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

Aseem Rastogi
Matthew Hammer, Michael Hicks

To Appear in IEEE S&P 2014 (Oakland)
What is Secure Computation

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

A

B

CryptoGraphic Protocol

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

• 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

Secure State

Local

Secure State

Local

Local

Secure State
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
```

```
Two-party Millionaire’s Example

\[
\text{par}(A) \\
\text{let } a = \text{read}() \text{ in} \\
\text{par}(B) \\
\text{let } b = \text{read}() \text{ in} \\
\text{sec}(A,B) \\
\text{let } o = a > b \text{ in}
\]
Two-party Millionaire’s Example

\[
\begin{align*}
\text{par}(A) & \quad \text{B's Local Computation} \\
\text{let } a = \text{read()} \text{ in} & \\
\text{par}(B) & \\
\text{let } b = \text{read()} \text{ in} & \quad \text{A's Local Computation} \\
\text{sec}(A, B) & \\
\text{let } o = a > b \text{ in} & \\
\end{align*}
\]
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
```

A’s Local Computation

B’s Local Computation

Secure Computation by (A,B)
Two-party Millionaire’s Example

\[
\text{par}(A) \\
\text{let } a = \text{read()} \text{ in} \\
\text{par}(B) \\
\text{let } b = \text{read()} \text{ in} \\
\text{sec}(A, B) \\
\text{let } o = a > b \text{ in} \\
o
\]

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

```python
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

\[
\begin{align*}
&\text{par}(A) \\
&\text{let } a = \text{read()} \text{ in} \\
&\text{par}(B) \\
&\text{let } b = \text{read()} \text{ in} \\
&\text{sec}(A,B) \\
&\text{let } o = \\
&\quad \text{let } g = a > b \text{ in} \\
&\quad \text{wire } A:g \\
&\text{in} \\
&\quad o \\
\end{align*}
\]

Wire Bundle \(\approx\) Map from Parties to Values
Passing Input via Wire Bundle

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

let b = read() in
par(B)

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

let o =
  let g = w3[A] > w3[B] in
  wire A:g
in

sec(A,B)
```

<table>
<thead>
<tr>
<th></th>
<th>A's View</th>
<th>B's View</th>
<th>sec(A,B)'s View</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td>{A:a}</td>
<td>{}</td>
<td>{A:a}</td>
</tr>
<tr>
<td>w2</td>
<td>{}</td>
<td>{B:b}</td>
<td>{B:b}</td>
</tr>
<tr>
<td>w3</td>
<td>{A:a}</td>
<td>{B:b}</td>
<td>{A:a, B:b}</td>
</tr>
</tbody>
</table>

Wire Concatenation

Wire Projection
Writing Richer as a Function

\[
\text{let } \text{richer } = \lambda x: W \{A, B\} \text{ nat .}
\]

\[
\text{let } o = x[A] > x[B] \text{ in}
\]

\[
\text{in}
\]

\[
\text{let } a = \text{read} () \text{ in}
\]

\[
\text{let } b = \text{read} () \text{ in}
\]

\[
\text{richer } (\text{wire } A:a ++ \text{wire } B:b)
\]

\[
W \{A,B\} \text{ nat: Dependently typed wire bundles}
\]
Key Concept - 2

Wire Bundle Abstraction for Input Output to Secure Blocks
Revisit Writing Richer as a Function

\[
\text{let richer} = \lambda x: W \{ A, B \} \rightarrow \text{nat} . \quad \text{sec}(A, B)
\]

\[
\text{let } o = x[A] > x[B] \text{ in }
\]

\[
o
\]

\[
in
\]

• 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:
  - \((\texttt{a} \to \texttt{b} \to \texttt{a}) \to \texttt{a} \to \texttt{b} \text{ list} \to \texttt{a}\)
  - \(\text{fold}(\texttt{f}, \texttt{x}, [2;1;3]) = \texttt{f}(\texttt{f}(\texttt{f}(\texttt{x},2),1),3)\)
  - \(\text{fold}(\text{fun} \ \texttt{x} \ \texttt{y} \rightarrow \text{if} \ \texttt{x} > \texttt{y} \ \text{then} \ \texttt{x} \ \text{else} \ \texttt{y}, 0, [2;1;3])\)

- Similar concept: Wire bundle fold \((\texttt{wfold})\)
  - Party sets are typed as \(\texttt{ps}\)
  - \(\texttt{W} \ \texttt{w} \ \texttt{a} \to \texttt{b} \to (\texttt{b} \to \texttt{ps} \to \texttt{a} \to \texttt{b}) \to \texttt{b}\)
  - Actually a bit more precise: \(\texttt{ps}\{\nu \subseteq \texttt{w}\}\)
  - \(\texttt{waps}: \texttt{W} \ \texttt{w} \ \texttt{a} \to (\texttt{a} \to \texttt{b}) \to \texttt{W} \ \texttt{w} \ \texttt{b}\)
Writing Richer as a *Generic* Function

```latex
let comb = \lambda x:ps.\lambda w:W \ x \ nat.
    \lambda a:ps.\nu x \ option.\lambda p:ps.\nu x.\lambda n:nat
match a with
| None => Some(p)
| Some(q) => if w[q] > n then a else
          Some(p)
in
let richer = \lambda x:ps .
    \lambda y:W x nat .
    \sec(x)
let o = wfold(y, None, comb x y) in
  o
in
```
Writing Richer as a \textit{Generic} Function

\[
\text{let } \text{comb} = \lambda x:ps . \lambda w:W x \text{ nat} .
\]
\[
\lambda a:ps \{v \subseteq x\} . \text{option} . \lambda p:ps \{v \subseteq x\} . \lambda n: \text{nat}
\]
\[
\text{match } a \text{ with}
\]
\[
| \text{None } => \text{Some}(p)
\]
\[
| \text{Some}(q) => \text{if } w[q] > n \text{ then } a \text{ else } \text{Some}(p)
\]
\[
\text{in}
\]
\[
\text{let } \text{richer} = \lambda x:ps .
\]
\[
\lambda y:W x \text{ nat} .
\]
\[
\text{sec}(x)
\]
\[
\text{let } o = \text{wfold}(y, \text{None}, \text{comb } x \ y) \text{ in}
\]
\[
o
\]
\[
\text{in}
\]
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 Metatheatery

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

Single-threaded  $C_1$  $\longrightarrow$  $C_2$

Multi-threaded  $\pi_1$  $\longrightarrow$  $\longrightarrow$  $\longrightarrow$  $\longrightarrow$  $\star$  $\pi_2$

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