, s))$. For a message $M \in \{0,1\}^*$, compute $m = H_q(M)$ and output 1 iff $(ms^{-1} \cdot G + rs^{-1} \cdot X).x \mod q = r$, otherwise output 0.

Indeed, if (r, s) is computed correctly on M, then $ms^{-1}{\cdot}G{+}rs^{-1}{\cdot}X = ms^{-1}{\cdot}G{+}rxs^{-1}{\cdot}G = (m{+}rx)s^{-1}{\cdot}G = (m{+}rx)\left(k^{-1}(m + rx)\right)^{-1}{\cdot}G = (m + rx)k(m + rx)^{-1} \cdot G = k \cdot G = R$ and so, projection to the x coordinate results with $R.x = r$.

The ECDSA functionality (Functionality 1) supports two interfaces, the key-generation interface is called once, followed by many, calls to the sign interface. We note that our robust protocol implements a slightly different functionality, in which the gray text is omitted, in that functionality the adversary does not get to decide on whether to forward outputs to the parties or not.

2.2 Shamir Sharing and Lagrange Interpolation

Secret sharing enables a dealer to split a secret x into n pieces or *shares*, such that only a sufficiently large subset of shares can be used to recover the secret. Shamir t-out-of-n secret sharing over the field $\mathbb{F}$ (where $t < n \in \mathbb{N}$) is defined by a tuple of algorithms $\mathsf{SS}_{\mathbb{F}} =$ (Share, Reconstruct), where $[x] = ([x]_1, \ldots, [x]_n) = \mathsf{Share}_{t,n}(x; r)$ denotes a sharing of x, and $x = \mathsf{Reconstruct}([x]_{i_1}, \ldots, [x]_{i_{t+1}})$ denotes the reconstruction using $t + 1$ shares, which may result with $\perp$ if the shares are inconsistent.

FUNCTIONALITY 1. (*The ECDSA Functionality: $\mathcal{F}_{ECDSA}$*)
The functionality is parameterized with the ECDSA group description $(\mathbb{G}, G, q)$ as well as a threshold parameter t, with $1 \leq t < n$. The functionality works with parties $P_1, \ldots, P_n$, P_c, and an adversary $\mathcal{S}$ as follows.

- Upon receiving (keygen) from all parties:
  (1) Generate an ECDSA key-pair (X, x) by choosing a random $x \leftarrow \mathbb{Z}_q^*$ and computing $X = x \cdot G$.
  (2) Choose a hash function $H_q : \{0,1\} \rightarrow \{0,1\}^{\lfloor \log q \rfloor}$.
  (3) If received $(\text{keygen}, \text{abort})$ from $\mathcal{S}$ then output $\perp$ and halt; otherwise, if received $(\text{keygen}, \text{continue})$ then continue.
  (4) Store (H_q, x), output X to all parties, and ignore future calls to keygen.
- Upon receiving $(\text{sign}, \text{sid}, M)$ from P_c and $t+1$ parties out of $\{P_1, \ldots, P_n\}$, if keygen was already called and sid was not already used:
  (1) Choose a random $k \in \mathbb{Z}_q^*$, compute $R \leftarrow k \cdot G$ and let $r = R.x \mod q$; then send R to all parties.
  (2) Let $m = H_q(M)$. Compute $s \leftarrow k^{-1}(m + rx) \mod q$.
  (3) If received $(\text{sign}, \text{sid}, \text{abort})$ from $\mathcal{S}$ then output $\perp$ and halt; otherwise, if received $(\text{sign}, \text{sid}, \text{continue})$ then continue.
  (4) Send (r, s) to all parties.

2.3 Schoenmakers's Publicly Verifiable *Random* Sharing Scheme

Verifiable secret sharing (VSS) enables a receiver to (1) check in the dealing phase that the share received from the dealer is consistent with a fully determined secret, and (2) check in the reconstruction phase that the shares published by other receivers are correct. Publicly VSS (PVSS) enables a receiver to check consistency not only of its own share, but also all receivers' shares; furthermore, it enables an external party to check that conditions (1) and (2) hold. While information theoretic schemes for PVSS schemes have been proposed [52], we use Schoenmakers's scheme that is based on the hardness of discrete logarithm, as it is the minimal assumption in our context anyway. Specifically, we use the *special* PVSS version in [54], in which *the secret is random*, which allows using a simpler protocol.[4] Thus, in the following we assume that x is uniformly random from $\mathbb{Z}_q$.

Schoenmakers's PVSS [54] over the group $(\mathbb{G}, G, q)$ is parameterized with the receivers' encryption keys, namely, the i-th receiver is associated with El-Gamal key-pair $(\mathsf{ek}_i, \mathsf{dk}_i)$. While the scheme supports any encryption scheme, the El-Gamal scheme leads to a very simple implementation and efficient proof. The dealer invokes the zero-knowledge functionality (see definition in Section C) with the relation

$$R_{\mathsf{PVSS},n,t} = \Big\{ \Big(\{\mathsf{ek}_i, c_i\}_{i=1}^n, \{A_j\}_{j=0}^t), (\{r_i\}_{i=1}^n, \{a_j\}_{j=0}^t \Big) \text{ s.t.}$$

$$\forall_{i=1}^n : c_i = \mathsf{EG.Enc}\left(\sum_{j=0}^t i^j \cdot a_j, r_i\right) \wedge \forall_{j=1}^t : A_j = a_j \cdot G \Big\}.$$

That is, the claim is that c_i is an encryption of $P(i) = \sum_{j=0}^t i^j \cdot a_j$ under the i-th public key ek_i, where $P's$ coefficients are $\log_G(A_0), \ldots, \log_G(A_t)$. Note that given A_j's anyone can compute $Q_i = P(i) \cdot G$ by $\sum_{j=0}^t i^j \cdot A_j$, thus, interpretting $c_i = (C_{i,1}, C_{i,2})$, this statement is reduced to the statement that $(Q_i, X, C_{i,2} \cdot (C_{i,1})^{-1})$ is a Diffie-Helman tuple, for every i. Indeed, the discrete logs are $x, P(i)$ and $x \cdot P(i)$, respectively. There exists standard NIZK for that statement.

Then, the Schoenmakers's scheme is defined by the tuple of algorithms $\mathsf{PVSS}_{(\mathbb{G},G,q),\{\mathsf{ek}_i\}_i} = (\mathsf{Share}, \mathsf{Reconstruct}, \mathsf{CheckDealer}, \mathsf{CheckShare})$:

- $(\{c_i\}_{i=1}^n, \{A_j\}_{j=0}^t, \pi) \leftarrow \mathsf{Share}_{t,n}(x)$. Set $a_0 = x$ and pick $a_1, \ldots, a_t \in \mathbb{F}$ and compute $[x] = \{[x]_1, \ldots, [x]_n\}$, where $[x]_i = P(i)$ and $P(x) = \sum_{j=0}^j a_j \cdot x^j$. Then, pick $r_i \leftarrow \mathbb{Z}_q^*$, compute $c_i = \mathsf{EG.Enc}_{\mathsf{ek}_i}([x]_i, r_i)$ for every $i \in [1, n]$, and compute $A_j = a_j \cdot G$ for every $j \in [0, t]$. Then, send $(\text{prove}, \text{sid}, \{\mathsf{ek}_i, c_i, r_i\}_i, \{A_j, a_j\})$ to $\mathcal{F}_{\mathsf{zk}}^{R_{\mathsf{PVSS}}}$ to obtain $\pi = (\text{proof}, \text{sid}, \{\mathsf{ek}_i, c_i\}_i, \{A_j\})$ and output $(\{c_i\}_{i=1}^n, \{A_j\}_{j=0}^t, \pi)$.
- $x = \mathsf{Reconstruct}([x]_{i_1}, \ldots, [x]_{i_{t+1}})$. Given $t+1$ shares $[x]_{i_1}, \ldots, [x]_{i_{t+1}}$, where $1 \leq i_1 < i_2 < \ldots < i_{t+1} \leq n$, for which $\mathsf{CheckShare}(\{A_j\}, [x]_{i_k}) = 1$ for all $k \in [1, t+1]$, interpolate a polynomial P such that $P(i_k) = [x]_{i_k}$ for all $k \in [1, t+1]$ and output $x = P(0)$.
- $b \leftarrow \mathsf{CheckDealer}(\{c_i\}_{i=1}^n, \{A_j\}_{j=0}^t \pi)$. Output $b = 1$ iff $\pi = (\text{proof}, \text{sid}, \{\mathsf{ek}_i, c_i\}_i, \{A_j\})$, and $b = 0$ otherwise.
- $b \leftarrow \mathsf{CheckShare}(\{A_j\}_{j=0}^t, [x]_k)$. For $k \in \mathbb{Z}_q^*$, output $b = 1$ iff $[x]_k \cdot G = \sum_{j=0}^t k^j \cdot A_j$, and $b = 0$ otherwise.

The scheme is a secure publicly verifiable secret sharing if the DDH problem is hard relative to $(\mathbb{G}, G, q)$.

2.4 Confidential Smart Contracts

A blockchain is a decentralized, distributed ledger that records transactions across a network of nodes, ensuring data integrity and transparency. Smart contracts are a fundamental component of many blockchain platforms, enabling users to automate the execution of agreements and facilitate trustless interactions between parties. These self-executing, deterministic[5] programs provide correctness and availability by ensuring that the code executes exactly as programmed without downtime, censorship, fraud, or third-party interference. However, traditional smart contracts do not inherently provide privacy, as their logic and data are visible to all network participants.

To address this issue, privacy-preserving blockchains have been developed , which enable executing *confidential smart contracts*. This means that these blockchains inherently hide sensitive input data fed into contracts, persistent state data, and depending on the use case, hide the output as well, even from the nodes operating the chain. Confidential smart contract-enabled blockchains may employ different privacy-preserving techniques, such as secret-sharing MPC (e.g., [69]), TEEs (e.g., [45]), or even HE ([57, 58]).

3 THRESHOLD ECDSA PROTOCOL

As explained in Section 1.3, current threshold ECDSA protocols require the use of expensive primitives (like HE or OT) and require at the very least four rounds of interactions, which in our model, translate to four consecutive writes to the blockchain. That kind of latency, and more importantly, the implied requirement from

[4]When the secret x is random it is possible for the dealer to publish $x \cdot G$, whereas in case x is not random the dealer has to publish a Pedersen commitment $x \cdot G + r \cdot H$ where H is another generator of $\mathbb{G}$ for which $\log_{\mathbb{G}} H$ is unknown.

[5]In our context, we require the contract to be non-deterministic in order to sample random values, a challenge we address in our implementation.

each user to sign four transactions in a row in order to produce a signature is too burdensome in practice.

In-line with our goals, we seek to construct a protocol that would be chain-friendly, and would minimize the number of writes each party has to perform. For all parties except one, we achieve the optimal of a single write per-party, which can be done non-interactively. A designated party (the signature initiator) needs to write twice. . The blockchain is modeled as an additional semi-honest and non-colluding party, denoted P_c. This enables us to take a different approach and leverage techniques from honest-majority MPC, even though the adversary may corrupt the majority of the parties $P_1, \dots, P_n$. We do this by assigning n shares to the parties, and t additional shares are held by P_c, for a total of $N = n+t$ shares. Our protocol ensures that as long as there are $t+1$ honest signers they will generate a valid signature; otherwise, no information is revealed.

In that sense, our protocol resembles the one by Damgard et al. [27], which is secure in the honest majority setting; however, we make significant changes to their protocol, greatly improving the number of writes and the communication costs. In particular, their protocol requires six writes (or four writes without fairness, which we obtain anyway), which is significantly more than ours.

For readability reasons, in the protocols below we write that P_i sends P_j a message although it is understood that P_i only communicates through P_c. That is, P_i sends a ciphertext to P_c under P_j's encryption key, and then P_j decrypts that message (implicitly implying PKI).

PROTOCOL 2. (*Key-Generation:* KeyGen)

(1) **Users' dealing**:
  (a) Party P_1 samples a random $x_u \leftarrow \mathbb{Z}_q$.
  (b) Party P_1 computes $[x_u] \leftarrow \mathsf{SS.Share}_{t,N}(x_u)$.
  (c) Party P_1 sends $[x_u]_i$ to P_i for all $i \in [1, n]$ and $[x_u]_i$ to P_c for all $i \in [n+1, N]$.

(2) **Center's dealing**:
  (a) Party P_c samples a random $x_c \leftarrow \mathbb{Z}_q$.
  (b) Party P_c computes $[x_c] \leftarrow \mathsf{SS.Share}_{t,N}(x_c)$, and sends $[x_c]_i$ to P_i for $i \in [1, n]$.

(3) **Compute key share**:
  (a) For each $j \in n+1, \dots, N$, P_c computes $[x]_j = [x_u]_j + [x_c]_j \mod q$ and $X_j \leftarrow [x]_j \cdot G$.
  (b) Each party P_i ($i \in [1, n]$) computes $[x]_i = [x_u]_i + [x_c]_i \mod q$ and $X_i = [x]_i \cdot G$.
  (c) Each party P_i ($i \in [1, n]$) sends X_i to P_c.

(4) **Public key**:
  (a) Let P be the polynomial defined by the $t+1$ points $(n, [x]_n), (n+1, [x]_{n+1}) \dots, (N, [x]_N)$, and let $\lambda_n^j, \lambda_{n+1}^j, \dots, \lambda_N^j$ be the Lagrange coefficients s.t. $P(j) = \sum_{k=n}^{N} \lambda_k^j \cdot [x]_k$.
  (b) Party P_c verifies that the keys are consistent: For every $j \in [1, n-1]$ compute $X'_j = P(j) \cdot G = \sum_{k=n}^{N} \lambda_k^j \cdot X_k$, then, abort if $X'_j \neq X_j$.
  (c) Otherwise (if all key shares are consistent) P_c broadcasts the public key $X = P(0) \cdot G = \sum_{k=n}^{N} \lambda_k^0 \cdot X_k$.

3.1 Key Generation

Our key generation protocol (Protocol 2) begins with a standard joint random secret sharing generation protocol having two dealers: P_1 and P_c. Given that the blockchain is semi-honest and non-colluding, we can avoid a more expensive coin-tossing protocol. This is a recurring theme we use in all of our protocols. After both P_1 and P_c deal their shares, each party computes their final share of the secret key $[x]_i$ and sends their share of the public key ($X_i := [x]_i \cdot G$) to P_c. Finally, P_c ensures that all shares of the public key are consistent by interpolating in the exponent. If any of the parties cheated, it aborts, otherwise it sends the generated public key X to all parties, which concludes the protocol successfully.

3.2 Signing Protocol

Similarly to key generation, the signature protocol (Protocol 3) begins with a two-dealer random secret-sharing protocol between P_1 and P_c, who jointly generate all required randomness for a single execution. These include t-sharings of fresh random values k, a, and $2t$-sharings of zero, denoted as z, z'. Intuitively, k is the usual ECDSA nonce produced for every signature, and the other values are used internally to mask $2t$-shares that are the product of two t-shares. For concrete efficiency, the protocol does not check consistency of any of these values. In fact, it may even be that the parties hold inconsistent sharings, or that $R \neq [k] \cdot G$. In the proof we show that the adversary cannot learn anything even if it cheats, and so it can only cause an abort.

After the parties obtain these sharings and $r := R.x$, they can locally compute their share of s_1, s_2, such that $[s_1]_i := [a]_i(m + r[x]_i) - [z]_i \mod q$ and $[s_2]_i := [k]_i[a]_i - [z']_i \mod q$. Notice that each s_1 and s_2 has a multiplicative depth of one, meaning that the resulting shares are lifted from a degree t polynomial to a degree $2t$ one. Furthermore, as these shares may no longer be properly random, each party also uses their share of z, z' to rerandomize their resulting shares.

Finally, each party sends $([s_1]_i, [s_2]_i)$ to P_c. After receiving $t+1$ shares, P_c can itself generate additional t shares of these values, and having $2t+1$ total shares of each, reconstruct s_1, s_2 to obtain the final $s := s_1 \cdot s_2^{-1} \mod q$. Finally, if (r, s) is a valid signature, P_c sends it to all parties.

It should be clear that the protocol takes only a single write (for producing the signature) by each party. The only exception is the dealer P_1, who needs to write twice (and can be pre-processed).

Fairness. Our protocol provides fairness, since we make sure that the first party to see a valid signature is P_c, which we know follows the protocol. Therefore, if P_c releases the signature to others, then we know it is indeed a valid signature. Another benefit of our construction is that P_c, a de-facto smart contract, can use incentives (e.g., penalties) to encourage parties to behave correctly and participate in the protocol [43, 44].

We prove the following theorem in Section A.1.

THEOREM 3.1. *Protocols 2-3 securely compute the ECDSA functionality (Functionality 1) with perfect security with abort, against a static malicious adversary who corrupts at most t parties (which are the majority) of $\{P_1, \dots, P_n\}$ or a semi-honest adversary who corrupts P_c.*

Security follows since we can perfectly simulate the adversary's view by picking random values for its shares. One challenge is to align all parties' shares (those of the adversary as well as those of the honest parties) with the values obtained in from the ECDSA functionality (like the public key X, the random nonce R and the

PROTOCOL 3. (*Signing:* Sign $(M, (\mathbb{G}, G, q), sid)$)

Inputs.

(1) Each party P_i, $i \in [1, n]$, holds $([x]_i, X)$.
(2) Party P_c holds X and $[x]_i$ for all $i \in [n+1, N]$.
(3) The parties Compute $m = H_q(M)$ and verify that sid has not been used before (otherwise the protocol is not executed).

The protocol.

(1) **Users' dealing**:
 (a) Party P_1 samples a random $k_u, a_u \leftarrow \mathbb{Z}_q$.
 (b) Party P_1 computes $[k_u] \leftarrow \text{SS.Share}_{t,N}(k_u)$ and $[a_u] \leftarrow \text{SS.Share}_{t,N}(a_u)$.
 (c) Party P_1 computes $[z_u] \leftarrow \text{SS.Share}_{2t,N}(0)$ and $[z'_u] \leftarrow \text{SS.Share}_{2t,N}(0)$.
 (d) Party P_1 sends $([k_u]_i, [a_u]_i, [z_u]_i, [z'_u]_i)$ to party P_i where $i \in [1, n]$ and to P_c where $i \in [n+1, N]$.
 (e) Party P_1 sends $R_u = k_u \cdot G$ to P_c.
(2) **Center's dealing**:
 (a) Party P_c computes $k_c = \mathcal{H}(x_c \| \text{sid})$.
 (b) Party P_c samples a random $a_c \leftarrow \mathbb{Z}_q$.
 (c) Party P_c computes $[k_c] \leftarrow \text{SS.Share}_{t,N}(k_c)$ and $[a_c] \leftarrow \text{SS.Share}_{t,N}(a_c)$.
 (d) Party P_c computes $[z_c] \leftarrow \text{SS.Share}_{2t,N}(0)$ and $[z'_c] \leftarrow \text{SS.Share}_{2t,N}(0)$.
 (e) P_c sends $([k_c]_i, [a_c]_i, [z_c]_i, [z'_c]_i)$ to party P_i for $i \in [1, n]$.
 (f) P_c sends $R = k_c \cdot G + R_u$ to everyone.
(3) **Partial signature.**
 (a) Every party P_i for $i \in [1, n]$, and P_c for $i \in [n+1, N]$:
  (i) Computes $[\alpha]_i = [\alpha_u]_i + [\alpha_c]_i \mod q$, for $\alpha \in \{k, a, z, z'\}$.
  (ii) Computes $[s_1]_i = [a]_i(m + r[x]_i) - [z]_i \mod q$ and $[s_2]_i = [k]_i[a]_i - [z']_i \mod q$.
 (b) P_i for $i \in [1, n]$ sends $(m, [s_1]_i, [s_2]_i)$ to P_c.
(4) **Finalization.** Upon receiving $t + 1$ messages, $\{(m, [s_1]_{i_j}, [s_2]_{i_j})\}_{j=1}^{t+1}$, party P_c:
 (a) Computes $s_1 = \text{SS.Reconstruct}(\{[s_1]_{i_j}\}_{j=1}^{t+1}, \{[s_1]_j\}_{j=n+1}^{N})$ and $s_2 = \text{SS.Reconstruct}(\{[s_2]_{i_j}\}_{j=1}^{t+1}, \{[s_2]_j\}_{j=n+1}^{N})$.
 (b) Computes $s = s_1 \cdot s_2^{-1} \mod q$.
 (c) Broadcasts (r, s) if it is a valid signature on MSG, otherwise it broadcasts $\perp$.

signature s), in which case we first make sure that the adversary's shares are consistent with those values, and then 'interpolate' the other parties' shares to reside on the same, fully determined, polynomial. Another challenge is that P_u picks a secret and shares it first (before this is done by P_c), however, when simulating P_c we need to know P_c's secret (be it x_c in the key generation protocol or k_c in the signing protocol) before simulating P_u's dealing. To this end, in the protocol we instruct P_c to derive its secret from $\mathcal{H}$, which is modeled as a random oracle that is programmable by the simulator. Interestingly, since P_c is semi-honest (and follows the protocol) we can program the random oracle apriori. That is, we can choose the secret values x_c and k_c on behalf of P_c even before it queried the random oracle for them. This was not possible if P_c is malicious, since P_c could have query the random oracle multiple times (or not at all), and the simulator could not know which one was the right one.

From ROM to the standard model. We stress that the protocol can be described in a way that is secure in the standard model, without the random oracle, by having P_c commit to a PRF key as a first step in the key generation protocol, and then this PRF can be used as a random oracle. The simulator extracts that PRF key, as it takes the role of the commitment functionality, and can reproduce any value that P_c produces during the protocol.

4 ROBUST THRESHOLD ECDSA

Note that Protocols 2 and 3 are fair, but not robust. They are fair because either all or none of the parties $P_1, \ldots, P_n$ obtain the result verification key X and signatures. However, robustness is not guaranteed, that is, if P_1 cheats in its dealing then the protocols abort and the parties will not learn the public key or signatures. We can overcome that by using a publicly verifiable secret sharing (cf. Section 2.3) in two different approaches: (1) Let P_1 be the only dealer (apart from P_c) as before, and if it cheats, repeat with P_2 as the dealer, and so on. This process will end by at most $t+1$ writes, as at least one of $P_1, \ldots, P_{t+1}$ is honest; (2) Let all $P_1, \ldots, P_{t+1}$ be dealers simultaneously which ensures that by one write this dealing is complete. While optimistically the first approach entails only one party to write to the blockchain, and hence the overall protocol's message complexity is $O(n)$ (i.e., we consider P_i sending a share to P_j as one message), in the worst case there are $O(t)$ rounds and $O(n^2)$ messages. In the second approach there is still $O(n^2)$ messages, but they are all happen in parallel and so this approach is completed in one round. Protocols 4 and 5 follow the second approach.

Note that ensuring correctness of sharing is not sufficient for robustness - one has to make sure that the computation of $s_1 = a(m + rx)$ and $s_2 = ka$ of the partial signatures by each party are computed correctly. Since these values are the result of a non-linear function, they could not be verified against existing values, m, r, A, K and X, that are already public. To this end, the parties provide additional auxiliary information M_1 and M_2, such that $M_1 = \log(A) \cdot \log(X) \cdot G$ and $M_2 = \log(A) \cdot \log(K) \cdot G$, then, everyone can check that s_1 and s_2 are computed correctly by verifying the equalities $s_1 \cdot G = r \cdot M_1 + m \cdot A$ and $s_2 \cdot G = M_2$. The last piece is verifying that M_1 and M_2 are indeed computed correctly. This can be done by having the parties provide a simple zero-knowledge proof that (A, X, M_1) and (A, K, M_2) are Diffie-Helman tuples (DHT), where the DHT relation is defined by

$$R_{\mathsf{DHT}} = \{(A, B, C) \text{ s.t. } a = \log(A), b = \log(B), ab = \log(C)\}.$$

Note that we use PVSS for the computation of P_c even though it is not needed as P_c is semi-honest, we do this as the interface already gives us the public values required for the messages of parties $1, \ldots, n$ to be publicly verified.

We prove the following in Section A.2.

Theorem 4.1. *Assuming the decisional Diffie-Helman (DDH) problem is hard relative to $(\mathbb{G}, G, q)$, Protocols 4 and 5 securely compute the ECDSA functionality (Functionality 1) with guaranteed outupt delivery, against a static malicious adversary who corrupts at most t parties (which is the majority of) of $\{P_1, \ldots, P_n\}$ or a semi-honest adversary who corrupts P_c.*

In addition to the challenges aforementioned above for the non-robust protocol, which we solve in the same way here, simulating the robust protocol introduces a new challenge because the use of

Shoenmakers's PVSS scheme, which involves El-Gamal encryptions. This extra challenge is introduced only when P_c is corrupted, since when it is not (and we are in the first case in which a subset of $P_1, \dots, P_n$ are corrupted, and so the simulator simulates message arriving from P_c) the simulator has to simulate only P_c's messages, which are not publicly verifiable, but are guaranteed to be correct due to the fact that P_c behaves honestly, thus, there is no need to simulate encryptions of unknown plaintexts. In contrast, when P_c is corrupted, we need to simulate publicly verifiable messages from parties $P_1, \dots, P_{t+1}$, let's focus on one of them, P_u. Then, in the key generation, the simulator knows the public key X (as received from the ECDSA functionality) as well as the complementary part of the public key X_c (which is extracted by the technique described above), therefore the simulator knows $X_u = X - X_c$. However, for a perfect simulation the simulator has to share $x_u = \log(X_u)$ using the PVSS scheme. Now, in contrast to the non-publicly verifiable secret sharing in which each receiver receives its own share only, in PVSS the dealer has to broadcast the encryptions of *all* shares under their respective key, and prove that they are consistent with the commitment of the polynomial. In our case, the simulator does not know x_u and so it cannot produce a polynomial P s.t. $P(0) = x_u$. Instead of providing encryptions of the shares $P(1), \dots, P(N)$, which are obviously unknown to the simulator, the simulator picks random shares $[x_u]_{n+1}, \dots, [x_u]_N$ intended for P_c and encrypts those correctly. Then, the simulator produces the commitment to the polynomial $A_0, \dots, A_t$, where $A_0 = X_u$ since the polynomial must evaluate to x_u at 0, and the values $A_1, \dots, A_t$ are computed from the linear system with t equations and t variables, where the j-th equation is $\sum_{j=0}^{t} i^j \cdot A_j = [x_u]_i \cdot G$. By solving that system the simulator obtains $A_1, \dots, A_t$ and so it has all information required to make all P_c's values be consistent with X and X_c. Finally, for the encryptions of parties $P_1, \dots, P_n$, that are also sent to P_c, the simulator simply encrypts the value $0 \in \mathbb{Z}_q$, which is indistinguishable from an encryption of the actual value $P(i)$ that should have been encrypted, from the CPA-security of El-Gamal.

PROTOCOL 4. (*Robust Key-Generation:* KeyGen)

(1) **User's dealing:** Every P_ℓ, $(\ell \in \{1, \dots, t+1\})$:
 (a) Samples $x_\ell \leftarrow \mathbb{Z}_q$ and computes and broadcasts
 $$(\{c_i^\ell\}_{i=1}^N, \{A_j^\ell\}_{j=0}^t, \pi^\ell) \leftarrow \mathsf{PVSS.Share}_{t,N}(x_\ell).$$
 (b) Let $u \in [1, t+1]$ be the first index for which
 $$1 = \mathsf{PVSS.CheckDealer}(\{c_i^u\}_{i=1}^N, \{A_j^u\}_{j=0}^t, \pi^u).$$
 Denote these values by $\{c_i\}_{i=1}^N, \{A_j\}_{j=0}^t$ (i.e., dropping the supertext u)
(2) **Center's dealing**:
 (a) P_c computes $[x_c] \leftarrow \mathsf{SS.Share}_{t,N}(x_c)$ for $x_c \leftarrow \mathcal{H}(\tilde{x})$ where $\tilde{x} \leftarrow \{0,1\}^\kappa$.
 (b) P_c sends $[x_c]_i$ to P_i for $i \in [1, n]$.
 (c) P_c broadcasts $X = x_c \cdot G + A_0$ and $X_i = [x_c]_i \cdot G + \sum_{j=0}^t i^j \cdot A_j$ for $i \in [1, n]$.
(3) **Compute secret key shares**: Each party P_i computes $[x_u]_i = \mathsf{EG.Dec}_{\mathsf{dk}_i}(c_i)$ and $[x]_i = [x_u]_i + [x_c]_i \mod q$.

5 A SOLUTION FOR A SINGLE USER

So far the chain-assisted protocols were designed to support a *group of signers*, but are not extended to the case in which there is *only one signer*. To see this, observe that for the smallest possible threshold $t = 1$, we need at least two parties that are not P_c. We therefore need to utilize a different protocol between the user and P_c directly. This reduces to a two-party ECDSA protocol between a user P_u and P_c. One of the current state of the art protocols for two-party ECDSA is that of Lindell's [47]. Luckily, when taking into account that our model allows for one of the parties to be semi-honest, we can gain some performance improvements for this setting as well, discussed shortly.

PROTOCOL 5. (*Robust Signing:* Sign $(M, (\mathbb{G}, G, q), sid)$)

Inputs.

(1) Each party P_i, $i \in [1, n]$, holds $([x]_i, X)$.
(2) Party P_c holds X and $[x]_i$ for all $i \in [n+1, N]$.
(3) The parties Compute $m = H_q(M)$ and verify that sid has not been used before (otherwise the protocol is not executed).

The protocol.

(1) **User's dealing:** Every P_ℓ, $(\ell \in \{1, \dots, t+1\})$:
 (a) Samples $k_\ell, a_\ell \leftarrow \mathbb{Z}_q$ and computes and broadcasts
 $$(\{c_{k,i}^\ell\}_{i=1}^N, \{K_j^\ell\}_{j=0}^t, \pi_k^\ell) \leftarrow \mathsf{PVSS.Share}_{t,N}(k_\ell),$$
 $$(\{c_{a,i}^\ell\}_{i=1}^N, \{A_j^\ell\}_{j=0}^t, \pi_a^\ell) \leftarrow \mathsf{PVSS.Share}_{t,N}(a_\ell),$$
 $$(\{c_{z,i}^\ell\}_{i=1}^N, \{Z_j^\ell\}_{j=0}^t, \pi_z^\ell) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0),$$
 $$(\{c_{z',i}^\ell\}_{i=1}^N, \{Z'^\ell_j\}_{j=0}^t, \pi_{z'}^\ell) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0).$$
 (b) Let $u \in [1, t+1]$ be the first index for which
 $$1 = \mathsf{PVSS.CheckDealer}(\{c_{\alpha,i}^u\}_{i=1}^N, \{\alpha_j^u\}_{j=0}^t, \pi_\alpha^u)$$
 for all $\alpha \in \{k, a, z, z'\}$.
(2) **Center's dealing**:
 (a) P_c computes $k_c = \mathcal{H}(x_c \| \mathrm{sid})$, samples $a_c \leftarrow \mathbb{Z}_q$ and computes $[k_c] \leftarrow \mathsf{SS.Share}_{N,t}(k_c)$, $[a_c] \leftarrow \mathsf{SS.Share}_{N,t}(a_c)$, $[z_c] \leftarrow \mathsf{SS.Share}_{N,2t}(0)$, and $[z'_c] \leftarrow \mathsf{SS.Share}_{N,2t}(0)$
 (b) P_c sends $([k_c]_i, [a_c]_i, [z_c]_i, [z'_c]_i)$ to P_i for $i \in [1, n]$
 (c) P_c broadcasts $K = k_c \cdot G + K_0^u$ and (K_i, A_i, Z_i, Z'_i) for all $i \in [1, n]$, where $E_i = [e_c]_i \cdot G + \sum_{j=0}^t i^j \cdot E_j$ for every $(E, e) \in \{(K, k), (A, a), (Z, z), (Z', z')\}$.
(3) **Local computation.**
 (a) P_i $(i \in [1, N])$ computes $[\alpha]_i = [\alpha_u]_i + [\alpha_c]_i \mod q$ for $\alpha \in \{k, a, z, z'\}$, where $[\alpha_u]_i = \mathsf{EG.Dec}_{\mathsf{dk}_i}(c_{\alpha,i}^u)$.
 (b) P_i $(i \in [1, N])$ computes $[s_1]_i = [a]_i(m + r[x]_i) - [z]_i \mod q$ and $[s_2]_i = [k]_i[a]_i - [z']_i \mod q$.
 (c) P_i $(i \in [1, n])$ computes $M_{i,1} = ([a]_i \cdot [x]_i) \cdot G$ and $M_{i,2} = ([a]_i \cdot [k]_i) \cdot G$.
 (d) Everyone computes $r = K.x \mod q$.
(4) **Partial signature.**
 (a) P_i $(i \in [1, n])$ sends $(\mathrm{prove}, \mathrm{sid}\|1, A_i, X_i, M_{i,1}, a_i, x_i)$ and $(\mathrm{prove}, \mathrm{sid}\|2, A_i, K_i, M_{i,2}, a_i, k_i)$ to $\mathcal{F}_{\mathrm{zk}}^{R_{DHT}}$.
 (b) P_i $(i \in [1, n])$ sends $(m, [s_1]_i, [s_2]_i, M_1, M_2)$ to P_c.
(5) **Finalization.** Upon receiving at least $t+1$ messages $(m, [s_1]_i, [s_2]_i, M_{i,1}, M_{i,2})$ for which $[s_1]_i \cdot G = r \cdot M_{i,1} + m \cdot A_i - Z_i$, $[s_2]_i \cdot G = M_{i,2} - Z'_i$, and proofs $(\mathrm{proof}, \mathrm{sid}\|1, A_i, X_i, M_{i,1})$ and $(\mathrm{proof}, \mathrm{sid}\|2, A_i, K_i, M_{i,2})$ were received from $\mathcal{F}_{\mathrm{zk}}^{R_{DHT}}$, denote these indices by I. Then party P_c:
 (a) Computes $s_1 = \mathsf{SS.Reconstruct}(\{[s_1]_i\}_{i \in I}, \{[s_1]_j\}_{j=n+1}^N)$ and $s_2 = \mathsf{SS.Reconstruct}(\{[s_2]_i\}_{i \in I}, \{[s_2]_j\}_{j=n+1}^N)$.
 (b) Broadcasts $s = s_1 \cdot s_2^{-1} \mod q$.

First note that the functionality is a bit different than a typical 2PC ECDSA: since P_c is only an assistant, party P_u is the only one who can ask for key generation or signatures. The formal description appears in Functionality 8 (Section B). Second, note that we employ the same technique for extracting P_c's secret inputs x_c, k_c as done in the multiparty protocols above. As explained, however, this technique can be replaced with a standard model technique using a commitment on a PRF key. Third, since P_c is semi-honest in our model, and so it is guaranteed to choose its nonce randomly and independently of P_u's message, which is not the case in Lindell's protocol. This way, in our model the two-party protocol enjoys *non-interactive signing*, or in other words, requires only one write. As briefly discussed below, that fact also enables simulation of both parties without the additional non-standard 'Paillier-EC' assumption that is used in [47]. The reason for that is that we assign P_c the role of the party who performs the linear evaluation on the encryption of P_u's secret key share (c_{key}). Now, since P_c follows the protocol's description, it is guaranteed to not cheat and produce an encryption of $(k_c)^{-1}(m+xr)$ exactly as described. This removes the need of (1) guessing whether P_c will abort or not, (2) adding an expensive zero-knowledge proof on P_c's last message, or (3) relying on a non-standard assumption as Paillier-EC. Except of the changes mentioned above, our protocol resembles that of Lindell.

We prove the following theorem in Section A.3.

Theorem 5.1. *Protocols 6 and 7 securely compute the ECDSA functionality (Functionality 8) against a static malicious adversary who corrupts P_u or a semi-honest adversary who corrupts P_c.*

PROTOCOL 6. (*Two-Party Key-Generation:* KeyGen)

(1) **P_c's randomness setup.**
  (a) P_c picks a random value $v \leftarrow \{0,1\}^\kappa$ and computes $v_x = \mathcal{H}(v)$.
  (b) P_c sends v_x to P_u.
(2) **Party P_u's message**:
  (a) P_u samples a random $x_u \leftarrow \mathbb{Z}_q^*$ and computes $X_u = x_u \cdot G$.
  (b) P_u generates a Paillier key-pair (pk, sk) where $pk = N = P \cdot Q$ with κ'-bit primes P, Q, and computes $c_{key} = \mathsf{Enc}_{pk}(x_u)$. ($\kappa'$ is the bit-length of the factors of N for the Paillier encryption scheme to be secure).
  (c) P_u sends X_u, $pk = N$ and c_{key} to P_c.
  (d) P_u proves in zero-knowledge that $N \in L_P$ and that it knows a witness (x_u, P, Q) such that $(c_{key}, N, X_u) \in L_{PDL}$, by sending $(\mathsf{prove}, c_{key}, N, X_u, x_u, P, Q)$ to $\mathcal{F}_{zk}^{\mathsf{keygen}}$.
(3) **Party P_c's message**: Upon receiving $(\mathsf{proof}, c_{key}, N, X_u)$ from $\mathcal{F}_{zk}^{\mathsf{keygen}}$:
  (a) Verify that $c_{key} \in \mathbb{Z}_{N^2}^*$ and that N is of length at least $2\kappa'$.
  (b) P_c computes $x_c = \mathcal{H}(v \| \mathsf{keygen})$, and $X_c = x_c \cdot G$ and $X = x_c \cdot X_u$.
  (c) Send X to P_u.
(4) **Output:**
  (a) P_u outputs (pk, sk, x_u, X).
  (b) P_c outputs (pk, x_c, X, c_{key}).

PROTOCOL 7. (*2P Signing:* Sign $(M, (\mathbb{G}, G, q), sid)$)

Inputs.

(1) Party P_u holds (pk, sk, x_u, X).
(2) Party P_c holds (pk, x_c, X, c_{key}).
(3) The parties Compute $m = H_q(M)$ and verify that sid has not been used before (otherwise the protocol is not executed).

The protocol.

(1) **Party P_u's message**:
  (a) P_u chooses $k_u \leftarrow \mathbb{Z}_q$ and computes $R_u = k_u \cdot G$.
  (b) P_u sends R_u to P_c.
  (c) P_u sends $(\mathsf{prove}, \mathsf{sid}, R_u, k_u)$ to $\mathcal{F}_{zk}^{\mathsf{DL}}$ to proves knowledge of k_u.
(2) **P_c's message:** Upon receiving $(\mathsf{proof}, \mathsf{sid}, R_u)$ from $\mathcal{F}_{zk}^{\mathsf{DL}}$:
  (a) P_c computes $k_c = \mathcal{H}(v \| \mathsf{sid})$ and computes $R = k_c \cdot R_u$ and $r = R.x \mod q$.
  (b) P_c chooses $\rho \leftarrow \mathbb{Z}_{q^2}$ and $\tilde{r} \leftarrow \mathbb{Z}_N^*$.
  (c) P_c computes:
    (i) $c_1 = \mathsf{Enc}_{pk}(\rho q + [(k_c)^{-1} m \mod q], \tilde{r})$,
    (ii) $v = (k_c)^{-1} \cdot r \cdot x_c \mod q$,
    (iii) $c_2 = c_1 \oplus (v \odot c_{key})$
  (d) P_c sends R and c_2 to P_u.
(3) **Output:**
  (a) P_u computes $s' = (k_u)^{-1} \cdot \mathsf{Dec}(sk, c_2) \mod q$ and $r = R.x \mod q$.
  (b) P_u outputs (r, s) where $s = \min(s', q - s')$.

6 APPLICATIONS

Unstoppable wallets serve as a foundational component for a diverse array of applications. To demonstrate their applicability, we developed and implemented two examples of applications that possess real-world value. These applications were deployed to Secret Network's mainnet under contract addresses: *secret1lge6kdh078u7yc778whz8wjdc39ce78knqjfjh* *secret1lkvhyg4723fxreeyrm0mk7pkzgd4qaztmx4ztw*. At their core, these wallets are governed by a smart contract, meaning that they may have all kinds of other use-cases as well.

6.1 Multisignature Wallet with Policy Checks

In the traditional banking system, accounts often have various checks and limits on spending to enhance security and control. One can imagine a similar use case for cryptocurrency transactions, integrating such checks and constraints within a multisignature wallet.

Threshold ECDSA inherently supports a multisignature transaction approval structure already, necessitating $(t+1)$-out-of-n parties to endorse signing a transaction. On top of this, with unstoppable wallets, we can introduce further layers of spending policies into the smart-contract component of the protocol, such as per-transaction spending limits, daily spending limits, or a combination of both. These policies offer increased control and security over transactions involving cryptocurrency.

One can think of more elaborate schemes and use-cases as well, that clearly benefit from the blockchain's role as a public bulletin board. For example, decentralized autonomous organizations (DAOs) are often assumed to be governed by all token holders, but their treasuries are in practice controlled by a small committee of signers[6]. By leveraging unstoppable wallets, the community could define clear spending limits in a smart contract to prevent a DAO committee from abusing their mandate.

[6]As a concrete example, as of Sep, 2022, Frax treasury of 1.2B USD was unilaterally controlled by the team's multisig (https://www.blockworksresearch.com/research/risk-assessment-frax-governance).

To demonstrate the concept of a multisig wallet with policy checks, we developed a contract that not only requires a quorum of at least $t+1$ approvals, but also verifies the transaction as a valid Ethereum transaction with a spending limit of 1 ETH. The flow of producing a spending transaction in this scenario is illustrated in Figure 2.

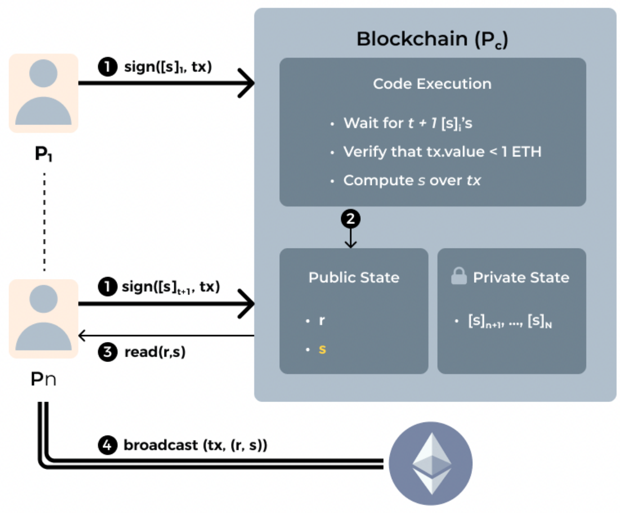


Figure 2: Multisignature Wallet with Policy Checks

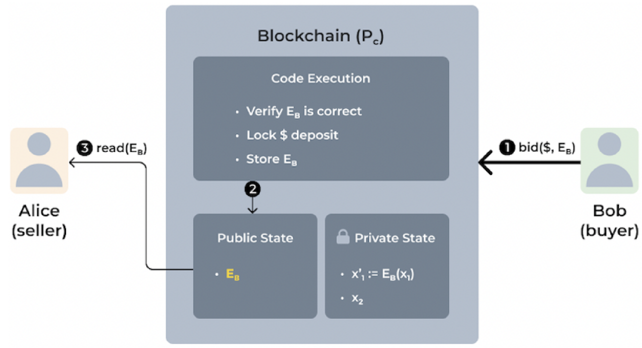
6.2 Wallet Exchange

Typically, users exchange cryptocurrencies, such as swapping BTC for ETH between two parties. However, here we propose an alternative model: instead of exchanging assets, what if we could exchange the wallet itself directly? This concept, a wallet exchange, is not merely theoretical. For instance, venture capital funds often enter illiquid deals for tokens that do not yet exist or have a certain lockup, making selling the asset itself infeasible.

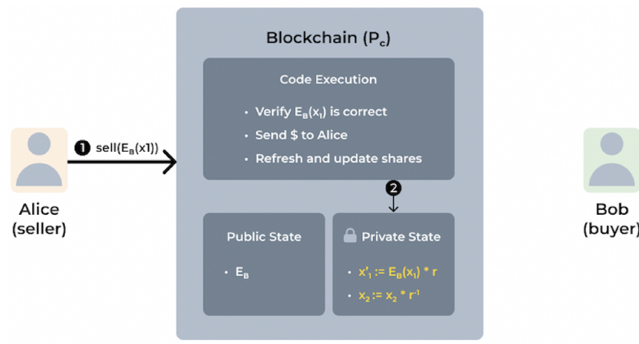
One could envision a wallet exchange platform that allows sellers to list their wallets instead of their assets, and sell these to buyers, who can be reassured that the seller provably loses access after the transaction concludes. In light of the recent collapse of large exchanges and centralized lenders like FTX and Celsius, an exchange that allows creditors to sell their claims (likely at a discount) becomes more appealing. Such exchanges have already started to emerge[7], and a wallet exchange mechanism could provide a more secure way to facilitate this process.

Equipped with this motivation, we present an implementation of a contract that enables selling a wallet from the current owner (the seller) to an interested buyer. Initially, the wallet is jointly held by the seller and the chain. A prospective buyer can send a bid to the contract governing the wallet, which the seller can either accept or ignore. The buyer can set a timeout to release their deposited bid if they have not received a response from the seller after some time.

If the seller accepts the bid, they must re-encrypt their share of the key with the buyer's key and send it to the contract in a separate transaction that concludes the sale. The chain, after verifying that neither party has cheated, assists in refreshing the shares and revoking the seller's share. The contract also atomically finalizes the payment, completing the wallet exchange process securely. This contract flow is illustrated in Figure 3.



(a) Step 1: Buyer initiates a bid for the wallet



(b) Step 2: Seller approves the sale

Figure 3: Wallet Exchange Application Flow

7 IMPLEMENTATION AND EVALUATION

We provide an overview of the implementation and evaluation of our proposed underlying threshold ECDSA protocols. We implement the main threshold ECDSA protocol in 2, 3, and the protocol for a single user. Using these as building blocks, we implement the applications discussed in Section 6. We also discuss the practical aspects of implementing cryptographic primitives on a confidential smart-contract enabled blockchain and delve into the performance analysis of our approach in terms of gas costs associated with on-chain transactions, which is the main performance bottleneck in addition to the number of consecutive writes (i.e., transactions) each user has to perform.

7.1 Implementation Details

Our implementation is tied and optimized for the *secp256k1* curve, as that is the most commonly used curve related to cryptocurrencies. However, our protocols are generic and our implementation can be extended to support other curves as well. The implementation is divided into two main parts: the local execution by users, and the on-chain execution on the blockchain. Our code is written in Rust,

[7]https://opnx.com/

but it is important to note that any language could be used for the client.

For the on-chain part of our proposed protocols, the spectrum of options is more constrained, as we needed a blockchain that supports confidential smart contracts. We chose the Secret Network [55], a blockchain platform that relies on TEEs for confidentiality and has been running in production for several years. Another benefit of choosing Secret Network is that it exemplifies a blockchain that guarantees correctness, availability and privacy with *different levels of confidence*. Namely, while the system's correctness and availability guarantees have never been broken, its privacy guarantees were broken multiple times due to attacks on the underlying hardware (e.g., [15, 60]). Attacks on privacy (but not on the availability nor the correctness) may happen even when confidential smart contracts are implemented using (publicly auditable) MPC protocols. In such cases corrupted parties may 'silently' break privacy, but cannot break correctness or availability. This is the source for our motivation to not store the entire signing key within the smart contract.

We made our implementation open-source [8]. Including our modifications below to existing repositories ,it consists of roughly 6,500 lines of code.

7.2 Cryptographic Primitives on Chain

In order to allow our protocols to run inside of a smart contract, we needed to implement several cryptographic building blocks in a way that allows them to run on-chain. In particular, we needed libraries that support secret sharing (over secp256k1's specified field), elliptic curve operations (over the same curve), and Paillier encryption.

This turned out to be especially challenging, since we had to make sure these building blocks are efficient, do not use randomness generated by the operating system, and do not use floating-point types. The last two are practical constraints present in any blockchain environment, which needs to be deterministic due to consensus. Porting existing cryptographic libraries was especially challenging, since practically all libraries need to generate randomness at one point, and this issue propagates up the dependency tree. We modified all relevant libraries to take in a custom PRG instead, and we used that as a hook to plug in a deterministic PRG that is purpose-built for Secret Network contracts. Overall, we modified approximately 1,350 lines of code across five open-source repositories.

7.3 Performance Evaluation

In this subsection, we assess the performance of our proposed threshold ECDSA protocols by focusing on the gas costs associated with on-chain transactions. Gas costs represent the computational resources necessary to execute a transaction on a blockchain, and are a popular cost metric on all smart-contracts chains, starting with Ethereum [17]. These costs not only impact users monetarily but also impose limitations on the number of gas-intensive transactions a blockchain can process in a single block, as blockchains have inherent constraints in terms of computational resources.

7.3.1 Multiparty Protocol Evaluation. In Table 2a we show an evaluation for $n = 5$, $t = 4$. *init* marks the contract's initialization (for each wallet we deploy a different contract), *keygen* is the dealing portion of the key generation protocol, *presig* marks the dealing part of the signing protocol where shared randomness and the nonce are produced, and *sign_i* marks the cost for each signing party. On a per user basis, the costs are negligible at the time of writing, and amount to roughly one-tenth of a cent per user (with the exception of the dealer who pays roughly three-tenths of a cent). Since the actual cost was calculated based on the price of SCRT, a volatile asset used to pay fees in Secret Network, it is also useful to compare the unitless gas used metric between threshold wallets and other common types of smart contract executions. We reference these in Table 3 and note that surprisingly our results are very appealing given that we have essentially implemented an MPC protocol on-chain.

Table 2: Benchmarks for Multiparty and Two-party ECDSA

(a) Table (a)

Tx Type	Time (ms)	Tx size (bytes)	Gas Used	Tx Cost (¢)
init	0.07	43	45,227	0.04¢
Keygen	7.93	1,206	132,792	0.11¢
Presig	11.65	4,335	237,195	0.19¢
Sign_1	1.62	295	138,865	0.11¢
Sign_2	1.55	295	140,599	0.11¢
Sign_3	1.51	295	142,328	0.11¢
Sign_4	1.85	295	144,046	0.12¢
Sign_5	12.95	295	187,238	0.15¢

(b) Table (b)

Tx Type	Time (ms)	Tx size (bytes)	Gas Used	Tx Cost (¢)
Keygen	175.35	2,707	856,051	0.68¢
Sign	313.75	287	1,882,619	1.51¢

Table 3: Gas cost baselines

Tx Type	Gas Used
Token transfer	55,877
NFT Mint/Transfer	150,833
Token Swap (direct)	595,916
Token Swap (2-hops)	1,553,937

We also found that costs scale very well (practically linearly, as expected) with the number of parties, making this scheme highly efficient in terms of scalability. We capture this close-to-linear relation in Figure 4, which examines how the average gas expenditure changes (on average) per party, as we increase the number of parties (and assume the maximum corruption threshold of $n = t - 1$). We make the same comparison for a fixed $n = 15$ and a dynamic threshold in Figure 5, and reach a similar result.

7.3.2 Two-Party Protocol Evaluation. Interestingly, as shown in 2b, our performance evaluation reveals that the multiparty protocol, even when accommodating numerous parties, incurs significantly lower costs per party compared to the two-party protocol. This

[8] https://github.com/scrtlabs/unstoppable-secrets

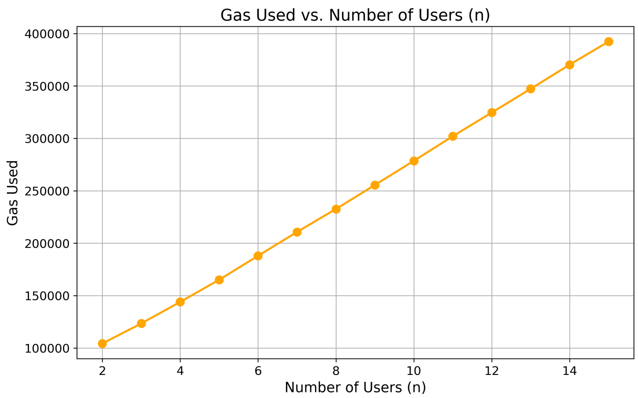


Figure 4: Gas Used vs. Number of Users (n)

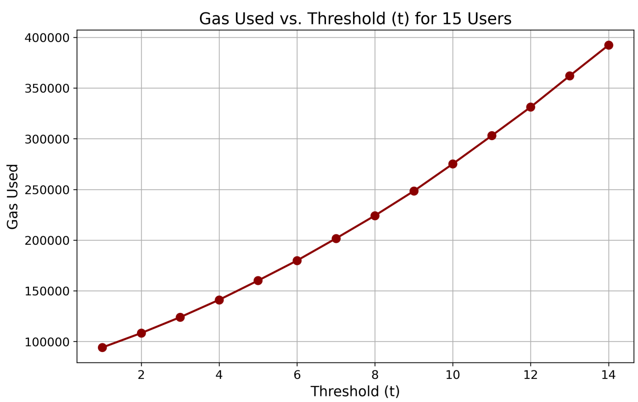


Figure 5: Gas Used vs. Threshold (t) for 15 users

finding can be attributed to the relatively resource-intensive Paillier Encryption used in the two-party protocol, which is used for a single user. It is also worth mentioning that we have not implemented the expensive zero-knowledge proofs necessary for this protocol on-chain, which would undoubtedly widen the gap even more. Based on our results, and assuming the maximum amount of corruptions, we extrapolate that it would take around $n = 82$ users for the gas costs of the multiparty protocol to match the two party one.

Also, given current gas limits in Secret Network, and given that state-of-the-art multiparty threshold ECDSA protocols (e.g., [18]) requires even more homomorphic operations and many more zero-knowledge proofs, it is fair to assume any existing multiparty variant would not even run on-chain. These results support the need of devising chain-friendly threshold ECDSA protocols, as demonstrated in this paper.

ACKNOWLEDGMENTS

We wish to thank Itzik Grossman and Assaf Morami from SCRT Labs for their contribution to the implementation of the results in this paper.

REFERENCES

[1] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. 2022. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2554–2572.

[2] Aritra Banerjee, Michael Clear, and Hitesh Tewari. 2021. zkhawk: Practical private smart contracts from mpc-based hawk. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 245–248.

[3] Carsten Baum, James Hsin-yu Chiang, Bernardo David, and Tore Kasper Frederiksen. 2022. Eagle: Efficient Privacy Preserving Smart Contracts. *Cryptology ePrint Archive* (2022).

[4] Carsten Baum, Ivan Damgård, and Claudio Orlandi. 2014. Publicly auditable secure multi-party computation. In *Security and Cryptography for Networks: 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings 9*. Springer, 175–196.

[5] Carsten Baum, Bernardo David, and Rafael Dowsley. 2020. Insured MPC: Efficient secure computation with financial penalties. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 404–420.

[6] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. 2020. Efficient constant-round MPC with identifiable abort and public verifiability. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II*. Springer, 562–592.

[7] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. 2020. Can a public blockchain keep a secret?. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*. Springer, 260–290.

[8] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2019. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1521–1538.

[9] Iddo Bentov and Ranjit Kumaresan. 2014. How to use bitcoin to design fair protocols. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II 34*. Springer, 421–439.

[10] Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. 2022. Efficient Asymmetric Threshold ECDSA for MPC-based Cold Storage. *IACR Cryptol. ePrint Arch.* (2022), 1296.

[11] Dan Boneh, Rosario Gennaro, and Steven Goldfeder. 2019. Using level-1 homomorphic encryption to improve threshold DSA signatures for bitcoin wallet security. In *Progress in Cryptology–LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America, Havana, Cuba, September 20–22, 2017, Revised Selected Papers*. Springer, 352–377.

[12] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 947–964.

[13] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. 2018. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *arXiv preprint arXiv:1805.08541* (2018).

[14] Russell Brandom. [n. d.]. Impersonation Attack on MyEtherWallet. https://www.theverge.com/2018/4/24/17275982/myetherwallet-hack-bgp-dns-hijacking-stolen-ethereum

[15] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[16] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. 2020. Zether: Towards privacy in a smart contract world. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers*. Springer, 423–443.

[17] Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/.

[18] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. 2020. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1769–1787.

[19] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2019. Two-party ECDSA from hash proof systems and efficient instantiations. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 191–221.

[20] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2020. Bandwidth-efficient threshold EC-DSA. In *Public-Key Cryptography–PKC 2020: 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4–7, 2020, Proceedings, Part II*. Springer, 266–296.

[21] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2023. Bandwidth-efficient threshold EC-DSA revisited: Online/offline extensions, identifiable aborts proactive and adaptive security. *Theoretical Computer Science* 939 (2023), 78–104.

[22] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 185–200.

[23] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. 2021. Fluid MPC: secure multiparty computation with dynamic participants. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41*. Springer, 94–123.

[24] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. 2017. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 719–728.

[25] Richard Cleve. 1986. Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract). In *STOC*, Juris Hartmanis (Ed.).

[26] Anders Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. 2020. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part II 25*. Springer, 654–673.

[27] Ivan Damgård, Thomas P Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergaard. 2022. Fast threshold ECDSA with honest majority. *Journal of Computer Security* 30, 1 (2022), 167–196.

[28] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. 2022. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2518–2534.

[29] Didem Demirag and Jeremy Clark. 2021. Absentia: Secure Multiparty Computation on Ethereum. In *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 381–396.

[30] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. 2018. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 980–997.

[31] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. 2019. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1051–1066.

[32] Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. 2022. POSE: Practical Off-chain Smart Contract Execution. *arXiv preprint arXiv:2210.07110* (2022).

[33] Adam Gągol, Jędrzej Kula, Damian Straszak, and Michał Świętek. 2020. Threshold ecdsa for decentralized asset custody. *Cryptology ePrint Archive* (2020).

[34] Rosario Gennaro and Steven Goldfeder. 2018. Fast multiparty threshold ECDSA with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1179–1194.

[35] Rosario Gennaro and Steven Goldfeder. 2020. One round threshold ECDSA with identifiable abort. *Cryptology ePrint Archive* (2020).

[36] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 1996. Robust threshold DSS signatures. In *Advances in Cryptology-EUROCRYPT'96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings 15*. Springer, 354–371.

[37] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. 2021. YOSO: You Only Speak Once: Secure MPC with Stateless Ephemeral Roles. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II*. Springer, 64–93.

[38] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. 2022. Storing and retrieving secrets on a blockchain. In *Public-Key Cryptography–PKC 2022: 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8–11, 2022, Proceedings, Part I*. Springer, 252–282.

[39] Vipul Goyal, Elisaweta Masserova, Bryan Parno, and Yifan Song. 2021. Blockchains enable non-interactive MPC. In *Theory of Cryptography: 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8–11, 2021, Proceedings, Part II 19*. Springer, 162–193.

[40] Banashri Karmakar, Nishat Koti, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. 2023. sfAsterisk: Super-fast MPC with a Friend. *IACR Cryptol. ePrint Arch.* (2023), 1098.

[41] Uri Kirstein, Shelly Grossman, Michael Mirkin, James Wilcox, Ittay Eyal, and Mooly Sagiv. 2021. Phoenix: A formally verified regenerating vault. *arXiv preprint arXiv:2106.01240* (2021).

[42] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 839–858.

[43] Ranjit Kumaresan and Iddo Bentov. 2016. Amortizing secure computation with penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 418–429.

[44] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. 2016. Improvements to secure computation with penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 406–417.

[45] Rujia Li, Qin Wang, Qi Wang, David Galindo, and Mark Ryan. 2022. SoK: TEE-assisted confidential smart contract. *arXiv preprint arXiv:2203.08548* (2022).

[46] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 63–79.

[47] Yehuda Lindell. [n. d.]. Fast Secure Two-Party ECDSA Signing. In *CRYPTO, 2017 (Lecture Notes in Computer Science, Vol. 10402)*. 613–644.

[48] Yehuda Lindell and Ariel Nof. 2018. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1837–1854.

[49] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 887–903.

[50] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. 2019. CHURP: dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2369–2386.

[51] Ofri Nevo, Ni Trieu, and Avishay Yanai. 2021. Simple, Fast Malicious Multiparty Private Set Intersection. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1151–1165.

[52] Torben P Pedersen. 1991. A threshold cryptosystem without a trusted party. In *Advances in Cryptology-EUROCRYPT'91*. Springer, 522–526.

[53] Marc Rivinius, Pascal Reisert, Daniel Rausch, and Ralf Küsters. 2022. Publicly accountable robust multi-party computation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2430–2449.

[54] Berry Schoenmakers. 1999. A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic. In *CRYPTO*, Vol. 1666. Springer, 148–164.

[55] SCRT. 2021. The Secret Network Graypaper. https://scrt.network/graypaper.

[56] Shraddha Sharma. [n. d.]. Crypto Custodian Prime Trust Frozen Amid Massive Fund Deficits and Missing Keys. https://beincrypto.com/crypto-custodian-prime-trust-missing-keys

[57] Ravital Solomon and Ghada Almashaqbeh. 2021. smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption. *Cryptology ePrint Archive* (2021).

[58] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. 2022. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 179–197.

[59] Samuel Steffen, Benjamin Bichsel, and Martin Vechev. 2022. Zapper: Smart Contracts with Data and Identity Privacy. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2735–2749.

[60] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. 2022. SoK: SGX. Fail: How Stuff Get eXposed.

[61] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. 2022. Cobra: Dynamic proactive secret sharing for confidential bft services. In *2022 IEEE symposium on security and privacy (SP)*. IEEE, 1335–1353.

[62] Kristof Gazso Namra Patel Dror Tirosh Shahaf Nacson Tjaden Hess Vitalik Buterin, Yoav Weiss. 2021. Account Abstraction Using Alt Mempool. https://eips.ethereum.org/EIPS/eip-4337. Accessed: May 5, 2023.

[63] Harry WH Wong, Jack PK Ma, Hoover HF Yin, and Sherman SM Chow. [n. d.]. Real Threshold ECDSA. ([n. d.]).

[64] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. 2022. VERI-ZEXE: Decentralized private computation with universal setup. *Cryptology ePrint Archive* (2022).

[65] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. 2021. Efficient online-friendly two-party ECDSA signature. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 558–573.

[66] Hang Yin, Shunfan Zhou, and Jun Jiang. 2019. Phala network: A confidential smart contract network based on polkadot.

[67] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security.* 270–282.

[68] Guy Zyskind, Oz Nathan, et al. 2015. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*. IEEE, 180–184.

[69] Guy Zyskind, Oz Nathan, and Alex Pentland. 2015. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471* (2015).

A SECURITY PROOFS

A.1 Proof of Theorem 3.1

The proof below is separated to the two cases mentioned in the Theorem, for each of which we present a perfect simulation. Note that the use of $\mathcal{H}$ in the protocol is merely to easily extract P_c's randomly chosen x_c. It is possible to remove this random oracle usage by standard commitment techniques. In both cases it is easy to see that the joint distributions of the honest parties' output and the adversary's view in the real and ideal worlds are identically distributed.

Case 1. Let $\mathcal{A}$ be a malicious real world adversary who corrupts P_1 and a subset of $\{P_2, \dots, P_n\}$ of size $t-1$. Denote the set of corrupted parties by C and the rest of the parties by $H = \{P_1, \dots, P_n\} - C$. We present an ideal world adversary $\mathcal{S}$ that does as follows:

- **Key Generation.**
  (1) Send (keygen) to $\mathcal{F}_{\text{ECDSA}}$.
  (2) Run $\mathcal{A}$ internally and simulates all other parties:
    (a) Receive all shares $[x_u]_i$ for all $i \in H \cup [n+1, N]$.
    (b) Reconstruct x_u using the above $|H| + t$ shares.
    (c) If reconstruction fails then send $(\mathsf{keygen}, \mathsf{abort})$ to $\mathcal{F}_{\text{ECDSA}}$ and halt. Otherwise, compute $[x_u]_i$ for all $P_i \in C$.
    (d) Compute $[x_c] \leftarrow \mathsf{SS.Share}(x_c, t, N)$ for a random $x_c \leftarrow \mathbb{Z}_q$, and send $[x_c]_i$ to every party $P_i \in C$.
    (e) Compute secret key shares $[x]_i = [x_u]_i + [x_c]_i \mod q$ for all $i \in [1, N]$. (Note that this x is not the actual secret key $\log_G(X)$ obtained by the functionality, however, the simulator uses it in order to checks whether the adversary cheats when computing the signature.)
    (f) Receive X_i for all $i \in C$ and compute $X_i = [x]_i \cdot G = ([x_u]_i + [x_c]_i) \cdot G$ for all $i \in H \cup [n+1, N]$.
    (g) Check consistency of all X_i as done in the protocol, if the check fails then send $(\mathsf{keygen}, \mathsf{abort})$ to $\mathcal{F}_{\text{ECDSA}}$ and halt.
    (h) Send $(\mathsf{keygen}, \mathsf{continue})$ to $\mathcal{F}_{\text{ECDSA}}$ and obtain X and H_q.
    (i) Broadcasts X and H_q.
    (j) Output whatever $\mathcal{A}$ outputs.
- **Sign.**
  (1) Send $(\mathsf{sign}, \mathsf{sid})$ to $\mathcal{F}_{\text{ECDSA}}$ and obtain R.
  (2) Run $\mathcal{A}$ internally and simulates all other parties:
    (a) Receive all shares $[k_u]_i$ and $[a_u]_i$ for all $i \in H \cup [n+1, N]$.
    (b) Receive all shares $[z_u]_i$ and $[z'_u]_i$ for all $i \in H \cup [n+1, N]$.
    (c) Receive R_u.
    (d) Sample $k_c, a_c \leftarrow \mathbb{Z}_q$ and compute $[k_c] \leftarrow \mathsf{SS.Share}(k_c, t, N)$, $[a_c] \leftarrow \mathsf{SS.Share}(a_c, t, N)$, $[z_c] \leftarrow \mathsf{SS.Share}(0, 2t, N)$ and $[z'_c] \leftarrow \mathsf{SS.Share}(0, 2t, N)$
    (e) Send $[k_c]_i, [a_c]_i, [z_c]_i, [z'_c]_i$ to P_i for all $i \in C$.
    (f) Compute $[k]_i = [k_u]_i + [k_c] \mod q$ and $[a]_i = [a_u]_i + [a_c] \mod q$ for all $i \in H \cup [n+1, N]$.
    (g) Send R to all $P_i \in C$.
    (h) Receive $[s_1]_i$ and $[s_2]_i$ from all $P_i \in C$.
    (i) Compute $[s_1]_i$ and $[s_2]_i$ using values r, m and the shares $[k]_i, [a]_i, [x]_i$ for all $P_i \in H \cup [n+1, N]$.
    (j) Reconstruct s_1 and s_2 using the the shares received from the adversary (for parties in C) and the shares computed above (for the parties in $H \cup [n+1, N]$). If reconstruction (of a $2t$-degree polynomial) failed then send $(\mathsf{sign}, \mathsf{sid}, \mathsf{abort})$ and halt.
    (k) Check whether r and $s = s_1 \cdot s_2^{-1} \mod q$ is a valid signature on M using the secret key x that was computed in the key-generation phase (recall, this is not the actual secret key used by the functionality).
    (l) If the check fails then send $(\mathsf{sign}, \mathsf{sid}, \mathsf{abort})$ and halt.
    (m) Send $(\mathsf{sign}, \mathsf{sid}, \mathsf{continue})$ and obtain (r, s). Broadcast (r, s) and output whatever $\mathcal{A}$ outputs.

Case 2. Let $\mathcal{A}$ be a semi-honest real world adversary who corrupts P_c. We present an ideal world adversary $\mathcal{S}$ that does as follows:

- **Key Generation.**
  (1) Send (keygen) and $(\mathsf{keygen}, \mathsf{continue})$ to $\mathcal{F}_{\text{ECDSA}}$, and obtain X.
  (2) Run $\mathcal{A}$ internally and simulate parties $(P_1, \dots, P_n)$:
    (a) Sample $x_u \leftarrow \mathbb{Z}_q$, compute $[x_u] \leftarrow \mathsf{SS.Share}(x_u, t, N)$ and send $[x_u]_i$ to P_c, for all $i \in [n+1, N]$.
    (b) Receive $[x_c]_i$ from P_c for all $i \in [1, n]$, reconstruct x_c (always succeeds because $\mathcal{A}$ follows the protocl) and compute $[x_c]_i$ for all $i \in [n+1, N]$.
    (c) Let λ_0^j and $\{\lambda_i^j\}_{i \in [n+1, N]}$ be the Lagrange coefficients for a polynomial evaluation on j, using points at 0 and the indices in $[n+1, N]$ ($t+1$ points in total).
    (d) For every $j \in [1, n]$ compute $X_j = \lambda_0^j \cdot X + \sum_{i \in [n+1, N]} \lambda_i^j \cdot X_i$.
    (e) Send X_j to P_c for every $i \in [1, n]$. (The above computation ensures that the consistency verification goes through.)
    (f) Output whatever $\mathcal{A}$ outputs.
- **Sign.**
  (1) Send $(\mathsf{sign}, \mathsf{sid})$ and $(\mathsf{sign}, \mathsf{sid}, \mathsf{continue})$ to $\mathcal{F}_{\text{ECDSA}}$ and obtain R and (r, s).
  (2) Run $\mathcal{A}$ internally and simulates all other parties:
    (a) Sample $k_u, a_u \leftarrow \mathbb{Z}_q$ and compute $[k_u] \leftarrow \mathsf{SS.Share}(k_u, t, N)$, $[a_u] \leftarrow \mathsf{SS.Share}(a_u, t, N)$, $[z_u] \leftarrow \mathsf{SS.Share}(0, 2t, N)$ and $[z'_u] \leftarrow \mathsf{SS.Share}(0, 2t, N)$
    (b) Send $[k_u]_i, [a_u]_i, [z_u]_i, [z'_u]_i$ to P_c for all $i \in [n+1, N]$.
    (c) Sample $k_c \leftarrow \mathbb{Z}_q$ and program $\mathcal{H}(x_c \| \mathsf{sid}) \leftarrow k_c$.
    (d) Compute $R_c = k_c \cdot G$ and $R_u = R - R_c$.
    (e) Send R_u to P_c.
    (f) Receive all shares $[k_c]_i$ and $[a_c]_i$ for all $i \in [1, n]$.
    (g) Receive all shares $[z_c]_i$ and $[z'_c]_i$ for all $i \in [1, n]]$.
    (h) Receive R.
    (i) Compute $[\alpha]_i = [\alpha_u]_i + [\alpha_c]_i \mod q$ for all $i \in [n+1, N]$ and for all $\alpha \in \{k, a, z, z'\}$.
    (j) Compute $[s_1]_i = [a]_i(m + r[x]_i) - [z]_i \mod q$ and $[s_2]_i = [k]_i[a]_i - [z']_i \mod q$ for all $i \in [n+1, N]$.

(k) Sample random $2t$-degree polynomials S_1 and S_2, such that $S_b(0) = s_b$ and $S_b(i) = [s_b]_i$, for all $i \in [n+1, N]$ and $b \in \{1, 2\}$.
(l) Send $(m, [s_1]_i, [s_2]_i)$ to P_c for all $i \in [1, n]$, where $[s_1]_i = S_1(i)$ and $[s_2]_i = S_2(i)$.

A.2 Proof of Theorem 4.1

The proof below is separated to the two cases mentioned in the Theorem, for each of which we present a perfect simulation. As mentioned above, we use $\mathcal{H}$ as a random oracle in order to easily extract P_c's randomly chosen x_c, but it is possible to replace it with standard commitment techniques.

Case 1. Let $\mathcal{A}$ be a malicious real world adversary who corrupts P_1 and a subset of $\{P_2, \ldots, P_n\}$ of size $t-1$. Without loss of generality, let that subset be $P_1, \ldots, P_t$. We present an ideal world adversary $\mathcal{S}$ that does as follows.

- **Key Generation.**
  (1) Send (keygen) to $\mathcal{F}_{\mathrm{ECDSA}}$, then send $(\mathsf{keygen}, \mathsf{continue})$ to $\mathcal{F}_{\mathrm{ECDSA}}$ and obtain X and H_q.
  (2) Run $\mathcal{A}$ internally and simulates all other parties (knowing their encryption key-pair, so it is possible to decrypt ciphertexts under their key):
    (a) Choose $x_{t+1} \leftarrow \mathbb{Z}_q$, and send

$$(\{c^{\ell}_{t+1}\}_{i=1}^{N}, \{A_j^{t+1}\}_{j=0}^{t}, \pi^{t+1}) \leftarrow \mathsf{PVSS.Share}_{t,N}(x_{t+1}),$$

    to the adversary.
    (b) Receive $(\{c^{\ell}_{\ell}\}_{i=1}^{N}, \{A_j^{\ell}\}_{j=0}^{t}, \pi^{\ell})$ from the adversary for all $\ell \in [1, t]$.
    (c) Let $u \in [1, t+1]$ be the first index for which

$$1 = \mathsf{PVSS.CheckDealer}(\{c_i^u\}_{i=1}^{N}, \{A_j^u\}_{j=0}^{t}, \pi^u).$$

    Denote these values by $\{c_i\}_{i=1}^{N}, \{A_j\}_{j=0}^{t}$ (i.e., dropping the supertext u). Note that there must be such u, as the above certainly holds for $u = t+1$ (as this is the honest party simulated here.
    (d) Extract the secret x_u by decrypting c_i for $t+1$ parties (which is possible because there are at least $t+1$ parties under the control of the simulator). Note that this also enables obtaining $\log(A_j)$ for all $j \in [0, t]$ sent by P_u.
    (e) Compute $[x_c] \leftarrow \mathsf{SS.Share}(x_c, t, N)$ for a random $x_c \leftarrow \mathbb{Z}_q$.
    (f) Send $[x_c]_i$ to the adversary for every $i \in [1, t]$.
    (g) Set $X_0 = X$ and compute $X_i = ([x_u]_i + [x_c]_i) \cdot G$ for every $i \in [1, t]$. Then compute $X_i = \sum_{j=0}^{t} i^j \cdot X_j$ for every $i \in [t+1, n]$.
    (h) Broadcast X and X_i for every $i \in [1, n]$.
    (i) Output whatever $\mathcal{A}$ outputs.
- **Sign.**
  (1) Send $(\mathsf{sign}, \mathsf{sid})$ to $\mathcal{F}_{\mathrm{ECDSA}}$ and obtain R, then send $(\mathsf{sign}, \mathsf{sid}, \mathsf{continue})$ and obtain (r, s).
  (2) Run $\mathcal{A}$ internally and simulates all other parties:
    (a) Choose $k_{t+1}, a_{t+1} \leftarrow \mathbb{Z}_q$, and send to the adversary

$$(\{c_{k,i}^{t+1}\}_{i=1}^{N}, \{K_j^{t+1}\}_{j=0}^{t}, \pi_k^{t+1}) \leftarrow \mathsf{PVSS.Share}_{t,N}(k_{t+1}),$$
$$(\{c_{a,i}^{t+1}\}_{i=1}^{N}, \{A_j^{t+1}\}_{j=0}^{t}, \pi_a^{t+1}) \leftarrow \mathsf{PVSS.Share}_{t,N}(a_{t+1}),$$
$$(\{c_{z,i}^{t+1}\}_{i=1}^{N}, \{Z_j^{t+1}\}_{j=0}^{t}, \pi_z^{t+1}) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0),$$
$$(\{c_{z',i}^{t+1}\}_{i=1}^{N}, \{Z'^{t+1}_j\}_{j=0}^{t}, \pi_{z'}^{t+1}) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0).$$

    (b) For every $i \in [1, t]$, receive from the adversary

$$(\{c_{k,i}^{i}\}_{i=1}^{N}, \{K_j^{i}\}_{j=0}^{t}, \pi_k^{i}) \leftarrow \mathsf{PVSS.Share}_{t,N}(k_i),$$
$$(\{c_{a,i}^{i}\}_{i=1}^{N}, \{A_j^{i}\}_{j=0}^{t}, \pi_a^{i}) \leftarrow \mathsf{PVSS.Share}_{t,N}(a_i),$$
$$(\{c_{z,i}^{i}\}_{i=1}^{N}, \{Z_j^{i}\}_{j=0}^{2t}, \pi_z^{i}) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0),$$
$$(\{c_{z',i}^{i}\}_{i=1}^{N}, \{Z'^{i}_j\}_{j=0}^{2t}, \pi_{z'}^{i}) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0).$$

    (c) Let $u \in [1, t+1]$ be the first index for which all sharings above are verified.
    (d) Denote the public values of P_u by $\{K_j, A_j\}_{j=0}^{t}$ and $\{Z_j, Z'_j\}_{j=0}^{2t}$.
    (e) Extract the values k_u, a_u and z_u, z'_u (the values z_u and z'_u are extractable via the zero knowledge functionality).
    (f) Generate the sharings $[k_c]$, $[a_c]$, $[z_c]$ and $[z'_c]$ as in the protocol, and send the adversary $\{[k_c]_i, [a_c]_i, [z_c]_i, [z'_c]_i\}$ for every $i \in [1, t]$.
    (g) Broadcast R (as received from the ECDSA functionality).
    (h) Set $K_0 = R$ and compute $K_i = ([k_u]_i + [k_c]_i) \cdot G$ for every $i \in [1, t]$. Then compute $K_i = \sum_{j=0}^{t} i^j \cdot K_j$ for every $i \in [t+1, n]$.
    (i) Compute $A_i = ([a_u]_i + [a_c]_i) \cdot G$, $Z_i = ([z_u]_i + [z_c]_i) \cdot G$ and $Z'_i = ([z'_u]_i + [z'_c]_i) \cdot G$ for every $i \in [1, n]$.
    (j) Broadcast (K_i, A_i, Z_i, Z'_i) for every $i \in [1, n]$.
    (k) Send $(\mathsf{proof}, \mathsf{sid}\|1, A_{t+1}, X_{t+1}, M_{t+1,1})$ and $(\mathsf{proof}, \mathsf{sid}\|2, A_{t+1}, K_{t+1}, M_{t+}$ to the adversary, in addition, receive and verify the adversary's proof on its $M_{i,1}, M_{i,2}$ for every $i \in [1, t]$.
    (l) When received $t+1$ messages $([s_1]_i, [s_2]_i, M_{i,1}, M_{i,2})$ for i for which the proof is verified, broadcast the signature (r, s) as received from the ECDSA functionality.
    (m) Output whatever $\mathcal{A}$ outputs.

First note that the honest parties's output are identically distributed in both real and ideal world. We now argue that the adversary's views in both world are computationally indistinguishable. The only difference between the views is that in the simulation the values X_i and K_i for $i \in [t+1, n]$ that are observed by the adversary (since P_c broadcasts them) are not computed correctly by $([x_u]_i + [x_c]_i) \cdot G$ and $([k_u]_i + [k_c]_i) \cdot G$; rather, they are computed (interpolated) directly from the values $X_0, \ldots, X_t$ and $K_0, \ldots, K_t$ (if they were not interpolated this way then it would have been easy to detect this). Now, since the adversary does not have any information about $([x_u]_i + [x_c]_i)$ or $([k_u]_i + [k_c]_i)$ it cannot tell the difference and so the views are identically distributed.

Case 2. Let $\mathcal{A}$ be a semi-honest real world adversary who corrupts P_c. We present an ideal world adversary $\mathcal{S}$ that does as follows:

- **Key Generation.**
  (1) Send (keygen) and $(\mathsf{keygen}, \mathsf{continue})$ to $\mathcal{F}_{\mathrm{ECDSA}}$, and obtain X.
  (2) Run $\mathcal{A}$ internally and simulate parties $(P_1, \ldots, P_n)$:
    (a) Choose $x_c \leftarrow \mathbb{Z}_q$ (on behalf of P_c).

(b) Compute $X_u = X - x_c \cdot G$.
(c) Choose random values $[x_u]_i \leftarrow \mathbb{Z}_q$ and compute $c_i \leftarrow \mathsf{EG.Enc}_{\mathsf{ek}_i}([x_u]_i)$ for $i \in [n{+}1, N]$; and $c_i \leftarrow\leftarrow \mathsf{EG.Enc}_{\mathsf{ek}_i}(1)$ for every other $i \in [1, n]$. Finally compute $A_1, \ldots, A_t$ such that $\sum_{j=0}^{t} i^j A_j = [x_u]_i \cdot G$ for every $i \in [n+1, N]$ (this is a linear system of t equations with t variables).
(d) Broadcast $\{c_i\}_{i=1}^{N}$, $\{A_j\}_{j=0}^{t}$, and π, where π is generated by the HVZK simulator associated with the zero-knowledge proof.
(e) Receive a call to $\mathcal{H}$ from the adversary and respond with x_c chosen above.
(f) Receive $[x_c]_i$ from the adversary for every $i \in [1, n]$.
(g) Receive X and X_i for every $i \in [1, n]$.
(h) Output whatever the adversary outputs.

- **Sign.**
  (1) Send (sign, sid) and (sign, sid, continue) to $\mathcal{F}_{\text{ECDSA}}$ and obtain R and (r, s).
  (2) Run $\mathcal{A}$ internally and simulates all other parties:
  (a) Choose $k_c \leftarrow \mathbb{Z}_q$ (on behalf of P_c).
  (b) Compute $R_u = R - k_c \cdot G$.
  (c) Choose random values $[k_u]_i \leftarrow \mathbb{Z}_q$ and compute $c_i \leftarrow \mathsf{EG.Enc}_{\mathsf{ek}_i}([k_u]_i)$ for $i \in [n{+}1, N]$; and $c_i \leftarrow\leftarrow \mathsf{EG.Enc}_{\mathsf{ek}_i}(1)$ for every other $i \in [1, n]$. Finally compute $K_1, \ldots, K_t$ such that $\sum_{j=0}^{t} i^j K_j = [k_u]_i \cdot G$ for every $i \in [n+1, N]$ (this is a linear system of t equations with t variables).
  (d) Broadcast $\{c_{k,i}\}_{i=1}^{N}$, $\{K_j\}_{j=0}^{t}$, and π_k, where π_k is generated by the HVZK simulator associated with the zero-knowledge proof.
  (e) Choose random $a_u \leftarrow \mathbb{Z}_q$ and compute

$$(\{c_{a,i}\}_{i=1}^{N}, \{A_j\}_{j=0}^{t}, \pi_a) \leftarrow \mathsf{PVSS.Share}_{t,N}(a_u),$$
$$(\{c_{z,i}\}_{i=1}^{N}, \{Z_j\}_{j=0}^{t}, \pi_z) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0),$$
$$(\{c_{z',i}\}_{i=1}^{N}, \{Z'u_j\}_{j=0}^{t}, \pi_{z'}) \leftarrow \mathsf{PVSS.Share}_{2t,N}(0).$$

  (f) Broadcast the PVSS results above.
  (g) Receive a call to $\mathcal{H}$ from the adversary and respond with k_c chosen above.
  (h) Receive $([k_c]_i, [a_c]_i, [z_c]_i, [z'_c]_i)$ from P_i for $i \in [1, n]$, and extract a_c (z_c and z'_c could not be extracted since they are shared using a sharing of degree $2t$).
  (i) Receive K and (K_i, A_i, Z_i, Z'_i) for all $i \in [1, n]$.
  (j) At this point the simulator knows the values $[s_1]_i, [s_2]_i$ for every $i \in [n{+}1, N]$ that are computed by the adversary in the local computation step.
  (k) The simulator generates random sharings of degree $2t$ for random values s_1, s_2 such that: (1) the shares at points $i \in [n+1, N]$ are those computed by the adversary; (2) it holds that $s_1 \cdot s_2^{-1} = s$ and s is the value received from the ECDSA functionality.
  (l) The simulator also compute the values $M_{i,1}, M_{i,2}$ according to the constraints implied in the protocol. Note that these values will not meet the constraints required by the zero-knowledge proof, however, the proof will be successfully verified since it is simulated using the HVZK simulator associated with it.
  (m) The simulator sends $[s_1]_i, [s_2]_i, M_{i,1}, M_{i,2}$ to the adversary for all $i \in [1, n]$.
  (n) Receive s from the adversary and output whatever it outputs.

Note that here the view of the adversary under the simulation is identical to its view in the real world, except the fact that the ciphertext that are published under the encryption keys of parties $P_1, \ldots, P_n$ are incorrect, that is, they encrypt 0 instead of the actual value. That value that should have been encrypted is unknown to the simulator and hence could not be used. This however is computationally indistinguishable by the adversary and hence it will proceed with the protocol exactly as it would have proceed if these ciphertext were encrypting the correct messages, as otherwise we could have used that adversary in order to break the CPA-security of El-Gamal (which relies on the DDH assumption).

A.3 Proof of Theorem 5.1

The two-party $\mathcal{F}_{\text{ECDSA}}$ is slightly different than the one presented in Functionality 1. For the two-party, the functionality works only with P_u, P_c and an adversary $\mathcal{S}$, *who cannot abort the execution* (but is mentioned in the functionality solely to emphasize this). This is possible because the first (and only) message sent in the protocol from P_u to P_c fully determines whether the adversary will abort or not (by verifying the zero-knowledge proofs), and if so, the honest party refuses to participate. In the ideal world, such refusal is expressed by not invoking $\mathcal{F}_{\text{ECDSA}}$ at all. Finally, since this case could not be translated to a honest majority protocol we could not achieve fairness, and only P_u obtains the result signature from the functionality. For completeness, the modified version is presented in Functionality 8.

We separately present a simulator to the case of malicious P_u and semi-honest P_c.

Case 1. Let $\mathcal{A}$ be a malicious real world adversary who corrupts P_u, consider an ideal world adversary $\mathcal{S}$ that does as follows:

- **Key Generation.**
  (1) Run $\mathcal{A}$ internally and simulate the honest party P_c:
  (a) Receive (X_u, pk, c_{key}) and $(\text{prove}, c_{key}, pk, X_u, x_u, P, Q)$ from P_u, set $sk = (P-1)(Q-1)$ and verify that (1) $X_u = x_u \cdot G$, (2) P, Q are primes of length κ', (3) $N = PQ$, (4) $x_u = \text{Dec}(sk, c_{key})$. If verification fails then halt, otherwise continue.
  (b) Send (keygen) to $\mathcal{F}_{\text{ECDSA}}$ and receive X.
  (c) Compute $X_c = (x_u)^{-1} \cdot X_u$ and send X to $\mathcal{A}$.
  (d) Output whatever $\mathcal{A}$ outputs.
- **Sign.**
  (1) Run $\mathcal{A}$ internally and simulate the honest party P_c:
  (a) Receive R_u and $(\text{prove}, \text{sid}, R_u, k_u)$ from P_u, verify that $R_u = k_u \cdot G$. If verification fails then halt, otherwise continue.
  (b) Send $(\text{sign}, \text{sid}, M)$ to $\mathcal{F}_{\text{ECDSA}}$ and receive R and (r, s).
  (c) Choose $\rho \leftarrow \mathbb{Z}_{q^2}$ and $\tilde{r} \leftarrow \mathbb{Z}_N^*$, and compute $c_2 = \text{Enc}(pk, \rho q + [k_u \cdot s \mod q])$, where s is the signature received from $\mathcal{F}_{\text{ECDSA}}$.
  (d) Send c_2 to $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs.

Observe that the view of P_u under simulation and in the real execution are identically distributed, except of the value c_2: in the simulation it is an encryption of $z'_1 = \rho q + [k_u \cdot s \mod q]$ whereas in the real execution it is an encryption of $z_2' = \rho q + [(k_c)^{-1}m \mod q] + [(k_c)^{-1}rx_c \mod q] \cdot x_u$, where ρ is a random value from $\{0, \dots, q^2-1\}$. Denote by z_1, z_2 the values wihtout the addition of a random multiple of q, that is, $z_1 = k_u \cdot s \mod q$ and $z_2 = [(k_c)^{-1}m \mod q] + [(k_c)^{-1}rx_c \mod q] \cdot x_u$. Note that we consider z_1 and z_2 over the integers, rather than over $\mathbb{Z}_q$. In [47] the values z_1' and z_2' are shown to be statistically close (as long as all conditions on X_u, pk and c_{key} are met, which is guaranteed by using an ideal functionality for zero-knowledge). We present this analysis here for completeness.

Consider the real world value z_2, it is an integer result of the addition of an element from $\mathbb{Z}_q$ (namely $(k_c)^{-1}m \mod q$) with the product of of two elements from $\mathbb{Z}_q$ (namely $[(k_c)^{-1}rx_c \mod q] \cdot x_u$), and we know that by reducing that integer modulo q we get $k_u \cdot s \mod q$ (where (r, s) the ECDSA signature on M obtained by the functionality), thus there exists some $\ell \in \mathbb{N}$ such that $[k_u \cdot s \mod q] + \ell \cdot q = z_2$. Also, note that $0 \le \ell < q$ since $z_2 < q(q-1)$, so the difference between the simulation and the real world is:

- Real: ciphertext c_2 encrypts $z_2' = [k_u \cdot s \mod q] + \ell \cdot q + \rho \cdot q$, and
- Simulation: ciphertext c_2 encrypts $z_1' = [k_u \cdot s \mod q] + \rho \cdot q$.

We show that with a random choice of $\rho \in \mathbb{Z}_{q^2}$ the values z_1' and z_2' are statistically close. Fix k_u and s, then for every $0 \le \zeta < q$ define $v = [k_u \cdot s \mod q] + \zeta \cdot q$, we have:

- If $0 \le \zeta < \ell$ then $\Pr[z_1' = v] = 1/q^2$ but $\Pr[z_2' = v] = 0$ (because $z_2' > [k_u \cdot s \mod q] + \ell \cdot q$).
- If $q^2 - 1 < \zeta < \ell + q^2$ then $\Pr[z_2' = v] = \Pr[\rho = q^2 - 1 - \ell] = 1/q^2$ but $\Pr[z_1' = v] = 0$ (because $z_1' \le [k_u \cdot s \mod q] + (q^2-1)q$).
- If $\ell \le \zeta \le q^2 - 1$ then $\Pr[z_1' = v] = \Pr[\rho = \zeta] = 1/q^2$ and $\Pr[z_2' = v] = \Pr[\rho = \zeta - \ell] = 1/q^2$.

We get that $\Delta(z_1', z_2') = \sum_{\zeta=0}^{\ell+q^2-1} \left|\Pr[z_1' = v] - \Pr[z_2' = v]\right| = \frac{2\ell}{q^2}$, which is negligible.

Case 2. Let $\mathcal{A}$ be a semi-honest real world adversary who corrupts P_c, consider an ideal world adversary $\mathcal{S}$ that does as follows:

- **Key Generation.**
  (1) Run $\mathcal{A}$ internally and simulate the honest party P_u:
    (a) Receive the oracle call and obtain v, forward v to the RO and obtain v_x, forward v_x back to $\mathcal{A}$.
    (b) Receive v_x from $\mathcal{A}$.
    (c) Compute $x_c = \mathcal{H}(v \| \text{keygen})$, $X_c = x_c \cdot G$ and $X_u = (x_c)^{-1} \cdot X$.
    (d) Generate a Paillier key-pair (pk, sk) where $pk = N = P \cdot Q$, with κ'-bit primes P, Q, and compute $c_{key} = \mathsf{Enc}(pk, 0)$.
    (e) Send (X_u, pk, c_{key}) and $(\text{proof}, c_{key}, N, X_u)$ to P_c.
    (f) Send (proof,
    (g) Receive X from $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs.
- **Sign.**
  (1) Run $\mathcal{A}$ internally and simulate the honest party P_c:
    (a) Receive R from $\mathcal{F}_{\text{ECDSA}}$.
    (b) Compute $k_c = \mathcal{H}(v \| \text{sid})$, and computes $R_u = (k_c)^{-1} \cdot R$.
    (c) Send R_u and $(\text{proof}, \text{sid}, R_u)$ to $\mathcal{A}$.
    (d) Receive c_2 from $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs.

The views of $\mathcal{A}$ in the real execution and under the simulation of the key generation protocol are computationally indistinguishable: the value X_u (and therefore X) are identically distributed in $\mathbb{G}$ and the key-pairs generated in both worlds are identically distributed. The only difference is in the generation of ciphertext c_{key}: in the real execution this is an encryption of x_u and in the simulation this is an encryption of zero, and since Paillier encryption scheme is CPA-secure it follows that that the two views are computationally indistinguishable.

In addition the views of $\mathcal{A}$ in the real execution and under the simulation of the signing protocol are identically distributed, in both cases it only receives R_u and $(\text{proof}, \text{sid}, R_u)$, such that $k_c \cdot R_u = R$, with R chosen by the functionality. Note that unlike in [47], since we assume $\mathcal{A}$ is semi-honest it always reply with a ciphertext that holds a correct evaluation on c_{key} and so we do not need to guess whether to abort or not, neither to rely on the 'Paillier-EC' assumption [47, Def. 5.2].

B FUNCTIONALITY FOR TWO-PARTY ECDSA

FUNCTIONALITY 8. (*2P ECDSA Functionality: $\mathcal{F}_{ECDSA}$*)
The functionality is parameterized with the ECDSA group description $(\mathbb{G}, G, q)$ and works with parties P_u, P_c, and an adversary $\mathcal{S}$ as follows.

- Upon receiving (keygen) from P_u:
  (1) Generate an ECDSA key-pair (X, x) by choosing a random $x \leftarrow \mathbb{Z}_q^*$ and computing $X = x \cdot G$.
  (2) Choose a hash function $H_q : \{0,1\} \to \{0,1\}^{\lfloor \log q \rfloor}$.
    (a) Store (H_q, x).
    (b) Output X to P_u and P_c.
    (c) Ignore future calls to keygen.
- Upon receiving $(\text{sign}, \text{sid}, M)$ from P_u, if keygen was already called and sid was not already used:
  (1) Choose a random $k \in \mathbb{Z}_q^*$
  (2) Compute $R \leftarrow k \cdot G$ and let $r = R.x \mod q$; then send R to P_u and P_c.
  (3) Let $m = H_q(M)$. Compute $s \leftarrow k^{-1}(m + rx) \mod q$.
  (4) Send (r, s) to P_u and $\mathcal{S}$.

C ZERO KNOWLEDGE PROOF OF KNOWLEDGE

For an NP-relation R, we use the $\mathcal{F}_{\mathsf{zk}}^R$ functionality (Functionality 9 below). The protocols we use to realize $\mathcal{F}_{\mathsf{zk}}^R$ are public coin, therefore they can be instantiated with a non-interactive version in the random oracle model via the Fiat-Shamir transform.

FUNCTIONALITY 9. (*The ZKPoK Functionality: $\mathcal{F}_{\mathsf{zk}}^R$*)
The functionality works with a prover $\mathcal{P}$ and verifiers $\vec{\mathcal{V}}$.

- Upon receiving $(\text{prove}, \text{sid}, x, w)$ from $\mathcal{P}$, if $(x, w) \in R$ and sid has never been used before, send $(\text{proof}, \text{sid}, x)$ to $\vec{\mathcal{V}}$.

Received 22 August 2023; accepted 16 November 2023