

Two-party ECDSA Signing *at* Constant Communication Overhead

Yashvanth Kondi

yash (at) ykondi.net



eprint.iacr.org/2025/1813

S:LENCE
LABORATORIES

NIST MPTS 2026
Presented Jan 27 2026

ECDSA

- Elliptic Curve Digital Signature Algorithm
- Devised by Scott Vanstone in 1992, standardised by NIST
- Differs from Schnorr enough so that patent doesn't apply
- Widespread adoption across the internet



Threshold ECDSA: Challenges

SchnorrSign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(R||m)$$

⋮

ECDSASign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(m)$$

Threshold ECDSA: Challenges

SchnorrSign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(R||m)$$

⋮

ECDSASign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(m)$$

Standard nonce
sampling

Threshold ECDSA: Challenges

SchnorrSign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(R||m)$$

$$s = k - \mathbf{sk} \cdot e$$

$$\sigma = (s, R)$$

output σ

.....

ECDSASign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(m)$$

Standard nonce
sampling

Threshold ECDSA: Challenges

SchnorrSign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(R||m)$$

$$s = k - \mathbf{sk} \cdot e$$

$$\sigma = (s, R)$$

output σ

⋮

ECDSASign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(m)$$

Standard nonce
sampling

Threshold ECDSA: Challenges

SchnorrSign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(R||m)$$

$$s = k - \mathbf{sk} \cdot e$$

$$\sigma = (s, R)$$

output σ

⋮

ECDSASign(**sk**, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(m)$$

$$s = \frac{e + \mathbf{sk} \cdot r_x}{k}$$

output $\sigma = (s, R)$

Standard nonce
sampling

Threshold ECDSA: Challenges

ECDSASign(\mathbf{sk}, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(m)$$

Multiplication of
secret values

$$s = \frac{e + \mathbf{sk} \cdot r_x}{k}$$

output $\sigma = (s, R)$

Division (Modular inverse)

Threshold ECDSA: Challenges

ECDSASign(\mathbf{sk}, m) :

$$k \leftarrow \mathbb{Z}_q$$

$$R = k \cdot G$$

$$e = H(m)$$

$$s = \frac{e + \mathbf{sk} \cdot r_x}{k}$$

output $\sigma = (s, R)$

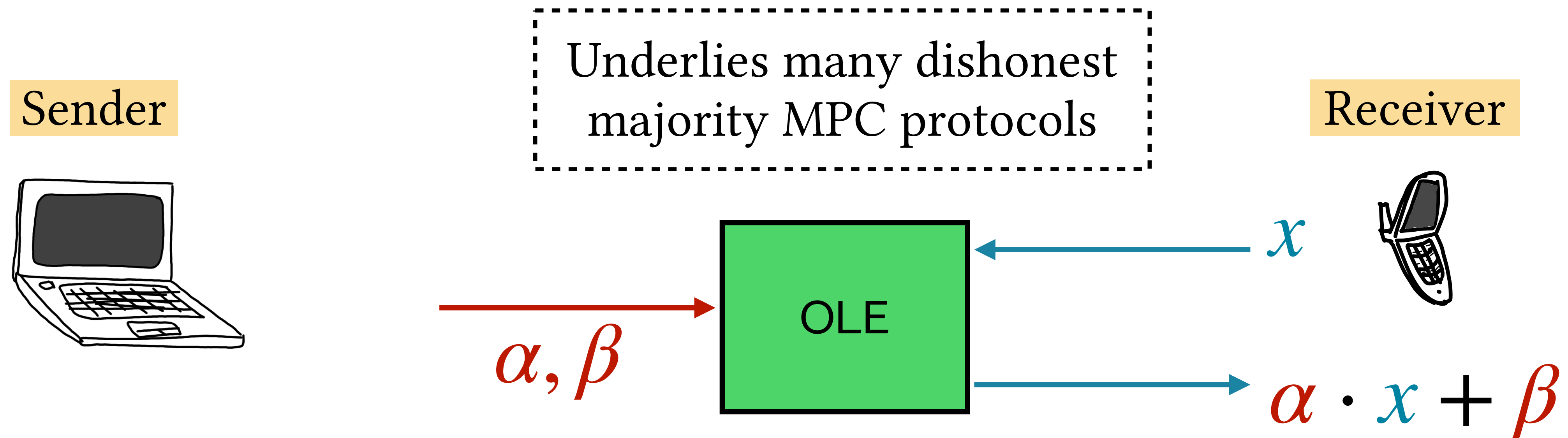
Reduces to a classic MPC primitive without a clear winning instantiation

Multiplication of secret values

Division (Modular inverse)

Secure Two-Party Multiplication

a.k.a. Oblivious Linear Evaluation (OLE), Mult2Add



Efficient constructions from:
OT, Paillier, Class Groups

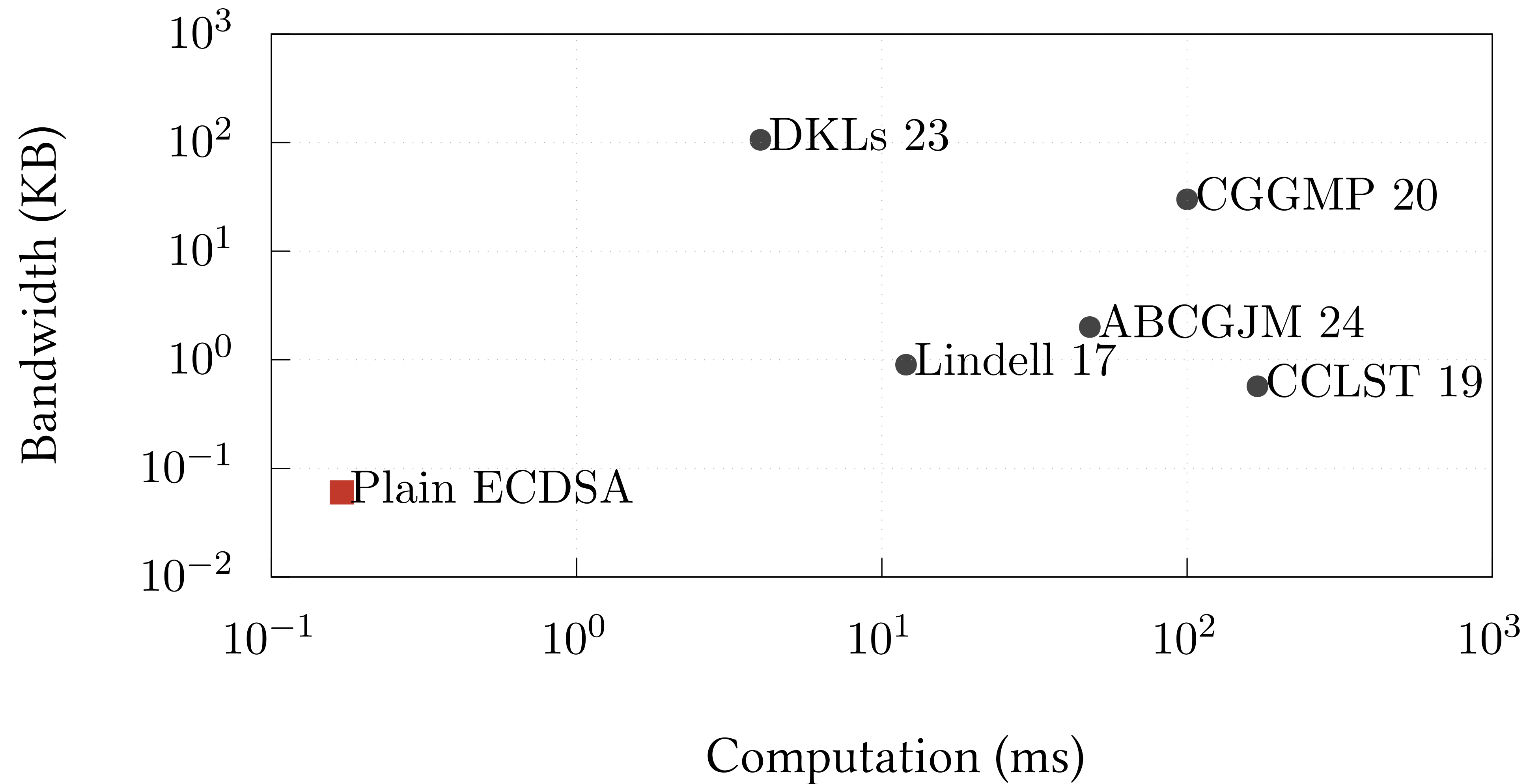
Two-party ECDSA Signing

Protocol	OLE	Bandwidth (KB)	Computation (ms)	Rounds
[Lindell 17]	Paillier	0.9	12	4
[CCLST 19]	Class groups	0.57	170	4
[CGGMP 20]	Paillier	30	100+	3-4
[D K Ls 23]	OT	106	4	3
[ABCGJM 24]	Paillier	2	48	2

Two-party ECDSA Signing

Protocol	OLE	Bandwidth (KB)	Computation (ms)	Rounds
[Lindell 17]	Paillier	0.9	12	4
[CCLST 19]	Class groups	0.57	170	4
[CGGMP 20]	Paillier	30	100+	3-4
[D K Ls 23]	OT	106	4	3
[ABCGJM 24]	Paillier	2	48	2
<i>Plain</i> ECDSA		0.06	0.17	

Two-party ECDSA Signing



Overhead of Two-party ECDSA

- All schemes incur substantial overhead relative to *plain* ECDSA
- This is (evidently) tolerable for many applications
...but for true ubiquity, orders of magnitude overhead unviable
- Security always comes at some cost—but how much is inherent?

Overhead of Two-party ECDSA

- All schemes incur substantial overhead relative to *plain* ECDSA
- This is (evidently) tolerable for many applications
...but for true ubiquity, orders of magnitude overhead unviable
- Security always comes at some cost—but how much is inherent?

Central question of this work:

How close to plain ECDSA can two-party signing be?

Overhead of Two-party ECDSA

- Consider asymptotic complexity as a starting point
 - $O(\kappa)$ bits to transmit an ECDSA signature
 - $O(1)$ EC, scalar operations to compute one
- Existing schemes' OLE pose an immediate barrier
 - OT-OLE: $\Omega(\kappa \log \kappa)$ bits transmitted through OTs
 - Homomorphic Encryption: ciphertexts alone $\omega(\kappa)$ bits in size
- Computation of 2P-schemes difficult to characterize meaningfully

Our New 2P-ECDSA Framework

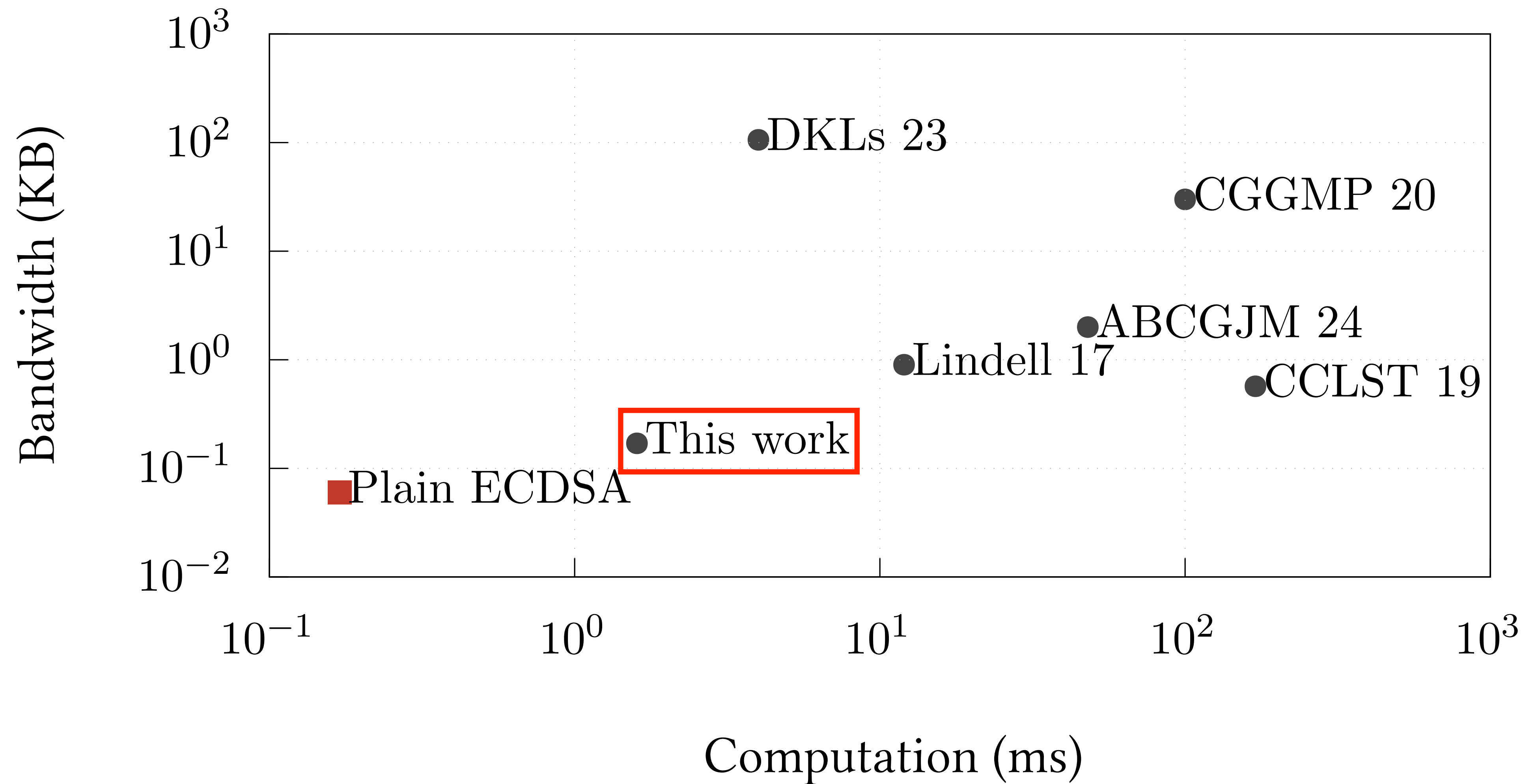
- **Efficiency:**

- $O(\kappa)$ bits sent on network to sign $\Rightarrow O(1)$ overhead
Concretely $\sim 2 |\text{ECDSA sig}|$ in each direction (170B total)
- Best known computation, 1.6ms on M1 MacBook ($\sim 10\times$ overhead)

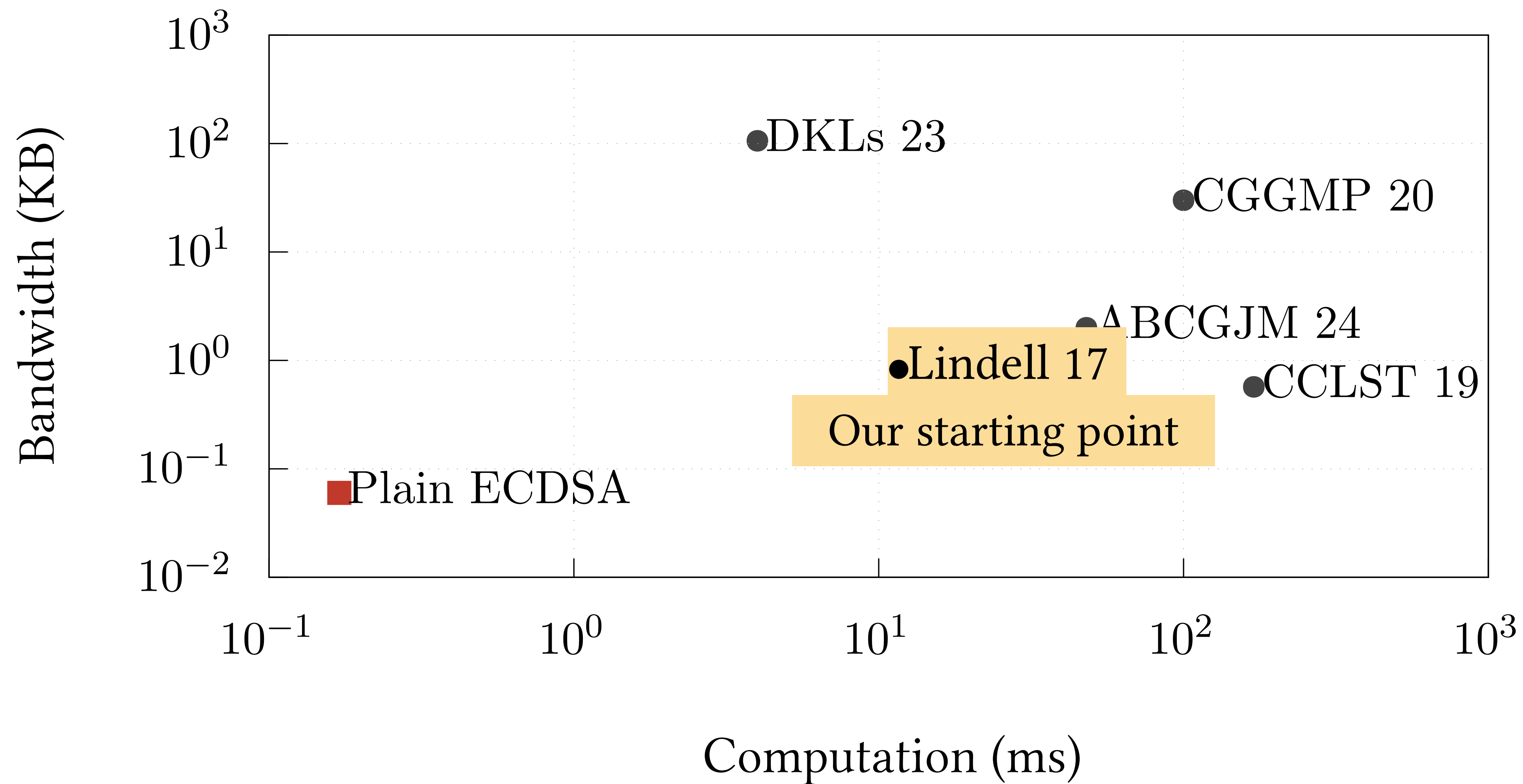
- **Simple, ECDSA-native tools:**

- Any PRF during signing phase
 \Rightarrow benefit from hardware acceleration on many platforms
- KeyGen: Oblivious Transfer (w. Diffie-Hellman in ECDSA curve)

Two-party ECDSA Now



Constructing Our Scheme



MPC for ECDSA Recipe [DKLs 23]

- Retrospectively interpret protocols per their framework
- Standard recipe in the literature:
 1. Rewrite ECDSA signing equation to an “MPC-friendly” equivalent i.e. only additions and multiplications of secret values
 2. Cryptographic Machinery for secure multiplication
 3. Verify that all operations were performed honestly

MPC for ECDSA Recipe [DKLs 23]

- Retrospectively interpret protocols per their framework
- Applied to [Lindell 17]:
 1. “Multiplicative” rewriting of ECDSA
 2. Paillier-based OLE
 3. Verify that all operations were performed honestly

Multiplicative Rewriting of ECDSA

Originally [MR01], refined by [Lin17] and [ABCGJM24]

$$R = k \cdot G$$

$$s = \frac{r_x \cdot sk + h}{k}$$

Multiplicative Rewriting of ECDSA

Originally [MR01], refined by [Lin17] and [ABCGJM24]

$$\text{sk} = \text{sk}_0 \cdot \text{sk}_1$$

$$k = k_0 \cdot k_1$$

$$R = k_0 \cdot k_1 \cdot G$$

$$s = \frac{r_x \cdot \text{sk}_0 \cdot \text{sk}_1 + h}{k_0 \cdot k_1}$$

Multiplicative Rewriting of ECDSA

Originally [MR01], refined by [Lin17] and [ABCGJM24]

$$\text{sk} = \text{sk}_0 \cdot \text{sk}_1$$

$$k = k_0 \cdot k_1$$

$$\alpha = r_x \text{sk}_1 / k_1$$

$$\beta = h / k_1$$

$$R = k_0 \cdot k_1 \cdot G$$

$$s = s' / k_0$$

$$s' = \alpha \cdot \text{sk}_0 + \beta$$

Multiplicative Rewriting of ECDSA

Originally [MR01], refined by [Lin17] and [ABCGJM24]

$$sk = sk_0 \cdot sk_1$$

$$k = k_0 \cdot k_1$$

$$\alpha = r_x sk_1 / k_1$$

$$\beta = h / k_1$$

$$R = k_0 \cdot k_1 \cdot G$$

$$s = s' / k_0$$

$$s' = \alpha \cdot sk_0 + \beta$$

sk_1, k_1



sk_0, k_0

Multiplicative Rewriting of ECDSA

Originally [MR01], refined by [Lin17] and [ABCGJM24]

$$sk = sk_0 \cdot sk_1$$

$$k = k_0 \cdot k_1$$

$$\alpha = r_x sk_1 / k_1$$

$$\beta = h / k_1$$

$$R = k_0 \cdot k_1 \cdot G$$

$$s = s' / k_0$$

$$s' = \alpha \cdot sk_0 + \beta$$

sk_1, k_1



sk_0, k_0

$$s = s' / k_0$$

Final step

Multiplicative Rewriting of ECDSA

Originally [MR01], refined by [Lin17] and [ABCGJM24]

$$sk = sk_0 \cdot sk_1$$

$$k = k_0 \cdot k_1$$

$$\alpha = r_x sk_1 / k_1$$

$$\beta = h / k_1$$

$$sk_1, k_1$$



$$R = k_0 \cdot k_1 \cdot G$$

$$s = s' / k_0$$

$$s' = \alpha \cdot sk_0 + \beta$$

OLE correlation



$$sk_0, k_0$$

$$s = s' / k_0$$

Final step

Multiplicative Rewriting of ECDSA

Originally [MR01], refined by [Lin17] and [ABCGJM24]

$$\begin{aligned} \text{sk} &= \text{sk}_0 \cdot \text{sk}_1 \\ k &= k_0 \cdot k_1 \end{aligned}$$

$$\begin{aligned} \alpha &= r_x \text{sk}_1 / k_1 \\ \beta &= h / k_1 \end{aligned}$$

$$R = k_0 \cdot k_1 \cdot G$$

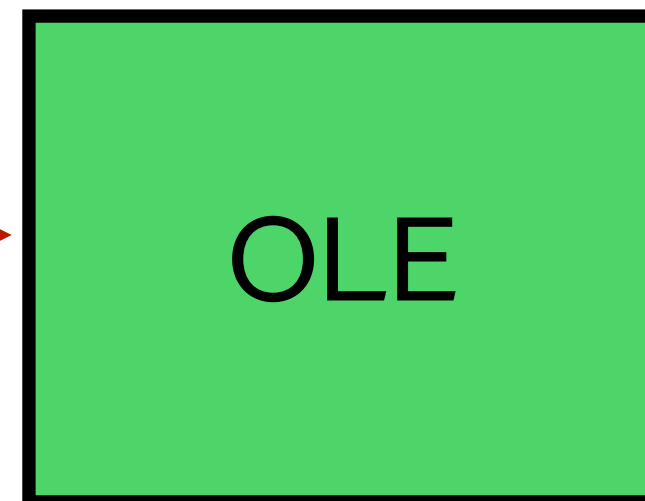
$$s = s' / k_0$$

$$s' = \alpha \cdot \text{sk}_0 + \beta$$

OLE correlation

sk_1, k_1 

$\xrightarrow{\alpha, \beta}$



$\leftarrow \text{sk}_0$

$\rightarrow s'$

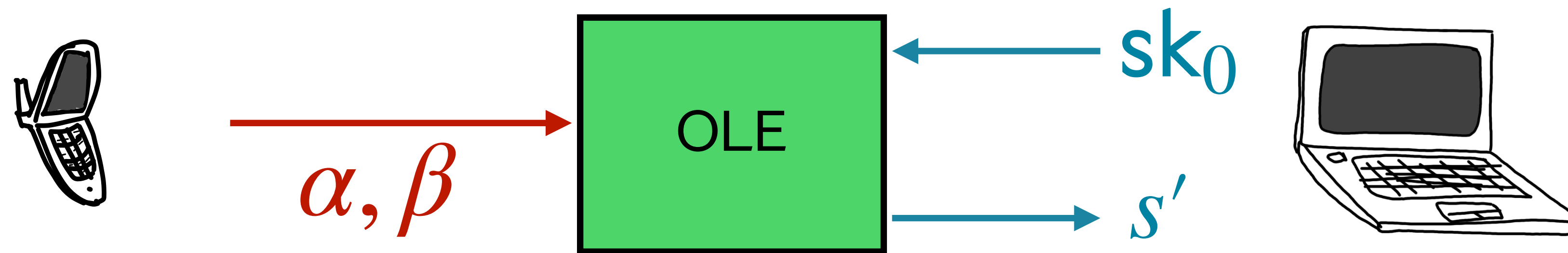


sk_0, k_0

$$s = s' / k_0$$

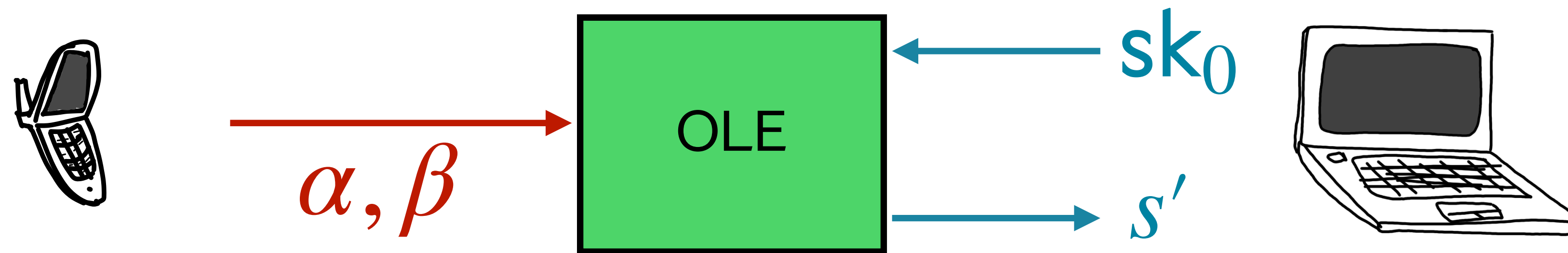
Final step

OLE Structure



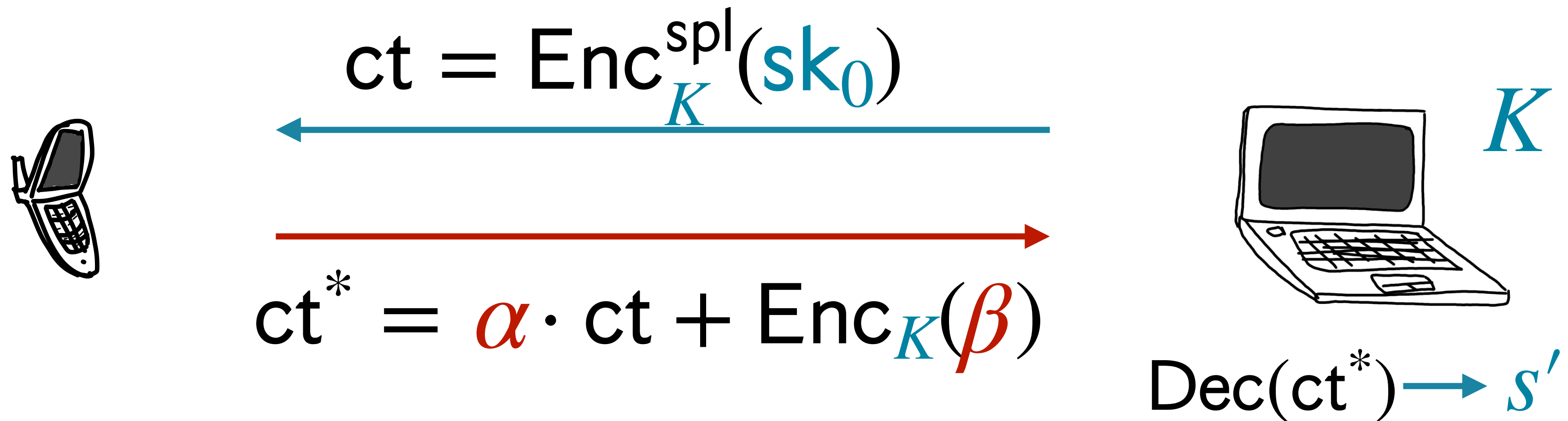
- [Lin17]: Paillier encryption, following classic protocol [Gil99]
- [Lin17] observation: Paillier-based OLE can be very efficient for signing, with heavy work offloaded to key generation

OLE Structure



- [Lin17]: Paillier encryption, following classic protocol [Gil99]
- [Lin17] observation: Paillier-based OLE can be very efficient for signing, with heavy work offloaded to key generation

OLE Structure



- [Lin17]: Paillier encryption, following classic protocol [Gil99]
- [Lin17] observation: Paillier-based OLE can be very efficient for signing, with heavy work offloaded to key generation

OLE Structure



Heavy: One-time during KeyGen

$$ct = \text{Enc}_K^{\text{spi}}(sk_0)$$



$$ct^* = \alpha \cdot ct + \text{Enc}_K(\beta)$$

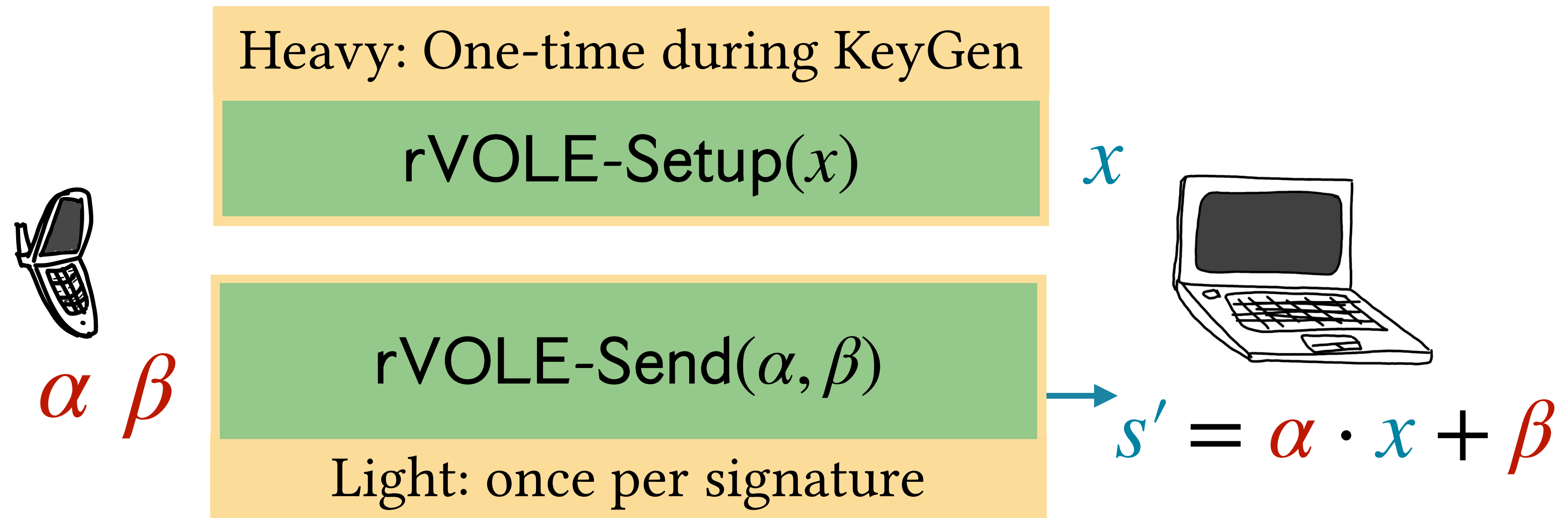
Light: once per signature



$$\text{Dec}(ct^*) \rightarrow s'$$

- [Lin17]: Paillier encryption, following classic protocol
- [Lin17] observation: Paillier-based OLE can be very efficient for signing, with heavy work offloaded to key generation

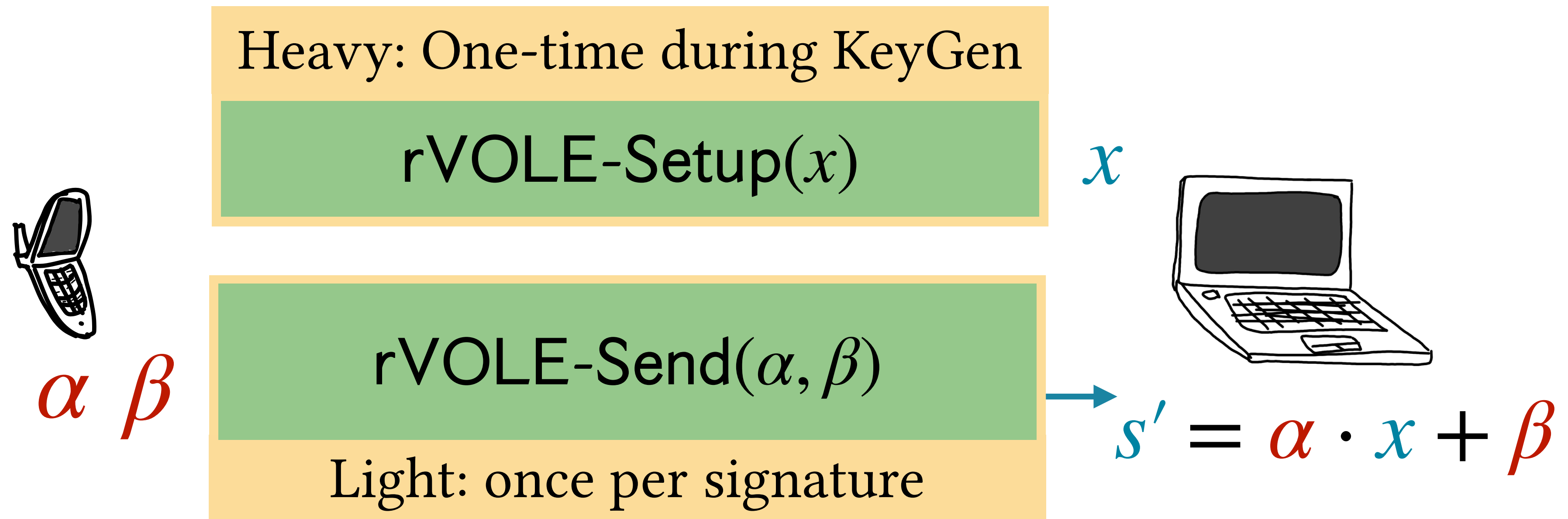
OLE Structure



- [Lin17]: Paillier encryption, following classic protocol
- [Lin17] observation: Paillier-based OLE can be very efficient for signing, with heavy work offloaded to key generation

This work Generalize the concept to reactive Vector Oblivious Linear Evaluation

OLE Structure

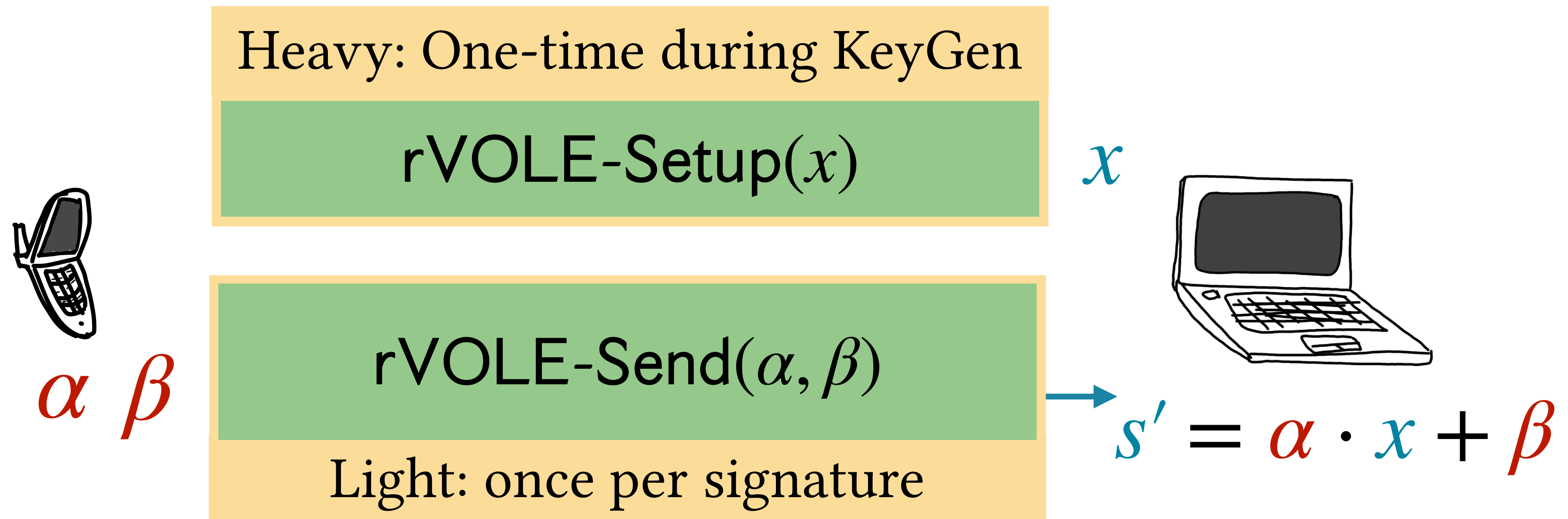


Sender provides a *vector* of inputs $(\vec{\alpha}_i, \vec{\beta}_i)$ applied to a single fixed receiver input x

Well known: $\text{VOLE}(n) < n \times \text{OLE}$

This work Generalize the concept to reactive Vector Oblivious Linear Evaluation

OLE Structure



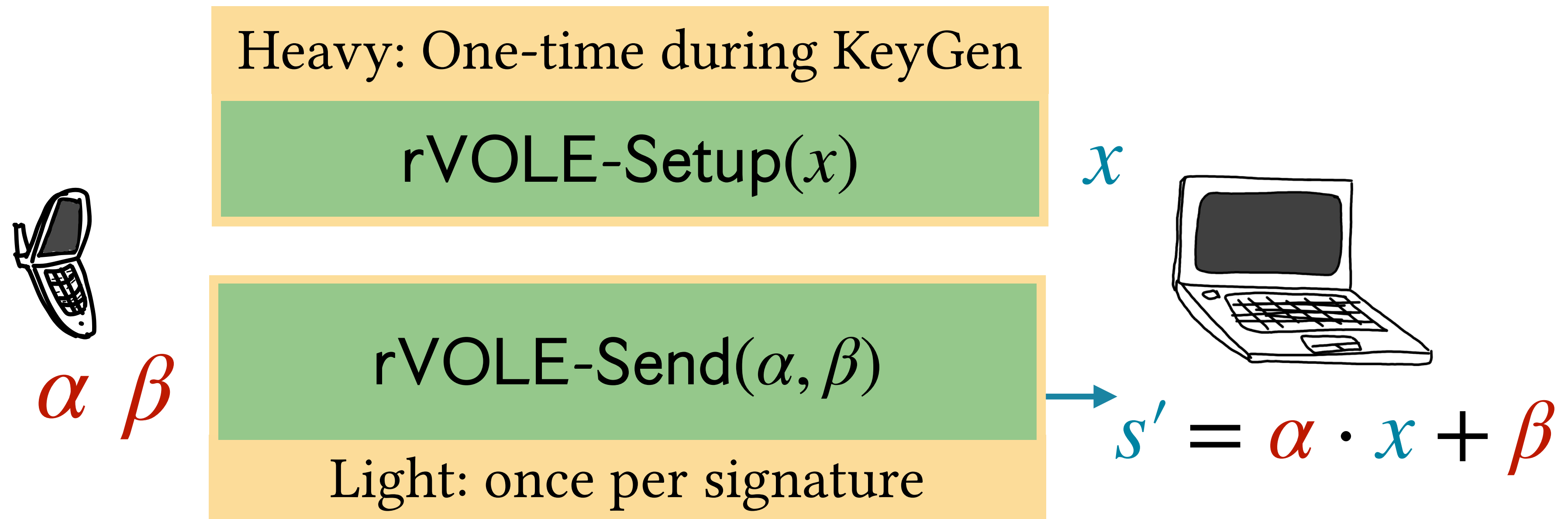
Sender's vector can be extended on demand

Sender provides a *vector* of inputs $(\vec{\alpha}_i, \vec{\beta}_i)$ applied to a single fixed receiver input x

Well known: $\text{VOLE}(n) < n \times \text{OLE}$

This work Generalize the concept to reactive Vector Oblivious Linear Evaluation

OLE Structure



Sender's vector can be extended on demand

Sender provides a *vector* of inputs $(\vec{\alpha}_i, \vec{\beta}_i)$ applied to a single fixed receiver input x

This work Generalize the concept to reactive Vector Oblivious Linear Evaluation

New construction with general, simple tools

rVOLE-Setup: Oblivious Transfer

rVOLE-Send: Any PRF + small field arithmetic

Constructing rVOLE

- Insight: rVOLE-Send can be instantiated with simpler, weaker tools than OLE (i.e. no need for homomorphic encryption or OT)
- [Roy22]: rVOLE over \mathbb{Z}_p (for small p)
 - Send is non-interactive, only local PRF eval and \mathbb{Z}_p add.s
- We need rVOLE over \mathbb{Z}_q per signing curve group order (large q)

How do we boost rVOLE(small p) \rightarrow rVOLE(big q)?

Constructing rVOLE

- Chinese Remainder Theorem: Classic method of emulating large integer arithmetic with smaller integers

$$M = \prod_{i \in [n]} p_i \text{ for a set of } n \text{ primes } p_1, p_2, \dots, p_n$$

$$\mathbb{Z}_M \cong \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \dots \times \mathbb{Z}_{p_n}$$

$$\text{rVOLE}(M) \cong \text{rVOLE}(p_1) \times \text{rVOLE}(p_2) \times \dots \times \text{rVOLE}(p_n)$$

- CRT used to decompose full OLE in [CCDKLRs20], [DHIM25]

Constructing rVOLE

- Chinese Remainder Theorem: Classic method of emulating large integer arithmetic with smaller integers

$$\text{rVOLE}(M) \cong \text{rVOLE}(p_1) \times \text{rVOLE}(p_2) \times \cdots \text{rVOLE}(p_n)$$

Roy22

Zero-communication PRF-based $\text{rVOLE}(p_i)$



Zero-communication PRF-based $\text{rVOLE}(M)$



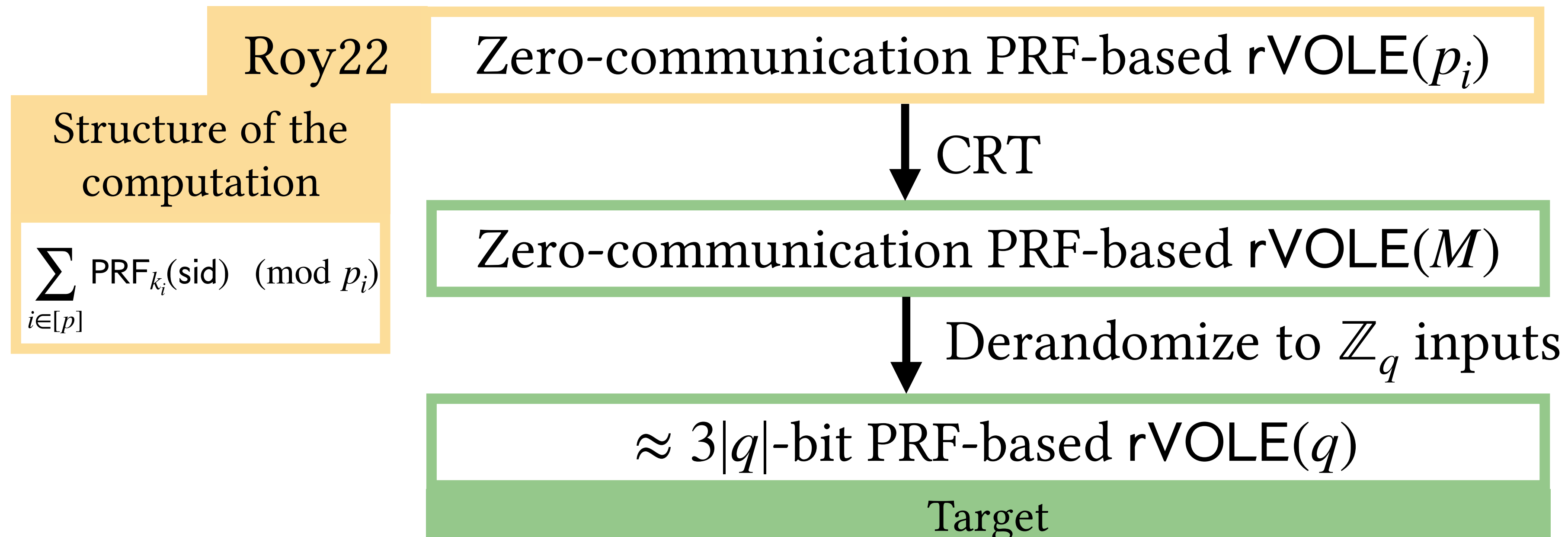
$\approx 3|q|$ -bit PRF-based $\text{rVOLE}(q)$

Target

Constructing rVOLE

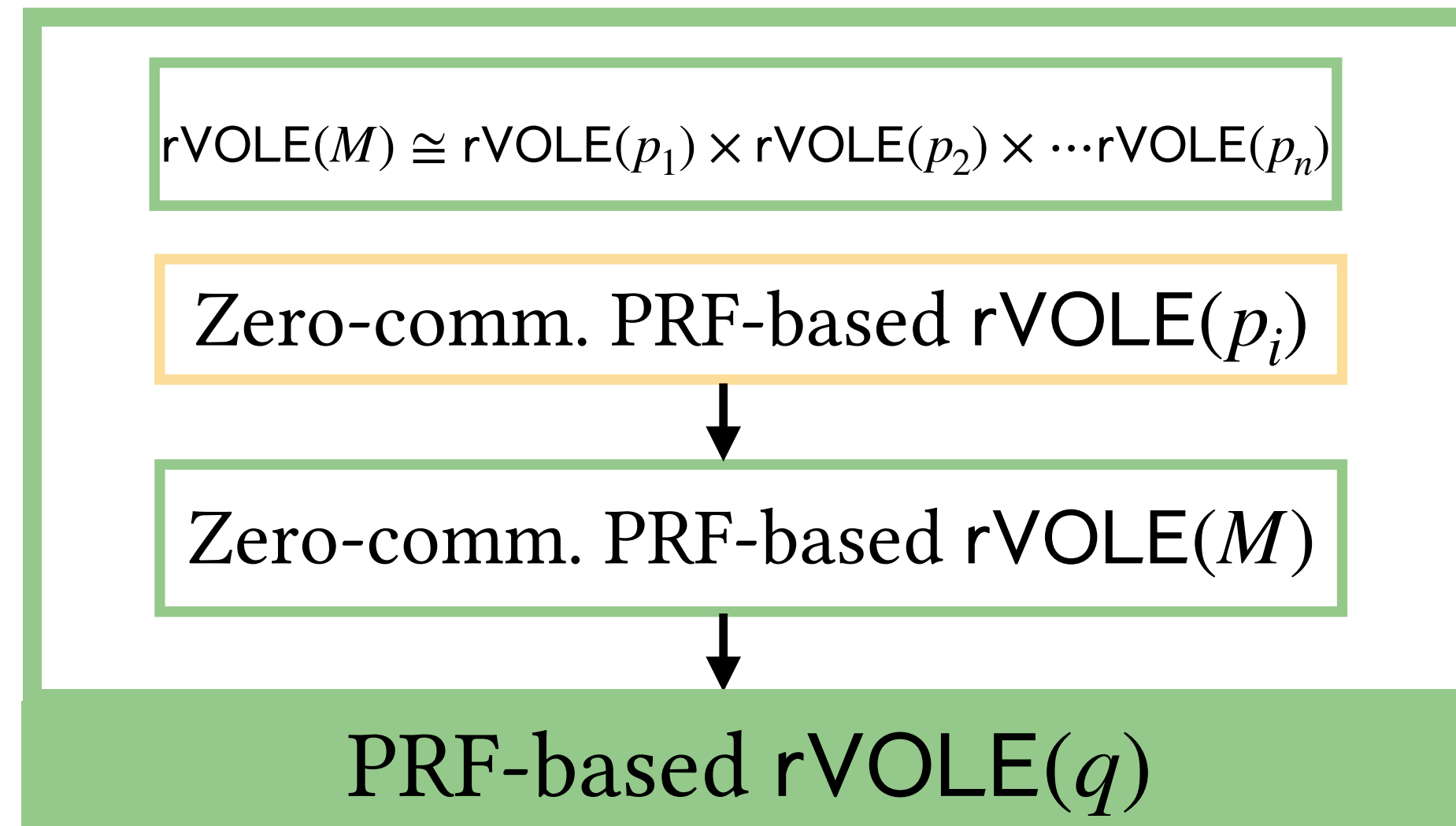
- Chinese Remainder Theorem: Classic method of emulating large integer arithmetic with smaller integers

$$\text{rVOLE}(M) \cong \text{rVOLE}(p_1) \times \text{rVOLE}(p_2) \times \cdots \text{rVOLE}(p_n)$$



Constructing rVOLE

- End result:



- rVOLE-Send that is cheaper and simpler than full OLE
 - $O(|q|)$ bits comm.: same as ECDSA itself
 - For a 32-byte curve: rVOLE-Send generates an 80 byte message in 0.5ms (compare against AHE ciphertexts at hundreds of bytes, several ms compute)
- rVOLE-Setup: OT-based protocol, 550ms, 490KB

MPC for ECDSA Recipe [DKLs 23]

- Retrospectively interpret protocols per their framework
- Applied to [Lindell 17]:
 1. “Multiplicative” rewriting of ECDSA
 2. Paillier-based OLE
 3. Verify that all operations were performed honestly

MPC for ECDSA Recipe [DKLs 23]

- Retrospectively interpret protocols per their framework
- Applied to [Lindell 17]:
 1. “Multiplicative” rewriting of ECDSA
 2. Paillier-based OLE
 3. Verify output to be a valid signature

MPC for ECDSA Recipe [DKLs 23]

- Retrospectively interpret protocols per their framework

- Applied to [Lindell 17]:

1. “Multiplicative” rewriting of ECDSA

2. Paillier-based OLE

3. Verify output to be a valid signature

We generalize the principle and apply it to prove malicious security of our scheme as well

Malicious Security Caveats

- Our proof inherits caveats of [Lin17]
- rVOLE inputs $(\alpha, \beta), x$ must be in the correct range
 - x is verified with an explicit one-time range proof at setup
 - α, β implicitly when signing: check for valid ECDSA sig
in case of fail, **abort all sessions, stop using key**
- Security reduces to ECDSA unforgeability, with #sessions loss
- As in [Lin17], this loss can be avoided with a custom assumption:

$$(x \cdot G, \text{abort}_x) \approx_c (x' \cdot G, \text{abort}_x)$$

Notes

- Incorporating nonce sampling, this yields:
 - 3-round signing protocol with uniform nonces
 - 2-round signing with biased nonces
(“doubly enhanced” unforgeability [ABCGJM 24])
- Our fully optimized construction makes use of Knowledge of Exponent to save some bytes
- Rust implementation, benchmarked ~1.6ms on M1 MacBook

In Conclusion

- New 2P-ECDSA signing that achieves $O(1)$ bandwidth overhead relative to simply exchanging plain ECDSA signatures.
- Insight: reactive VOLE suffices, simpler than full OLE
⇒ instantiable from OT+PRF, generalized analysis from [Lin17]
- Concrete bandwidth and computation costs (0.17KB, 1.6ms)
closest to plain ECDSA signing so far.



eprint.iacr.org/2025/1813

Thanks!

Thanks **Eysa Lee** for

