Yao’s Garbled Circuits
Recent Directions and Implementations

Pete Snyder
Outline

1. Context
2. Security definitions
3. Oblivious transfer
4. Yao’s original protocol
5. Security improvements
6. Performance improvements
7. Implementations
8. Conclusion
Outline

1. Context
2. Security definitions
3. Oblivious transfer
4. Yao’s original protocol
5. Security improvements
6. Performance improvements
7. Implementations
8. Conclusion
1. Context for Yao’s Protocol

- Secure function evaluation
- Computing functions with hidden inputs
- “Millionaires’ problem”
Yao and SFE

- Initially only considered theoretically interesting
- Later became focus of practical work
- Yao never published protocol
Outline

1. Context

2. Security definitions

3. Oblivious transfer

4. Yao's original protocol

5. Security improvements

6. Performance improvements

7. Implementations

8. Conclusion
2. Definitions and Assumptions

- Properties of a “secure” SFE protocol
- Adversary models
2.1. SFE Properties

• Could try to fully define what a SFE system can and cannot leak
  • Might quickly devolve into long arbitrary lists

• Instead, compare a solution to a best-possible 3rd party / ideal - oracle
Ideal Oracle

\[ u \leftarrow f(i_{p1}, i_{p2}) \]
Validity

• A SFE protocol must provide the same result as an ideal oracle

• Does not require:
  • correct answer
  • any answer at all
Privacy

• A SFE protocol must not allow parties to learn more about each other’s inputs than they would with an ideal oracle

• Does not require:
  • That parties cannot learn inputs
  • ex: integer multiplication
Fairness

• A SFE protocol must not allow one party to learn result while keeping it from the other.

• Tricky…
2.2. Adversary Models

**Semi-Honest**
- Follows protocol
- Will take advantage where allowed
- Has transcript of entire protocol

**Malicious**
- Arbitrarily deviates from protocol
- Will take any beneficial actions
- More “real-world”
Outline

1. Context

2. Security definitions

3. **Oblivious transfer**

4. Yao's original protocol

5. Security improvements

6. Performance improvements

7. Implementations

8. Conclusion
3. Oblivious transfer

- What is oblivious transfer
- Simple protocol
What is Oblivious Transfer

- OTs is category of 2-party protocols
  - P1 has some values
  - P2 learns some values but not others
  - P1 doesn’t know what P2 learns
- Yao’s protocol builds on OT
1-out-of-2 Oblivious Transfer

Inputs

- **P1**: $S = \{s_0, s_1\}$
- **P2**: $i \in \{0, 1\}$

Receives

- **P1**: Nothing
- **P2**: $S_i$ but not $S_{i-1}$
Example OT Protocol

S = \{s_0, s_1\}

i \in \{0, 1\}

(k^{pub}, k^{prv}), (k^\perp, \perp)

k^{pub}_i = k^{pub}, k^{pub}_{i-1} = k^\perp

c_i = E_{k^{pub}_i}(s_i), c_{i-1} = E_{k^{pub}_{i-1}}(s_{i-1})

S_i = D_{k^{pri}_i}(c_i), \perp = D_{\perp}(c_{i-1})
Outline

1. Context
2. Definitions and assumptions
3. Oblivious transfer
4. Yao’s original protocol
5. Security improvements
6. Performance improvements
7. Implementations
8. Conclusion
4. Yao’s Protocol

• “Intuitive” description (hopefully…)

• Detailed description
Yao’s Garbled Circuits

1. P1 and P2 want to securely compute \( f \)

2. P1: Creates circuit representation of \( f \)

3. P1: “garbles” the circuit so that P2 can execute the circuit, but not learn intermediate values

4. P1: Sends P2 the garbled circuit and his garbled input bits

5. P2: Uses OT to receive P2’s input bits

6. P2: Evaluates circuit
1. Generating equivalent boolean circuit for the function

- Create circuit $c$ such that $\forall x, y \rightarrow f(x, y) = c(x, y)$
- Beyond this talk (compiler theory, etc.)
- Implementations use domain specific high level languages
2. Garbling the circuit

• Goal is to allow P2 to compute circuit w/o knowing intermediate values of circuit

• Garbling means mapping binary values to encryption keys, and encrypting outputs of gates

• Pre-garbling: Gates are \( \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\} \)

• Post-garbling: \( f(\{0, 1\}^{|k|}, \{0, 1\}^{|k|}) \rightarrow \{0, 1\}^{|k|} \)
Preparing one gate

\[ g_1 \]

\[
\begin{array}{ccc}
<table>
<thead>
<tr>
<th>w_0</th>
<th>w_1</th>
<th>w_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\end{array}
\]

\[
\begin{array}{cccc}
<table>
<thead>
<tr>
<th>k_0^0</th>
<th>k_1^0</th>
<th>k_2^0</th>
<th>E_{k_0^0}(E_{k_1^0}(k_2^0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>k_0^0</td>
<td>k_1^0</td>
<td>k_2^0</td>
<td>E_{k_0^0}(E_{k_1^1}(k_2^0))</td>
</tr>
<tr>
<td>k_0^1</td>
<td>k_1^1</td>
<td>k_2^0</td>
<td>E_{k_0^1}(E_{k_1^0}(k_2^0))</td>
</tr>
<tr>
<td>k_0^1</td>
<td>k_1^1</td>
<td>k_2^0</td>
<td>E_{k_0^1}(E_{k_1^1}(k_2^0))</td>
</tr>
</tbody>
</table>
\end{array}
\]
3. Garbling P1’s Input

- **P1** has garbled circuit
- **P1** has original $i_{p1}$
- **P2** has original $i_{p2}$
- Circuit only contains garbled / mapped values
Garbling $i_{p_1}$

Original $i_{p_1}$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>1</td>
</tr>
<tr>
<td>w</td>
<td>1</td>
</tr>
<tr>
<td>w</td>
<td>0</td>
</tr>
</tbody>
</table>

Garbled $i_{p_1}$

<table>
<thead>
<tr>
<th></th>
<th>$k^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>$k^1$</td>
</tr>
<tr>
<td>w</td>
<td>$k^1$</td>
</tr>
<tr>
<td>w</td>
<td>$k^0$</td>
</tr>
</tbody>
</table>
4. Garbling P2’s input

• **P2** has garbled circuit, garbled $i_{p1}$, original $i_{p2}$

• **P1** has mappings boolean → garbled mappings

• To compute circuit, **P2** needs garbled input values
Garbling $i_p^2$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w$</td>
<td>$k^0$</td>
</tr>
<tr>
<td>$w$</td>
<td>$k^0$</td>
</tr>
<tr>
<td>$w$</td>
<td>$k^0$</td>
</tr>
<tr>
<td>$w$</td>
<td>$k^0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$</th>
<th>garbled</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w$</td>
<td>0</td>
</tr>
<tr>
<td>$w$</td>
<td>0</td>
</tr>
<tr>
<td>$w$</td>
<td>1</td>
</tr>
<tr>
<td>$w$</td>
<td>0</td>
</tr>
</tbody>
</table>
Garbling $i_{p2}$

1-out-of-2 OT

$N = \{k_2^0, k_2^1\}$  $i = 0$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>$k_0$</td>
<td>$k_1$</td>
</tr>
<tr>
<td>w</td>
<td>$k_0$</td>
<td>$k_1$</td>
</tr>
<tr>
<td>w</td>
<td>$k_0$</td>
<td>$k_1$</td>
</tr>
<tr>
<td>w</td>
<td>$k_0$</td>
<td>$k_1$</td>
</tr>
<tr>
<td>w</td>
<td>$k_0$</td>
<td>$k_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$i$</th>
<th>garbled</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>w</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>w</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>w</td>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>
Garbling $i_{p2}$

1-out-of-2 OT

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>$k^0_2$</td>
<td>$k^1_2$</td>
</tr>
<tr>
<td>w</td>
<td>$k^0_2$</td>
<td>$k^1_2$</td>
</tr>
<tr>
<td>w</td>
<td>$k^0_2$</td>
<td>$k^1_2$</td>
</tr>
<tr>
<td>w</td>
<td>$k^0_2$</td>
<td>$k^1_2$</td>
</tr>
<tr>
<td>w</td>
<td>$k^0_2$</td>
<td>$k^1_2$</td>
</tr>
</tbody>
</table>

$\begin{array}{|c|c|c|}
\hline
i & \text{garbled} \\
\hline
w & 0 & k^0_2 \\
\hline
w & 0 & ? \\
\hline
w & 1 & ? \\
\hline
w & 0 & ? \\
\hline
\end{array}$
5. Computing the circuit

- **P2**: Garbled circuit, $i_{p1}, i_{p2}$

- **P2**: Tries each row in table to see what key the inputs unlock

Assume **P1**’s input is 1 and **P2**’s input is 0
Outline

1. Context
2. Security definitions
3. Oblivious transfer
4. Yao’s original protocol
5. **Security improvements**
6. Performance improvements
7. Implementations
8. Conclusion
5. Security improvements

• Yao is only secure against semi-honest adversaries

• Areas for improvement
  1. Securing oblivious transfer
  2. Securing circuit construction
  3. Securing against corrupt inputs

• Remaining issues...
Securing oblivious transfer

- Problem with existing implementation:
  - Initially $P_2$ generates $(k^{\text{pub}}, k^{\text{prv}}), (k^\bot, \bot)$
  - $P_1$ can’t verify that $P_2$ holds only one private key
  - $P_2$ can learn garbled values of 0 and 1 bits for $P_2$’s input wires
  - Allows for violations of privacy SFE principal in malicious case
Securing oblivious transfer

- Solution:
  - \textbf{P2} needs to provably bind itself from being able to decrypt both sent values
  - \textbf{P1} still cannot learn \textbf{P2}'s selected value
Securing oblivious transfer

- Selects $C \in \mathbb{Z}_q^*$ such that $P_2$ does not know discrete log of $C$
- Selects $i \in \{0, 1\}$
- Selects $x_i$, $0 \leq i < q-2$
- $\beta_i = g^{x_i}$, $\beta_{i-1} = C^* (g^{x_i})^{-1}$
- Verifies that $\beta_i \cdot \beta_{i-1} = C$
- If so, proceed similarly to previous protocol
Securing circuit construction

• Problem with existing implementation:
  • $\textbf{P1}$ can construct a garbled circuit that computes $f'$ instead of $f$
  • $f'$ could echo $i_{p2}$ (or something more subtle)
  • $\textbf{P1}$ could learn $\textbf{P2}$'s input
  • Allows for violations of privacy SFE principal in malicious case
Securing circuit construction

- Zero-Knowledge Proofs
  - Too expensive for practical use
- Cut-and-Choose
  - P1 garbles multiple circuits, P2 checks some
  - Cat and mouse game
Cut-and-Choose v1.0

- Uniquely garbles $m$ versions of the circuit
- Un-garbles selected circuits
- Selects $m-1$ circuits to verify
- Verifies $m-1$ circuits are correct

Protocol continues as normal
Cut-and-Choose v1.0

- Reduces $P_1$’s chance to successfully cheat to $1/m$
- $1/m$ might not be enough security
- Verifying circuits is expensive, generating circuits is expensive
- Would be nice to get $\geq 1-(1/m)$ confidence for $\leq$ work
Cut-and-Choose v2.0

- Uniquely garbles $m$ versions of the circuit
- Un-garbles selected circuits
- Selects $m/2$ circuits to verify
- Verifies $m/2$ circuits
- Compute remaining $m/2$ circuits, abort if differences

Protocol continues as normal
Cut-and-Choose v2.0

• **P1** will only succeed in attack if:
  • **P1** generates $m/2$ corrupt circuits
  • None of these $m/2$ circuits are among the $m/2$ **P2** selects to be revealed

• **P1**’s chance of success is tiny…

• But opens up a new early abort attack from **P1**…
Securing against corrupt inputs

- **P1** submits malicious input in OT:
  - 0 = valid garbled bit of $i_{P2}$, 1 = ⊥
- If **P2** returns, $i_{P2b} = 0$, if **P2** aborts, $i_{P2b} = 1$
- **P1** learns 1 bit of $i_{P2}$, violating *privacy* SFE principal
Securing against corrupt inputs

- Augment circuits with $s$ additional input bits leading into XOR gates
- Gives $P_2$ $2^{s-1}$ ways to generate true desired input bit
- $P_1$ can still force abort, but learns nothing from it
Ensuring P2 returns anything

- *Fairness* SFE principal requires that P2 not be able to learn anything P1 cannot

- No solutions to add this assurance to Yao

- Yao’s protocol is not *fair*, and so not secure, in *malicious* case

- Focus on second best: ensuring that if P2 does return, result is correct
  - Return encrypted values that P1 has key for
  - Signature based solutions
Outline

1. Context
2. Security definitions
3. Oblivious transfer
4. Yao’s original protocol
5. Security improvements
6. Performance improvements
7. Implementations
8. Conclusion
6. Performance improvements

- Yao’s protocol is “efficient” but expensive
- State of the art implementation takes 8 hours to compute large string edit distance
- Billions of gates, gigs or more of memory per circuit
Areas for improvement

- Communication optimizations
- Execution optimizations
- Circuit optimizations
Communication optimizations

• Recall cut-and-check requires $m$ circuits

• $m$ circuits *
  billions of gates *
  4 multi byte values for each gate =
  gigabytes to terabytes of overhead

• Can we do something about $m$?
Communication optimizations

• “Random Seed Checking”

• Don’t randomly assign keys

• Do so pseudo-randomly from initial random seed

• Instead of sending $m/2$ verification circuits, $P_1$ send commitments of circuit construction and then initial random seed

• $P_2$ reconstructs circuit from random seed and checks that it matches the commitment
Execution optimizations

- Fast table lookups
- Pipelined circuit execution
Fast table lookups

Assume \( P_1 \)'s input is 1 and \( P_2 \)'s input is 0.
Fast table lookups

- Two index bits (one from each input wire) uniquely identify rows in each gate
- Slight increase in circuit construction cost
- Circuit execution now only needs one decryption per gate, instead of on average 2
Pipelined circuit execution

- Standard version of Yao’s protocol has
  - $P_1$ garbles, $P_2$ waits
  - $P_2$ evaluates, $P_1$ waits
Pipelined circuit execution

- Construction of input gates
- Oblivious Transfer
- Completing circuit construction
- Circuit evaluation
Circuit optimizations

- Circuit simplification
- Free XORs
- "Garbled row reduction"
Circuit simplification

• removing errors in the \( f \rightarrow \) circuit conversion
• Remove dead chunks of the circuit
• Reduce sub-circuits that can be more efficiently represented by a smaller number of gates
• 60% reduction in circuit size for some circuit constructing tools (ex Fairplay)
Free XORs

- By default all garbled values are independent
- Take advantage of this by fixing input values to XOR gates with single random $R$
- Replace XOR gates with an XOR function
- Remove 4 garbled values for each XOR gate
Free XORs

Graph:
- Node g3 with inputs w2 and w5
- Node g1 with inputs w0 and w1
- Node g2 with inputs w3 and w4

Labels:
- P1
- P2
Free XORs

XOR $w_6$

<table>
<thead>
<tr>
<th>$w_2$</th>
<th>$w_5$</th>
<th>$w_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

OR

<table>
<thead>
<tr>
<th>$w_0$</th>
<th>$w_1$</th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

AND

<table>
<thead>
<tr>
<th>$w_3$</th>
<th>$w_4$</th>
<th>$w_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$P_1$ $P_2$
### XOR

<table>
<thead>
<tr>
<th>(w_2)</th>
<th>(w_5)</th>
<th>(w_6)</th>
<th>“unpacked”</th>
<th>“simplified”</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k_2^0)</td>
<td>(k_5^0)</td>
<td>(k_2^0 \oplus k_5^0)</td>
<td>(k_2^0 \oplus k_5^0)</td>
<td>(k_2^0 \oplus k_5^0)</td>
</tr>
<tr>
<td>(k_2^0)</td>
<td>(k_5^1)</td>
<td>(k_2^0 \oplus k_5^1)</td>
<td>(k_2^0 \oplus (k_5^0 \oplus R))</td>
<td>((k_2^0 \oplus k_5^0) \oplus R)</td>
</tr>
<tr>
<td>(k_2^1)</td>
<td>(k_5^0)</td>
<td>(k_2^1 \oplus k_5^0)</td>
<td>((k_2^0 \oplus R) \oplus k_5^0)</td>
<td>((k_2^0 \oplus k_5^0) \oplus R)</td>
</tr>
<tr>
<td>(k_2^0)</td>
<td>(k_5^1)</td>
<td>(k_2^0 \oplus k_5^1)</td>
<td>((k_2^0 \oplus R) \oplus (k_5^0 \oplus R))</td>
<td>((k_2^0 \oplus k_5^0) \oplus R)</td>
</tr>
</tbody>
</table>

### OR

<table>
<thead>
<tr>
<th>(w_0)</th>
<th>(w_1)</th>
<th>(w_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k_0^0)</td>
<td>(k_1^0)</td>
<td>(k_2^0)</td>
</tr>
<tr>
<td>(k_0^0)</td>
<td>(k_1^1)</td>
<td>(k_2^0 \oplus R)</td>
</tr>
<tr>
<td>(k_0^1)</td>
<td>(k_1^0)</td>
<td>(k_2^0 \oplus R)</td>
</tr>
<tr>
<td>(k_0^1)</td>
<td>(k_1^1)</td>
<td>(k_2^0 \oplus R)</td>
</tr>
</tbody>
</table>

### AND

<table>
<thead>
<tr>
<th>(w_3)</th>
<th>(w_4)</th>
<th>(w_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k_3^0)</td>
<td>(k_4^0)</td>
<td>(k_5^0)</td>
</tr>
<tr>
<td>(k_3^0)</td>
<td>(k_4^1)</td>
<td>(k_5^0)</td>
</tr>
<tr>
<td>(k_3^1)</td>
<td>(k_4^0)</td>
<td>(k_5^0)</td>
</tr>
<tr>
<td>(k_3^1)</td>
<td>(k_4^1)</td>
<td>(k_5^0)</td>
</tr>
<tr>
<td>(k_3^1)</td>
<td>(k_4^1)</td>
<td>(k_5^0 \oplus R)</td>
</tr>
</tbody>
</table>
Garbled row reduction

• Similar to free XOR trick, but saves just one row

• Used for AND and OR gates

• Relies on the “fast table lookups” optimization

• Special cases garbled output value for one gate index, ex (0, 0)

• key is a function of input keys
Garbled row reduction
Garbled row reduction

AND \( w_6 \)

\[
\begin{array}{c|c|c}
w_2 & w_5 & w_6 \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

OR

W_2

AND

W_5

\[
\begin{array}{c|c|c}
w_0 & w_1 & w_2 \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
w_3 & w_4 & w_5 \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

W_0 W_1 W_3 W_4

P1 P2 P1 P2
Garbled row reduction

\begin{align*}
\text{AND } w_6
\end{align*}

\begin{align*}
\text{OR}
\end{align*}

\begin{align*}
\text{AND}
\end{align*}

\begin{align*}
\text{AND}
\end{align*}
Garbled row reduction

<table>
<thead>
<tr>
<th>w₂</th>
<th>w₅</th>
<th>w₆</th>
<th>garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k₀</td>
<td>k₀</td>
<td>k₀</td>
<td>$H(k₀^0</td>
</tr>
<tr>
<td>k₀</td>
<td>k₁</td>
<td>k₁</td>
<td>$H(k₀^0</td>
</tr>
<tr>
<td>k₁</td>
<td>k₀</td>
<td>k₀</td>
<td>$H(k₁^0</td>
</tr>
<tr>
<td>k₁</td>
<td>k₁</td>
<td>k₁</td>
<td>$H(k₁^1</td>
</tr>
</tbody>
</table>

w₂ AND w₆

<table>
<thead>
<tr>
<th>w₀</th>
<th>w₁</th>
<th>w₂</th>
<th>garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k₀</td>
<td>k₀</td>
<td>k₀</td>
<td>$H(k₀^0</td>
</tr>
<tr>
<td>k₀</td>
<td>k₁</td>
<td>k₂</td>
<td>$H(k₀^0</td>
</tr>
<tr>
<td>k₁</td>
<td>k₀</td>
<td>k₀</td>
<td>$H(k₀^0</td>
</tr>
<tr>
<td>k₁</td>
<td>k₁</td>
<td>k₂</td>
<td>$H(k₁^0</td>
</tr>
</tbody>
</table>

w₀ OR w₁

<table>
<thead>
<tr>
<th>w₃</th>
<th>w₄</th>
<th>w₅</th>
<th>garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k₀</td>
<td>k₀</td>
<td>k₀</td>
<td>$H(k₀^0</td>
</tr>
<tr>
<td>k₀</td>
<td>k₁</td>
<td>k₄</td>
<td>$H(k₀^0</td>
</tr>
<tr>
<td>k₃</td>
<td>k₄</td>
<td>k₀</td>
<td>$H(k₃^0</td>
</tr>
<tr>
<td>k₃</td>
<td>k₄</td>
<td>k₅</td>
<td>$H(k₃^0</td>
</tr>
</tbody>
</table>

w₃ AND w₅

W₀ W₁ W₃ W₄

P₁ P₂ P₁ P₂
Outline

1. Context
2. Security definitions
3. Oblivious transfer
4. Yao’s original protocol
5. Security improvements
6. Performance improvements
7. Implementations
8. Conclusion
7. Implementations

• FairPlay (2004)
• Huang, Evans, Katz, Malka (2011)
• Kreuter, shelat, Shen (2012)
<table>
<thead>
<tr>
<th>Year</th>
<th>Security</th>
<th>Largest Circuit</th>
<th>Problems</th>
<th>Introduced Performance Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Semi-Malicious</td>
<td>4.3k</td>
<td>Very simple</td>
<td>Fast Table Lookups, Performance OT Protocols</td>
</tr>
<tr>
<td>2011</td>
<td>Semi-Honest</td>
<td>1 billion</td>
<td>Edit Distances AES</td>
<td>Free XORs, Garbled Row Reduction, Pipelined circuit execution</td>
</tr>
<tr>
<td>2012</td>
<td>Malicious</td>
<td>5.9 billion</td>
<td>AES RSA Signing Dot Product</td>
<td>Hardware optimizations, Random seed checking, Pipelining optimizations for above</td>
</tr>
</tbody>
</table>
Outline

1. Context
2. Security definitions
3. Oblivious transfer
4. Yao’s original protocol
5. Security improvements
6. Performance improvements
7. Implementations

8. Conclusion
8. Conclusion

• Multi-party extensions for Yao
• Performance optimizing OT protocols
• Gateway to other areas
• much, much, much, much, much more…
Mission Accomplished

Any questions?