MPC Compiler

Chunxu Tang

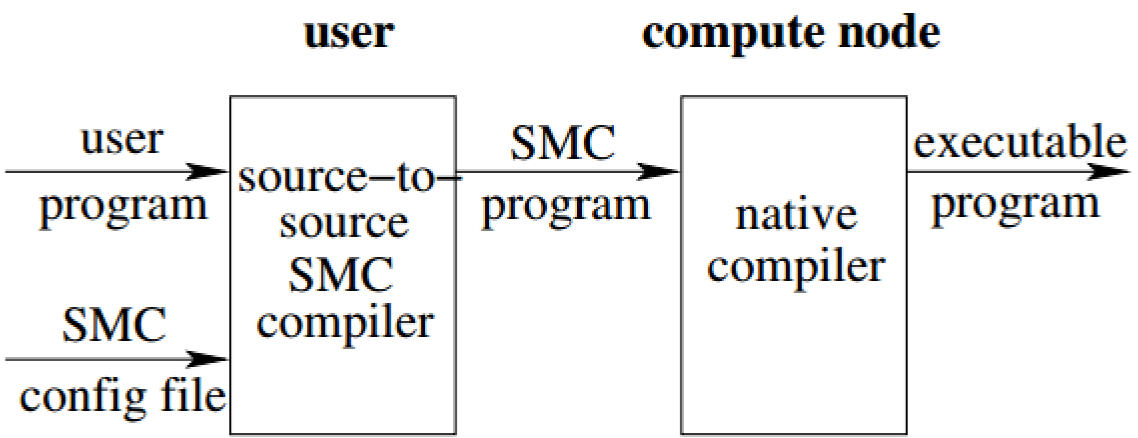
Haoyi Shi

PICCO: A General-Purpose Compiler for Private Distributed Computation

Chunxu Tang

PICCO (Private Distributed Computation Compiler)

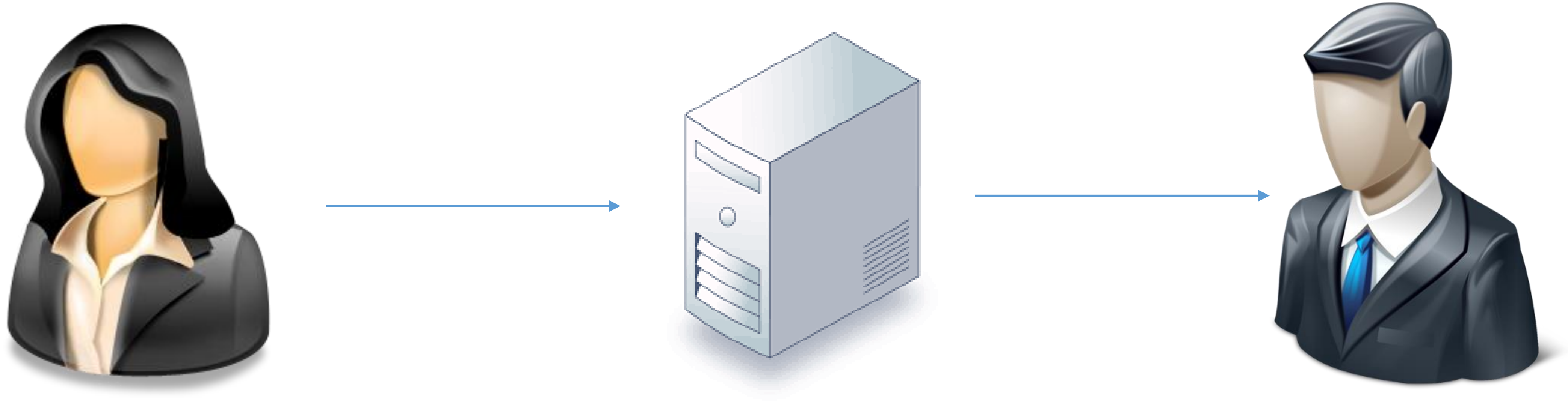
- A source-to-source compiler that translates a program written in an extension of the C programming language with provisions for annotating private data to its secure distributed implementation in C.



(a) Compilation

Framework

Input party Computational party Output party



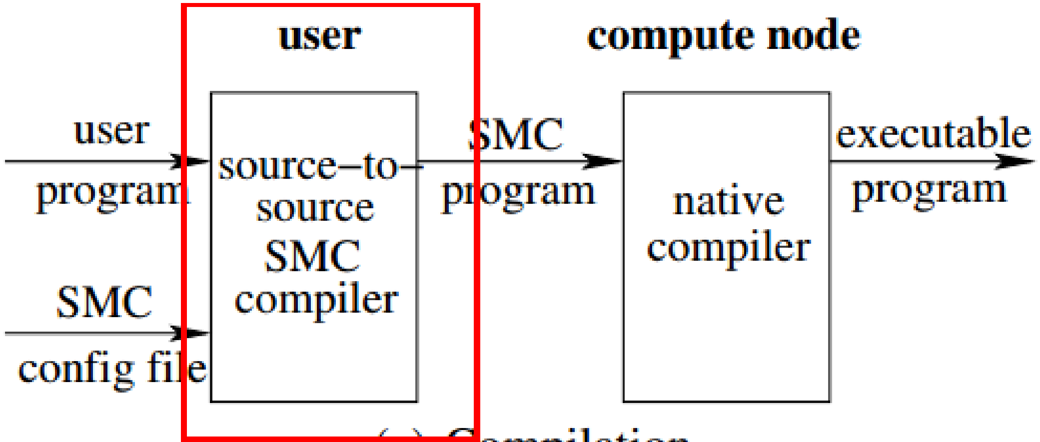
Framework (Cont.)

- (n, t) – secret sharing scheme
  - Any private value is secret-shared among n parties such that any t+1 shares can be used to reconstruct the secret

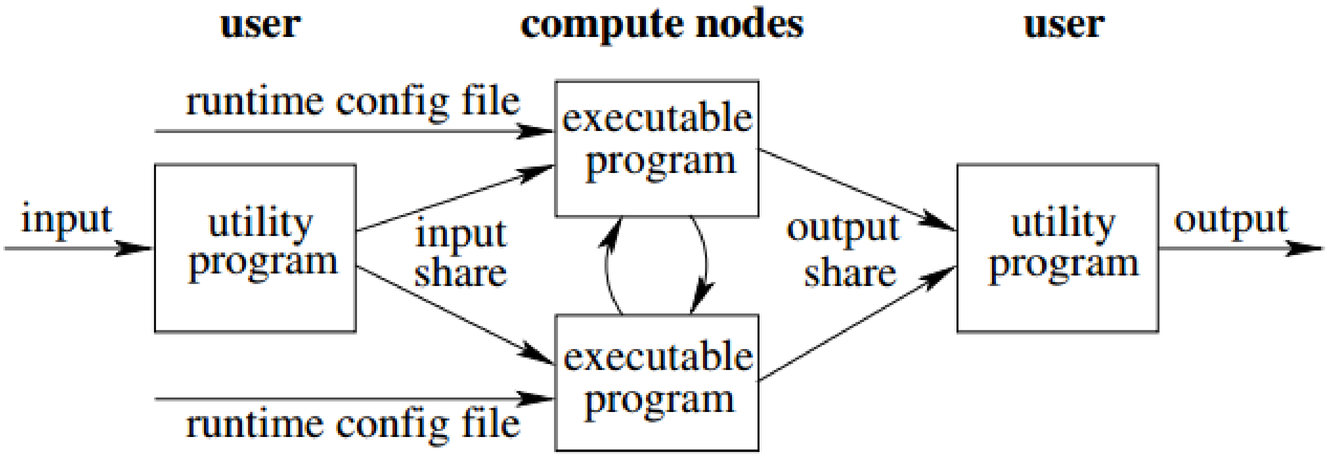
- Shamir secret sharing scheme
  - A secret value s is represented by a random polynomial of degree t with the free coefficient set to s.

- Participants are semi-honest

Overview



(a) Compilation



(b) Execution

Specifications of user programs

- Private and public variable qualifiers
  - private int x;
  - int x;

```
public int main() {
    public int i, M;
    smcinput(M, 1, 1);
    private int<1> A[M], B[M];
    private int<10> dist = 0;

    smcinput(A, 1, M);
    smcinput(B, 1, M);
    for (i = 0; i < M; i++)
      dist += A[i] ^ B[i];

    smcoutput(dist, 1);
    return 0;
}
```

Private data types

- A programmer can specify the length of the numeric data types in bits.

```
public int main() {
   public int i, M;
   smcinput(M, 1, 1);
   private int<1> A[M], B[M];
   private int<10> dist = 0;

   smcinput(A, 1, M);
   smcinput(B, 1, M);
   for (i = 0; i < M; i++)
     dist += A[i] ^ B[i];

   smcoutput(dist, 1);
   return 0;
}
```

Built-in I/O functions

- smcinput(name, id)
  - name: name of the variable to read
  - id: id of the input party
  - smcinput(x, 1);

- smcoutput(name, id)
  - name: name of the output variable
  - id: id of the output party

```
public int main() {
    public int i, M;
    smcinput(M, 1, 1);
    private int<1> A[M], B[M];
    private int<10> dist = 0;

    smcinput(A, 1, M);
    smcinput(B, 1, M);
    for (i = 0; i < M; i++)
      dist += A[i] ^ B[i];

    smcoutput(dist, 1);
    return 0;
}
```

?

Array operations

- A @ B
  - element-wise multiplication
- smcinput (A, 1, 100)
  - read 100 values into array A from
  - the data of party 1

```
public int main() {
    public int i, M;
    smcinput(M, 1, 1);
    private int<1> A[M], B[M];
    private int<10> dist = 0;

    smcinput(A, 1, M);
    smcinput(B, 1, M);
    for (i = 0; i < M; i++)
      dist += A[i] ^ B[i];

    smcoutput(dist, 1);
    return 0;
}
```

Enforcement of secure data flow

- Statements that assign an expression that contains private values to a public variable are not allowed.

```
x = a;

x: public
a: private
```

- For conditional statements with a private condition, assignments to public variables within the scope of such statements are not allowed.

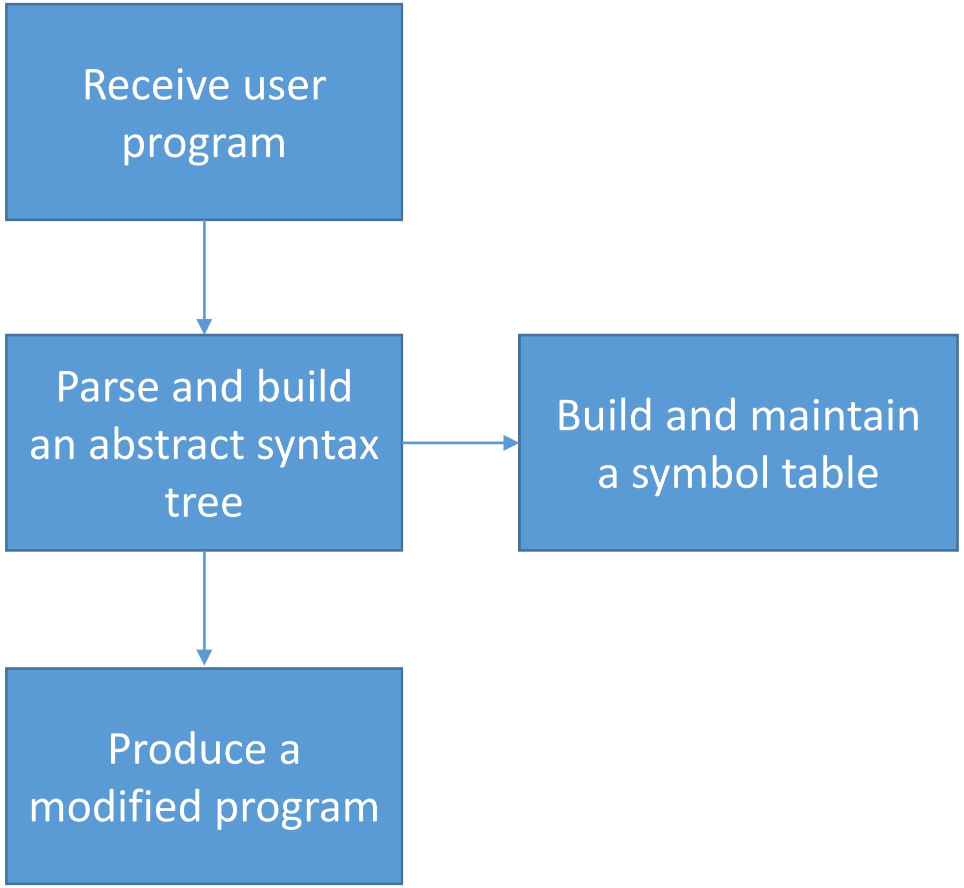
```
if (x > 0)
  a = 1;

x: private
a: public
```

Support for concurrent execution

- for (statement; condition; statement)
-     [statement; …]

- [statement1;]
- [statement2;]

Processing of user programs



Program transformations

- GMP library
  - GNU Multiple Precision Arithmetic library
- Private variables -> GMP large-precision type mpz_t
- Changing arguments of functions with private return values

Handling of program input and output

- Take smcinput(*var*, *i*) as an example:
  - The compiler looks up the type of variable *var* in the symbol table that stores all declared variables.
  - Replace with instructions to read data from party *i*.
  - Type of variable *var* determines how many fields are used to represent the variable and length.

Handling of private data types in assignments

- Produce a terminal error if a private expression is being assigned to a public variable.

- A function call is used, but its return type is not known, the compiler displays a warning of a potential violation of secure data flow.

Handling of conditional statements

- if-statements with private conditions are not allowed to contain observable public actions in their body.

- Produce a terminal error when a violation is found.

Handling of conditional statements (Cont.)

- Determine all variables whose values are modified, and preserve their values in temporary variables.

- Update each affected variable *v* by setting its value to $c \cdot v + (1 - c)v_{orig}$
  - c : private bit corresponding to the result of evaluating the condition
  - v_{orig}: original value of v prior to executing the body of if-statement.

Handling of conditional statements (Cont.)

```
if (t>0)
  for (i=0; i<n; i+=5)
    a[i]=a[i]+1;
```

→

```
mpz_t cond1;
mpz_t tmp1;
smc_gt(t,0,cond1);
for (i=0; i<n; i+=5) {
  tmp1=a[i];
  a[i]=a[i]+1;
  a[i]=cond1*a[i]+(1-cond1)*tmp1;
}
```

Modulus computation

- Compute the maximum bit length of all declared variables and maximum bit length necessary for carrying out the specified operations.

Evaluation

Experiment	Modulus p length (bits)	Basic functionality		Optimized functionality		Sharemind		Two-party compiler [29]	
		LAN (ms)	WAN (ms)	LAN (ms)	WAN (ms)	LAN (ms)	WAN (ms)	LAN (ms)	WAN (ms)
100 arithmetic operations	33	1.40	315	**0.18**	**31.6**	71	203	1,198	1,831
1000 arithmetic operations	33	13.4	3,149	**0.60**	**32.3**	82	249	3,429	5,823
3000 arithmetic operations	33	42.7	9,444	**1.60**	**34.5**	127	325	10,774	11,979
5×5 matrix multiplication	33	17.7	3,936	**0.27**	**31.6**	132	264	3,419	5,244
8×8 matrix multiplication	33	67.8	16,126	**0.45**	**32.1**	168	376	18,853	21,843
20×20 matrix multiplication	33	1,062	251,913	**2.41**	**35.7**	1,715	2,961	N/A	N/A
Median, mergesort, 32 elements	81	703.7	98,678	**256.7**	6,288	7,115	22,208	4,450	**5,906**
Median, mergesort, 64 elements	81	1,970	276,277	**649.6**	**12,080**	15,145	47,636	N/A	N/A
Median mergesort, 256 elements	81	13,458	1,894,420	**3,689**	**47,654**	66,023	203,044	N/A	N/A
Median mergesort, 1024 elements	81	86,765	–	**20,579**	**170,872**	317,692	869,582	N/A	N/A
Hamming distance, 160 bits	9	21.2	5,038	**0.17**	**31.1**	72	188	793	816
Hamming distance, 320 bits	10	42.3	10,092	**0.22**	**31.3**	102	203	850	1,238
Hamming distance, 800 bits	11	105.8	25,205	**0.35**	**31.5**	117	254	933	989
Hamming distance, 1600 bits	12	212.7	50,816	**0.57**	**31.8**	132	284	1,037	1,265
AES, 128-bit key and block	8	319.1	75,874	**35.1**	**3,179**	652 [34]	N/A	N/A	N/A
Edit distance, 100 elements	57	48,431	9,479,330	**4,258**	**116,632**	69,980	214,286	N/A	N/A
Edit distance, 200 elements	57	201,077	–	**16,038**	**432,456**	196,198	498,831	N/A	N/A
Fingerprint matching, 20 minutiae	66	3,256	541,656	**830**	**74,704**	24,273	75,820	N/A	N/A
Fingerprint matching, 40 minutiae	66	13,053	2,140,630	**2,761**	172,455	55,088	**172,266**	N/A	N/A

Thank you!

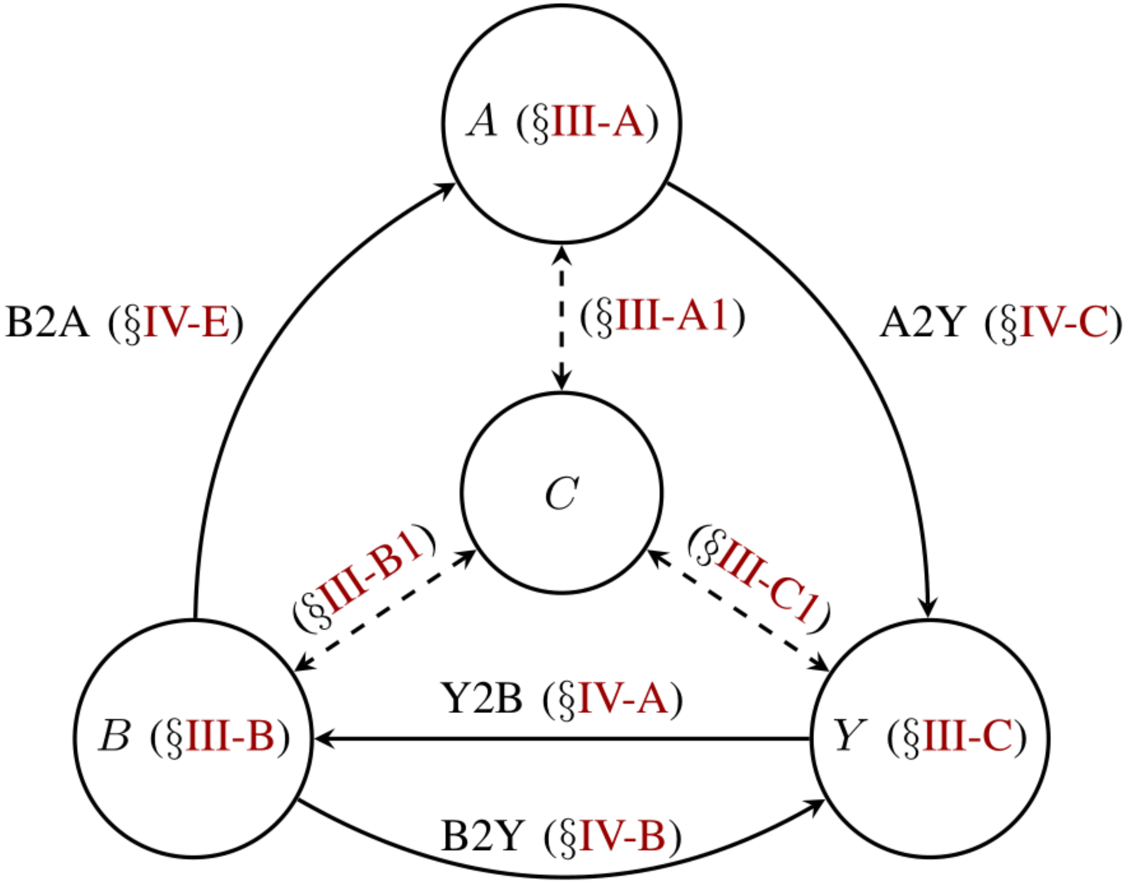
A Framework for Efficient Mixed-Protocol Secure Two-Party computation

Haoyi Shi

ABY Framework

- Two-party framework
- Mixed protocols
  - Overcome the dependence on an efficient function representation
  - **A**rithmetic Sharing
  - **B**oolean Sharing
  - **Y**ao’s garbled circuit

overview



A (§III-A)
B2A (§IV-E)
(§III-A1)
A2Y (§IV-C)
C
(§III-B1)
(§III-C1)
Y2B (§IV-A)
B (§III-B)
Y (§III-C)
B2Y (§IV-B)

Arithmetic Sharing

- Shared value: $\langle x\rangle_0^A + \langle x\rangle_1^A \equiv x \pmod{2^\ell}$ $\langle x\rangle_0^A, \langle x\rangle_1^A \in \mathbb{Z}_{2^\ell}$
- Sharing: $\mathsf{Shr}_i^A(x)$ P_i chooses $r \in_R \mathbb{Z}_{2^\ell}$, sets $\langle x\rangle_i^A = x - r$, and sends r to P_{1-i}, who sets $\langle x\rangle_{1-i}^A = r$
- Reconstruction: $\mathsf{Rec}_i^A(x)$ P_{1-i} sends its share $\langle x\rangle_{1-i}^A$ to P_i who computes $x = \langle x\rangle_0^A + \langle x\rangle_1^A$.
- Addition:

$\langle z\rangle^A = \langle x\rangle^A + \langle y\rangle^A$: P_i locally computes $\langle z\rangle_i^A = \langle x\rangle_i^A + \langle y\rangle_i^A$.

Arithmetic Sharing

- Multiplicaton: $\langle z\rangle^A = \langle x\rangle^A \cdot \langle y\rangle^A$
  - Pre-computed triple: $\langle c\rangle^A = \langle a\rangle^A \cdot \langle b\rangle^A$

    P_i sets $\langle e\rangle_i^A = \langle x\rangle_i^A - \langle a\rangle_i^A$ and $\langle f\rangle_i^A = \langle y\rangle_i^A - \langle b\rangle_i^A$,
    both parties perform $\mathsf{Rec}^A(e)$ and $\mathsf{Rec}^A(f)$, and P_i sets

    $$\langle z\rangle_i^A = i \cdot e \cdot f + f \cdot \langle a\rangle_i^A + e \cdot \langle b\rangle_i^A + \langle c\rangle_i^A.$$

  - Use OT to generate multiplication triple.

Sharing Conversion

- Yao to Boolean Sharing(Y2B)
  - The permutation bits of $\langle x \rangle_0^Y$ and $\langle x \rangle_1^Y$ $\langle x \rangle_1^Y[0] = 1 - \langle x \rangle_0^Y[0]$
  - For P_i, $\langle x \rangle_i^B = Y2B(\langle x \rangle_i^Y) = \langle x \rangle_i^Y[0]$.
- Boolean to Yao Sharing (B2Y)
  - Let $x_0 = \langle x \rangle_0^B$ and $x_1 = \langle x \rangle_1^B$.
  - P0 samples $\langle x \rangle_0^Y = k_0 \in_R \{0,1\}^\kappa$. Both parties run OT_κ^1 where P_0 acts as sender with inputs $(k_0 \oplus x_0 \cdot R; k_0 \oplus (1 - x_0) \cdot R)$, whereas P_1 acts as receiver with choice bit x_1 and obliviously obtains $\langle x \rangle_1^Y = k_0 \oplus (x_0 \oplus x_1) \cdot R = k_x$.

Sharing Conversion

- Arithmetic to Yao Sharing (A2Y)
  - Let $x_0 = \langle x \rangle_0^A$ and $x_1 = \langle x \rangle_1^A$

    $\langle x_0 \rangle^Y = \mathsf{Shr}_0^Y(x_0)$ and $\langle x_1 \rangle^Y = \mathsf{Shr}_1^Y(x_1)$ and compute $\langle x \rangle^Y = \langle x_0 \rangle^Y + \langle x_1 \rangle^Y$.
- Arithmetic to Boolean Sharing (A2B)

  $$\langle x \rangle^B = A2B(\langle x \rangle^A) = Y2B(A2Y(\langle x \rangle^A))$$
- Yao to Arithmetic Sharing (Y2A)

  $$\langle x \rangle^A = Y2A\left(\langle x \rangle^Y\right) = B2A\left(Y2B\left(\langle x \rangle^Y\right)\right)$$

- Boolean to Arithmetic Sharing
  - Perform an OT for each bit.

P_0 P_1

randomly chooses $r_i \in_R \{0,1\}^\ell$

$s_{i,0} = \left(1 - \langle x \rangle_0^B[i]\right) \cdot 2^i - r_i$

$s_{i,1} = \langle x \rangle_0^B[i] \cdot 2^i - r_i$

$(s_{i,0}, s_{i,1})$ ⟶

$\langle x \rangle_1^B[i]$

$s_{\langle x \rangle_1^B[i]} = \left(\langle x \rangle_0^B[i] \oplus \langle x \rangle_1^B[i]\right) \cdot 2^i - r_i$

- Finally, P_0 compute $\langle x \rangle_0^A = \sum_{i=1}^{\ell} r_i$
- P_1 compute $\langle x \rangle_1^A = \sum_{i=1}^{\ell} s_{\langle x \rangle_1^B[i]} = x - \langle x \rangle_0^A$

Benchmark the primitive operations

- In local settings, conversion cost is small.
  - E.g, converting from Yao to Arithmetic shares, multiplying, and converting back to Yao, is more efficient than performing muiltiplication in Yao sharing.

Sharing	**MUL**		**CMP**		**MUX**	
	size	rounds	size	rounds	size	rounds
Arithmetic	$\boldsymbol{\ell}$	1	—	—	—	—
Boolean	$2\ell^2$	ℓ	3ℓ	$\log_2 \ell$	**1**	1
Yao	$2\ell^2$	**0**	$\boldsymbol{\ell}$	**0**	ℓ	**0**

Benchmark the primitive operations

- Latency(seq)
  - The best performance for sequential functions depends on the latency.
  - E.g, multiplication in Yao is more efficient in the cloud settings.

- Throughput(par)
  - Arithmetic and Boolean sharing benefit more than Yao sharing.

Arithmetic Sharing			
Op	**Local**	**Cloud**	**Comm**
ADD	0	0	0
MUL	17	1 712	1 152

B*2*A		
Local	**Cloud**	**Comm**
14	722	512

A*2*Y		
Local	**Cloud**	**Comm**
39	2 289	2 048

Boolean Sharing			
Op	**Local**	**Cloud**	**Comm**
ADD	118	1 889	7 424
MUL	622	5 209	64 512
XOR	0	0	0
AND	11	1 071	1 024
CMP	39	1 293	2 848
EQ	14	1 074	992
MUX	1	295	32

Y*2*B		
Local	**Cloud**	**Comm**
6	7	0

B*2*Y		
Local	**Cloud**	**Comm**
13	816	512

Yao Sharing			
Op	**Local**	**Cloud**	**Comm**
ADD	40	2 313	1 536
MUL	1 003	11 402	96 768
XOR	18	1 148	0
AND	38	2 309	1 536
CMP	38	2 296	1 536
EQ	38	2 305	1 488
MUX	39	2 292	1 536

Fig. 2: Setup time (in μs) and communication (in Bytes) for a single atomic operation on $\ell = 32$-bit values in a local and cloud scenario, averaged over 1 000 operations using long-term security parameters.

Arithmetic Sharing

Op	Local		Cloud		Comm
	Seq	Par	Seq	Par	
ADD	1	1	0	0	0
MUL	138	2	237 411	209	4

A (§III-A)

$B2A$

Local		Cloud		Comm
Seq	Par	Seq	Par	
14	5	419 120	484	66

$A2Y$

Local		Cloud		Comm
Seq	Par	Seq	Par	
27	8	434 588	466	1 028

$Y2B$ (Seq / Par)

Local	Cloud	Comm
4 / 2	8 / 2	0

B (§III-B)

Y (§III-C)

Boolean Sharing

Op	Local		Cloud		Comm
	Seq	Par	Seq	Par	
ADD	1 879	5	2 469 216	2 346	116
MUL	9 703	15	6 740 331	6 575	1 008
XOR	3	1	5	1	0
AND	136	2	192 133	231	16
CMP	717	4	1 078 922	1 076	45
EQ	614	3	1 128 954	1 087	16
MUX	120	1	224 880	216	8

$B2Y$ (Seq / Par)

Local	Cloud	Comm
7 / 5	478 959 / 479	516

Yao Sharing

Op	Local		Cloud		Comm
	Seq	Par	Seq	Par	
ADD	28	7	41	10	0
MUL	970	143	1 280	165	0
XOR	5	2	7	3	0
AND	11	5	16	7	0
CMP	27	7	32	9	0
EQ	19	7	28	7	0
MUX	20	7	25	7	0

Fig. 3: Online time (in μs) and communication (in Bytes) for one atomic operation on $\ell = 32$-bit values in a local and cloud scenario, averaged over 1 000 sequential / parallel operations using long-term security parameters.

Biometric Matching

- One party provides a biometric sample.
- The other party (DB) provides several biometric samples.
- Matching: Euclidean distance.

$$min\left(\sum_{i=1}^{d}(S_{i,1}-C_i)^2,\cdots,\sum_{i=1}^{d}(S_{i,n}-C_i)^2\right)$$

- 4 instantiations
  - B-only
  - Y-only
  - A+Y
  - A+B

Biometric Matching

- Mixed protocols perform better
  - Communication improves by at least a factor of 20
- Arithmetic sharing(OT-based) is better than homomorphic encryption

	Local			**Cloud**			**Comm. [MB]**	**#Msg**
	S	O	T	S	O	T		
Y-only	2.24	0.31	2.55	23.78	0.84	24.62	147.7	**2**
B-only	2.15	0.28	2.43	10.34	29.07	39.41	99.9	129
A+Y	0.14	**0.05**	**0.19**	2.98	**0.44**	**3.42**	5.0	8
A+B	0.08	0.13	0.21	2.34	24.07	26.41	**4.6**	101