A Multiparty Computation Approach to Threshold ECDSA

Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat
Northeastern University

December 17, 2018

Abstract

The Elliptic Curve Digital Signature Algorithm (ECDSA) is one of the most widely used schemes in deployed cryptography. Through its applications in code and binary authentication, web security, and cryptocurrency, it is likely one of the few cryptographic algorithms encountered on a daily basis by the average person. Standardizing a design for a threshold variant of ECDSA will be significant progress toward standardizing building blocks for threshold cryptosystems at large.

However, the design of ECDSA is such that executing multi-party or threshold signatures in a secure manner is challenging: unlike other, less widespread signature schemes, secure multi-party ECDSA requires custom protocols, which has heretofore implied reliance upon additional cryptographic assumptions and primitives such as the Paillier cryptosystem.

This paper reports on new protocols (appearing in [DKLs18, DKLs19]) for multi-party ECDSA key-generation and signing with arbitrary thresholds, that are secure against malicious adversaries in the Random Oracle Model assuming only the Computational Diffie-Hellman Assumption. We instantiate our protocols using the same hash function and elliptic curve group used by the ECDSA signature being computed. Our threshold t scheme requires $\log(t) + 6$ rounds of communication with scope for adjustment to constant rounds if desired, and when $t = 2$ we provide an optimized two message protocol.

We evaluate our implementations and find that the wall-clock time for computing a signature through our two-party protocol comes to within a factor of 18 of local signatures. Concretely, two parties can jointly sign a message in just over three milliseconds.

We also demonstrate the feasibility of signing with a low-power device (as in the setting of 2-factor authentication) by computing a signature between two Raspberry Pi devices in under 60 milliseconds.

1 Introduction

Threshold Digital Signature Schemes, a classic notion in the field of Cryptography [Des87], allow a group of individuals to delegate their joint authority to

sign a message to any subcommittee among themselves that is larger than a certain size. Though extensively studied, threshold signing is seldom used in practice, in part because until recently, threshold techniques for standard signatures tended to be highly inefficient, reliant upon unacceptable assumptions, or otherwise undesirable, while bespoke threshold schemes continue incompatible with familiar and widely-accepted standards.

Consider the specific case of the Elliptic Curve Digital Signature Algorithm (ECDSA), perhaps the most widespread of signatures schemes: almost all existing threshold techniques for generating ECDSA signatures require the invocation of heavy cryptographic primitives such as Paillier encryption [Lin17, GGN16, BGG17, GG18]. This leads both to poor performance and to reliance upon assumptions that are foreign to the mathematics on which ECDSA is based.

This is troublesome, because performance concerns and avoidance of certain assumptions often motivate the use of ECDSA in the first place. To address this probem, in [DKLs18, DKLs19] we propose an approach to computing threshold ECDSA signatures that makes use of Multiparty Computation (MPC) techniques, augmented by a simple linear check in the exponent to enforce security against malicious adversaries. Our techniques can be securely instantiated by relying on the hardness of the Computational Diffie-Hellman problem in the same curve as the ECDSA signature itself, in addition to using the same hash function as ECDSA. We implement and benchmark our protocols, showing that using native ECDSA assumptions for threshold ECDSA does not have to compromise on concrete efficiency; indeed our wall-clock times are better than all other current approaches in both the LAN and WAN settings.

The recent protocol of Lindell and Nof [LN18] is constructed under the Decisional Diffie-Hellman assumption, making use of a multiplier which can be instantiated using our CDH-based one (although this is not the one that they choose to implement for their benchmarks). However due to the large number of public key operations required by the zero-knowledge proofs in their protocol, we believe that our approach will compare favourably if their threshold ECDSA scheme were to be implemented using our multiplier.

In this paper we provide the necessary background to motivate the standardization of threshold ECDSA signature schemes, briefly describe our approach of using MPC techniques to achieve this task, and report the efficiency of our implementation to demonstrate that our protocol is already practical in a number of real-world settings. The technical details can be found in our published work [DKLs18, DKLs19].

2 Background

ECDSA is a standardized [Nat13, Ame05, Bro10] derivative of the earlier Digital Signature Algorithm (DSA), devised by David Kravitz [Kra93]. Where DSA is based upon arithmetic modulo a prime, ECDSA uses elliptic curve operations over finite fields. Compared to its predecessor, it has the advantage of

being more efficient and requiring much shorter key lengths for the same level of security. In addition to the typical use cases of authenticated messaging, code and binary signing, remote login, &c., ECDSA has been eagerly adopted where high efficiency is important. For example, it is used by TLS [BWBG$^+$06], DNSSec [HW12], and many cryptocurrencies, including Bitcoin [Bit17b] and Ethereum [Woo17].

A t-of-n threshold signature scheme is a set of protocols which allow n parties to jointly generate a single public key, along with n private shares of a joint secret key, and then privately sign messages if and only if t (some predetermined number) of those parties participate in the signing operation. In addition to satisfying the standard properties of signature schemes, it is necessary that threshold signature schemes be secure in a similar sense to other protocols for multi-party computation. That is, it is necessary that no malicious party can subvert the protocols to extract another party's share of the secret key, and that no subset of fewer than t parties can collude to generate signatures.

The concept of threshold signatures originates with the work of Yvo Desmedt [Des87], who proposed that multi-party and threshold cryptographic protocols could be designed to mirror societal structures, and thus cryptography could take on a new role, replacing organizational policy and social convention with mathematical assurance. Although this laid the motivational groundwork, it was the subsequent work of Desmedt and Frankel [DF89] that introduced the first true threshold encryption and signature schemes. These are based upon a combination of the well-known ElGamal [ElG84] and Shamir Secret-Sharing [Sha79] primitives, and carry the disadvantage that they require a trusted party to distribute private keys. Pedersen [Ped91] later removed the need for a trusted third party.

Desmedt and Frankel [DF89] recognized the difficulties inherent in designing threshold systems for standard signature schemes. Nevertheless, they later returned to the problem [DF91], proposing a non-interactive threshold system for RSA signatures [RSA78]. This was subsequently improved and proven secure in a series of works [DF92, GJKR96a, DSDFY94, Sho00]. Threshold schemes were also developed for Schnorr [Sch89, SS01] and DSA [Lan95, GJKR96b, MR01] signatures. Many of these schemes were too inefficient to be practical, however.

The efficiency and widespread acceptance of ECDSA make it a natural target for similar work, and indeed threshold ECDSA signatures are such a useful primitive that many cryptocurrencies are already implementing a similar concept in an ad-hoc manner [Bit17a]. Unfortunately, the design of the ECDSA algorithm poses a unique problem: the fact that it uses its nonce in a multiplicative fashion frustrates attempts to use typical linear secret sharing systems as primitives.

It has only very recently become practically feasible to compute a threshold ECDSA signature along with setup; of the broadly two approaches, one using homomorphic encryption [GG18, LN18] and the other MPC [DKLs18, DKLs19] a clear winner is yet to emerge. The homomorphic encryption based approaches are more efficient in terms of communication, while the MPC approach is more efficient in computation. While the works of [GG18, LN18] require a constant

number of rounds (as opposed to $\log(t)$ in the MPC approach) the MPC approach can in principle be made constant-round at the expense of some computation and communication, by means of the Bar-Ilan and Beaver inversion technique [BB89] (work in progress). We note that in terms of wall-clock time (accounting for network costs, etc), in both the WAN and LAN settings, our benchmarks for signature generation significantly outperform those of [GG18, LN18]. While the benchmarks of [LN18] were performed on a single threaded machine, we do not believe that additional cores on commodity hardware will bridge the difference in performance to be comparable to the MPC approach. The setup times for our MPC approach is orders of magnitude more efficient than the ones based on homomorphic encryption.

3 Our Techniques

We present two sets of protocols; a general t-of-n multiparty threshold ECDSA scheme [DKLs19] that requires $\log(t) + 6$ rounds to sign, and a specialized minimally interactive 2-of-n scheme [DKLs18] that requires only 2 rounds (one in each direction) to sign. Both have similar setup protocols.

Recall the signing equation for ECDSA,

$$\sigma = \frac{H(m) + \mathsf{sk} \cdot r_x}{k}$$

where m is the message, H is a hash function, sk is the secret key, k is the instance key, and r_x is the x-coordinate of the elliptic curve point $R = k \cdot G$ (G being the generator for the curve). At a high level, our approach is constituted by the following recipe:

1. Setup:
   (a) **Key generation.** To establish the common signing key sk, each party chooses a random additive share of sk, of which they compute a t-of-n Shamir sharing and communicate said shares to other parties privately. The validity of shares received is checked by having each party broadcast their share in the exponent, and verifying that every possible interpolation in the exponent yields the same value (which can be done in time $\mathcal{O}(n)$).
   (b) **Base Oblivious Transfers.** Along with signing key generation, the setup required for the OT-based multiplier (consisting of the base OTs to be extended later) is done at this time.
2. Signing a message m:
   (a) **Nonce generation.** A key-agreement protocol is used to produce a nonce R, of which the discrete logarithm k is multiplicatively shared among the parties. In the two party setting this is done by Diffie-Hellman, whereas in the multiparty setting a t-party multiplier is used for this task.

(b) **Multiplication.** A t-party multiplication protocol is used to compute additive shares of sk/k and $1/k$, from which an additive sharing of a complete signature σ can be obtained by simple linear combination.

To compute these multiplications, one could apply generic multiparty computation over arithmetic circuits, but generic MPC techniques incur large practical costs in order to achieve malicious security. Instead, we construct a new two-party multiplication protocol, based upon the semi-honest Oblivious-Transfer (OT) multiplication technique of Gilboa [Gil99], which we harden to tolerate malicious adversaries. Note that even if the original Gilboa multiplication protocol is instantiated with a malicious-secure OT protocol, it is vulnerable to a simple selective failure attack whereby the OT sender (Alice) can learn one or more bits of the secret input of the OT receiver (Bob). We mitigate this attack by encoding Bob's input randomly, such that Alice must learn more than a statistical security parameter number of bits in order to determine his unencoded input.

(c) **Consistency check.** In order to verify that consistent inputs were supplied to the multipliers, we introduce a simple *consistency check* mechanism. In essence, the parties combine their shares with the secret key and instance key *in the exponent*, such that if the shares are consistent then they evaluate to a constant value. This check is a novel and critical element of our protocols, and we conjecture that it can be applied to other domains. The security of this mechanism relies on the hardness of the CDH problem in the elliptic curve group used by the signature itself.

4 Implementation

We created proof-of-concept implementations of our two-party and t-of-n setup and signing protocols in the Rust language. Our implementation uses the secp256k1 curve, as standardized by NIST [Bro10]. Additionally, we chose our statistical security parameter as 80. In all cases where our protocols permitted a choice among different methods for instantiating a particular primitive, we chose the option that most closely mirrors the ECDSA specification. For example, we chose SHA-256 as our hash function, and we implemented Oblivious Transfer Extensions using a combination of the protocol of Keller et al. [KOS15] and our own varant of the protocol of Chou and Orlandi [CO15], since this method works over the same elliptic curve as ECDSA and requires no assumptions beyond those necessary to prove our signing protocols secure.

In our implementation, each party parallelizes its interactions with its counterparties, using a number of threads equal to the number of parties, up to a specified maximum. While it is hypothetically possible for our setup protocol to parallelize key-generation and one-time Oblivious Transfer Extension initialization, our implementation runs these two phases sequentially, and thus the

n/t Range	n/t Step	Samples (Signing)	Samples (Setup)
$[2, 8]$	1	16000	2000
$(8, 16]$	2	8000	1000
$(16, 32]$	4	4000	500
$(32, 64]$	8	2000	250
$(64, 128]$	16	1000	125
$(128, 256]$	32	500	62

Table 1: **LAN Benchmark Parameters**. For signing we varied t according to these parameters, and for setup we varied n, fixing $t = \lfloor (n + 1)/2 \rfloor$.

round count of setup is increased from five to eight.

We benchmarked our implementation using a set of Google Cloud Platform `n1-highcpu-8` nodes, each running Ubuntu 18.04 with kernel 4.15.0. Each node of this type has four physical cores clocked at 2.0 GHz, and is capable of executing eight threads simultaneously. Each party participating in a benchmark was allocated one node, and the parties communicated via Google's internal network. We compiled our code using the nightly version of Rust 1.28, with the default level of optimization. Parallelism was provided by the Rayon crate and, as each node can execute eight threads simultaneously, we limited the number of threads used in signing to ten (having arrived at this number empirically). Our hash function implementations were written in C using compiler intrinsics, and were compiled with GCC 8.2.0. Our benchmarking programs were designed to establish insecure connections among the parties one time only, and then run a batch of setup or signing operations, measuring the wall clock time for the entire batch. Thus, they record overhead due to latency and bandwidth constraints, but they do not record overhead due to private or authenticated channels.

4.1 LAN Benchmarks

For benchmarks in the LAN setting, we created a set of 256 nodes in Google's South Carolina datacenter. Among these nodes, we measured the bandwidth to be generally between 5 and 10 Gbits/sec, and the round-trip latency to be approximately 0.3 ms. Using these nodes, we collected data for both our setup and signing protocols using combinations of parameters as specified in Table 1. For signing benchmarks, all costs are independent of n, the number of parties in the larger group from whom the signing parties are selected. Consequently, we varied only t, the number of parties actually participating in signing. For setup, only computation costs depend upon t, and not bandwidth; consequently we varied n and set $t = \lfloor (n+1)/2 \rfloor$, which we determined to be the most expensive value relative to a particular choice of n. Our aim in choosing sample counts was to ensure each benchmark took five to ten minutes in total, in order to smooth out artifacts due to transient network conditions. Our results for setup are reported in Figure 1, and our results for signing are reported in Figure 2.

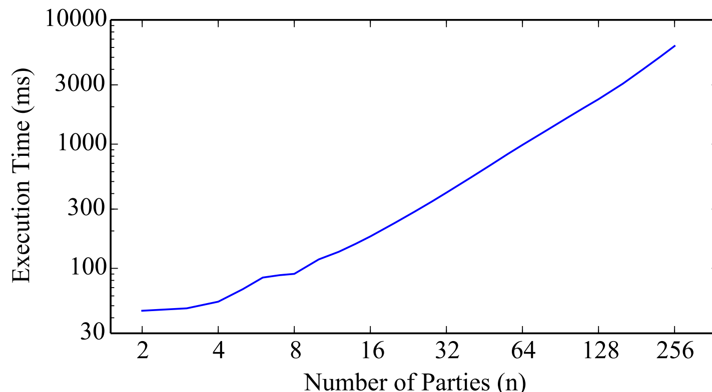

Figure 1: **Wall Clock Times for n-Party Setup over LAN**. Note that all parties reside on individual machines in the same datacenter, and latency is on the order of a few tenths of a millisecond.

4.2 Comparison

In a similar benchmark environment to our own, but without parallelism, the 2-of-2 protocol of Lindell [Lin17] was reported to require 36.8 ms for signing with only two parties, and 2435 ms for key-generation. Our two-party protocol requires only 5.8ms for 2-of-2 signing, and allowing parallelism, our t-of-n protocol is capable of signing with 20 parties in 31.6 ms (a slightly shorter time envelope). For setup (which includes key-generation) our protocols require 45 ms with two parties and 232 with 20.

In the arbitrary-threshold context, a number of prior and contemporary works exist. As with Lindell's protocol, we did not benchmark their protocols in our environment, and so no truly fair comparison is possible. Nevertheless, all of them report benchmarks among 2 to 20 LAN-connected parties on broadly similar hardware to our own, and we believe it possible to draw some loose conclusions by comparing their results. The protocol of Gennaro and Goldfeder [GG18] appears to be the fastest prior or concurrent work: they do not count network costs or report benchmarks for their key-generation protocol, but claim that in terms of computation their signing protocol requires 77 ms among two parties and 509 ms among 20. Lindell and Nof [LN18] report benchmarks that count network costs, but do not take advantage of parallelism. They demonstrate signing times of 305 ms among two parties and 5 seconds among 20, and key-generation times of 11 and 28 seconds, respectively. The protocols of Gennaro et al. [GGN16] and Boneh et al. [BGG17] are somewhat slower; for explicit comparisons with these, we refer the reader to Doerner et al. [DKLs18, DKLs19]. In all parameter regimes reported, all prior and concurrent works are at least one order of magnitude slower than our own in terms of both key-generation and signing, and in some cases we improve upon them by two or more orders of magnitude. We stress again that as these benchmarks

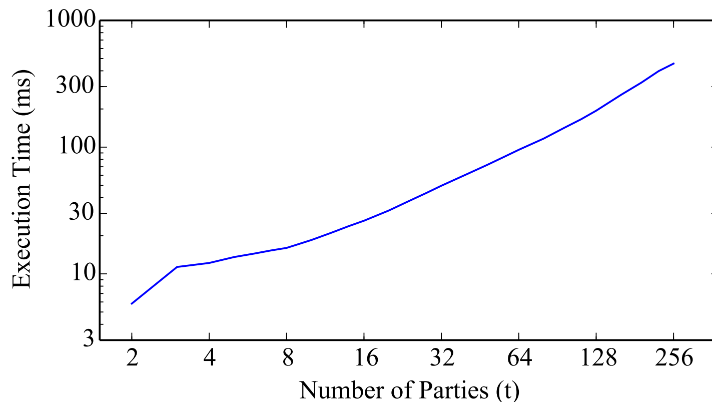

Figure 2: **Wall Clock Times for t-Party Signing over LAN**. Note that all parties reside on individual machines in the same datacenter, and latency is on the order of a few tenths of a millisecond.

were not run in identical environments, they do not constitute a fair comparison. Nevertheless, we do not believe that environmental differences account for the performance discrepancy.

4.3 WAN Benchmarks

Our protocol is at a disadvantage relative to other works in terms of round count and bandwidth cost. In order to demonstrate the practical implications of this fact, we ran an additional benchmark in the WAN setting. We chose 16 Google datacenters (otherwise known as *zones*) that offer instances with current-generation CPUs; these are located on a map in Figure 3. Five were located inside the United States, in South Carolina, Virginia, Oregon, California, and Iowa. Among these, the longest leg was between Oregon and South Carolina, with a round-trip latency of 66.5 ms and bandwidth of 353 Mbits/sec. The remaining 11 were located in Montreál, London, Frankfurt, Belgium, the Netherlands, Finland, Sydney, Taiwan, Tokyo, Mumbai, and Singapore. Among the complete set, the longest leg was between Belgium and Mumbai, with a round-trip latency of 348 ms and a bandwidth of 53.4 MBits/sec. We tested two configurations: one with only the five US datacenters participating, and another with all 16. For each configuration, we performed benchmarks with one party in each participating datacenter, and with eight parties in each participating datacenter. In all cases, we collected 125 samples. Results are reported in Table 2, along with comparative data from our LAN benchmarks.

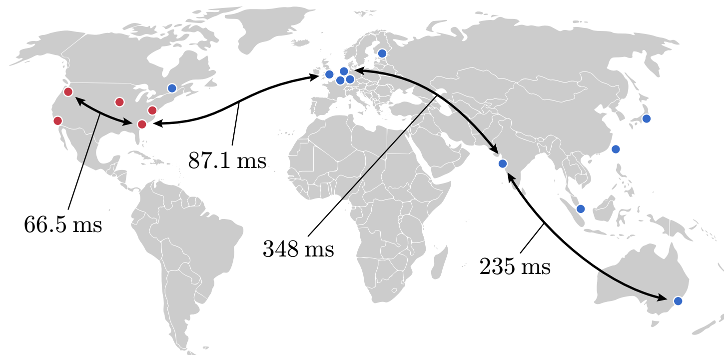

Figure 3: **Map of Datacenter Locations used for WAN Benchmarks**, with latency figures along a few of the longer routes. The subgroup of five zones inside the US are highlighted in red.

Parties/Zones	Signing Rounds	Signing Time	Setup Time
5/1	9	13.6	67.9
5/5	9	288	328
16/1	10	26.3	181
16/16	10	3045	1676
40/1	12	60.8	539
40/5	12	592	743
128/1	13	193.2	2300
128/16	13	4118	3424

Table 2: **Wall-clock Times in Milliseconds over WAN**. The benchmark configurations used are described in Section 4.3. For signing we varied t according to these parameters, and for setup we varied n, fixing $t = \lfloor (n+1)/2 \rfloor$. Benchmarks involving only a single zone are LAN benchmarks, for comparison.

4.4 Low-power Benchmarks

Finally, we performed a set of benchmarks on a group of three Raspberry Pi model 3B+ single-board computers in order to demonstrate the feasibility of evaluating our protocols on small, low-powered devices. Each board has a single, quad-core ARM-based processor clocked at 1.4 GHz. The boards were loaded with Raspbian Linux (kernel 4.14) and connected to one another via ethernet. As an optimization for the embedded setting, we abandoned SHA-256 (except where required by ECDSA) in favor of the BLAKE2 hash function [ANWOW13], using assembly implementations provided by the BLAKE2 authors. To simulate the setting wherein an embedded device signs with a more powerful one, we used a 2013 15" Macbook Pro running Mac OS 10.13 (i.e. one author's laptop). This machine was engaged in other tasks at the time of benchmarking, and no attempt was made to prevent this. We benchmarked 2-of-2 signing and setup

Configuration	Benchmark	Setup Time	Signing Time
Macbook/RPi	2-of-2	1419	52.6
2×RPi	2-of-2	1960	58.5
3×RPi	3-of-3	2277	162

Table 3: **Wall-clock Times in Milliseconds for Raspberry Pi**. The benchmark configurations used are described in Section 4.4.

between the Macbook and a single Raspberry Pi, and t-of-n setup and signing among the group of Pis, with t and n set as both 2 and 3. For setup, we collected 50 samples, and for signing, we collected 250. Results are presented in Table 3. We observe that in spite of the limitations of the hardware on which these benchmarks were run, the signing time remains much less than a second, and setup requires only a few seconds. Thus we expect our protocol to be computationally efficient enough to run even on embedded devices such as hardware tokens or smartwatches, and certainly on more powerful mobile devices such as phones.

5 Conclusion

Threshold signature schemes are a powerful cryptographic primitive that are natural building blocks for any threshold cryptosystem, direct applications notwithstanding. Unfortunately, ECDSA which is one of the most widely deployed signature schemes, does not permit a simple threshold variant. Candidate solutions for this problem have only recently entered the realm of practical feasibility, therefore formulating a standard for this task will be substantial progress toward widespread deployment of threshold variants of signatures already in use today.

In this paper we make the case for the use of MPC techniques to solve this problem, demonstrating protocols to compute threshold ECDSA signatures that add no assumptions foreign to ECDSA itself. Our benchmarks indicate that our threshold ECDSA scheme achieves the best wall-clock times of such schemes known to date, showing that being conservative in assumptions need not come at the cost of concrete efficiency.

References

[Ame05] American National Standards Institute. X9.62: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[ANWOW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. `https://blake2.net/blake2.pdf`, 2013.

[BB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 201–209, New York, NY, USA, 1989. ACM.

[BGG17] Dan Boneh, Rosario Gennaro, and Steven Goldfeder. Using level-1 homomorphic encryption to improve threshold dsa signatures for bitcoin wallet security. In *LATINCRYPT*, 2017.

[Bit17a] Bitcoin Wiki. Multisignature. `https://en.bitcoin.it/wiki/Multisignature`, 2017. Accessed Oct 22, 2017.

[Bit17b] Bitcoin Wiki. Transaction. `https://en.bitcoin.it/wiki/Transaction`, 2017. Accessed Oct 22, 2017.

[Bro10] Daniel R. L. Brown. Sec 2: Recommended elliptic curve domain parameters, 2010.

[BWBG+06] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve digital signature algorithm (dsa) for dnssec. `https://tools.ietf.org/html/rfc4492`, 2006.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *LATINCRYPT*, 2015.

[Des87] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *CRYPTO*, 1987.

[DF89] Yvo G. Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO*, 1989.

[DF91] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures (extended abstract). In *CRYPTO*, 1991.

[DF92] Yvo Desmedt and Yair Frankel. Parallel reliable threshold multisignature, 1992.

[DKLs18] J. Doerner, Y. Kondi, E. Lee, and a. shelat. Secure two-party threshold ecdsa from ecdsa assumptions. In *39th IEEE Symposium on Security and Privacy*, 2018.

[DKLs19] J. Doerner, Y. Kondi, E. Lee, and a. shelat. Threshold ecdsa from ecdsa assumptions: The multiparty case. In *40th IEEE Symposium on Security and Privacy (to appear)*, 2019.

[DSDFY94] Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. How to share a function securely. In *STOC*, 1994.

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, 1984.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ecdsa with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1179–1194, New York, NY, USA, 2018. ACM.

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. *Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security*. 2016.

[Gil99] Niv Gilboa. Two party rsa key generation. In *CRYPTO*, 1999.

[GJKR96a] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust and efficient sharing of rsa functions. In *CRYPTO*, 1996.

[GJKR96b] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold dss signatures. In *EUROCRYPT*, 1996.

[HW12] P. Hoffman and W.C.A. Wijngaards. Elliptic curve digital signature algorithm (dsa) for dnssec. `https://tools.ietf.org/html/rfc6605`, 2012.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO*, 2015.

[Kra93] D.W. Kravitz. Digital signature algorithm, jul 1993. US Patent 5,231,668.

[Lan95] Susan K. Langford. Threshold dss signatures without a trusted party. In *CRYPTO*, 1995.

[Lin17] Yehuda Lindell. Fast secure two-party ecdsa signing. In *CRYPTO*, 2017.

[LN18] Yehuda Lindell and Ariel Nof. 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*, CCS '18, pages 1837–1854, New York, NY, USA, 2018. ACM.

[MR01] Philip MacKenzie and Michael K. Reiter. Two-party generation of dsa signatures. In *CRYPTO*, 2001.

[Nat13] National Institute of Standards and Technology. FIPS PUB 186-4: Digital Signature Standard (DSS). `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`, 2013.

[Ped91] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *EUROCRYPT*, 1991.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 1978.

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO*, 1989.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 1979.

[Sho00] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, 2000.

[SS01] Douglas R. Stinson and Reto Strobl. Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *ACISP*, 2001.

[Woo17] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2017.