International Conference on Practice and Theory of Public-Key Cryptography (PKC) 2020

MonZa: fast maliciously-secure 2-party computation on the ring $\mathbb{Z}_{2^k}$

Dario Catalano[1], Mario Di Raimondo[1], Dario Fiore[2] and **Irene Giacomelli**[3]

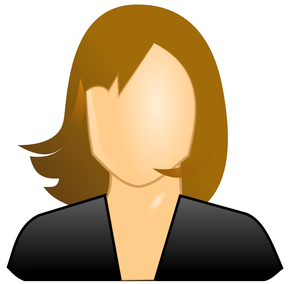
[1] Università di Catania,
[2] IMDEA Software Institute,
[3] Protocol Labs.

2-party secure computation

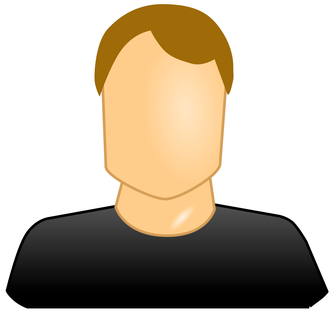
Problem:

Two parties, Alice and Bob, with private inputs, **a** and **b**, want to compute **c = f(a,b)** without revealing extra info on the private inputs.

Alice with private input a



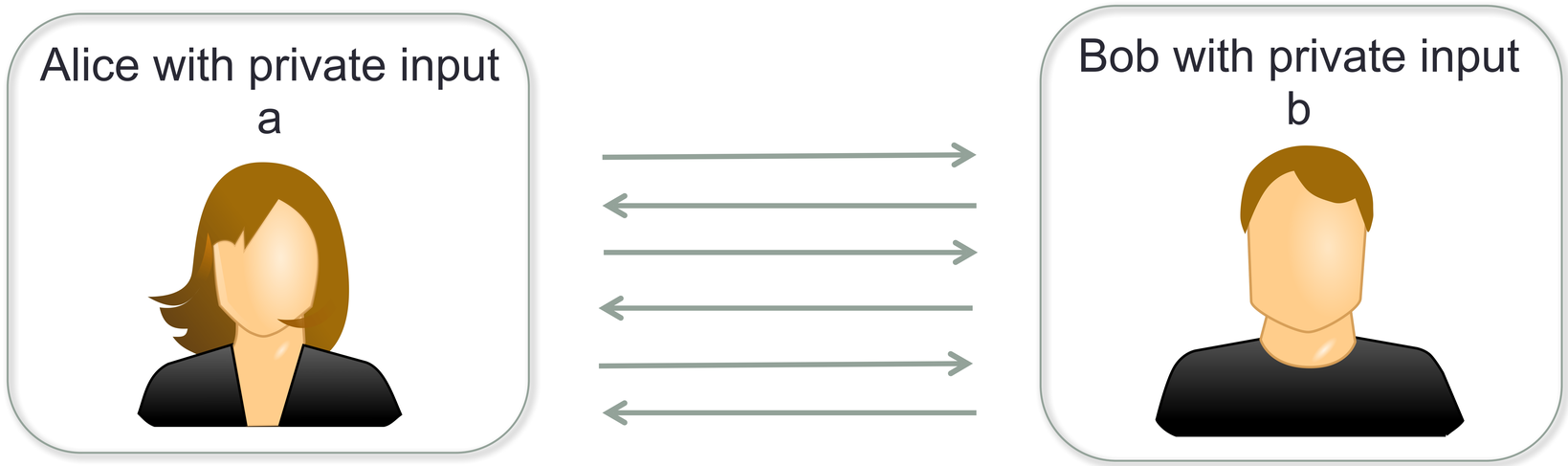
Bob with private input b



2-party secure computation

Solution:

Design an **interactive protocol** for Alice and Bob such that, at the end of its execution, they learn c = f(a,b) and nothing else.



Active security: executing the protocol in presence of a maliciously party is as secure as sending inputs to a trusted party who computes and returns only the output.

2-party secure computation over a ring

- Common: the function f is represented as
  - binary circuit
  - circuit over a finite field

- This work: f is represented as an arithmetic circuit over the **ring** $\mathbb{Z}_{2^k}$ (integers modulo 2^k)

$$f : (\mathbb{Z}_{2^k})^n \times (\mathbb{Z}_{2^k})^m \rightarrow (\mathbb{Z}_{2^k})^u$$

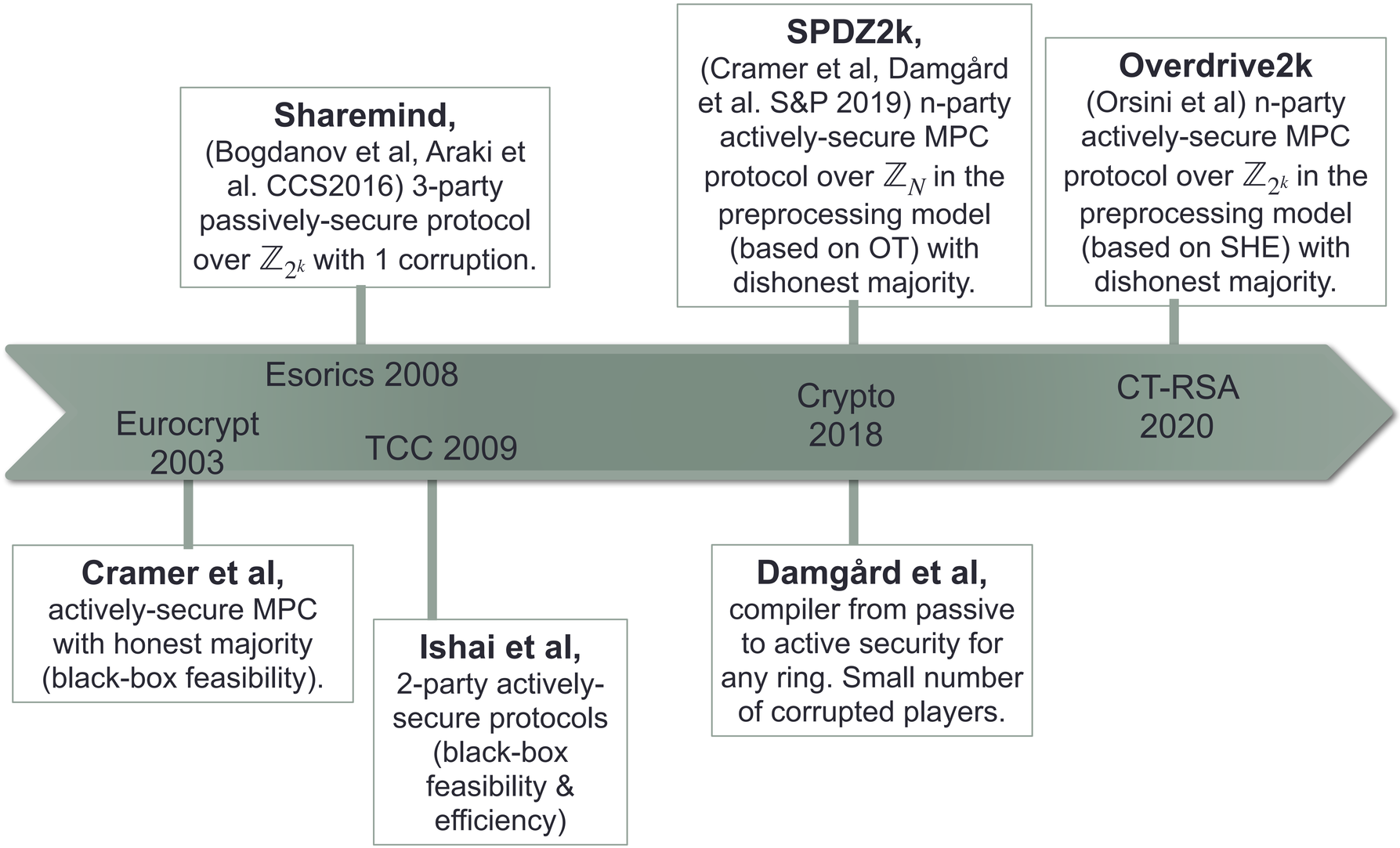
Why focus on $\mathbb{Z}_{2^k}$?

Integer arithmetic on standard CPUs is done modulo 2^k (eg, 32/64 bits), so an MPC protocol design that mirrors this can:

- *simplify implementation*
  (no need for modular arithmetic or to compensate modular reduction)

- *use optimizations* that are possible/done for CPU computations and that are often expensive to emulate modulo p.

MPC over a ring



Our contribution

MonZa

Efficient 2-party actively-secure protocol over $\mathbb{Z}_{2^k}$ in the preprocessing model

- Online phase: a la SPDZ2k
- Preprocessing (new!): efficient generation of **random triples** and **random elements** via the **Joye-Libert encryption scheme**
    1. linearly homomorphic
    2. works naturally with elements in $\mathbb{Z}_{2^k}$ and many instances with the same plaintext space
    3. efficient: fast encryption/decryption and small rate

        For example, decrypting a 120-bit plaintext using a 2048-bit modulus takes 4.8 ms (the equivalent with Paillier's scheme takes 9 to 5 ms, if exploiting CRT).
    4. no need of ZK proofs of plaintext knowledge/range-proofs

Everything solved? Nope…
Challenge: design a ZK proof of correct multiplication. Stay tuned!

Our contribution

MonZa

Efficient 2-party actively-secure protocol over $\mathbb{Z}_{2^k}$ in the preprocessing model

- Implemented in C, benchmarks on two servers Intel Xeon 8124M CPU runningat 3.0 GHz
- **Throughput:**

Latency (ms)	0.5 (LAN)	17 (WAN)	100 (WAN)
Triples/sec	19	18	17
Rand values/sec	134	132	121

input bit-length = 64
computational security = 112 bits
statistical security = 56 bits
batch size = 1000

★Notice (computational complexity): the pre-processing phase of MonZa is **asymmetric** (Alice has to decrypt, but Bob uses only faster operations)

→ MonZa can be used for applications in the server-client model, (one party has less computational power than the other one).

SPDZ-like protocol

- **Additive secret-sharing**: to hide the inputs and store the intermediate results.

$$\text{share}(a) = (a_1 , a_2) \text{ with } a_1 + a_2 = a$$

- **Information-theoretic MAC**: to guarantee active security

$$\text{MAC}(a) = \Delta \cdot a = m(a)_1 + m(a)_2$$

$\Delta = \Delta_1 + \Delta_2$, global random MAC-key.

Used in the SPDZ family for computation over a field,
adapted to work for computation over the ring $\mathbb{Z}_{2^k}$ by Cramer et al (Crypto 2018).

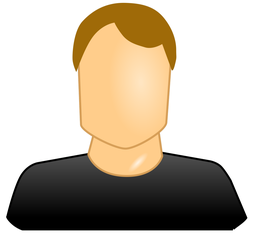
SPDZ2k - value representation

Key idea: to securely compute over $\mathbb{Z}_{2^k}$, share and authenticate over $\mathbb{Z}_{2^{k+s}}$

Δ random value in $\mathbb{Z}_{2^s}$ (fixed for the protocol) shared as $\Delta = \Delta_1 + \Delta_2 \bmod 2^{k+s}$

- a' in $\mathbb{Z}_{2^{k+s}}$ such that
  $a' = a \bmod 2^k$, $a' = a_1 + a_2 \bmod 2^{k+s}$, $m(a)_1 + m(a)_2 = \Delta \cdot a' \bmod 2^{k+s}$

MAC key share: Δ_1 (fixed)
Shares in $\mathbb{Z}_{2^{k+s}}$ *:* a_1 , b_1
MAC shares: $m(a)_1$, $m(b)_1$



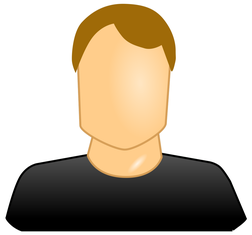
MAC key share: Δ_2 (fixed)
Shares in $\mathbb{Z}_{2^{k+s}}$*:* a_2 , b_2
MAC shares: $m(a)_2$, $m(b)_2$

SPDZ2k - online phase

Both the MAC and the secret-sharing scheme are homomorphic, so **linear operations** can be easily computed with no interaction!

Compute $a + b \bmod 2^k$:

MAC key share: Δ_1 (fixed)
shares: $a_1 + b_1$
MAC shares: $m(a)_1 + m(b)_1$

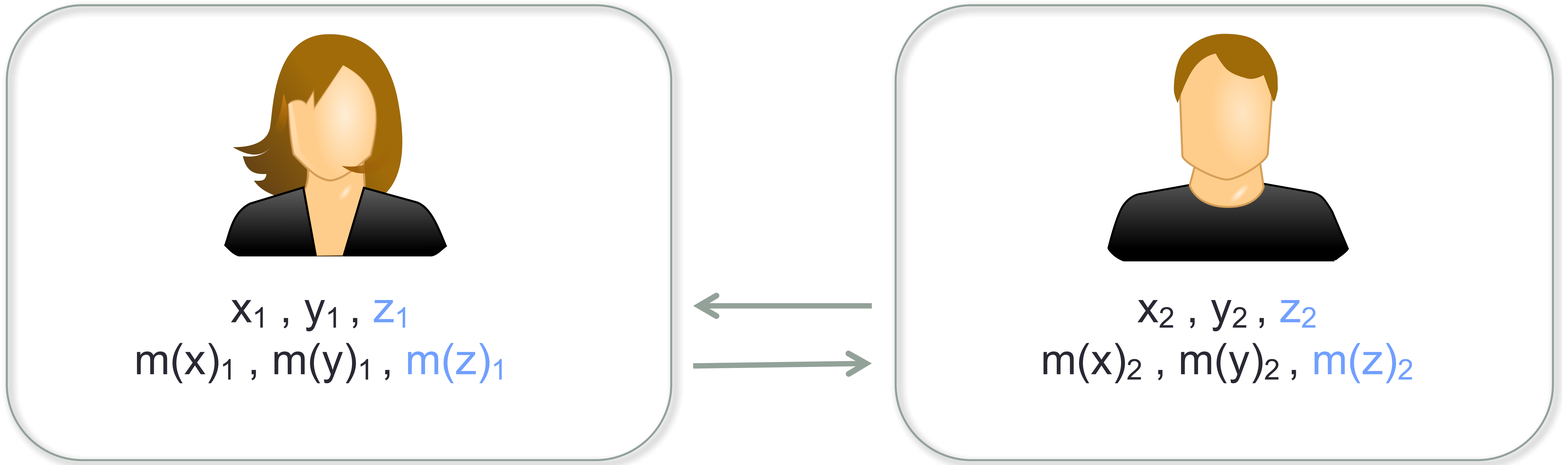


MAC key share: Δ_2 (fixed)
shares: $a_2 + b_2$
MAC shares: $m(a)_2 + m(b)_2$

SPDZ2k - online phase

Both the MAC and the secret-sharing scheme are homomorphic, so **linear operations** can be easily computed with no interaction!

Compute a + b mod 2^k:



Multiplication is harder, it needs a **random triple**: **x, y and z random elements** (in shared & authenticated form) **such that z = x · y**

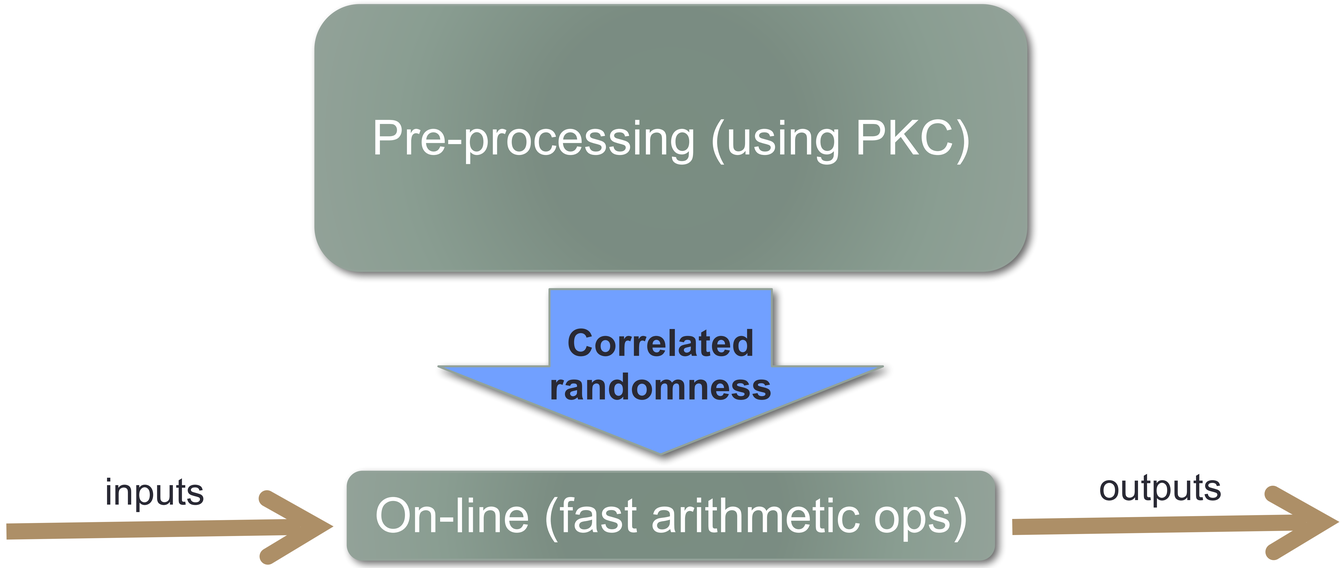
Given a triple, computing a · b mod 2^k can be done using Beaver's formula:

$$a \cdot b = (a+x) \cdot (b+y) + (a+x) \cdot y + (b+y) \cdot y + z$$

opened values

Preprocessing model

Random triples (and other correlated randomness) are created during a preprocessing phase (no inputs).



Triple construction

1. Take x and y at random

   (easy, each party choses its share at random)

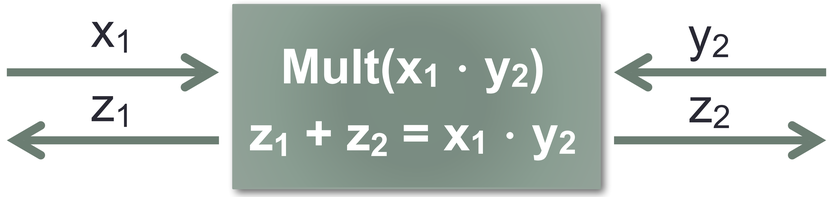
   $x = x_1 + x_2 \bmod 2^{k+s}$, $y = y_1 + y_2 \bmod 2^{k+s}$

   and compute shares of $MAC(x) = \Delta \cdot x \bmod 2^{k+s}$ and $MAC(y) = \Delta \cdot y \bmod 2^{k+s}$

2. Compute the shares of $z = x \cdot y \bmod 2^{k+s}$

3. Compute the shares of $MAC(z) = \Delta \cdot z \bmod 2^{k+s}$

For all compute, we need a protocol for multiplying two secret values!



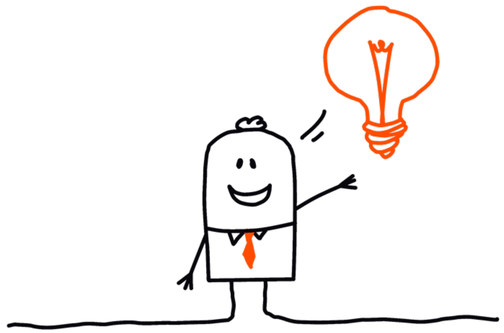
Multiplication of secret values

1) ***Linearly-homomorphic encryption*** (e.g., BeDOZa,Overdrive)
   - ZK proofs of plaintext knowledge (and range-proofs)
   - ZK proofs of correct multiplication (BeDOZa) or “SPDZ-sacrifice”

2) ***Somewhat homomorphic encryption*** (e.g., SPDZ, Overdrive2k)
   - ZK proofs of plaintext knowledge (and plaintext range)
   - Relatively expensive computation, RAM-intense

3) ***Oblivious transfer*** (e.g., Mascot, SPDZ2k)
   - Cheap computation with OT extension, but bandwidth intense
   - Need to mitigate selective failure

Our approach



Use **Joye-Libert (JL) scheme!**
(linearly-homomorphic encryption)

- The message space is $\mathbb{Z}_{2^n}$
- The public key is (N, g), where N = pq and g is an element of maximal order in $(\mathbb{Z}_N)^*$ and whose Jacobi symbol is 1

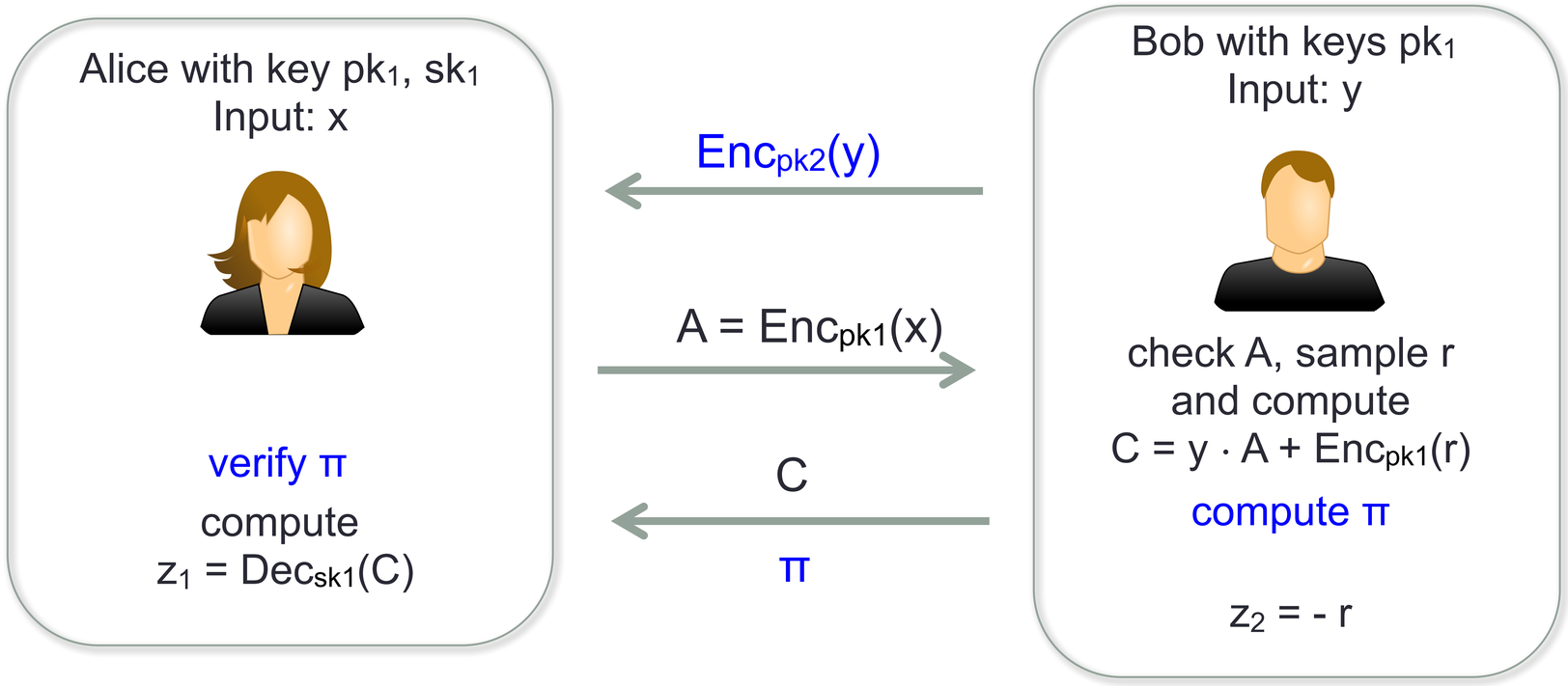
$$Jac_N(g) = Leg_p(g) \times Leg_q(g) = (g^{(p-1)/2} \bmod p) \times (g^{(q-1)/2} \bmod q)$$

- To encrypt $m \in \mathbb{Z}_{2^n}$, choose a random $x \in (\mathbb{Z}_N)^*$ and set

$$\mathbf{C = g^m \cdot x^{2^n} \bmod N}$$

Mult(x,y)

(Gilboa-like protocol)



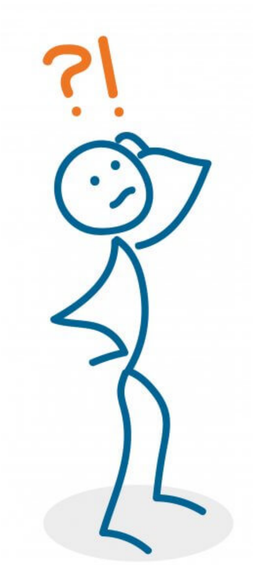
Correctness: $C = Enc_{pk1}(x \cdot y + r)$ and $z_1 + z_2 = Dec_{sk1}(Enc_{pk1}(x \cdot y + r)) - r = x \cdot y \bmod 2^n$

Security for Bob: easy!

Security for Alice: Bob needs to prove that the ciphertext C is computed in the correct way via a ZK proof π proving $C = y \cdot A + Enc_{pk1}(r)$ and $B = Enc_{pk2}(y)$ (y and r private inputs).

Challenge:
Design π, the ZK proof for correct multiplication with JL!

No such protocol exists for JL !



- Overdrive's approach needs an encryption with enhanced CPA (i.e., non-linear operations on ciphertexts are not possible)

- Standard Schnorr-like protocol techniques do not work due to the message space not being a field (or $\mathbb{Z}_{pq}$).
  In $\mathbb{Z}_{2^n}$ there are several and efficiently-findable **noninvertible** elements, so novel techniques needed to prove **soundness!**

Goal: ZK-proof for correct multiplication with JL scheme

Bob's witness: messages y and r in $\mathbb{Z}_{2^n}$

Public inputs: ciphertexts A, B and C

Statement: $C = y \cdot A + Enc_{pk1}(r)$ and $B = Enc_{pk2}(y)$

For the sake of simplicity, in this talk I'll focus on:
ZK-proof of knowledge for a JL plaint text

Bob's witness: messages m in $\mathbb{Z}_{2^n}$

Public inputs: ciphertext **C**

Statement: C = Enc(m)

ZK-proof for JL scheme

Bob's witness: messages m in $\mathbb{Z}_{2^n}$

Public inputs: ciphertext **C**

Statement: $C = Enc(m) = g^{m} \cdot x^{2^n} \bmod N$

A Schnorr-like protocol goes like this:

sample e in $\mathbb{Z}_{2^s}$

$g^{z} \cdot y^{2^n} = S \cdot C^{e} \bmod N$?

S = Enc(s)

e

z ,y

sample s and w

$S = g^{s} \cdot w^{2^n} \bmod N$

compute

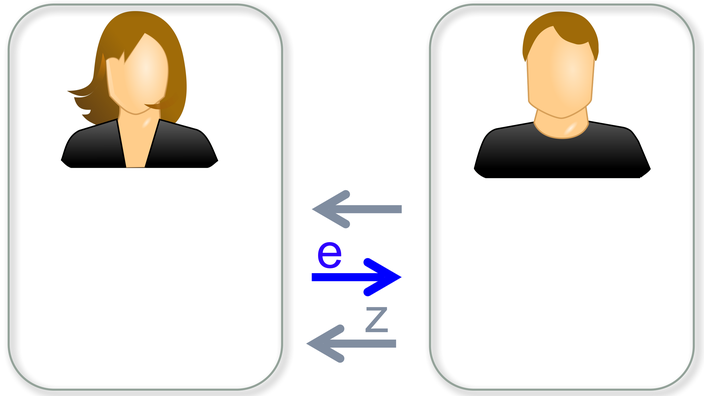
$z = s + m \cdot e \bmod 2^n$

$y = g^{t} \cdot w \cdot x^{e} \bmod N$

with t s.t. $t\ 2^n = s + m \cdot e - z$

ZK-proof for JL scheme

Soundness:
if Bob can answer to two challenges $e \neq e'$, then
m is computed as $m = (z - z') \cdot (e - e')^{-1} \bmod 2^n$.



Problem: in $\mathbb{Z}_{2^n}$ the value $e - e' \neq 0$ can be non invertible!

Solution: we show that **g.c.d.(e - e', 2^n) = 2^t for some $t \leq s < n$**
and how to **extract n-s bits** of the message m.

Conclusion:
to prove over $\mathbb{Z}_{2^n}$ we need to work with JL with a larger message space, $\mathbb{Z}_{2^{n+s}}$!
This is not an efficiency problem, ciphertext length stays the same!

Triple construction – all together

Parties have MAC-key shares, Δ_1 and Δ_2, and input shares

$x = x_1 + x_2 \bmod 2^n$, $y = y_1 + y_2 \bmod 2^n$

1. Run Mult(Δ_i,x_{3-i}) and Mult(Δ_i,y_{3-i}) with i=1,2 to compute MAC(x) and MAC(y)
2. Run Mult(x_1,y_2) and Mult(x_2,y_1) to compute $z_1 + z_2 = x \cdot y \bmod 2^n$
3. Run Mult(Δ_i,z_{3-i}) with i=1,2 to compute MAC(z)

Notice:

- we need two extra ZK proofs to prove that a party uses the correct value in step 3 (variant of the ZK proof of correct multiplication)
- “SPDZ sacrifice” is not needed!

Bandwidth

k = bit size of the inputs
s = stat. security parameter
N = ciphertext bit length

1 triple = 78 N + 18 (k+2s) bits sent between Alice and Bob
(reduced to 56 N + 18 (k+2s) using the random-oracle)

For example, for 80-bit computational security and s = 40, < 70 kbit for a triple in Z2^128.

				$\mathbf{SPD}\mathbb{Z}_{2^k}$		$\mathbf{Mon}\mathbb{Z}_{2^k}$ **a base**		$\mathbf{Mon}\mathbb{Z}_{2^k}$ **a optim.**	
S	$\lvert N\rvert$	k	s	triple	input	triple	input	triple	input
80	1024	32	32	79.87	**3.17**	81.60	9.41	**59.07**	6.34
		64	40	177.41	**5.90**	82.46	9.50	**59.94**	6.43
		128	40	362.75	8.53	94.22	10.86	**68.70**	**7.38**
112	2048	32	32	**79.87**	**3.17**	161.47	18.62	116.42	12.48
		64	56	267.52	**10.03**	162.91	18.78	**117.86**	12.64
		128	56	487.68	13.68	164.06	18.91	**119.01**	**12.77**
128	3072	32	32	**79.87**	**3.17**	241.34	27.84	173.76	18.62
		64	64	319.49	**12.48**	243.07	28.03	**175.49**	18.82
		128	64	557.06	**16.64**	244.22	28.16	**176.64**	18.94

(S: comp. sec. level; N: JL-schemes modulus; k: message bit-length; s: stat. sec. level)

Cost in kbit

Future work

- Extension to n parties
- Design batch verifications for the ZK-proofs of correct multiplication
- Exploit JL unique properties to design sub-protocols (e.g., secure comparison)

Thanks for the attention!

https://eprint.iacr.org/2019/211