Wysteria:
Secure Multiparty Computation via Functional Programming

Matthew A. Hammer
Aseem Rastogi, Michael Hicks

University of Maryland, Computer Science Dept.
December 2013
Outline

• Background:
  • Secure Multiparty Computation

• Wysteria

• Design elements via Examples
• Language Theory and Meta Theory
• Implementation & Experimental Results
Background: Secure Multiparty Computation
Secure Multi-party Computation (SMC)

- SMC between A and B is an abstraction
- Simulates a trusted third party T
- obliviously computes a public function f
- input/output is private data of A and B
Secure Multi-party Computation (SMC)

1. Send inputs
   - A → T
   - T → B

2. Compute $f$
   - T

3. Recv outputs
   - A ← T
   - B → T
Simple Examples of $f$

- Millionaire's problem
  - Who is richer?
- Private set intersection
  - What elements are common?
- Statistical calculations (e.g., median)
  - Which element is ranked in the middle?
Privacy Model for SMC

• Ideal functionality: All data computed is unknown*, up to knowledge implied by output of $f$
  
  (*) computationally infeasible

• Each output generally depends on all inputs

• Information leakage is inherent to SMC

• SMCs “declassify” private knowledge

• **Design goal:** Protocols should be clear & concise!
Example of SMC in Practice

- Danish Beet Auction (2008--?)
- Three servers: Danisco, DKS, SIMAP
- 1200 bidders
- 30 min to compute market clearing price for beets, via a double auction
Mixing SMC with Local Computation

1. Send inputs
2. Compute $f$
3. Recv outputs

What about multiple interactions with $T$?
(Potential) Reactive Examples

- **Card games** with betting (poker, bridge, etc.)
  - SMC used to *deal random cards*
  - Secure state consists of *card piles, private hands*
  - Shares force *commitment to private decisions*

- **Strategy games** with shared, private maps:
  - SMC used to *roll dice and simulate physics*
  - Secure state consists of a *shared map*
  - Shares enforce *coherence of private maps*
Strategy games

Shared Map

(partially unknown)

Detailed view
Related Work

- **Fairplay** [USENIX 2004]
- **FairplayMP** [CCS 2008]
- **SMCL** [PLAS 2007]
- **TASTY** [CCS 2010]
- **L1** [COMPSAC 2011]
- **PCF** [USENIX 2013]
- **Jif/Split** [~2002]

No functional languages!
Library reuse?
Compositionality?
No formal semantics!
How to prove Correctness?
Wysteria: Design Elements and Examples
Wysteria

- **Wysteria** is an experimental PL design

- Design goal:
  - Compositional PL abstractions for generic, mixed-mode SMC protocols

- Design formula:
  - Simply-Typed Lambda Calculus + ???
Example: Millionaire's Problem

```
let a = par(Alice) = read () in
let b = par(Bob) = read () in
let out = sec({Alice,Bob}) = ( a > b ) in
print out
```
Example: Millionaire's Problem

```
let a = par(Alice) = read () in
let b = par(Bob) = read () in
let out = sec({Alice,Bob}) = ( a > b ) in
print out
```

This is mixed mode

Not yet generic
Abstracting the Millionaire's Problem

is_richer =
\[\lambda x : W \text{ Alice int.} \]
\[\lambda y : W \text{ Bob int.} \]
\[\text{let out } = \text{sec}({\text{Alice, Bob}}) = x[\text{Alice}] > y[\text{Bob}] \]
in out

Function abstracts over private inputs via wire bundles

let a = \text{par(Alice)} = \text{read()} \text{ in}
let b = \text{par(Bob)} = \text{read()} \text{ in}
let out = \text{par(Alice, Bob)} = \text{is_richer (wire Alice a) (wire Bob b)} \text{ in}
print out
Abstracting the Millionaire’s Problem

is_richer =
\lambda x : W \{Alice, Bob\} \text{ int.}
let out = sec({Alice, Bob}) =
x[Alice] > x[Bob]
in out

let a = \text{par(Alice)} = \text{read () in}
let b = \text{par(Bob)} = \text{read () in}
let out = \text{par(Alice, Bob)} =
is_richer (((wire Alice a) ++ (wire Bob b))
in print out
Abstracting the Millionaire’s Problem

Generic protocol:
Abstracts over variable principal set

\[
\text{richest\_of} = \lambda \text{ms} : \text{ps}. \ \lambda x : \mathbb{W} \ \text{ms} \ \text{int}.
\]

\[
\text{let out} = \text{sec}(\text{ms}) = \text{wfold} \ \text{None} \ x
\]

\[
\lambda \text{richest}. \ \lambda p. \ \lambda n. \ \text{match} \ \text{richest} \ \text{with}
\]

\[
| \text{None} \Rightarrow \text{Some} \ p
\]

\[
| \text{Some} \ q \Rightarrow \text{if} \ n > x[q] \ \text{then} \ \text{Some} \ p \ \text{else} \ \text{Some} \ q
\]

\[\text{in} \ \text{out}\]
Abstracting the Millionaire's Problem

let all = \{A, B\} in
let input = (wire A a) ++ (wire B b) in
let r : ps\{singl \& \subseteq all\} option = (richest_of all) input in

richest_of = \lambda ms : ps. \lambda x : W ms int.
let out = sec(ms) =
  wfold None x
  \lambda richest. \lambda p. \lambda n. match richest with
  | None ⇒ Some p
  | Some q ⇒ if n > x[q] then Some p else Some q
in out
Abstracting the Millionaire’s Problem

let all = \{A, B, C\} in
let input = (wire A a) ++ (wire B b) ++ (wire C c) in
let r : ps\{singl ∧ ⊆ all\} option = (richest_of all) input in

richest_of = λms : ps. λx : W ms int.
let out = sec(ms) =
  wfold None x
  λrichest. λp. λn. match richest with
  | None ⇒ Some p
  | Some q ⇒ if n > x[q] then Some p else Some q

in out
Two-round Betting Game (1/2)

Round 1

let \( a_1 = \text{par}(A) = \text{read}() \) in
let \( b_1 = \text{par}(B) = \text{read}() \) in
let \( \text{in}_1 = (\text{wire} A a_1) \oplus (\text{wire} B b_1) \) in
let \((\text{higher}_1, \text{sa}, \text{sb}) = \text{sec}(A,B) = \)
  let \( c = \text{if} \ \text{in}_1[A] > \text{in}_2[B] \text{ then } A \text{ else } B \) in
  \((c, \text{makesh} \ \text{in}_1[A], \text{makesh} \ \text{in}_2[B])\)
in print higher_1

...
Two-round Betting Game (2/2)

Round 2

... ...
let \(a_2 = \text{par}(A) = \text{read}()\) in
let \(b_2 = \text{par}(B) = \text{read}()\) in
let \(\text{in}_2 = (\text{wire} \ A \ a_2) ++ (\text{wire} \ B \ b_2)\) in
let \(\text{higher}_2 = \text{sec}(A,B) =\)
  let \((a_1, b_1) = (\text{combsh} \ sa, \text{combsh} \ sb)\) in
  let \(\text{bida} = (a_1 + \text{in}_2[A]) / 2\) in
  let \(\text{bidb} = (b_1 + \text{in}_2[B]) / 2\) in
  let \(c = \text{if} \ \text{bida} > \text{bidb} \text{ then } A \text{ else } B\) in
in print higher_2
Wysteria’s Design Elements

- **Principals as data**
  - Principals can be inputs and outputs of computation

- **Mixed-mode** computation
  - Designates: Secure vs parallel
  - Designates: Participants

- **Wire bundles** of private data

- **Shares** of secure state
Wysteria Language: Theory & Meta Theory
Wysteria Language Theory

- **Type system** reasons about special abstractions
  - (i.e., principals, wire bundles, modal computation, shares, etc.)

- (Two) Operational semantics:
  - **Single-threaded view** for synchrony
  - **Multi-threaded view** for privacy + parallelism
Fig. 2. Value typing judgement.

\[
\Gamma \vdash_M \tau_1 \subseteq \tau_2
\]

(Subtyping)

Fig. 3. Subtyping and delegation judgements.

\[
\Gamma \vdash_M M > N
\]

(Mode \(M\) can delegate to mode \(N\))

\[
\Gamma \vdash_M e : \tau ; \varepsilon
\]

(Expression typing: “Under \(\Gamma\), expression \(e\) has type \(\tau\), and may be run at \(M\).”)
\[ \Gamma \vdash M \leftarrow \tau \varepsilon \]

Delegation

Values:
Standard stuff + Principal sets

Subtyping:
Refinement implication

Expression typing: "under \( \Gamma \), expression \( e \) has type \( \tau \), and may be run at \( M \)"

Fig. 2. Value typing judgement.

\[ \Gamma \vdash M \leftarrow \tau \varepsilon \]

Fig. 3. Subtyping and delegation judgements.

Standard stuff (datatypes and higher-order functions)

Standard stuff (function application)

Fig. 4. Expression typing judgements.
Single-threaded Semantics

\[ C \xrightarrow{\text{local stepping}} M \]

**C_1 \rightarrow C_2**

\[ M ::= \text{sec}(w) \mid \text{par}(w) \]

\[ C ::= M\{\sigma; k; \psi; e\} \]

store

stack

program counter

environment

fig. 5. \(\lambda_{stp}\) operational semantics of single-threaded configurations
Multi-Threaded Semantics

Protocols
\[ \Pi ::= \varepsilon \mid \Pi_1 \cdot \Pi_2 \mid A \]

Agents
\[ A ::= \text{single}(p)\{\sigma;\kappa;\psi;e\} \]
\[ \mid \text{secure}(w_1 / w_2)\{\sigma;\kappa;\psi;e\} \]

Fig. 6. λM’s operational semantics of multi-threaded target protocols
Wysteria Meta Theory

- **Theorem** (Type soundness):
  - Well-typed programs make progress and preserve types

- **Theorem** (Synchrony):
  - Single- & Multi-threaded operational views agree
Type Soundness

Theorem [Progress]:
Suppose $\Gamma \vdash C_1 : \tau$
then exists $C_2$ such that $C_1 \rightarrow C_2$

Theorem [Preservation]:
Suppose $\Gamma \vdash C_1 : \tau$ and that $C_1 \rightarrow C_2$
then $\Gamma \vdash C_2 : \tau$
Operational Synchrony

Single-threaded

C₁

C₂

Single-threaded

π₁

“Slices to”

π₂

Multi-threaded

Multi-threaded

Thursday, December 12, 2013
Wysteria System: Preliminary Evaluation
Prototype Implementation

Wysteria Client (A)

Public Wysteria program

Front-end Parser

Type Checker

VM

Z3

SMC Server

Private I/O

*SMC server currently uses GMW

Thursday, December 12, 2013
### Implemented Examples

<table>
<thead>
<tr>
<th>Application</th>
<th>$n$ - party?</th>
<th>mixed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Millionaire’s</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>2nd-price auction</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>2-round bidding</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>GPS</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Median</td>
<td>2-party</td>
<td>yes</td>
</tr>
<tr>
<td>PSI</td>
<td>2-party</td>
<td>yes</td>
</tr>
<tr>
<td>Card dealing</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
Multi-party Examples

Several research groups have looked at compiling high-level languages programs involving multiple parties' data to secure protocols. Our work is distinguished from all of these language programs expressed involving multiple parties' data in several respects.

The set sizes are assumed to have a match; it need not be compared in the rest of remaining iterations. Second, once an index in the inner loop is known to have a match, it can iterate over arbitrary but known lengths arrays of players. Just as FairPlayMP programs start to exhibit quadratic time behavior as in the secure-only protocol performs the intersection of their private sets. The set sizes are assumed to be treated as variable.

Support for mixed-mode computations themselves be treated as variable. For the two party cases of the back end of our language, there are libraries for building garbled circuits directly, but is limited to two parties. Our language specifies secure computations in their entirety, avoids entire classes of potential misuse compared to Lfi. Its programs are similar to the unoptimized version, these densities do not perform comparisons for every pair of input elements. These lines of work provide important building blocks for the two principals compute the intersection of their private sets. The set sizes are assumed to be present, moreover, in

specification of roles, we have secure and parallel computations secure multiparty computations. Our approach is less rigid in its surrounding computation that is run locally, in the clear. It also allows programmers to distinguish secure computation from other forms. Our language programs expressed involving multiple parties' data in several respects. The set sizes are assumed to have a match; it need not be compared in the rest of remaining iterations. Second, once an index in the inner loop is known to have a match, it can iterate over arbitrary but known lengths arrays of players. Just as FairPlayMP programs start to exhibit quadratic time behavior as in the secure-only protocol performs the intersection of their private sets. The set sizes are assumed to be treated as variable.

Support for mixed-mode computations themselves be treated as variable. For the two party cases of the back end of our language, there are libraries for building garbled circuits directly, but is limited to two parties. Our language specifies secure computations in their entirety, avoids entire classes of potential misuse compared to Lfi. Its programs are similar to the unoptimized version, these densities do not perform comparisons for every pair of input elements. These lines of work provide important building blocks for the two principals compute the intersection of their private sets. The set sizes are assumed to be present, moreover, in

specification of roles, we have secure and parallel computations secure multiparty computations. Our approach is less rigid in its surrounding computation that is run locally, in the clear. It also allows programmers to distinguish secure computation from other forms. Our language programs expressed involving multiple parties' data in several respects. The set sizes are assumed to have a match; it need not be compared in the rest of remaining iterations. Second, once an index in the inner loop is known to have a match, it can iterate over arbitrary but known lengths arrays of players. Just as FairPlayMP programs start to exhibit quadratic time behavior as in the secure-only protocol performs the intersection of their private sets. The set sizes are assumed to be treated as variable.
which result from the straightline expansion of two nested versions in a subset of $NSI$ (Alternatively at the lowest level). More recently, Bold et al. are records defining each participant's expected input and output on the two-party case. Airplay among more than two parties. Most prior work has focused on designs carefully considering support for secure computations in several respects.

Starting to exhibit quadratic time behavior as in the secure-only version, the mixed-mode version is far more performant. This can be seen in the figure, as the density of matching elements performing comparisons for every pair of input elements is common. For the unoptimized version, these densities do not adapt three different densities of matching elements $\times 4$, $\times 64$, and $\times 4096$ for the representation of parallel-mode loops and arrays. We easily express these optimizations in the language using built-in primitives for expressing parallel modes. It is known to have a match, it need not be compared in the rest of the remaining iterations. Second, once an index in the inner loop is fixed, the pairwise comparison protocol is performed. In contrast, when a matching element is found, the intersection of their private sets is computed. The set sizes are assumed to be public knowledge. As with the intersection of their private sets, the set sizes are assumed to be public knowledge.

The figure compares the secure-only and mixed-mode protocol in several respects. It is a circuit format language for expressing mixed-mode SM, but it is limited to two parties. This is different from Lfs, which is an intermediate language for mixed-mode SM, but provides more generality and ease of expression. These lines of work provide important building blocks towards a fully secure and parallel computation that is run locally, in the clear. It also allows programmers to specify secure computations in their entirety, avoiding entire classes of potential misuse.

Secure-only vs Mixed-mode (Two-party Median)
Private set intersection

Several research groups have looked at compiling high-level languages into secure-only or mixed-mode versions of secure protocols. Our work is distinguished from all of these by proposing two optimizations to this naive approach.

Private set intersection (PSI) is a protocol that allows two parties to compute the intersection of their private sets. The set sizes are assumed to be known to the parties, but there is no need for one party to know the entire set of the other. The PSI protocol performs comparisons for every pair of input elements from the two sets, making it computationally expensive, especially when the sets are large.

The PSI protocol can be coded in two ways: a secure-only pairwise comparison protocol or the unoptimized version. The densities of the sets (the proportion of elements that are a match) significantly affect the performance of the PSI protocol. When densities are low, the secure-only version is more performant than the mixed-mode version, as it avoids unnecessary comparisons for pairs of elements known not to match.

However, when densities are high, the mixed-mode version becomes more performant, as it can take advantage of the fact that if an element is found in the inner loop, the inner loop can be short circuited, avoiding the need to perform a pairwise comparison for that element.

To improve performance at lower densities, as the algorithm increases in size, the mixed-mode version is far more performant, as it can avoid performing comparisons for every pair of input elements, especially when the elements are primitives for expressing parallel mode loops and arrays.

In summary, our work focuses on optimizing the secure-only and mixed-mode versions of the PSI protocol to improve performance, considering the densities of the sets and the presence of primitives for expressing parallel mode loops and arrays.

The graph above illustrates the performance comparison between the secure-only and mixed-mode versions of the PSI protocol for different densities, showing that the mixed-mode version becomes more performant as the density of the sets increases.
Summary and Future Directions
Summary

• Secure Multiparty Computation is a useful abstraction

• SMC simulates interaction with a trusted third party

• Wysteria gives compositional PL abstractions for SMC

• Protocols are (higher-order) functional programs

• Protocols can be generic in the involved principals

• Protocols can mix modes, capturing Reactive SMCs
Future Directions

• More Applications:
  • Card game library: Poker, Bridge, etc.
  • RTS game engine (?)

• Front-end enhancements:
  • Integration into OCaml (as an EDSL)

• Back-end enhancements:
  • Malicious attackers (now: Semi-honest)
  • Integration with Bitcoin:
    • Protocols with monetary transactions
    • Fairness, and disincentive for early quit