A Verified DSL for MPC

in

Aseem Rastogi

Microsoft Research India

(Computer Aided Security Proofs, Aarhus, Denmark)

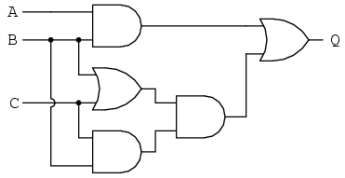
Secure Multi-party Computation (MPC)

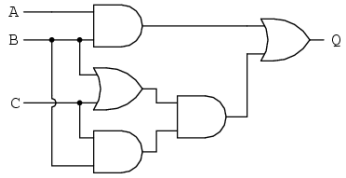
General n-party cryptographic protocols

Compute $f(x_1, x_2, x_3)$

Without revealing inputs

Input: x_2

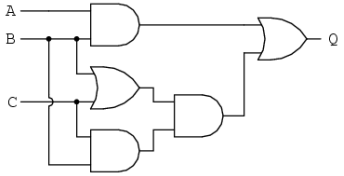




Circuit representation of f

Input: x_1

Input: x_3



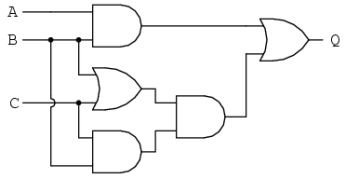
Secure Multi-party Computation (MPC)

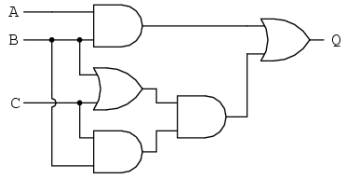
General n-party cryptographic protocols

Compute $f(x_1, x_2, x_3)$

Without revealing inputs

Input: x_2



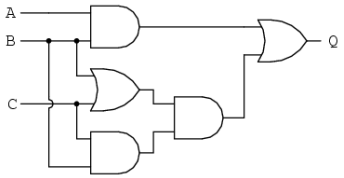


Circuit representation of f

Input: x_1

$f(x_1, x_2, x_3)$

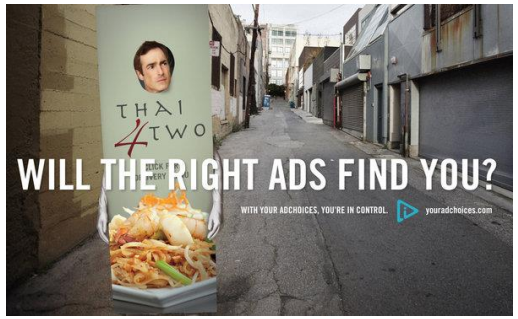
Input: x_3



Wide ranging applications of MPC

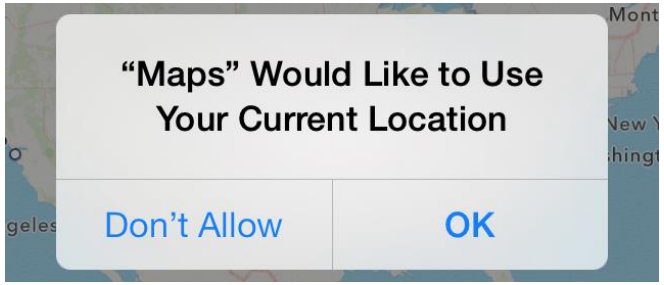
- Auctions

- Online ads



- Statistical computations over joint data

- Location privacy, and many more



How should we program MPC

- What programming interface (API) to use

- Requirements:
  - Should be high-level
  - Should simplify reasoning about the programs

*We consider only honest-but-curious threat model

MPC frameworks

- Circuit libraries

```
let g1 = Gate (AND, w1, w2) in
let g2 = Gate (XOR, w2, w3) in
…
```

Too low-level

- Circuit compilers

```
let r = x + y + z in
…
```

Only monolithic circuit code

What about *normal* computations around it ?

Unoptimized vs optimized median

Joint median computation

`(x1, x2) (y1, y2)  Assume: x1 < x2  y1 < y2  Distinct (x1, x2, y1, y2)`

```
let b  = compare x1 y1 in
let x3 = b ? x2 : x1   in
let y3 = b ? y1 : y2   in
smaller x3 y3
```

```
let b  = do_sec (compare, x1) in
let x3 = b ? x2 : x1          in
do_sec (smaller, x3)
```

```
let b  = do_sec (compare, y1) in
let y3 = b ? y1 : y2          in
do_sec (smaller, y3)
```

Writing one program for each party

Joint median computation

```
(x1, x2) (y1, y2)  Assume: x1 < x2  y1 < y2   Distinct (x1, x2, y1, y2)
```

```
let b  = do_sec (compare, x1) in
let x3 = b ? x2 : x1          in
do_sec (smaller, x3)
```

```
let b  = do_sec (compare, y1) in
let y3 = b ? y1 : y2          in
do_sec (smaller, y3)
```

- Not scalable
- Chances of errors, no tool support to catch them
- Reasoning about some program property as a whole is hard

Key Idea

- MPC Specific computation pattern
  - Parties compute their programs, and
  - Synchronize at secure computations

We use this insight to design a better MPC language

Wysteria: DSL for Programming MPC

- A new high-level MPC language
  - Supports n-party generic applications

- Wysteria programs are:
  - written like regular single-threaded programs, and
  - executed in a distributed, multi-party setting

Key Idea of Wysteria

We could write it in a single program

```
let b  = do_sec (compare, x1)
let x3 = a ? x2 : x1
do_sec (smaller, x3)
```

```
let (compare, x1) = as_sec {A, B}
let x2 = as_par {A}
let x3 = as_par {B}
as_sec {A, B} (smaller x3 y3)
```

```
let (compare, x1) = do_sec (compare, y1)
let y2 = a ? y1 : y2
do_sec (smaller, y3)
```

```
let (compare, y1) in
let y2 = a ? y1 : y2 in
do_sec (smaller, y3) in
```

- `as_par` and `as_sec` are two of the API functions exported by Wysteria
- `{A}, {B}, {A, B}` are party sets

Wysteria computational model

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

- Every party runs the same program
- `as_par ps e` – parties in `ps` perform `e` “locally, in-parallel”
- `as_sec ps e` – parties in `ps` perform `e` “jointly, using MPC”

Wysteria computational model

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

■

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

- Every party runs the same program
- `as_par ps e` – parties in `ps` perform `e` "locally, in-parallel"
- `as_sec ps e` – parties in `ps` perform `e` "jointly, using MPC"

Wysteria computational model

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

■

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

- Every party runs the same program
- `as_par ps e` – parties in `ps` perform `e` "locally, in-parallel"
- `as_sec ps e` – parties in `ps` perform `e` "jointly, using MPC"

Wysteria computational model

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

■

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

- Every party runs the same program
- `as_par ps e` – parties in `ps` perform `e` "locally, in-parallel"
- `as_sec ps e` – parties in `ps` perform `e` "jointly, using MPC"

Wysteria computational model

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

■

```
let b  = as_sec {A, B} (compare x1 y1) in
let x3 = as_par {A}    (b ? x2 : x1)   in
let y3 = as_par {B}    (b ? y1 : y2)   in
as_sec {A, B} (smaller x3 y3)
```

- Every party runs the same program
- `as_par ps e` – parties in `ps` perform `e` "locally, in-parallel"
- `as_sec ps e` – parties in `ps` perform `e` "jointly, using MPC"

Wysteria advantages over previous work

```
let b  = as_sec {A, B} (compare x1 y1)   in
let x3 = as_par {A}    (b ? x2 : x1)     in
let y3 = as_par {B}    (b ? y1 : y2)     in
as_sec {A, B} (smaller x3 y3)
```

- One, single-threaded program, scalable
- Design precludes a whole class of errors

Higher assurance (Formal verification?)

Unoptimized vs optimized median

```
as_sec {A, B}
  let b  = compare x1 y1 in
  let x3 = b ? x2 : x1   in
  let y3 = b ? y1 : y2   in
  smaller x3 y3
```

Unoptimized median

```
let b  = as_sec {A, B} (compare x1 y1)   in
let x3 = as_par {A}    (b ? x2 : x1)     in
let y3 = as_par {B}    (b ? y1 : y2)     in
as_sec {A, B} (smaller x3 y3)
```

Optimized median

(runs faster)

- Optimized version reveals more, by design
- Is it secure ? Does it leak more than the unoptimized one ?

Formal verification of Wysteria programs

- Wysteria is implemented as a DSL in F*

- Wysteria programs are *simply* F* programs
  - Programmers can use F* facilities to verify MPC programs

Sample Code for Median

(In emacs)

Formally verifying the median properties

```
as_sec {A, B}
  let b  = compare x1 y1 in
  let x3 = b ? x2 : x1   in
  let y3 = b ? y1 : y2   in
  smaller x3 y3
```

Unoptimized median

```
let b  = as_sec {A, B} (compare x1 y1)   in
let x3 = as_par {A}    (b ? x2 : x1)     in
let y3 = as_par {B}    (b ? y1 : y2)     in
as_sec {A, B} (smaller x3 y3)
```

Optimized median

(runs faster)

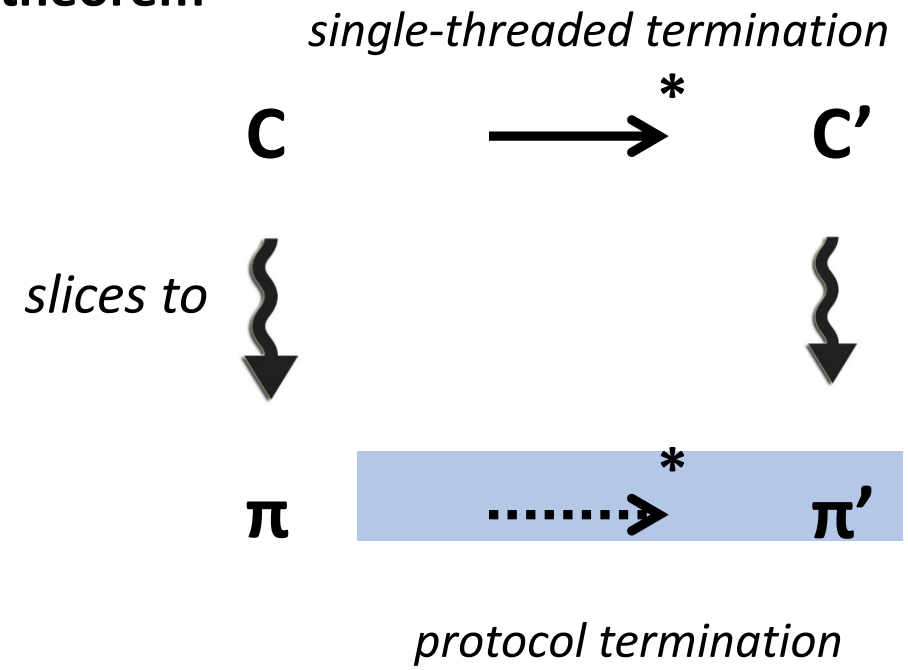
- Both versions compute the correct median ✓
- Optimized one does not leak more than the unoptimized one ✓

(Security verified in the idealized crypto model)

Wysteria metatheory

- Formalize two semantics
  - Single-threaded semantics (Wysteria semantics in F*)
  - Distributed semantics (for running the programs)

Soundness theorem



Define a slice relation that relates

a single-threaded configuration
to
its multi-party protocol

Wysteria metatheory

- Formalize two semantics
  - Single-threaded semantics (Wysteria semantics in F*)
  - Distributed semantics (for running the programs)

Soundness theorem

Define a slice relation that relates

a single-threaded configuration
to
its multi-party protocol

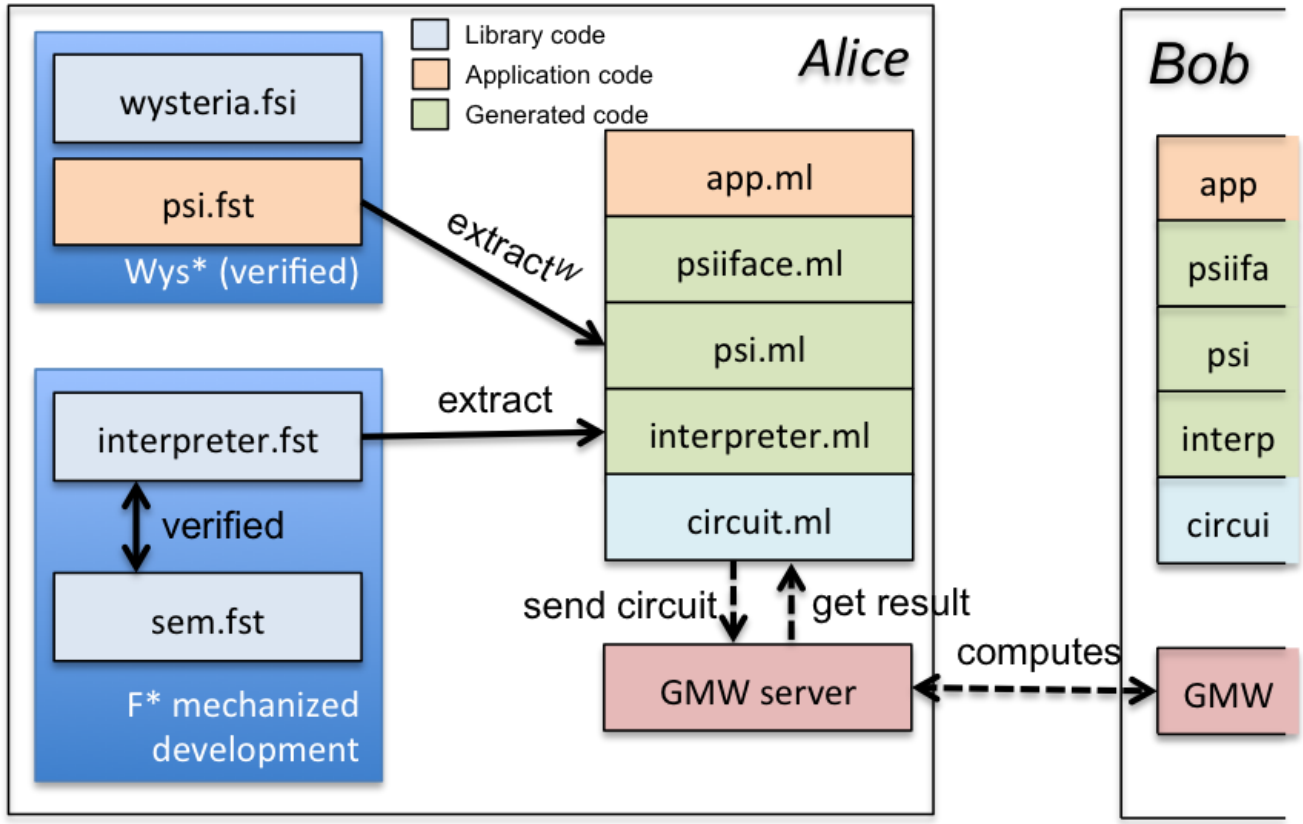
Properties verified in the source carry over when the programs are run in the distributed semantics

Theorem mechanically verified in F*

Sample Code for Metatheory

(In emacs)

Wysteria toolchain



GMW is an MPC crypto protocol

We use GMW implementation from Choi et. al.

Interpreter converts `as_sec` code to boolean circuits dynamically

FORK ME ON GITHUB!