MPC Compilers

Scott Constable

Overview

1. Yao’s Garbled Circuit Background (from Peter Snyder’s wonderful “Yao’s Garbled Circuits: Recent Directions and Implementations”)
2. FairplayMP
3. Wysteria

Secure Function Evaluation

“SFE refers to the problem of how two parties can collaborate to correctly compute the output of a function without either party needing to reveal their inputs to the function, either to each other or to a third party.” This definition can be condensed to a computation which satisfies three properties:

1. **Validity**
2. **Privacy**
3. **Fairness**

Adversary Models

- **Semi-Honest**: A semi-honest adversary is assumed to follow the protocol, but may seek to learn additional information while executing the protocol.
- **Malicious**: A malicious (untrusted) adversary is subject to no constraints, and may deviate from the protocol in any way that will allow him/her to learn additional information.

Garbled Circuits

- Transforms any Turing computable function into a boolean circuit, then masks each wire in the circuit so that the party executing the function cannot see the inputs or outputs of each gate.
- One solution to the SFE problem
- Can be adapted to either the semi-honest model (fast) or the malicious model (slow)

Oblivious Transfer

Protocol 1 Semi-Honest 1-out-of-2 Oblivious Transfer

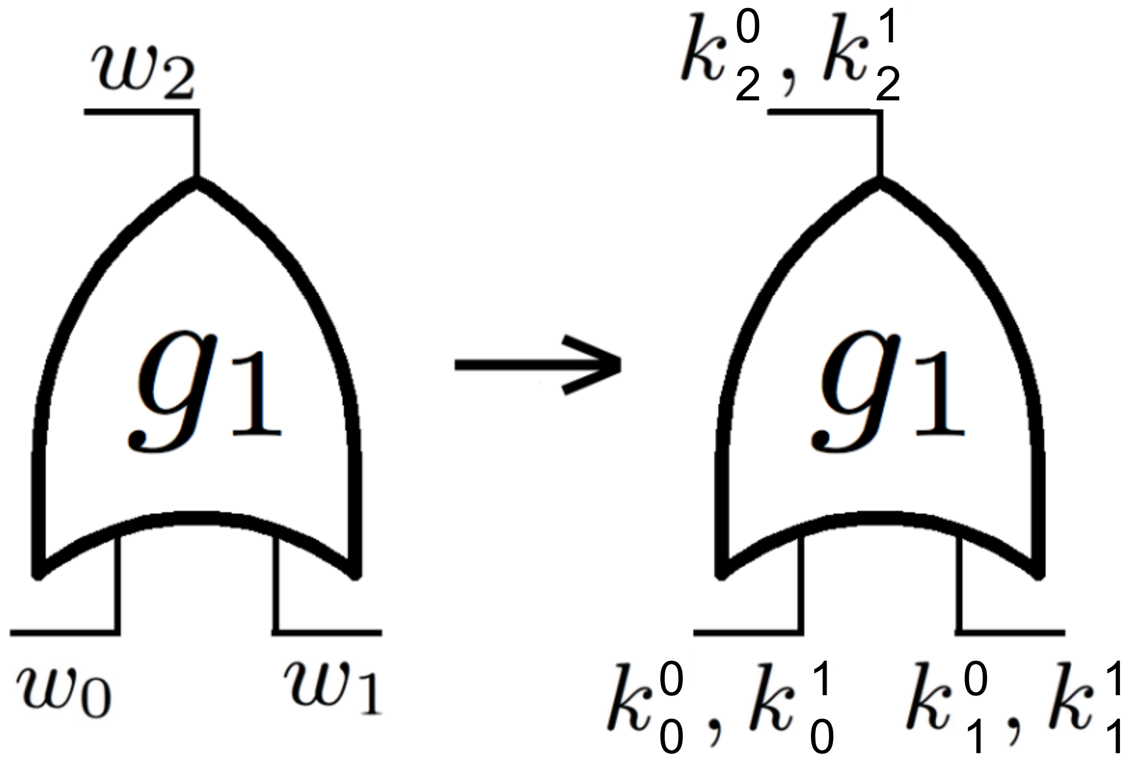
1: *P1* has a set of two strings, $S = \{s_0, s_1\}$.
2: *P2* selects $i \in \{0, 1\}$ corresponding to whether she wishes to learn s_0 or s_1.
3: *P2* generates a public / private key pair (k^{pub}, k^{pri}), along with a second value $k^{\perp}$ that is indistinguishable from a public key, but for which *P2* has no corresponding private key to decrypt with.
4: *P2* then advertises these values as public keys (k_0^{pub}, k_1^{pub}) and sets $k_i^{pub} = k^{pub}$ and $k_{i-1}^{pub} = k^{\perp}$.
5: *P1* generates $c_0 = E_{k_0^{pub}}(s_0)$ and $c_1 = E_{k_1^{pub}}(s_1)$, and sends c_0 and c_1 to *P2*.
6: *P2* computes $s_i = D_{k^{pri}}(c_i)$.

Garbled Circuit Protocol

Protocol 2 Yao's Garbled Circuits Protocol

1: *P1* generates a boolean circuit representation c_c of f that takes input i_{P1} from *P1* and i_{P2} from *P2*.
2: *P1* transforms c_c by garbling each gate's computation table, creating garbled circuit c_g.
3: *P1* sends both c_g and the values for the input wires in c_g corresponding to i_{P1} to *P2*.
4: *P2* uses *1-out-of-2 OT* to receive from *P1* the garbled values for i_{P2} in c_g.
5: *P2* calculates c_g with the garbled versions of i_{P1} and i_{P2} and outputs the result.

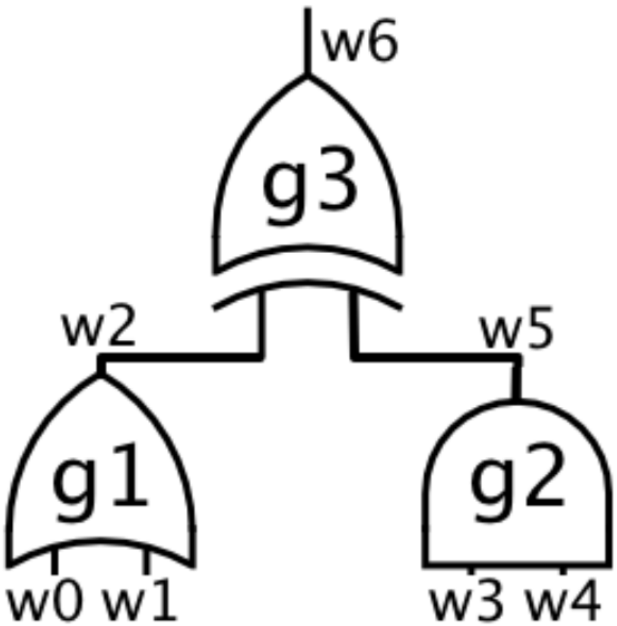
Example 1



w_0	w_1	w_2
0	0	0
0	1	1
1	0	1
1	1	1

w_0	w_1	w_2	garbled value
k_0^0	k_1^0	k_2^0	$H(k_0^0 \| k_1^0 \| g_1) \oplus k_2^0$
k_0^0	k_1^1	k_2^1	$H(k_0^0 \| k_1^1 \| g_1) \oplus k_2^1$
k_0^1	k_1^0	k_2^1	$H(k_0^1 \| k_1^0 \| g_1) \oplus k_2^1$
k_0^1	k_1^1	k_2^1	$H(k_0^1 \| k_1^1 \| g_1) \oplus k_2^1$

Example 2



w_0	w_1	w_2
0	0	0
0	1	1
1	0	1
1	1	1

w_0	w_1	w_2	garbled value
k_0^0	k_1^0	k_2^0	$H(k_0^0 \| k_1^0 \| g_1) \oplus k_2^0$
k_0^0	k_1^1	k_2^1	$H(k_0^0 \| k_1^1 \| g_1) \oplus k_2^1$
k_0^1	k_1^0	k_2^1	$H(k_0^1 \| k_1^0 \| g_1) \oplus k_2^1$
k_0^1	k_1^1	k_2^1	$H(k_0^1 \| k_1^1 \| g_1) \oplus k_2^1$

w_3	w_4	w_5
0	0	0
0	1	0
1	0	0
1	1	1

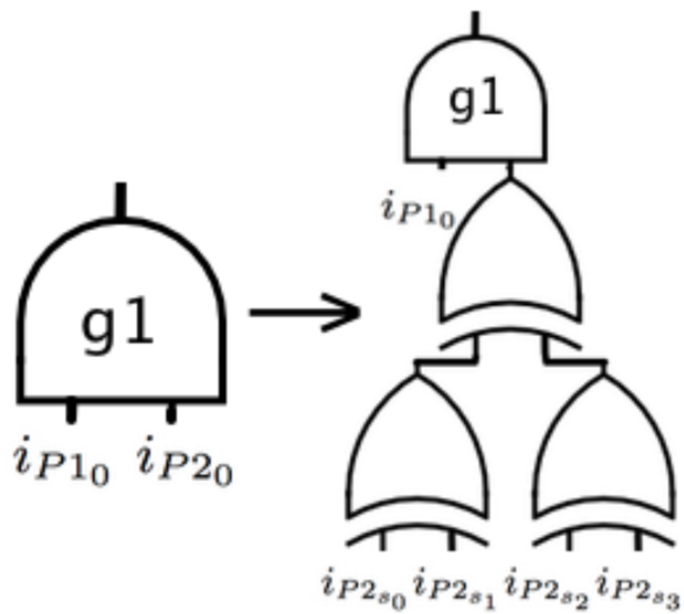
w_3	w_4	w_5	garbled value
k_3^0	k_4^0	k_5^0	$H(k_3^0 \| k_4^0 \| g_2) \oplus k_5^0$
k_3^0	k_4^1	k_5^0	$H(k_3^0 \| k_4^1 \| g_2) \oplus k_5^0$
k_3^1	k_4^0	k_5^0	$H(k_3^1 \| k_4^0 \| g_2) \oplus k_5^0$
k_3^1	k_4^1	k_5^1	$H(k_3^1 \| k_4^1 \| g_2) \oplus k_5^1$

w_2	w_5	w_6
0	0	0
0	1	1
1	0	1
1	1	0

w_2	w_5	w_6	garbled value
k_2^0	k_5^0	k_6^0	$H(k_2^0 \| k_5^0 \| g_3) \oplus k_6^0$
k_2^0	k_5^1	k_6^1	$H(k_2^0 \| k_5^1 \| g_3) \oplus k_6^1$
k_2^1	k_5^0	k_6^1	$H(k_2^1 \| k_5^0 \| g_3) \oplus k_6^1$
k_2^1	k_5^1	k_6^0	$H(k_2^1 \| k_5^1 \| g_3) \oplus k_6^0$

Security

1. OT Protocol
2. Securing Circuit Construction
    a. ZK Proofs
    b. Cut-and-Choose
3. Fairness
4. Corrupt Inputs



FairplayMP

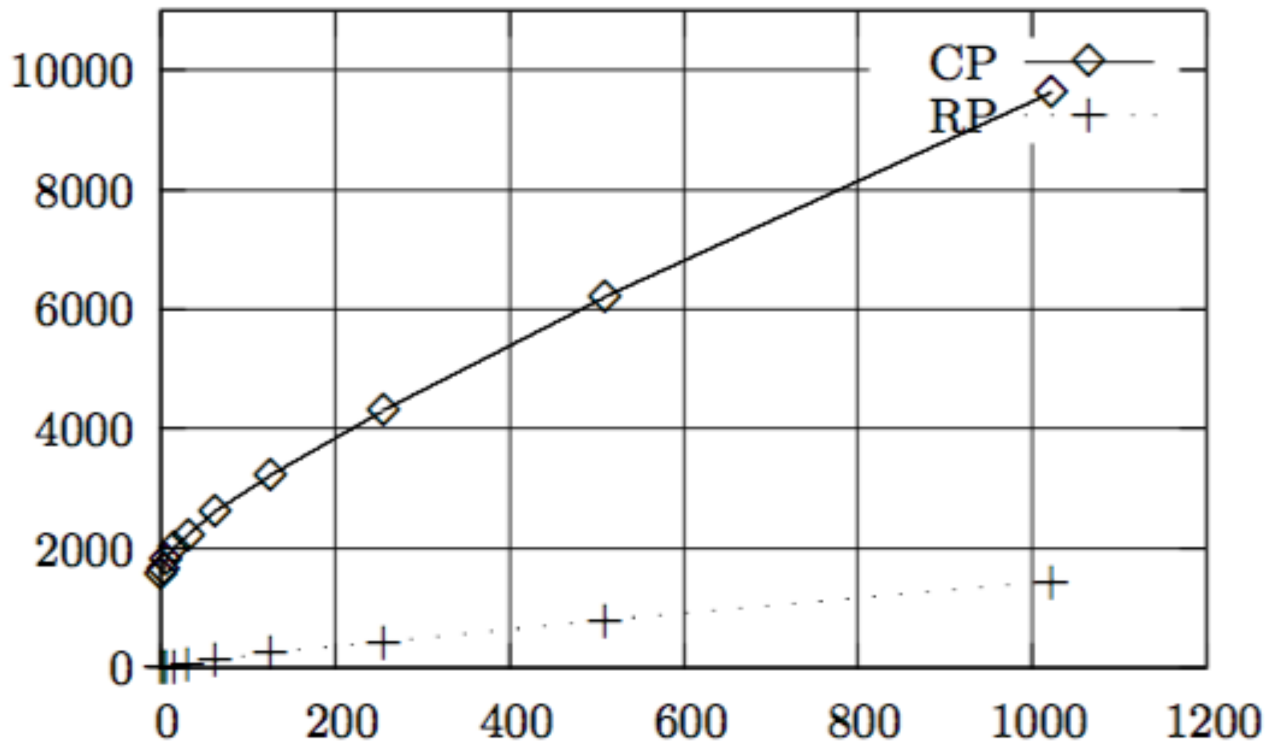
- Assumes a semi-honest adversary model
- Two components:
  - A compiler allows users to describe a SFE using a high-level language, SFDL 2.0
  - Cryptographic engine that executes the BMR protocol
- Constant number of communication rounds
- Allows for more than two parties to participate
- Different parties can see different outputs

```
/**
  Second Price Auction:
  Performs a 2nd price auction between 4
  bidders. Only the winning bidder and the
  seller learn the identity of the winner.
  Everyone knows the 2nd highest price.
**/
program SecondPriceAuction{
  const nBidders = 4;
  type Bid = Int<8>; //enough bits for a bid
  // enough bits to represent a winner.
  type WinningBidder = Int<3>;
  type SellerOutput =
  struct{WinningBidder winner,
         Bid winningPrice};
  //Seller has no input
  type Seller = struct{SellerOutput output};
  type BidderOutput =
  struct{Boolean win, Bid winningPrice};
  type Bidder =
  struct{Bid input, BidderOutput output};
  function void main(Seller seller,
                     Bidder[nBidders] bidder){
    var Bid high;
    var Bid second;
    var WinningBidder winner;
    winner = 0; high = bidder[0].input;
    second = 0;
    // Making the auction.
    for(i=1 to nBidders-1){
      if(bidder[i].input > high){
        winner = i;
        second = high;
        high = bidder[i].input;
      }
      else
        if(bidder[i].input > second)
          second = bidder[i].input;
    }
    // Setting the result.
    seller.output.winner = winner;
    seller.output.winningPrice = second;
    for(i=0 to nBidders-1){
      bidder[i].output.win = (winner == i);
      bidder[i].output.winningPrice = second;
    }
  }
}
```

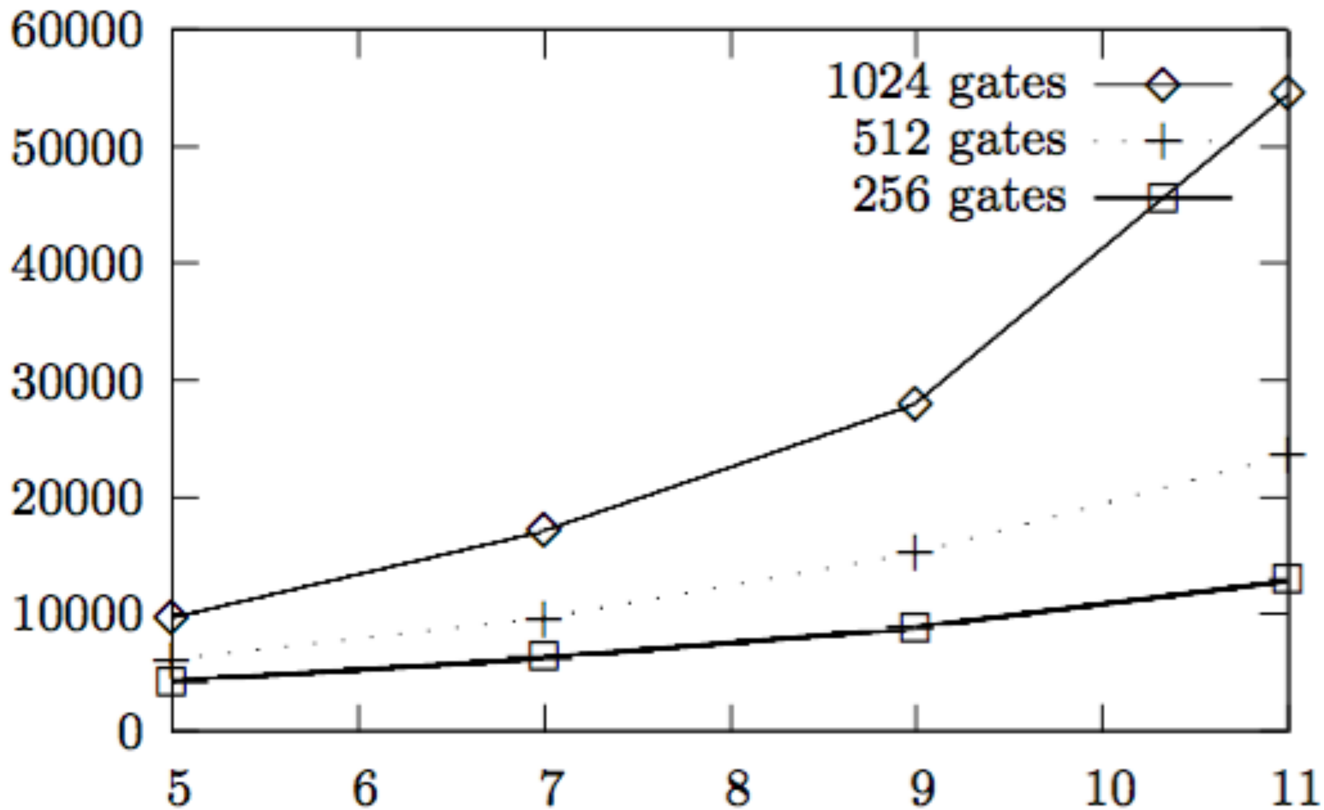
Steps

1. Users express SFE algorithm in SFDL 2.0
2. Program compiled into a boolean circuit
3. Users write a configuration file describing IP addresses, other settings required for protocol execution
4. Program executes the SMPC in two steps:
    a. Garbled circuit created from boolean circuit according to BMR protocol
    b. Garbled circuit evaluated by players who are supposed to receive the respective program outputs

Performance



Performance (Cont.)



Wysteria

Observation: In many SMPC scenarios, some (or most) of the overall computation need not be done in the "secure" mode, i.e. garbled circuits.

Wysteria by Example (1)

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

Wysteria by Example (2)

```
is_richer = λa: W {Alice} nat. λb: W {Bob} nat.
    let out =sec({Alice,Bob})= a[Alice] > b[Bob] in
    out
```

```
let a   =par({Alice})=     read() in
let b   =par({Bob})=       read() in
let out = is_richer (wire {Alice} a) (wire {Bob} b) in
out
```

Wysteria by Example (3)

```
is_richer = λv: W {Alice,Bob} nat.
    let out =sec({Alice,Bob})= v[Alice] > v[Bob] in
    out
```

```
is_richer ((wire {Alice} a) ++ (wire {Bob} b))
```

Wysteria by Example (4)

```
richest_of = λms:ps. λv: W ms nat.
  let out =sec(ms)=
    wfold(None, v,
      λrichest. λp. λn. match richest with
        | None   ⇒ Some p
        | Some q ⇒ if n > v[q] then Some p
                                else Some q) ) )
  in (wire ms out)
```

```
let all = {Alice,Bob,Charlie} in
let r : W all (ps{singl ∧ ⊆ all} option) =
          richest_of all (   wire {Alice}    alice_networth
                          ++ wire {Bob}      bob_networth
                          ++ wire {Charlie}  charlie_networth )
```

Wysteria by Example (5)

```
/* Bidding round 1 of 2: */
let a1  =par({Alice})= read () in
let b1  =par({Bob})= read () in
let in1 = (wire {Alice} a1) ++ (wire {Bob} b1) in
let (higher1, sa, sb) =sec({Alice,Bob})=
  let c = if in1[Alice] > in2[Bob] then Alice else Bob in
  (c, makesh in1[Alice], makesh in1[Bob])
in
print higher1 ;

/* Bidding round 2 of 2: */
let a2 =par({Alice})= read () in
let b2 =par({Bob})= read () in
let in2 = (wire {Alice} a2) ++ (wire {Bob} b2) in
let higher2 =sec({Alice,Bob})=
  let (a1, b1) = (combsh sa, combsh sb) in
  let bid_a = (a1 + in2[Alice]) / 2 in
  let bid_b = (b1 + in2[Bob]) / 2 in
  if bid_a > bid_b then Alice else Bob
in
print higher2
```

Syntax

$$
\begin{array}{llcl}
\text{Principal} & p,\ q & ::= & \textbf{Alice} \mid \textbf{Bob} \mid \textbf{Charlie} \mid \cdots \\
\text{Value} & v,\ w & ::= & x \mid n \mid \textbf{inj}_i\ v \mid (v_1, v_2) \mid p \mid \{w\} \mid w_1 \cup w_2
\end{array}
$$

Expression

$$
\begin{array}{rl}
e\ ::= & v_1 \oplus v_2 \mid \textbf{case}\,(v, x_1.e_1, x_2.e_2) \mid \textbf{fst}\,(v) \mid \textbf{snd}\,(v) \mid \lambda x.e \mid v_1\ v_2 \\
\mid & \textbf{fix}\ x.\lambda y.e \mid \textbf{array}(v_1, v_2) \mid \textbf{select}(v_1, v_2) \mid \textbf{update}(v_1, v_2, v_3) \\
\mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{let}\ x \overset{M}{=} e_1\ \textbf{in}\ e_2 \mid \textbf{wire}_w(v) \mid e_1 +\!\!+\ e_2 \mid v[w] \\
\mid & \textbf{wfold}_w(v_1, v_2, v_3) \mid \textbf{wapp}_w(v_1, v_2) \mid \textbf{waps}_w(v_1, v_2) \\
\mid & \textbf{wcopy}_w(v) \mid \textbf{makesh}(v) \mid \textbf{combsh}(v) \mid v
\end{array}
$$

$$
\begin{array}{llcl}
\text{Type environment} & \Gamma & ::= & . \mid \Gamma, x :_M \tau \mid \Gamma, x : \tau \\
\text{Mode} & M,\ N & ::= & m(w) \mid \top \\
\text{Modal operator} & m & ::= & \mathsf{p} \mid \mathsf{s} \\
\text{Effect} & \epsilon & ::= & \cdot \mid M \mid \epsilon_1, \epsilon_2
\end{array}
$$

$$
\begin{array}{llcl}
\text{Refinement} & \phi & ::= & \mathsf{true} \mid \textbf{singl}(\nu) \mid \nu \subseteq w \mid \nu = w \mid \phi_1 \wedge \phi_2 \\
\text{Type} & \tau & ::= & \textbf{nat} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \textbf{ps}\ \phi \mid \mathbf{W}\ w\ \tau \\
 & & & \mid \textbf{Array}\ \tau \mid \textbf{Sh}\ w\ \tau \mid x{:}\tau_1 \xrightarrow{\epsilon} \tau_2
\end{array}
$$

Typing Goals

1. Each variable can only be used in an appropriate mode
2. Delegated computations require that all participating principals are present in the current mode
3. Parallel local state must remain consistent across parallel principals
4. Code in secure blocks must be restricted so that it can be compiled to a boolean circuit

Type Semantics: Value Typing

T-VAR
$$\frac{x :_M \tau \in \Gamma \vee x : \tau \in \Gamma \quad \Gamma \vdash \tau}{\Gamma \vdash_M x : \tau}$$

T-NAT
$$\frac{}{\Gamma \vdash_M n : \mathbf{nat}}$$

T-INJ
$$\frac{\Gamma \vdash_M v : \tau_i \quad j \in \{1, 2\} \quad \tau_j \ \texttt{IsFlat} \quad \Gamma \vdash \tau_j}{\Gamma \vdash_M \mathbf{inj}_i\ v : \tau_1 + \tau_2}$$

T-PROD
$$\frac{\Gamma \vdash_M v_i : \tau_i}{\Gamma \vdash_M (v_1, v_2) : \tau_1 \times \tau_2}$$

T-PRINC
$$\frac{}{\Gamma \vdash_M p : \mathbf{ps}\ (\nu = \{p\})}$$

T-PSONE
$$\frac{\Gamma \vdash_M w : \mathbf{ps}\ (\mathbf{singl}(\nu))}{\Gamma \vdash_M \{w\} : \mathbf{ps}\ (\nu = \{w\})}$$

T-PSUNION
$$\frac{\Gamma \vdash_M w_i : \mathbf{ps}\ \phi_i}{\Gamma \vdash_M w_1 \cup w_2 : \mathbf{ps}\ (\nu = w_1 \cup w_2)}$$

T-PSVAR
$$\frac{\Gamma \vdash_M x : \mathbf{ps}\ \phi}{\Gamma \vdash_M x : \mathbf{ps}\ (\nu = x)}$$

T-MSUB
$$\frac{\Gamma \vdash M \quad \Gamma \vdash_M x : \tau \quad \Gamma \vdash M \rhd N \quad N = \mathsf{s}(\_) \Rightarrow \tau\ \texttt{IsSecIn}}{\Gamma \vdash_N x : \tau}$$

T-SUB
$$\frac{\Gamma \vdash_M v : \tau_1 \quad \Gamma \vdash \tau_1 <: \tau \quad \Gamma \vdash \tau}{\Gamma \vdash_M v : \tau}$$

Type Semantics: Delegation

D-REFL
$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu = w_1)}{\Gamma \vdash m(w_1) \rhd m(w_2)}$$

D-TOP
$$\frac{\Gamma \vdash w : \mathbf{ps}\ \phi}{\Gamma \vdash \top \rhd m(w)}$$

D-PAR
$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_1)}{\Gamma \vdash \mathsf{p}(w_1) \rhd \mathsf{p}(w_2)}$$

D-SEC
$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu = w_1)}{\Gamma \vdash \mathsf{p}(w_1) \rhd \mathsf{s}(w_2)}$$

Type Semantics: Subtyping

S-REFL
$$\frac{}{\Gamma \vdash \tau <: \tau}$$

S-TRANS
$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}$$

S-SUM
$$\frac{\Gamma \vdash \tau_i <: \tau_i'}{\Gamma \vdash \tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

S-PROD
$$\frac{\Gamma \vdash \tau_i <: \tau_i'}{\Gamma \vdash \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

S-PRINCS
$$\frac{[\![\Gamma]\!] \vDash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \mathbf{ps}\ \phi_1 <: \mathbf{ps}\ \phi_2}$$

S-WIRE
$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_1) \quad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash \mathbf{W}\ w_1\ \tau_1 <: \mathbf{W}\ w_2\ \tau_2}$$

S-ARRAY
$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \mathbf{Array}\ \tau_1 <: \mathbf{Array}\ \tau_2}$$

S-SHARE
$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu = w_1) \quad \Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \mathbf{Sh}\ w_1\ \tau_1 <: \mathbf{Sh}\ w_2\ \tau_2}$$

S-ARROW
$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \quad \Gamma, x : \tau_1' \vdash \tau_2 <: \tau_2'}{\Gamma \vdash x{:}\tau_1 \xrightarrow{\epsilon} \tau_2 <: x{:}\tau_1' \xrightarrow{\epsilon} \tau_2'}$$

Type Semantics: Expressions

T-BINOP
$$\frac{\Gamma \vdash_M v_i : \textbf{nat}}{\Gamma \vdash_M v_1 \oplus v_2 : \textbf{nat}; \cdot}$$

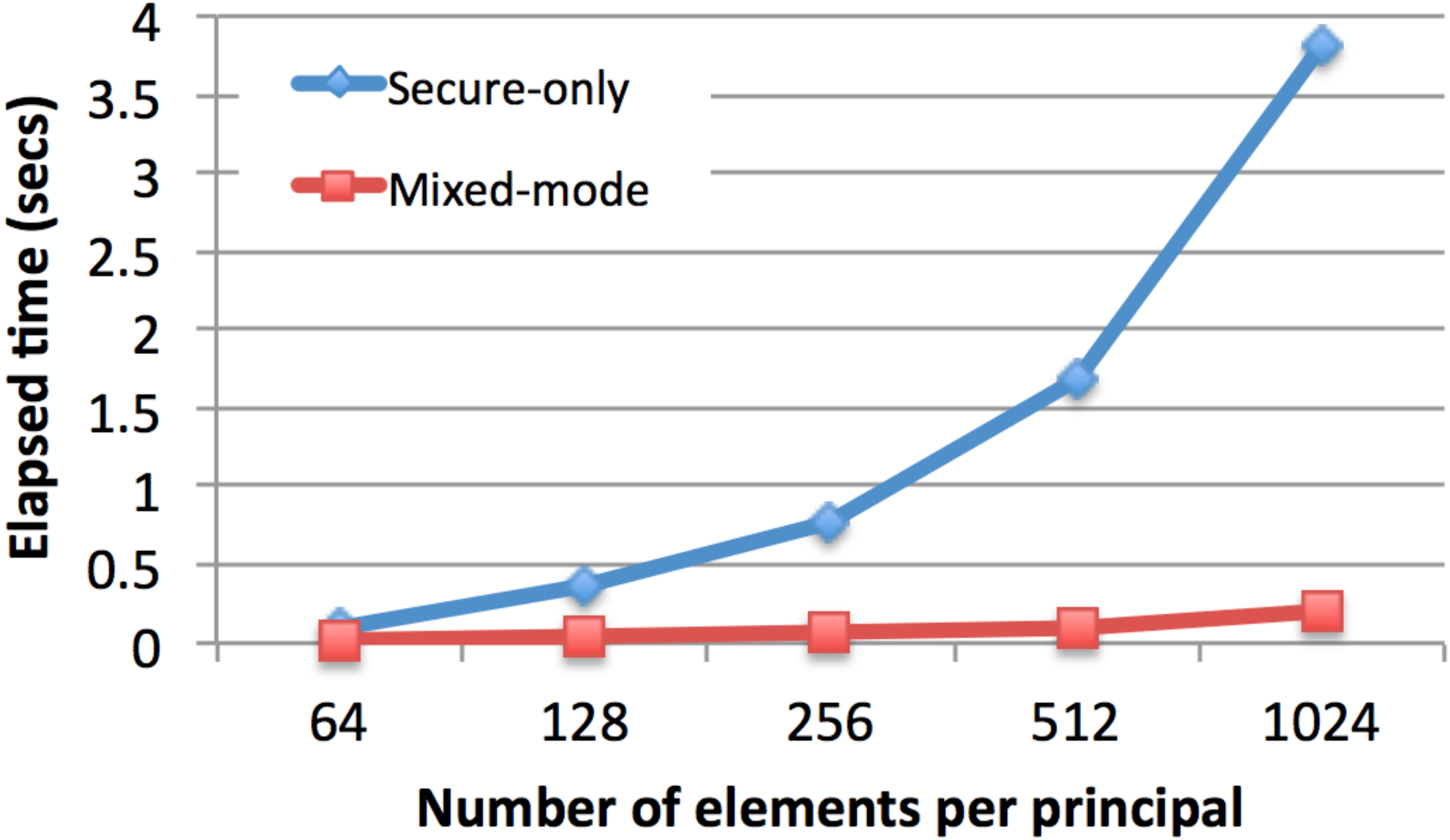
T-FST
$$\frac{\Gamma \vdash_M v : \tau_1 \times \tau_2}{\Gamma \vdash_M \textbf{fst}\,(v) : \tau_1; \cdot}$$

T-SND
$$\frac{\Gamma \vdash_M v : \tau_1 \times \tau_2}{\Gamma \vdash_M \textbf{snd}\,(v) : \tau_2; \cdot}$$

T-LAM
$$\frac{\Gamma \vdash \tau \qquad \Gamma, x : \tau \vdash_M e : \tau_1; \epsilon}{\Gamma \vdash_M \lambda x.e : (x{:}\tau \xrightarrow{\epsilon} \tau_1); \cdot}$$

T-APP
$$\frac{\begin{array}{c}\Gamma \vdash_M v_1 : x{:}\tau_1 \xrightarrow{\epsilon} \tau_2 \\ \Gamma \vdash v_2 : \tau_1 \qquad \Gamma \vdash \tau_2[v_2/x] \\ \Gamma \vdash M \rhd \epsilon[v_2/x] \qquad M = \mathtt{s}(\_) \Rightarrow \tau_2 \ \mathtt{IsFO}\end{array}}{\Gamma \vdash_M v_1\ v_2 : \tau_2[v_2/x]; \epsilon[v_2/x]}$$

Performance



Thanks! Questions?