Formal Verification of the *Fantastic Four** Secure Multiparty Computation Protocol

CS 599 Final Project

Eli Baum

Prateek Jain

* Dalskov, Anders, Daniel Escudero, and Marcel Keller. "Fantastic four: Honest-Majority Four-Party secure computation with malicious security." *30th USENIX Security Symposium (USENIX Security 21)*. 2021 (link)

Multiparty Computation

- Divide computation among multiple non-colluding parties
- Usually *information-theoretic* security
- Additive sharing of secret x:
  - `[x]`$_1$ `+ [x]`$_2$ `+ [x]`$_3$ `+ [x]`$_4$ `:= x`
- Replicated sharing: parties hold multiple shares
  - `P1:` `[x]`$_2$ `[x]`$_3$ `[x]`$_4$
  - `P2:` `[x]`$_1$ `[x]`$_3$ `[x]`$_4$
  - `P3:` `[x]`$_1$ `[x]`$_2$ `[x]`$_4$
  - `P4:` `[x]`$_1$ `[x]`$_2$ `[x]`$_3$
- Prior work (HKO+18) verified another MPC protocol in EasyCrypt

Haagh, H., Karbyshev, A., Oechsner, S., Spitters, B., & Strub, P.-Y. (n.d.). *Computer-aided proofs for multiparty computation with active security*. (link)

Representing Replicated Shares

	Sh0	Sh1	Sh2	Sh3
P0	.	x_1	x_2	x_3
P1	x_0	.	x_2	x_3
P2	x_0	x_1	.	x_3
P3	x_0	x_1	x_2	.

← each view looks random

Notational weirdness:
Pi holds all shares *except* *i*-th

In practice: *<u>Dealer</u> generates* (x_0, x_1, x_2) *randomly;*
set x_3 *such that* $\sum x_i = x$

Fantastic Four Protocol

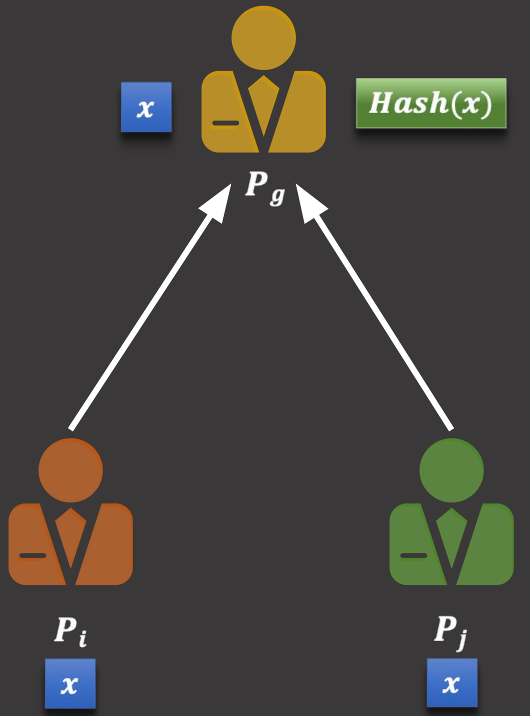
- JMP: Joint Message Passing
- INP: Shared Input
- MULT: Shared Multiplication

JMP (Joint Message Passing) Protocol

- Input: x known to P_i and P_j
- Output: P_g learns x
- $\mathtt{jmp(x, P_i, P_j, P_g)}$: P_i & P_j send x to P_g

	Sh0	Sh1	Sh2	Sh3
P0	.	0	0	x
P1	0	.	0	x
P2	0	0	.	?
P3	0	0	0	.

`jmp(x, P0, P1, P2)`

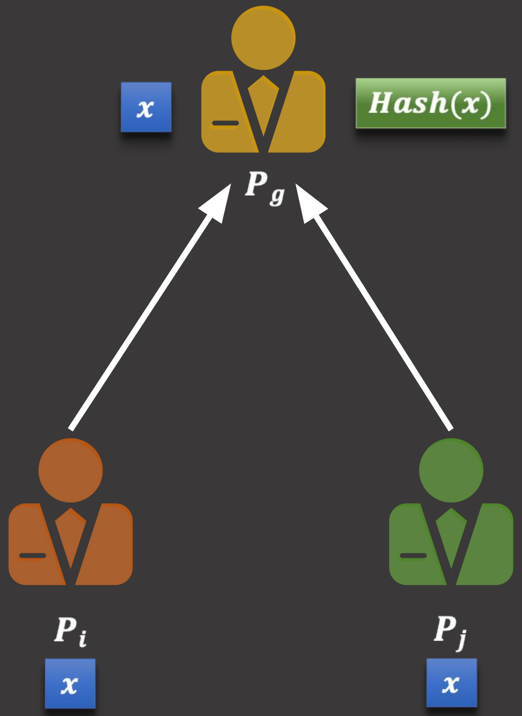


JMP (Joint Message Passing) Protocol

- Input: x known to P_i and P_j
- Output: P_g learns x
- $jmp(x, P_i, P_j, P_g)$: P_i & P_j send x to P_g

	Sh0	Sh1	Sh2	Sh3
P0	.	0	0	x
P1	0	.	0	x
P2	0	0	.	x
P3	0	0	0	.

jmp(x, P0, P1, P2)



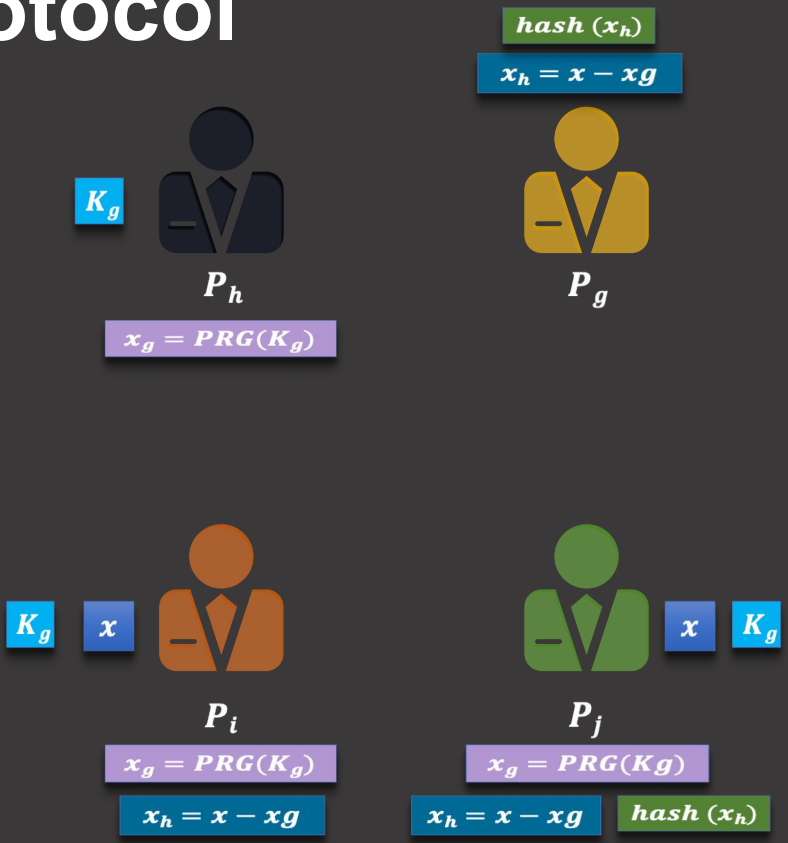
INP (Shared Input) Protocol

- P_i, P_j, P_h know a pre-shared key K_g
- Input : P_i and P_j both know a value x.
- Output : $[x]$, a secret-share matrix representing x
- Security : P_g and P_h views are uniform.

	i	j	g	h
i	.	0	0	x
j	0	.	0	x
g	0	0	.	?
h	0	0	0	.

	i	j	g	h
i	.	0	x_g	$x - x_g$
j	0	.	x_g	$x - x_g$
g	0	0	.	$x - x_g$
h	0	0	x_g	.

inp(x, P_i, P_j, P_g, P_h)



Multiplication Protocol

- Input : $[x]$ and $[y]$.
- Output : $[x.y]$.
- Example: For one round of INP - parties P_0 and P_1 both know x_2, x_3, y_2 and y_3 and run the protocol $[x_2y_3+x_3y_2] = INP(x_2y_3+x_3y_2, P_0, P_1)$.

	sh0	sh1	sh2	sh3
P0	.	x_1	x_2	x_3
P1	x_0	.	x_2	x_3
P2	x_0	x_1	.	x_3
P3	x_0	x_1	x_2	.

	sh0	sh1	sh2	sh3
P0	.	y_1	y_2	y_3
P1	y_0	.	y_2	y_3
P2	y_0	y_1	.	y_3
P3	y_0	y_1	y_2	.

$\xrightarrow{INP}$

	sh0	sh1	sh2	sh3
P0	.	0	r_{23}	$x_2y_3 + x_3y_2 - r_{23}$
P1	0	.	r_{23}	$x_2y_3 + x_3y_2 - r_{23}$
P2	0	0	.	$x_2y_3 + x_3y_2 - r_{23}$
P3	0	0	r_{23}	.

Multiplication Protocol

- Extrapolating to other parties, we can say that, Parties P_i and P_j both know x_g, x_h, y_g and y_h and run the protocol $[x_g y_h + x_h y_g] = INP(x_g y_h + x_h y_g, P_i, P_j)$.
- For every $g \in \{0,1,2,3\}$ parties call the non-interactive method and calculate their respective $[x_g y_g]$

	i	j	g	h
i	.	0	r_{gh}	$x_g y_h + x_g y_h - r_{gh}$
j	0	.	r_{gh}	$x_g y_h + x_g y_h - r_{gh}$
g	0	0	.	$x_g y_h + x_g y_h - r_{gh}$
h	0	0	r_{gh}	.

INP

	i	j	g	h
i	.	$x_1 y_1$	$x_2 y_2$	$x_3 y_3$
j	$x_0 y_0$	.	$x_2 y_2$	$x_3 y_3$
g	$x_0 y_0$	$x_1 y_1$	.	$x_3 y_3$
h	$x_0 y_0$	$x_1 y_1$	$x_2 y_2$	.

$[x_g y_g]$

Multiplication Protocol

- The parties locally add the shares
  $[x \cdot y] = \Sigma_{i \neq j}[x_i y_j + x_j y_i] + \Sigma[x_i y_i]$

- Expansion of the product:

x1*y1	+	x1*y2	+	x1*y3	+	x1*y4	+
x2*y1	+	x2*y2	+	x2*y3	+	x2*y4	+
x3*y1	+	x3*y2	+	x3*y3	+	x3*y4	+
x4*y1	+	x4*y2	+	x4*y3	+	x4*y4	

Computed *via* `INPLocal` ( ) and `INP` ( )

What we are proving

1. Correctness
2. Validity: *matrices are valid secret sharings*
    - diagonals are zero
    - off-diagonal, all columns equal (consistency)
3. Security: *adversary cannot distinguish between its view of the real protocol & a simulated execution*
    - ~ for any party *p*, row *p* of each matrix is the same

Correctness

```
op open(m : matrix) = x0 + x1 + x2 + x3.
```

Sum of the unique shares of all the parties is equal to the secret.

```
op valid(m : matrix) =
     diagonal entries are 0 /\
     for each share
          all parties' copy of share are equal
```

	i	j	g	h
i	.	x_1	x_2	x_3
j	x_0	.	x_2	x_3
g	x_0	x_1	.	x_3
h	x_0	x_1	x_2	.

Security

```
(* get party p's view *)
op view(m : matrix, p : party) = pth row of matrix.
```

Party **p** can view only its own share in the matrix represented by the **pth** row of the matrix.

```
equiv [F4.<proc> ~ Sim.<proc>:
<precondition> ===> view real_output p = view simu_output p]
```

The view of every party **p** will be the indistinguishable for the real and the simulated result.

share: protocol

```
proc F4.share(x : elem) : matrix = {
    var s0, s1, s2: elem;
    var shares : elem list;

    (* generate random *)
    s0, s1, s2 <$ randelem;

    shares <- [s0; s1; s2; x - (s0 + s1 + s2)];

    return matrix of shares;
}
```

share: simulator

```
proc Sim.share(x : elem) : matrix = {
    var s0, s1, s2, s3: elem;
    var shares : elem list;

    (* generate random *)
    s0, s1, s2, s3 <$ randelem;

    shares <- [s0; s1; s2; s3];

    return matrix of shares;
}
```

share: proofs

```
lemma share_correct(x_ : elem) :
    hoare[F4.share:
     x = x_
     ==> valid res /\ open res = x_].
```

```
lemma share_secure (p : party):
    equiv[F4.share ~ Sim.share :
     ={x} ==>
     ...
     view res{1} p = view res{2} p].
```

row p *of each matrix look the same*

jmp: protocol

```
(* si and sj share x with g. h learns nothing *)
proc jmp(x : elem, si sj g h : party) : matrix = {
  var mjmp : matrix;

  (* zero matrix, except share h, which is x *)
  mjmp <- offunm ((fun p s =>
      if s = p then zero else
      if s = h then x else zero), N, N);
  return mjmp;
}
```

jmp

- We do not simulate jmp: it is not a secure functionality in the semi-honest environment.
- Full protocol uses jmp to enforce cheating detection: Pi *and* Pj send; receiver Pg compares.
    - **Claim:** any malicious behavior can be caught via this detection
    - We leave implementing Fantastic Four in the malicious environment to future work.

inp: protocol

```
(* securely share x known by i & j to g & h *)
proc inp(x : elem, i j g h : party) : matrix = {
  var r, xh : elem;
  var pgm, minp : matrix;

  (* only known by Pi, Pj, Ph *)
  r <$ randelem;

  (* only know by Pi and Pj *)
  xh <- x - r;

  (* send xh from Pi, Pj to Pg *)
  pgm <@ jmp(xh, i, j, g, h);

  (* xi = xj = 0, xg = r, xh = x - r (within pgm) *)
  return matrix of shares [0; 0; r; xh];
}
```

Inp: protocol output matrix

Assume P_0 & P_1 know x and want to securely share it with P_2 & P_3

Views of P_2, P_3 look random

	Sh0	Sh1	Sh2	Sh3
P0	.	0	r	x - r
P1	0	.	r	x - r
P2	0	0	.	x - r
P3	0	0	r	.

inp: simulator

```
proc inp(x : elem, i j g h : party) : matrix = {
  var r, xh, xfake: elem;
  var pgm, minp : matrix;

  (* can’t fool Pi, Pj; behave*)
  r <$ randelem;
  xh <- x - r;
  (* but can fool Pg *)
  xfake <$ randelem;

  minp <- offunm ((fun p s =>
    if s = p then zero else (* diag *)
    if s = g then r else     (* share g is always r *)
    if s = h then (if p = g then xfake else xh) (* lie to Pg *)
    else zero (* all other shares 0 *)
    ), N, N);
  return minp;
}
```

Inp: simulator output matrix

Assume P_0 & P_1 know x and want to securely share it with P_2 & P_3

	Sh0	Sh1	Sh2	Sh3	
P0	.	0	r	x - r	← <u>don't</u> lie to P_0 and P_1 they know what to expect!
P1	0	.	r	x - r	
P2	0	0	.	x_{fake}	← <u>lie to party 2</u>
P3	0	0	r	.	

Inp : Proofs

```
lemma inp_correct(x_ : elem, i_ j_ g_ h_: party) :
    hoare[F4.inp : x=x_ ==> valid res /\ open res = x_].

lemma inp_secure(i_ j_ g_ h_ : party, p : party) :
    equiv[F4.inp ~ Sim.inp :
      ={x,i,j,g,h} /\ unique [i_; j_; g_; h_] ==>
      view real_output p = view simu_output p].
```

mult

simulated
or real

```
proc mult(mx, my : matrix) : matrix = {
  ...

  m23 <@ inp(mx.[0,1] * my.[1,0] + mx.[1,0] * my.[0,1], 2, 3, 0, 1);
  m13 <@ inp(mx.[0,2] * my.[1,0] + mx.[1,0] * my.[0,2], 1, 3, 0, 2);
  m12 <@ inp(mx.[0,3] * my.[1,0] + mx.[1,0] * my.[0,3], 1, 2, 0, 3);
  m03 <@ inp(mx.[0,2] * my.[0,1] + mx.[0,1] * my.[0,2], 0, 3, 1, 2);
  m02 <@ inp(mx.[0,3] * my.[0,1] + mx.[0,1] * my.[0,3], 0, 2, 1, 3);
  m01 <@ inp(mx.[0,3] * my.[0,2] + mx.[0,2] * my.[0,3], 0, 1, 2, 3);

  (* INP local: elementwise (hadamard) product *)
  mlocal <- mx ⊙ my;

  return m01 + m02 + m03 + m12 + m13 + m23 + mlocal;
}
```

Mult : Proofs

```
lemma mult_correct(x_ y_ : elem) :
    hoare[F4.mult_main : x = x_ /\ y = y_ ==>
        open res = x_ * y_ /\ valid res].

lemma mult_secure(p: party) :
    equiv[F4.mult ~ Sim.mult :
      ={x, y} ==> view real_output p = view simu_output p].
```

Proof Progress

Proved correctness & security for

- share
- jmp (correctness only; not a secure functionality)
- inp
- addition
- multiplication

…in the semi-honest environment, using both pHL and game models.

Statistics:

- ~1200 lines
- 100 sec to execute
- 107 calls to smt()
- 28 lemmas
- 0 admits

Easycrypt Games

- Adv
- GReal
    - Mimics the real world game where the true implementation of **`mult`** `and` **`add`** in the **`Fantastic4`** protocol is used.
- GIdeal
    - Mimics the ideal world game where the **`mult`** and **`add`** of the simulator is used.

Module type Adversary

```
module type ADV = {
  (* Ask adversary for element x: used in add_main *)
  proc getx(): elem

  (* Ask adversary for element y: used in add_main *)
  proc gety(): elem

  (* Ask adversary for matrix mx: used in mult_main *)
  proc getmx(): matrix

  (* Ask adversary for matrix mx: used in mult_main *)
  proc getmy(): matrix

  (* Adversary gets a view for a party p of the matrix and
  asked to differentiate between the real and the ideal game. *)
  proc put(view_mz : vector) : bool
}.
```

Real and Ideal Games with Call to Mult

```
module GReal (Adv : ADV) = {

  proc mult_main(): bool = {
    var mx, my, mz : matrix;
    var b : bool;

    mx <@ Adv.getmx();
    my <@ Adv.getmy();
    mz <@ F4.mult(mx, my);

    b <@ Adv.put(view mz p);

    return b;
  }
}.
```

```
module GIdeal (Adv : ADV) = {

  proc mult_main(): bool = {
    var mx, my, mz : matrix;
    var b : bool;

    mx <@ Adv.getmx();
    my <@ Adv.getmy();
    mz <@ Sim.mult(mx, my);

    b <@ Adv.put(view mz p);

    return b;
  }
}.
```

Security Proof of mult_main

```
local lemma GReal_GIdeal :
    equiv[GReal(Adv).mult_main ~ GIdeal(Adv).mult_main:
          ={glob Adv} /\ 0 <= p < N ==> ={res}].

lemma Sec_Mult_Main &m :
    0 <= p < N =>
    Pr[GReal(Adv).mult_main() @ &m : res] =
    Pr[GIdeal(Adv).mult_main() @ &m: res].

lemma Security_Mult_Main (Adv <: ADV{}) &m :
    0 <= p < N =>
    Pr[GReal(Adv).mult_main() @ &m : res] =
    Pr[GIdeal(Adv).mult_main() @ &m: res].
```

More powerful adversary: views full transcript

```
module GReal (Adv : ADV) = {
  proc mult_inp(): bool = {
    mx <@ Adv.getmx();
    my <@ Adv.getmy();
    m23 <@ F4.inp(...);
    b1 <@ Adv.put(view m23 p);
    m13 <@ F4.inp(...);
    b2 <@ Adv.put(view m13 p);
    m12 <@ F4.inp(...);
    b3 <@ Adv.put(view m12 p);
    m03 <@ F4.inp(...);
    b4 <@ Adv.put(view m03 p);
    m02 <@ F4.inp(...);
    b5 <@ Adv.put(view m02 p);
    m01 <@ F4.inp(...);
    b6 <@ Adv.put(view m01 p);
    mlocal <- create mlocal matrix;
    b7 <@ Adv.put(view mlocal p);
    mresult <- m01 + m02 + m03 + m12 + m13 + m23 + mlocal;
    b8 <@ Adv.put(view mresult p);
    b9 <- b1 /\ b2 /\ b3 /\ b4 /\ b5 /\ b6 /\ b7 /\ b8;
    return b9;
  }
}.
```

```
module GIdeal (Adv : ADV) = {
  proc mult_inp(): bool = {
    mx <@ Adv.getmx();
    my <@ Adv.getmy();
    m23 <@ Sim.inp(...);
    b1 <@ Adv.put(view m23 p);
    m13 <@ Sim.inp(...);
    b2 <@ Adv.put(view m13 p);
    m12 <@ Sim.inp(...);
    b3 <@ Adv.put(view m12 p);
    m03 <@ Sim.inp(...);
    b4 <@ Adv.put(view m03 p);
    m02 <@ Sim.inp(...);
    b5 <@ Adv.put(view m02 p);
    m01 <@ Sim.inp(...);
    b6 <@ Adv.put(view m01 p);
    mlocal <- create mlocal matrix;
    b7 <@ Adv.put(view mlocal p);
    mresult <- m01 + m02 + m03 + m12 + m13 + m23 + mlocal;
    b8 <@ Adv.put(view mresult p);
    b9 <- b1 /\ b2 /\ b3 /\ b4 /\ b5 /\ b6 /\ b7 /\ b8;
    return b9;
  }
}.
```

Security Proof of mult_inp

```
local lemma GReal_GIdeal :
    equiv[GReal(Adv).mult_inp ~ GIdeal(Adv).mult_inp:
          ={glob Adv} /\ 0 <= p < N ==> ={res}].

lemma Sec_mult_inp &m :
    0 <= p < N =>
    Pr[GReal(Adv).mult_inp() @ &m : res] =
    Pr[GIdeal(Adv).mult_inp() @ &m: res].

lemma Security_mult_inp (Adv <: ADV{}) &m :
    0 <= p < N =>
    Pr[GReal(Adv).mult_inp() @ &m : res] =
    Pr[GIdeal(Adv).mult_inp() @ &m: res].
```

Real and Ideal Games for Add

```
module GReal (Adv : ADV) = {

  proc add_main(): bool = {
    ...
    x <@ Adv.getx();
    y <@ Adv.gety();
    mx <@ F4.share(x);
    b1 <@ Adv.put(view mx p);

    my <@ F4.share(y);
    b2 <@ Adv.put(view my p);

    mz <- mx + my;
    b3 <@ Adv.put(view mz p);

    b <- b1 /\ b2 /\ b3;
    return b;
  }
}.
```

```
module GIdeal (Adv : ADV) = {

  proc add_main(): bool = {
    ...
    x <@ Adv.getx();
    y <@ Adv.gety();
    mx <@ Sim.share(x);
    b1 <@ Adv.put(view mx p);

    my <@ Sim.share(y);
    b2 <@ Adv.put(view my p);

    mz <- mx + my;
    b3 <@ Adv.put(view mz p);

    b <- b1 /\ b2 /\ b3;
    return b;
  }
}.
```

Security Proof of add_main

```
local lemma GReal_GIdeal :
    equiv[GReal(Adv).add_main ~ GIdeal(Adv).add_main:
          ={glob Adv} /\ 0 <= p < N ==> ={res}].

lemma Sec_Mult_Main &m :
    0 <= p < N =>
    Pr[GReal(Adv).add_main() @ &m : res] =
    Pr[GIdeal(Adv).add_main() @ &m: res].

lemma Security_Mult_Main (Adv <: ADV{}) &m :
    0 <= p < N =>
    Pr[GReal(Adv).add_main() @ &m : res] =
    Pr[GIdeal(Adv).add_main() @ &m: res].
```

Lessons Learned & Future Work

- Needed finite (Zmodp) types instead of infinite (int) types
  - Mysterious `algebra` tactic helped!
- Think carefully about statements to be proven
  - A single admit can get you anywhere
- Future work:
  - Malicious adversaries (this proof: semi-honest)
  - Experiment with probabilistic game proof for composable share-and-multiply protocol
  - Hoare equivalence for our current (buggy) implementation
  - Eventual goal: verify entire 4PC protocol as implemented