Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations

To Appear in IEEE S&P 2014 (Oakland)

Aseem Rastogi

Matthew Hammer, Michael Hicks

PLUM

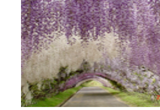
PROGRAMMING LANGUAGES UNIVERSITY OF MARYLAND

What is Secure Computation

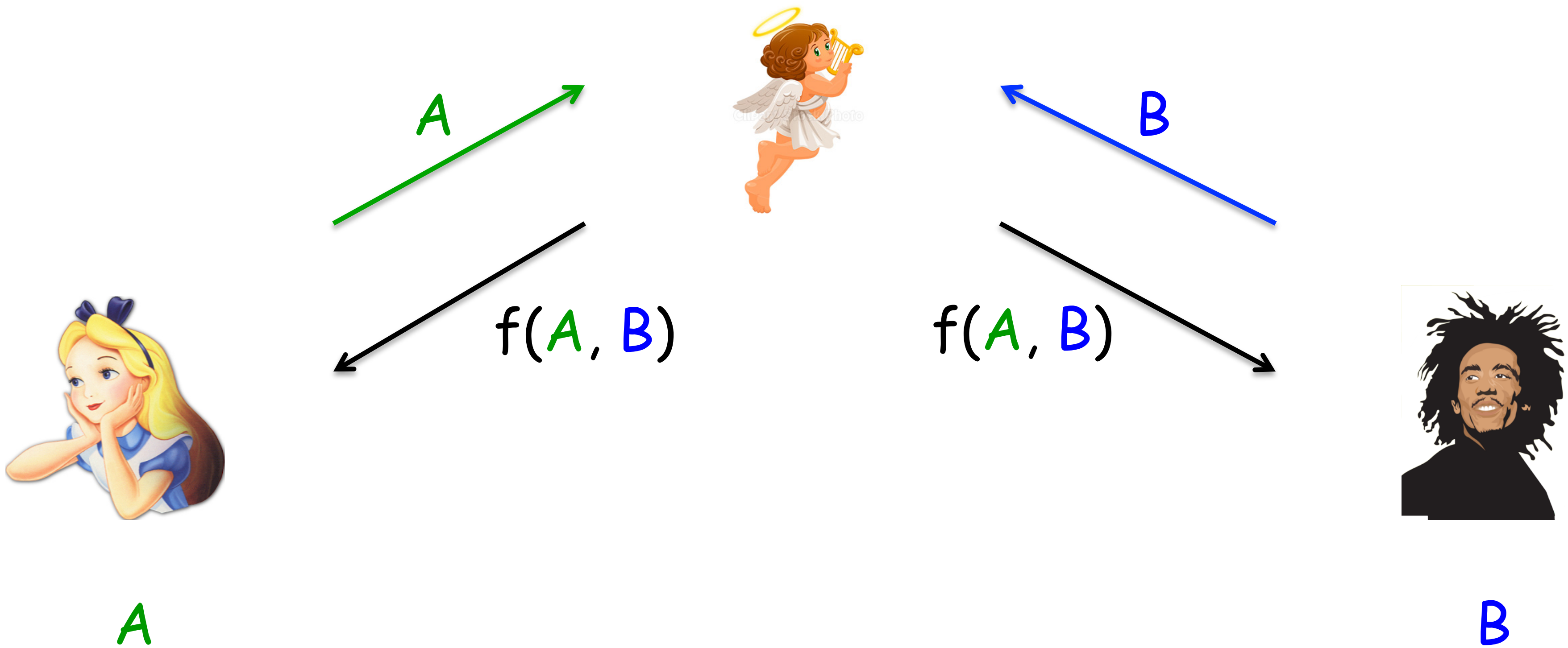
A B

Compute $f(A, B)$

Without revealing A to Bob and B to Alice

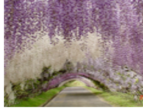


Using a Trusted Third Party

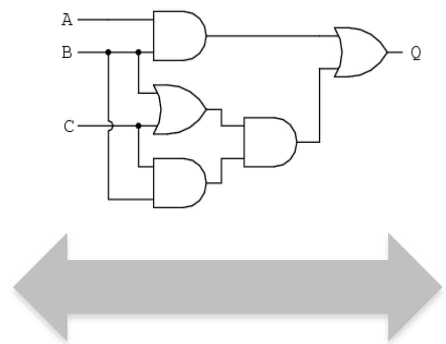


Compute f(A, B)

Without revealing A to Bob and B to Alice



Secure Computation Eliminates Trusted Third Party



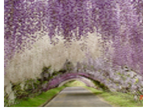
Cryptographic Protocol

A

B

Compute f(*A*, *B*)

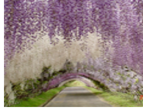
Without revealing *A* to Bob and *B* to Alice



Secure Computation Examples

- Richest Millionaire
  - Without revealing salaries
- Nearest Neighbor
  - Without revealing locations
- Auction
  - Without revealing bids
- Private Set Intersection
  - Without revealing sets

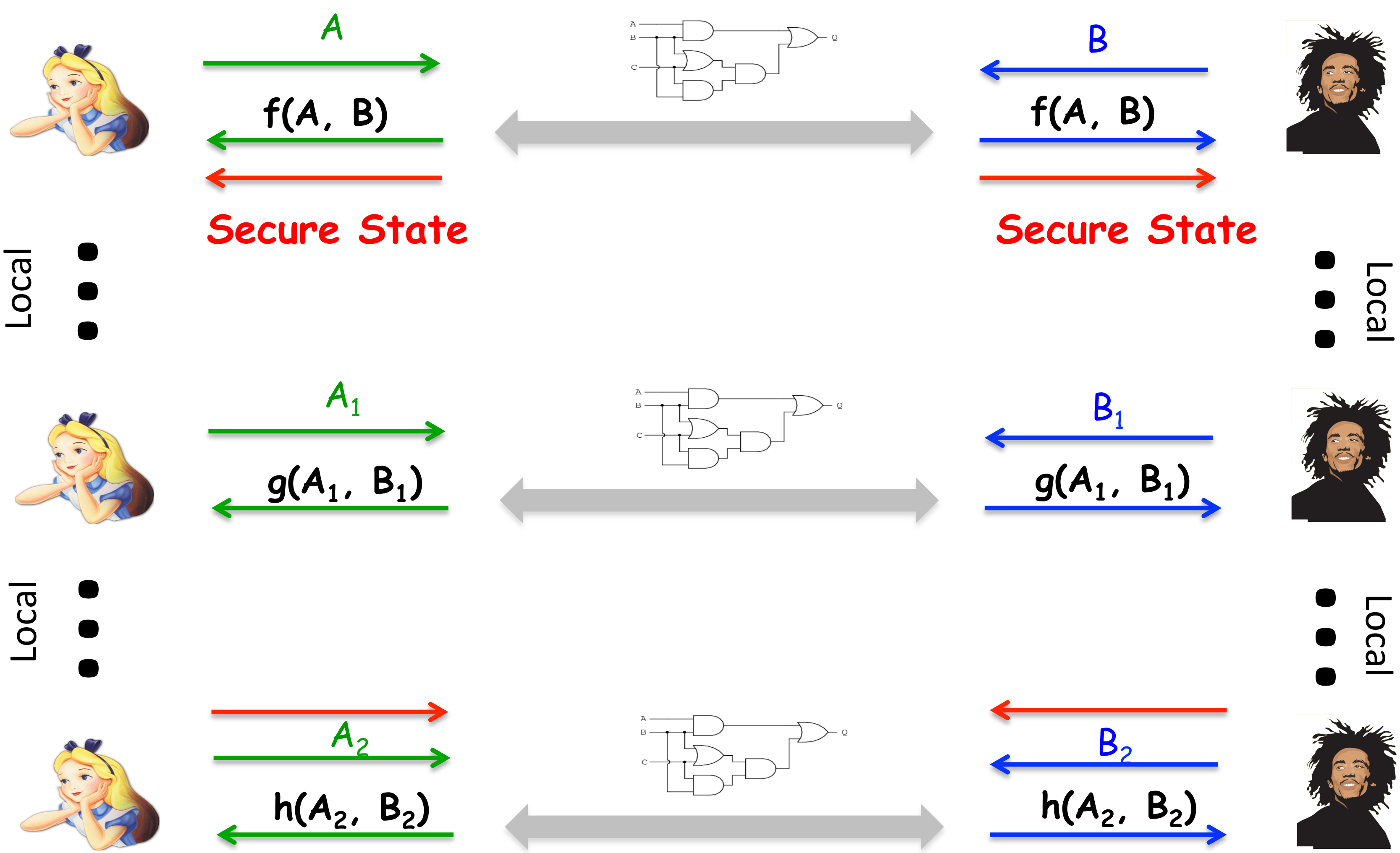
Real World Example

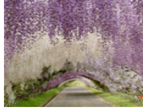


Let’s Go Beyond Toy Examples

- Card Games
  - E.g. Online Poker
  - Players trust (potentially malicious) house
  - Use Secure Computation to deal cards !
- Strategy Games
  - E.g. Dice Games
  - Use Secure Computation to roll dice !

Reactive Secure Computation



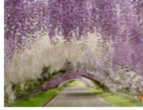


Computation Patterns for n-Party Case

- Parties could play asymmetric roles
  - Participate in some computations not others
- Asymmetric outputs
  - Only some parties get to know the output

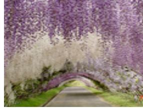
Wysteria Design Goals

- High-level language to write n-Party SMC
  - Single specification
  - Runtime compilation to circuits
- Support reactive computation patterns
  - Mixed-mode
  - Parties decide at runtime whether to participate
- Support generic code for n-parties
- High-level support for secure state
- Compositionality, statically typed, sound, …



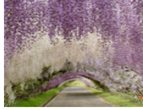
Needless to say, Wysteria has it all !

https://bitbucket.org/aseemr/wysteria/wiki/Home



Two-party Millionaire’s Example

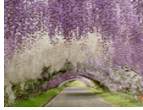
```
        par(A)
let a = read() in

        par(B)
let b = read() in

       sec(A,B)
let o = a > b in

o
```



Two-party Millionaire's Example

A's Local Computation

```
      par(A)
let a = read() in

      par(B)
let b = read() in

     sec(A,B)
let o = a > b in

o
```



Two-party Millionaire's Example

A's Local Computation

```
        par(A)
let a = read() in
```

B's Local Computation

```
        par(B)
let b = read() in

       sec(A,B)
let o = a > b in

o
```

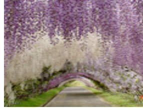
Two-party Millionaire's Example

```
        par(A)
let a = read() in

        par(B)
let b = read() in

        sec(A,B)
let o = a > b in

o
```

A's Local Computation

B's Local Computation

Secure Computation by (A,B)



Two-party Millionaire's Example

```
        par(A)
let a = read() in

        par(B)
let b = read() in

       sec(A,B)
let o = a > b in

o
```

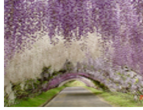
A's Local Computation

B's Local Computation

Secure Computation by (A,B)

Interpreter compiles it to boolean circuit at runtime

Both Parties Run the Same Program

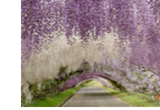


Key Concept - 1

Mixed-Mode Computations via Place Annotations

What If Only A Should Know the Output

```
     par(A)
let a = read() in

     par(B)
let b = read() in

     sec(A,B)
let o = a > b in

o
```



What If Only A Should Know the Output

```
        par(A)
let a =  read() in

        par(B)
let b =  read() in

        sec(A,B)
let o =
  let g = a > b in
  wire A:g
in
o
```

Wire Bundle
≈ Map from Parties to Values

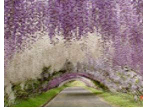
Passing Input via Wire Bundle

```
         par(A)
let a = read() in
         par(B)
let b = read() in

let w1 = wire A:a in
let w2 = wire B:b in
let w3 = w1 ++ w2 in

         sec(A,B)
let o =
   let g = w3[A] > w3[B] in
    wire A:g
in
o
```

	A's View	B's View	sec(A,B)'s View
w1	{ A : a }	{ }	{ A : a }
w2	{ }	{ B : b }	{ B : b }
w3	{ A : a }	{ B : b }	{ A : a, B : b }

Wire Concatenation

Wire Projection

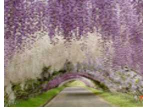


Writing Richer as a Function

```
let richer = λx:W {A,B} nat .
                     sec(A,B)
               let o = x[A] > x[B] in
               o
in
let a = read () in
let b = read () in
richer (wire A:a ++ wire B:b)
```

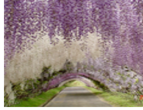
Projections are type checked

`W {A,B} nat:` Dependently typed wire bundles



Key Concept - 2

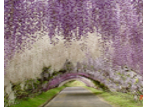
Wire Bundle Abstraction for Input Output to Secure Blocks



Revisit Writing Richer as a Function

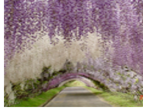
```
let richer = λx:W {A,B} nat .
                  sec(A,B)
             let o = x[A] > x[B] in
             o
in
```

- Applies only to A, B
- Not generic, not reusable for different parties



Wire Bundle Folding

- List fold:
  - `(‘a -> ‘b -> ‘a) -> ‘a -> ‘b list -> ‘a`
  - `fold(f,x,[2;1;3]) = f(f(f(x,2),1),3)`
  - `fold(fun x y -> if x > y then x else y, 0, [2;1;3])`



Wire Bundle Folding

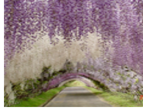
- List fold:
  - ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
  - fold(f,x,[2;1;3]) = f(f(f(x,2),1),3)
  - fold(fun x y -> if x > y then x else y, 0, [2;1;3])
- Similar concept: Wire bundle fold (**wfold**)
  - Party sets are typed as **ps**
  - **W w 'a -> 'b -> ('b -> ps -> 'a -> 'b) -> 'b**
  - Actually a bit more precise: **ps{ν ⊆ w}**
  - **waps: W w 'a -> ('a -> 'b) -> W w 'b**

Writing Richer as a *Generic* Function

```
let comb = λx:ps.λw:W x nat.
           λa:ps{ν  x} option.λp:ps{ν  x}.λn:nat
           match a with
           | None => Some(p)
           | Some(q) => if w[q] > n then a else
                          Some(p)
in
let richer = λx:ps .
               λy:W x nat .
                     sec(x)
               let o = wfold(y, None, comb x y) in
               o
in
```

Writing Richer as a *Generic* Function

```
let comb = λx:ps.λw:W x nat.
           λa:ps{ν ⊆ x} option.λp:ps{ν ⊆ x}.λn:nat
           match a with
           | None => Some(p)
           | Some(q) => if w[q] > n then a else
                          Some(p)
in
let richer = λx:ps .
               λy:W x nat .
                     sec(x)
               let o = wfold(y, None, comb x y) in
               o
in
```

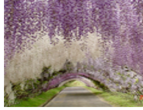


Key Concept - 3

- ***Parties are first-class values***
- ***Dependent types enable writing generic code***

Wysteria Metatheory

- Dependently typed language
  - Extensions to λ-calculus
  - Dependent types reason about SMC abstractions
- Two operational semantics
  - Single-threaded (conceptual), parties maintain synchrony
  - Multi-threaded (actual), parties execute independently, synchronizing at secure blocks



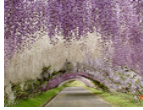
Wysteria Metatheory

- Standard progress and preservation theorems
  - “Well-typed programs don’t go wrong”
- Operational semantics correspondence

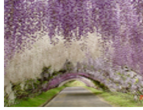
Single-threaded $C_1 \longrightarrow C2$

Multi-threaded $\pi_1 \longrightarrow \longrightarrow \longrightarrow \cdots \longrightarrow^{*} \pi_2$

slice operation



Demo !

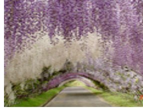


Next Steps: Write a Cool App

- E.g. write full poker game
- (We already have a card dealing prototype)
- Challenges: design and implement an FFI to interact with OCaml

Next Steps: Recursive Types

- Add recursive types, e.g. Trees, Lists
- Secure blocks invariants:
  - Always terminate
  - Each party generates same circuit *independently*
- How do we ensure these properties for recursive types ?
- Applications: binary search, list operations, etc. in secure blocks



More Details

https://bitbucket.org/aseemr/wysteria/wiki/Home