State-of-the-art Threshold ECDSA for Honest Majority and Honest Minority

SCN 2020 · CCS 2021

Hongrui Cui

August 2, 2022

* Some acknowledgments?

ECDSA

- Setup:

$$\mathsf{Gen}(1^\kappa) \mapsto (\mathbb{G}, q, G, H, F) \quad \texttt{secp256k1} \text{ in BitCoin}$$

- KeyGen:

$$x \leftarrow \mathbb{F}_q, \mathsf{sk} = x, \mathsf{pk} = x \cdot G$$

- Sign:

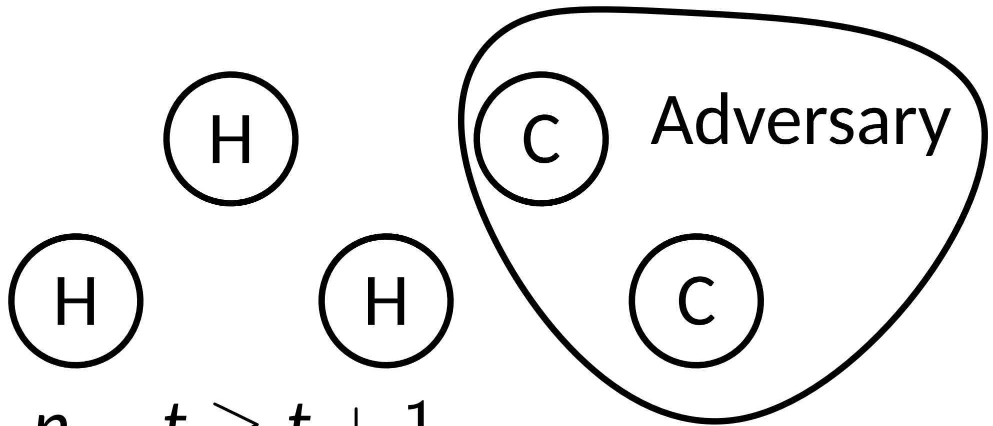
$$m \leftarrow H(\mathsf{msg}), k \leftarrow \mathbb{F}_q \setminus \{0\}$$
$$r \leftarrow F(H(k \cdot G)), s \leftarrow k^{-1}(m + rx)$$

- Verify:

$$\text{Checks } r = F(s^{-1} \cdot (m \cdot G + r \cdot \mathsf{pk}))$$

Threshold ECDSA with Honest Majority

- (n, t)-threshold, $n \geq 2t + 1$
- Example: (3, 1), (5, 2),...-threshold



$n - t \geq t + 1$

- Information Theoretic Protocol
- Malicious threshold ECDSA with abort, can be boosted to fairness

Main Tool: Shamir Secret Sharing

- (n,t)-Shamir-SS:

$$[s] := (f(1), f(2), ..., f(n))$$
$$f \leftarrow \mathbb{F}_q[X] \text{ s.t. } \deg(f) \leq t \wedge f(0) = s$$

- Encode is linear

$$\boxed{\begin{matrix} f(1) \\ f(2) \\ \vdots \\ f(n) \end{matrix}} = \boxed{\begin{matrix} 1^0\ 1^1\ 1^2 \dots 1^t \\ 2^0\ 2^1\ 2^2 \dots 2^t \\ \vdots \\ n^0\ n^1\ n^2 \dots n^t \end{matrix}} \times \boxed{\begin{matrix} s \\ c_1 \\ c_2 \\ \vdots \\ c_t \end{matrix}}$$

$$[s] = \text{Van(n,t)} \times \text{Coeff}$$

- Decode is linear

$$\text{Coeff} = \text{Van(t,t)}^{-1} \times [s]_{[1,t]}$$

Shamir Secret Sharing: Homomorphism

- Linear Homomorphism: for $[s_1] = f_1(1), ..., f_1(n), [s_2] = f_2(1), ..., f_2(n)$

$$f_1(1) + f_2(1), ..., f_1(n) + f_2(n) = (f_1 + f_2)(1), ..., (f_1 + f_2)(n)$$

$$\deg(f_1 + f_2) \leq t \wedge (f_1 + f_2)(0) = s_1 + s_2 \Rightarrow \text{Shamir-SS for } s_1 + s_2$$

- Limited Multiplicative Homomorphism

$$f_1(1) * f_2(1), ..., f_1(n) * f_2(n) = (f_1 * f_2)(1), ..., (f_1 * f_2)(n)$$

$$\deg(f_1 * f_2) \leq 2t \wedge (f_1 * f_2)(0) = s_1 * s_2 \Rightarrow \textbf{2t}\text{-Shamir-SS for } s_1 * s_2$$

- Higher degree reduces the error-correction capability

Protocol Details: KeyGen

- P_i samples and distributes $[x_i]$
- Define $[x] := [x_1] + ... + [x_n] = (f(1), ..., f(n))$
- P_i broadcasts $f(i) \cdot G$
- Consistency check:

$$([s]_{\{j\}} \cdot G) \;=\; \mathsf{Van}(\{j\},\mathsf{t}) \;\times\; \mathsf{Van}(\mathsf{t},\mathsf{t})^{-1} \;\times\; ([s]_{[1,t]} \cdot G)$$

Interpolation “in the exponent”

- P_i outputs $\mathsf{sk} = x_i$, $\mathsf{pk} =$

$$(s \cdot G) \;=\; \left(\mathsf{Van}(\mathsf{t},\mathsf{t})^{-1} \;\times\; ([s]_{[1,t]} \cdot G)\right)_0$$

Protocol Detail: Sign

- Step 1: Parties prepare random t-SS $[a]$, $[k]$, random $2t$-SS of 0 $[b]$, $[d]$, $[e]$
- Step 2: P_i prepares $R_i := k_i \cdot G, w_i := k_i a_i + b_i$ and broadcasts R_i, w_i
- Step 3: Check R_i's t-consistency and computes/broadcasts

$$\begin{aligned} R &= (k \cdot G) &= \left(\mathrm{Van(t,t)}^{-1} \times ([s]_{[1,t]} \cdot \mathrm{G}) \right)_1 \\ W_i &= a_i \cdot R \end{aligned}$$

- Step 4: Computes $r = F(R)$ and check W_i's t-consistency and

$$W = ak \cdot G = w \cdot G$$

- Step 5: Compute $m = H(\mathrm{msg})$, $c_i = e_i m + d_i$, and $s_i = w^{-1} a_i (m + r x_i) + c_i$
- Step 6: Reconstructs s from $2t$-sharing and verify signature

Fairness

- Use verifiable secret sharing $[[s]] := \Big((s_1, \mathrm{Com}_1), ..., (s_n, \mathrm{Com}_n)\Big)$
- $\mathrm{Com}_1 = g^{s_1}h^{r_1}$ where $\log_g(h)$ is unknown
- Generation:
  1. P_i prepares f_i and broadcasts $\mathrm{Com}(\text{coeff of } f_i)$
  2. P_i opens $f_i(j)$ to P_j using linear homomorphism of Com
  3. Define $f := f_1 + ... + f_n$
- KeyGen*: Generate $[[x]]$ as above
- Sign*:
  1. Generate $[[s]]_t, [[s]]_{2t}, [[b]]_t, [k^{-1}]_t, [[x]]_{2t}$
  2. P_i opens $b_i - k_i^{-1}$ and checks for t-consistency $\Rightarrow [[k^{-1}]]_t$
  3. P_i broadcasts $\mathrm{Com}(x_i k_i^{-1})$ and prove correctness using **ZK**
  4. Reconstructs $s - xk^{-1}$ with VSS $\Rightarrow [[xk^{-1}]]_t$
  5. $[[s]]_t := m[[k^{-1}]]_t + r[[xk^{-1}]]_t$

Performance

- Amazon EC2 `m5.xlarge`, single core, CentOS
- 1 docker of PostgreSQL to store sk share and 1 docker of server

Table 2. Latency per operation

	LAN			WAN		
n, t	keygen	presig	sign	keygen	presig	sign
$3, 1$	28.2 ms	34.2 ms	19.9 ms	1.22 s	1.47 s	0.73 s
$5, 2$	39.9 ms	44.8 ms	25.0 ms	1.47 s	1.71 s	0.98 s
$7, 3$	54.6 ms	60.0 ms	30.8 ms	1.48 s	1.72 s	0.98 s
$9, 4$	66.4 ms	74.0 ms	34.8 ms	1.48 s	1.72 s	1.00 s

Threshold ECDSA with 2 Parties

- Full-threshold case, needs cryptographic tools
- 1st tool: Multiplication-to-Addition Protocol

$$x_1 \in \mathbb{Z}_q \longrightarrow \quad \boxed{\textbf{MtA}} \quad \longleftarrow x_2 \in \mathbb{Z}_q$$

$$\xleftarrow{t_A \in \mathbb{Z}_q} \qquad \xrightarrow{t_B \in \mathbb{Z}_q}$$

$$t_A + t_B = x_1 \cdot x_2$$

- Realizations: OT (Gilboa) or AHE (Paillier)
- Advantages: Fast Small communication

- 2nd tool: Schnorr Proof
- Proving relations: $y = g^x$ for public g, y

$$
\begin{array}{lcl}
P_A & & P_B \\
a \leftarrow \mathbb{Z}_q & \xrightarrow{A = g^a} & \\
 & \xleftarrow{c \leftarrow \mathbb{Z}_q} & \\
 & \xrightarrow{z = a + cx} & \text{accepts if } g^z = A \cdot y^c
\end{array}
$$

- Public Coin protocol, able to apply Fiat-Shamir

2ECDSA: KeyGen

- Essentially coin-tossing

P_1		P_2
	......Keygen$(\mathbb{G}, P, q)$......	
$x_1 \leftarrow \mathbb{Z}_q, Q_1 = x_1 \cdot P$		
nizk1 := nizkPoK(Q_1, x_1)	$\xrightarrow{\text{f1} := \mathsf{H}(Q_1, \text{nizk1})}$	$x_2 \leftarrow \mathbb{Z}_q, Q_2 = x_2 \cdot P$
if Verifzk(nizk2) $= 0$, **return** $\perp$	$\xleftarrow{Q_2, \text{nizk2}}$	nizk2 := nizkPoK(Q_2, x_2)
	$\xrightarrow{Q_1, \text{nizk1}}$	**if** f1 $\neq \mathsf{H}(Q_1, \text{nizk1})$, **return** $\perp$
		if Verifzk(nizk1) $= 0$, **return** $\perp$
$Q = Q_1 + Q_2$		$Q = Q_1 + Q_2$

2ECDSA: Sign

(1) Commit P_2's Nonce

$k_2 \leftarrow \mathbb{Z}_q,\ R_2 = k_2 \cdot P$

$\xleftarrow{\mathsf{f2} := \mathsf{H}(R_2, \mathsf{nizk3})}$

$\mathsf{nizk3} := \mathsf{nizkPoK}(R_2, k_2)$

(2) MtA and Consistency

$x_1' \leftarrow \mathbb{Z}_q,\ Q_1' = x_1' \cdot P$ $\xrightarrow{x_1'}$ $\boxed{\textbf{MtA}}$ $\xleftarrow{k_2}$

$\xleftarrow{t_A}$ $\xrightarrow{t_B}$

$$t_B + cc = t_A + t_B + x_1' r_1 - x_1$$
$$= x_1'(r_1 + k_2) - x_1$$

$r_1 \leftarrow \mathbb{Z}_q$

$cc = t_A + x_1' r_1 - x_1 \mod q$ $\xrightarrow{Q_1', r_1, cc}$ if $(t_B + cc)P = (r_1 + k_2)Q_1' - Q_1$

$x_2' = x_2 - (t_B + cc) \mod q$

$$(r_1 + k_2)x_1' + x_2' = x$$

2ECDSA: Sign

(3) Nonce KE

$k_1 \leftarrow \mathbb{Z}_q, R_1 = k_1 \cdot P$

$$\text{nizk4} := \text{nizkPoK}(R_1, k_1) \quad \xrightarrow{R_1,\ \text{nizk4}} \quad \textbf{if } \text{Verifzk}(\text{nizk4}) = 0, \textbf{return } \perp$$

$$\textbf{if } \text{f2} \neq \text{H}(R_2, \text{nizk3}), \textbf{return } \perp \quad \xleftarrow{R_2,\ \text{nizk3}} \quad R := (r_1 + k_2) \cdot R_1$$

$\textbf{if } \text{Verifzk}(\text{nizk3}) = 0, \textbf{return } \perp$

$R := k_1 \cdot R_2 + k_1 r_1 \cdot P$ $\quad k = k_1(r_1 + k_2)$

(4) Online Sign

$$s = k_1^{-1}(s_2 + r x_1') \mod q \quad \xleftarrow{s_2} \quad s_2 = (r_1 + k_2)^{-1}(H(m) + r x_2') \mod q$$

$\textbf{if } \text{Verify}(m; (r, s)) = 0, \textbf{return } \perp$

$\textbf{else return } (r, s)$

$$s = k_1^{-1}(s_2 + rx_1')$$
$$s = k^{-1}(H(m) + rx_2')) + k_1^{-1}rx_1'$$

Performance

- Tested on local laptops, loopback network
- Mainly consider computation time

Table 3: Cost comparison of Paillier-based schemes.

Schemes	Computation		Communication	
	Offline	Online	Offline	Online
LNR18 [26]	461ms	302ms	12.1KB	6.6KB
GG18 [19]	1237ms	3ms	15.5KB	288B
CGGMP20 [6]	2037ms	0.2ms	44KB	32B
2ECDSA (Paillier)	226ms	0.2ms	6.3KB	32B
Lin17 [25] (Paillier-EC)	34ms	8ms	192B	768B
GG18 [19] (Paillier-EC)	360ms	3ms	6.6KB	288B
2ECDSA (Paillier-EC)	141ms	0.2ms	4.1KB	32B

Table 4: Cost comparison of OT-based schemes.

Schemes	Computation		Communication	
	Offline	Online	Offline	Online
DKLS18 [15]	2.9ms	0.2ms	169.8KB	32B
DKLS19 [16]	3.7ms	0.2ms	180KB	32B
2ECDSA (OT)	2.6ms	0.2ms	90.9KB	32B