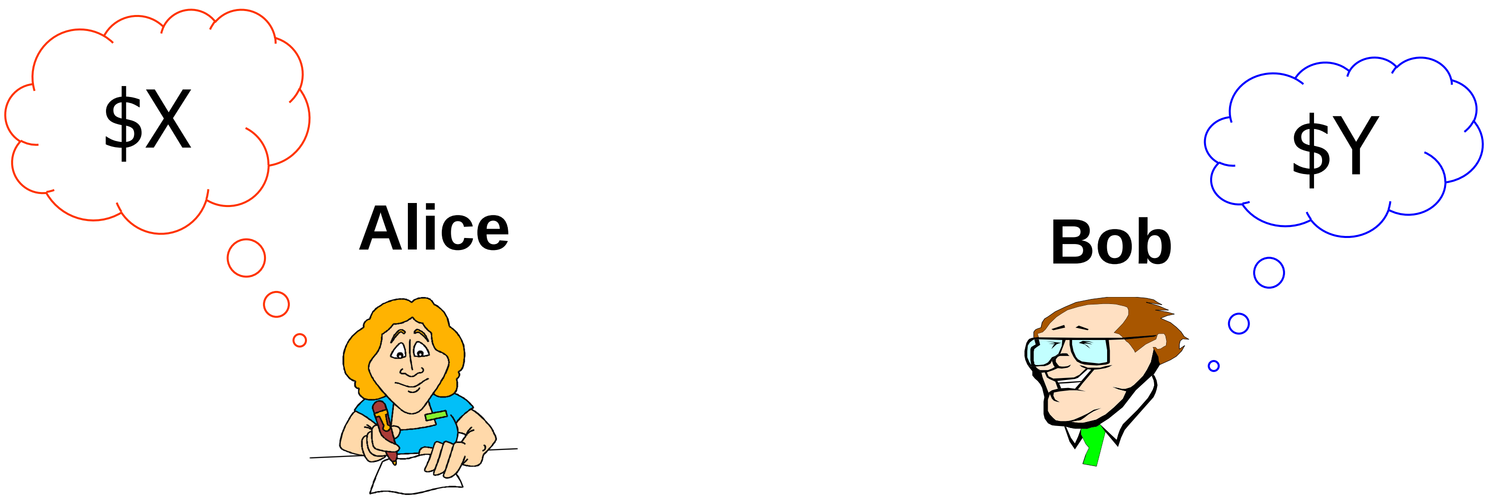
Implementing Secure Computation

Benny Pinkas, University of Haifa

Main theme

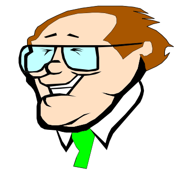
- Most MPC protocols were only designed to show feasibility.
- Implementations can give valuable insight
  - Identify bottlenecks and motivate researchers to focus on high-impact issues.
  - The area is full with opportunities for theory based observations that lead for optimizations.
- Quantitative improvements do add up.
  - Result in a qualitative improvement, which can bring secure computation to the masses.

A canonical example: The millionaires’ problem



- Want to find out if X > Y
- But leak no other information! (even to each other)
- Standard crypto tools (encryption) do not help in this case!

Secure two-party computation - definition



Input: x y

Output: F(x,y) *and nothing else*

As if… x y

Trusted third party

F(x,y) F(x,y)

Exact definitions based on this concept

Feasibility results in secure computation

- Any function can be computed securely [Yao,GMW]
- Two-party computation: Yao's seminal work
- *Multi-party*: many generic protocols
- Functions are not represented as programs, but rather as
  - Boolean circuits
  - Arithmetic circuits (+,* gates)
  - Other models (e.g., Damgard-Ishai)

Feasibility results in secure computation

- Any function can be computed securely [Yao,GMW]
- Two-party computation: Yao's seminal work
- *Multi-party*: many generic protocols
- Functions are not represented as programs, but rather as
  - Boolean circuits ☹
  - Arithmetic circuits (+,* gates) ☹☹
  - Other models (e.g., Damgard-Ishai) ☹?

Secure computation is not widely used

- Why isn't secure computation widely used? (compared to linear programming or data compression)
- Perhaps there is no real demand for this technology
- Real-world secure computation was not considered "practical"
- Therefore
  - Most results were only stated as mathematical theorems.
  - One had to read the relevant papers and implement them from scratch.
- Therefore
  - Secure computation is/was inaccessible to non-experts.
  - Implementation issues have not been addressed.

There is a long road from a feasibility result to a working system

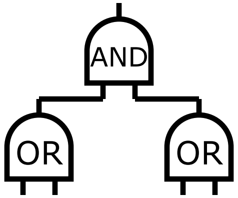
- The results are hard to understand
  - The techniques are quite complicated

- Feasibility results are hard to use
  - Focus on asymptotic results (e.g., O(1) is better than O(log n), even if this only holds for $n > 10^{12}$).
  - Constants don't matter.
  - Issues which are crucial for performance were not thoroughly investigated.
  - User interface can make or break a system.

Protocols

- We consider generic protocols rather than specific protocols for specific problems

- The basic technique of generic protocols:
  - Any function can be represented as a Boolean circuit or an algebraic circuit
  - Show how each gate can be securely evaluated
  - Applying this to layer after layer of the circuit, the entire function can be computed (without revealing any intermediate result)



<u>Background:</u> Fairplay [Malkhi,Nisan,Pinkas,Sella ’04]

Background: Fairplay
[Malkhi,Nisan,Pinkas,Sella ’04]

- The first (and only) generic system for secure two-party computation, implementing Yao’s protocol.

Background: Fairplay [Malkhi,Nisan,Pinkas,Sella '04]

- The first (and only) generic system for secure two-party computation, implementing Yao's protocol.
- Based on the compilation paradigm:
  - Users write programs in a high-level programming language (SFDL – Secure Function Definition Lang).

SFDL Example

```
program Millionaires {
   type int = Int<20>; // 20-bit integer
   type AliceInput = int;
   type BobInput = int;
   type AliceOutput = Boolean;
   type BobOutput = Boolean;
   type Output = struct {AliceOutput alice, BobOutput bob};
   type Input = struct {AliceInput alice, BobInput bob};

   function Output millionaires(Input input) {
      output.alice = input.alice > input.bob;
      output.bob =  input.bob > input.alice;
   }
}
```

SFDL Example

```
program Millionaires {
  type int = Int<20>; // 20-bit integer
  type AliceInput = int;
  type BobInput = int;
  type AliceOutput = Boolean;
  type BobOutput = Boolean;
  type Output = struct {AliceOutput alice, BobOutput bob};
  type Input = struct {AliceInput alice, BobInput bob};

  function Output millionaires(Input input) {
    output.alice = input.alice > input.bob;
    output.bob =  input.bob > input.alice;
  }
}
```

SFDL Example

```
program Millionaires {
    type int = Int<20>; // 20-bit integer
    type AliceInput = int;
    type BobInput = int;
    type AliceOutput = Boolean;
    type BobOutput = Boolean;
    type Output = struct {AliceOutput alice, BobOutput bob};
    type Input = struct {AliceInput alice, BobInput bob};

    function Output millionaires(Input input) {
        output.alice = input.alice > input.bob;
        output.bob =  input.bob > input.alice;
    }
}
```

Background: Fairplay [Malkhi,Nisan,Pinkas,Sella ’04]

- The use of a high-level programming language was a major innovation
  - Much easier than designing a circuit

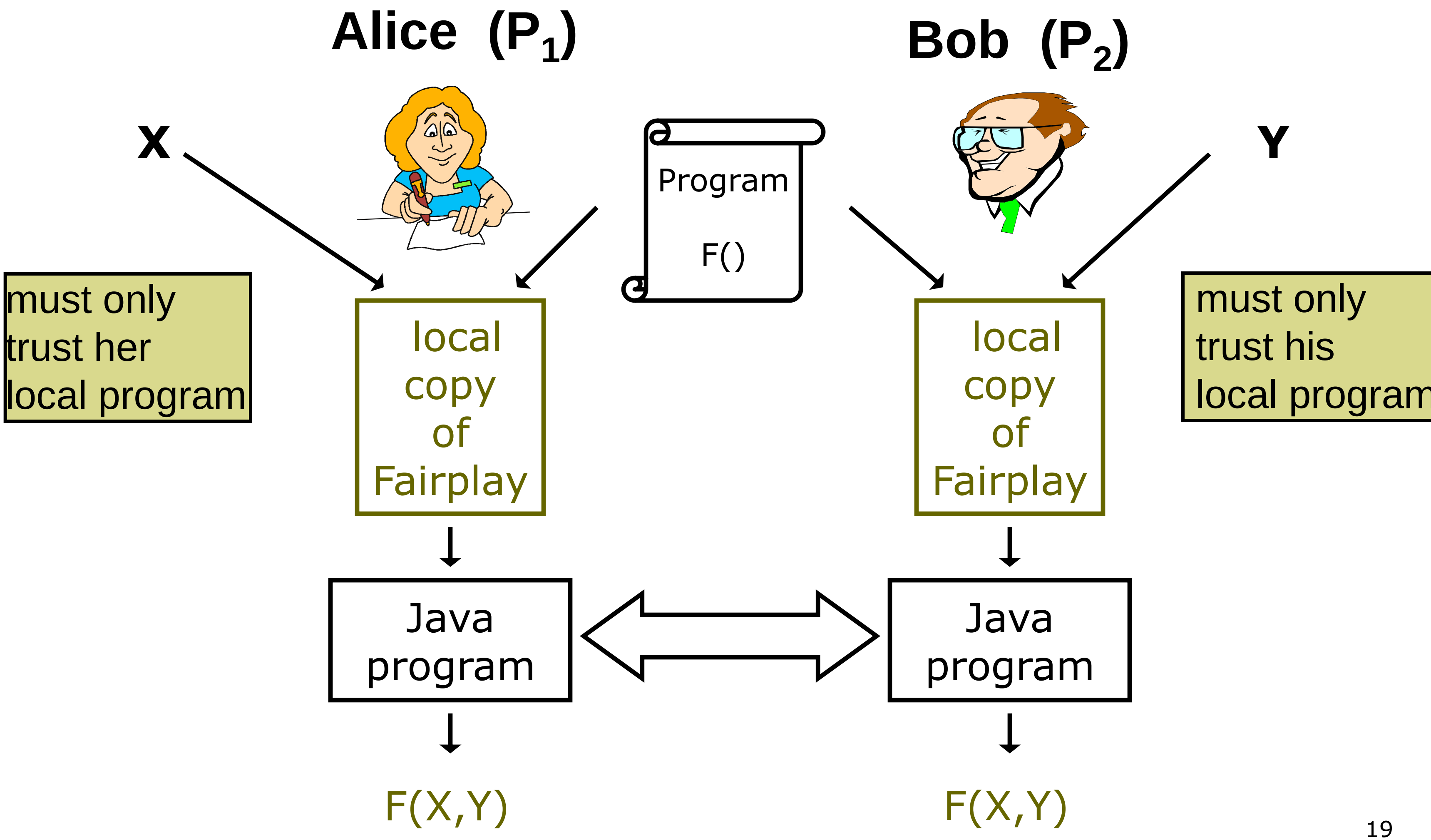
Background: Fairplay
[Malkhi,Nisan,Pinkas,Sella ’04]

- The first generic system for secure two-party computation, implementing Yao’s protocol.
- Based on the compilation paradigm:
  - Users write programs in a high-level programming language (SFDL – Secure Function Definition Lang).
  - Programs are translated by the system to a Boolean circuit, described in SHDL (Simple Hardware Definition Lang).

Background: Fairplay [Malkhi,Nisan,Pinkas,Sella '04]

- The first generic system for secure two-party computation, implementing Yao's protocol.
- Based on the compilation paradigm:
  - Users write programs in a high-level programming language (SFDL – Secure Function Definition Lang).
  - Programs are translated by the system to a Boolean circuit, described in SHDL (Simple Hardware Definition Lang).
  - The SHDL circuit is translated to Java programs implementing Yao's protocol.
  - The tool can be downloaded http://www.fairplayproject.net

The setting



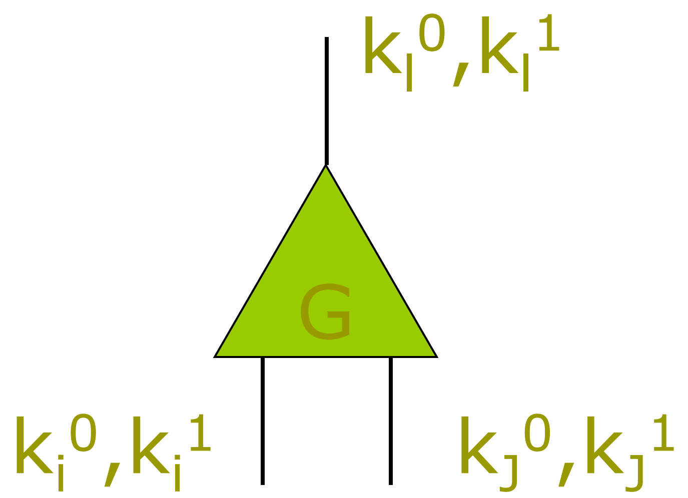
Alice (P1)
Bob (P2)
X
Y
Program
F()
must only trust her local program
local copy of Fairplay
local copy of Fairplay
must only trust his local program
Java program
Java program
F(X,Y)
F(X,Y)

Background:

Yao’s protocol

Secure two-party computation of general functions [Yao82,86]

- P_1 (aka Bob) constructs a binary circuit computing F, and then garbles it.
- Garbled values:



k_i^0 = 0 on wire i
k_i^1 = 1 on wire i

(P_2 will learn one string per wire, but not which bit it corresponds to.)

Gate tables

- P_1 defines garbled values for every wire.
- For every gate, every combination of garbled input values is used as a key for encrypting the corresponding output
  - Assume G=AND. P_1 constructs a table:
    - Keys k_i^0, k_J^0 encrypt key k_l^0
    - Keys k_i^0, k_J^1 encrypt key k_l^0
    - …Keys k_i^1, k_J^1 encrypt key k_l^1
- Result: given k_i^x, k_J^y, one can compute $k_l^{G(x,y)}$ and nothing else.

Gate tables

- P_1 defines garbled values for every wire.
- For every gate, every combination of garbled input values is used as a key for encrypting the corresponding output
  - Assume G=AND. P_1 constructs a table:
    - Encryption of k_l^0 using keys k_i^0,k_J^0
    - Encryption of k_l^0 using keys k_i^0,k_J^1
    - … Encryption of k_l^1 using keys k_i^1,k_J^1
- Result: given k_i^x,k_J^y, one can compute $k_l^{G(x,y)}$ and nothing else.

The Protocol (semi-honest case)

- P_1 sends to P_2
  - Tables encoding each circuit gate.
  - Garbled values (k's) of P_1's input values.
- For every wire i of P_2's input:
  - The parties run an oblivious transfer (OT) protocol

The Protocol (semi-honest case)

- P_1 sends to P_2
  - Tables encoding each circuit gate.
  - Garbled values (k's) of P_1's input values.
- For every wire i of P_2's input:
  - The parties run an oblivious transfer (OT) protocol

- Oblivious transfer:
  - P_2 has an input bit b
  - P_1 has two inputs X^0, X^1
  - P_2 learns X^b
  - P_1 learns nothing

implemented using public-key crypto

The Protocol (semi-honest case)

- P_1 sends to P_2
  - Tables encoding each circuit gate.
  - Garbled values (k's) of P_1's input values.
- For every wire i of P_2's input:
  - The parties run an oblivious transfer (OT) protocol, where
  - P_2's input is her input bit (*b*).
  - P_1's input is k_i^0, k_i^1
  - P_2 learns k_i^b

The Protocol (semi-honest case)

- P_1 sends to P_2
  - Tables encoding each circuit gate.
  - Garbled values (k's) of P_1's input values.
- For every wire i of P_2's input:
  - The parties run an oblivious transfer (OT) protocol, where
  - P_2's input is her input bit (b).
  - P_1's input is k_i^0, k_i^1
  - P_2 learns k_i^b
- Afterwards P_2 can compute the circuit by herself.
- Efficient for medium size circuits
- There is a full proof of security (after modifications) against *semi-honest* adversaries [LP06]

Fairplay – Implementation and Results

- Implementation:
  - Written in Java
  - Implements Yao's protocol
  - Crypto using the Java BigInteger libraries
  - El Gamal based OT

- Solving the billionaires problem (30 bit ints)
  - OTs accounted for 90% of running time on a LAN
  - For 50% of running time on a WAN
  - OT is the only public-key operation
  - Conjecture: OT is the bottleneck

Two-party Computation Secure against <u>Malicious</u> Adversaries

Yehuda Lindell
Benny Pinkas

Eurocrypt 2007

Potential adversarial behavior

- Possible adversarial behavior
  - Semi-honest: adversary follows the directions of the protocol, but tries to learn about the other side's inputs.
  - Malicious: adversary can behave arbitrarily.
- Ensuring security against malicious adversaries is much harder than against semi-honest adversaries.
- The original Fairplay system was only secure against semi-honest adversaries.

Approaches for obtaining security against malicious adversaries

- In the protocol, one party (P_1) constructs a garbled version of the circuit, and the other party (P_2) then computes this circuit.

- How can P_2 verify that the garbled version of the circuit is constructed correctly?
  - P_1 can be required to prove in zero-knowledge that the circuit is correct. This is in general not very efficient. ☹

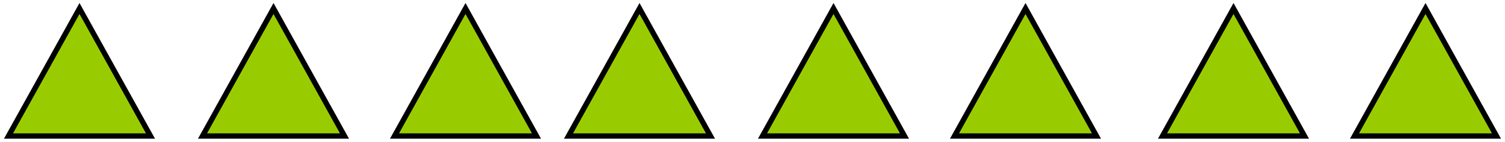
Approaches for obtaining security against malicious adversaries

- In the protocol, one party (P_1) constructs a garbled version of the circuit, and the other party (P_2) then computes this circuit.

- How can P_2 verify that the garbled version of the circuit is constructed correctly?
  - P_1 can be required to prove in zero-knowledge that the circuit is correct. This is in general not very efficient. ☹
- LP07 show an alternative and more efficient method for verifying the circuits.

Malicious Behavior and Cut-and-Choose

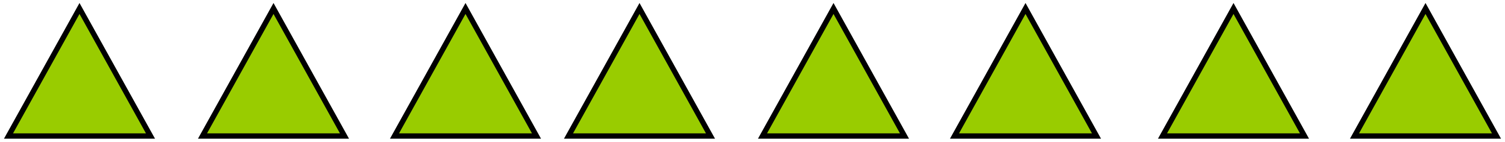
- Proving circuit is correct using “cut-and-choose”:
- P_1 constructs and commits to s circuits
  - Committed circuits are hidden from P_2, but cannot be changed anymore by P_1.



All circuits compute F, but each circuit is generated by an independent cryptographic encoding.

Cut-and-Choose: first attempt

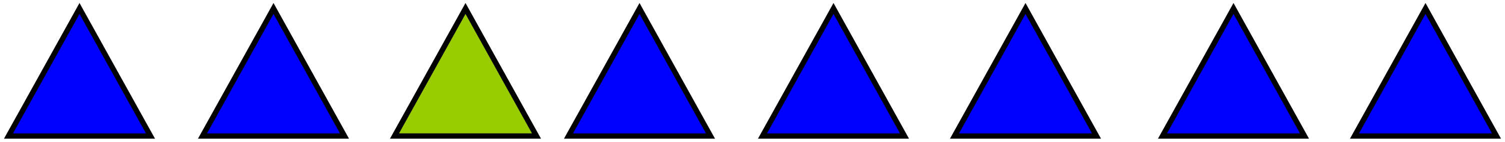
- Proving circuit is correct using "cut-and-choose":
- P_1 constructs and commits to s circuits
  - Committed circuits are hidden from P_2, but cannot be changed anymore by P_1.



- P_2 asks P_1 to open s-1 circuits, which P_2 then checks.

Cut-and-Choose: first attempt

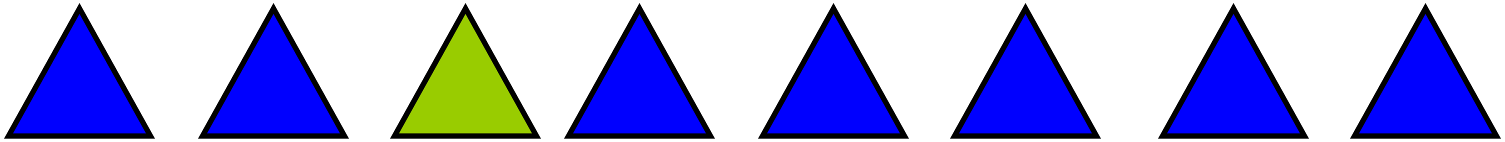
- Proving circuit is correct using "cut-and-choose":
- P_1 constructs and commits to s circuits
  - Committed circuits are hidden from P_2, but cannot be changed anymore by P_1.



- P_2 asks P_1 to open s-1 circuits, which P_2 then checks.

Cut-and-Choose: first attempt

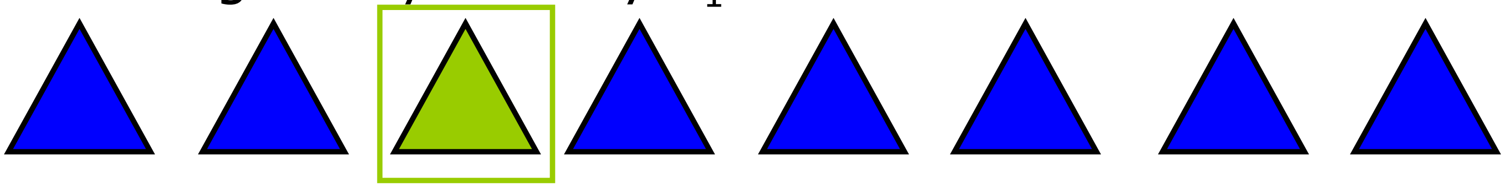
- Proving circuit is correct using "cut-and-choose":
- P_1 constructs and commits to s circuits
  - Committed circuits are hidden from P_2, but cannot be changed anymore by P_1.



- P_2 asks P_1 to open s-1 circuits, which P_2 then checks. If any of these is bad, P_2 aborts.

Cut-and-Choose: first attempt

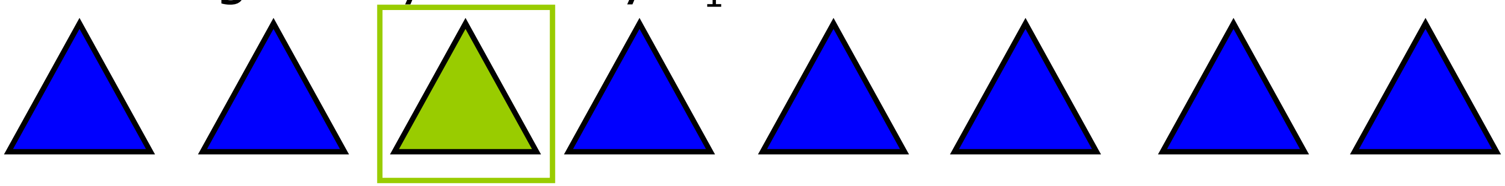
- Proving circuit is correct using “cut-and-choose”:
- P_1 constructs and commits to s circuits
  - Committed circuits are hidden from P_2, but cannot be changed anymore by P_1.



- P_2 asks P_1 to open s-1 circuits, which P_2 then checks. If any of these is bad, P_2 aborts.
- The parties then evaluate the remaining circuit

Cut-and-Choose: first attempt

- Proving circuit is correct using "cut-and-choose":
- P_1 constructs and commits to s circuits
  - Committed circuits are hidden from P_2, but cannot be changed anymore by P_1.

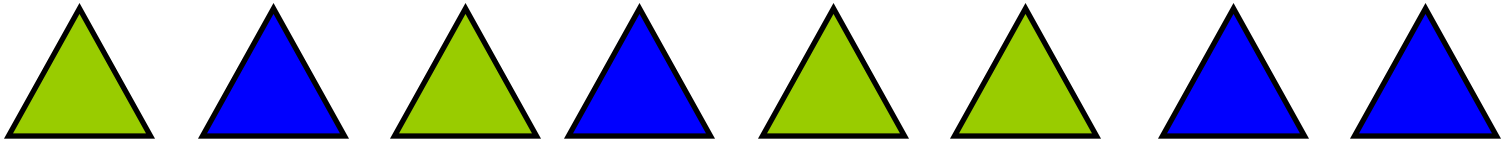


- P_2 asks P_1 to open s-1 circuits, which P_2 then checks. If any of these is bad, P_2 aborts.
- The parties then evaluate the remaining circuit
- A corrupt P_1 succeeds with prob. 1/s

Improving security of cut-and-choose

Improving security of cut-and-choose

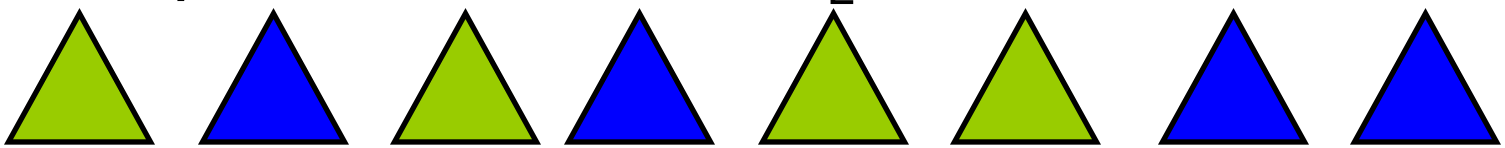
- P_2 asks P_1 to open a random subset of s/2 circuits, which P_2 checks.
- If any of them is bad, P_2 aborts.



- The protocol continues with the remaining s/2 circuits. P_2 outputs the value outputted by the majority of these circuits.

Improving security of cut-and-choose

- P_2 asks P_1 to open a random subset of s/2 circuits, which P_2 checks.
- If any of them is bad, P_2 aborts.



- The protocol continues with the remaining s/2 circuits. P_2 outputs the value outputted by the majority of these circuits.
- A corrupt P_1 succeeds with probability $2^{-s/4}$
  - In order to cheat, P_1 needs to corrupt a majority of the s/2 circuits, and that none of them is checked.

New problems: Inconsistent outputs

- What should P_2 do if *not* all s/2 evaluated circuits yield the same output?

New problems: Inconsistent outputs

- What should P_2 do if *not* all s/2 evaluated circuits yield the same output?
  - P_1 definitely cheated, but should P_2 abort?
  - Aborting reveals information to P_1.
  - For example
    - P_1 constructs *s-1* circuits computing F, and a single circuit computing F if and only if P_2's input is 0.
    - With probability ½, that circuit is not checked in the first stage. Then P_2 finishes the computation iff its input is 0.
- P_2 must therefore always output the majority value.

New problems: Inconsistent inputs

- P_1 might provide different inputs (of P_1) to different circuits among the s/2 evaluated circuits.

New problems: Inconsistent inputs

- P_1 might provide different inputs (of P_1) to different circuits among the s/2 evaluated circuits.

- Does this matter? Yes it does.
  - Cut-and-choose checks the circuits but not P_1's inputs.
  - Smart input choices by P_1 provide information on Y.

- Solution: must verify consistency of P_1's inputs (this step proved to be quite tricky).

Lindell-Pinkas 07

- The first truly practical two-party protocol secure against malicious adversaries.
- The protocol is proven to be secure according to the strongest security definition (Ideal/real simulation paradigm)
- The resulting protocol is rather efficient
  - Computational overhead as in semi-honest case ☺
  - Larger communication overhead ☹
- Competing approaches
  - Jarecki-Shmatikov (efficient ZK proof per gate)
  - Nielsen – Orlandi (LEGO)

Implementing secure computation

Lindell – Pinkas – Smart ’08

Pinkas – Smart – Schneider – Williams

Contributions

- Implemented the LP ’07 protocol
  - This was not a simple task.
  - Implemented a version based on random oracles, and a version in the standard model.
  - Optimized the circuit construction (note that for 2^{-40} security must send s=160 copies of it).

- Spoiler: obtained some interesting results regarding
  - Standard model vs. random oracle implementation.
  - Oblivious transfer as the bottleneck.

Optimizations

- Automatically optimized the circuit
- Example: 16-bit comparison.
  - Original circuit consisted of 61 2-to-1 gates.
  - Optimized circuit has 15 3-to-1 gates and one 2-to-1 gate (essentially computing X-Y and checking the sign).

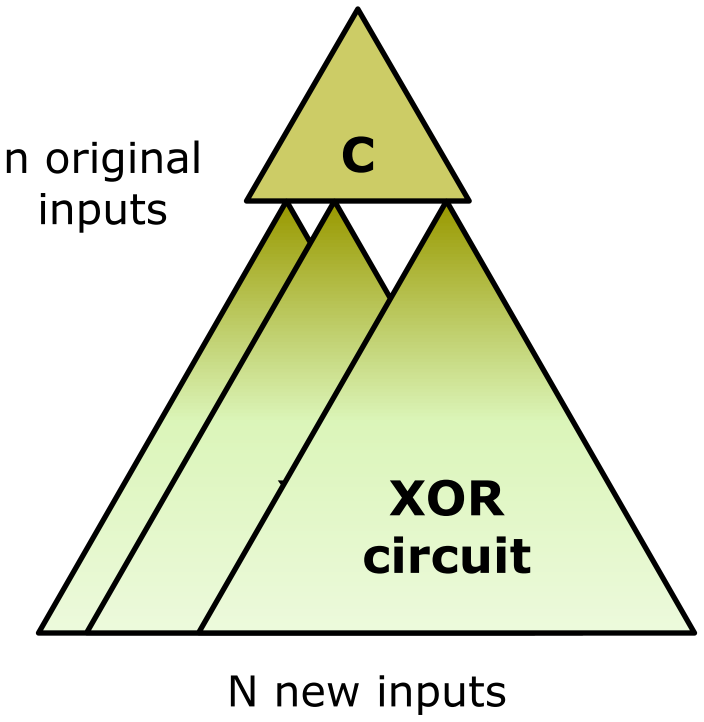
- Encountered interesting questions
  - Used a modified protocol which computes XOR gates for free [KS08].
  - Subsequent work built tools to modify circuit in order to maximize the number of XOR gates [KSS09].
  - Input coding…

Protecting P_2's inputs

- To protect P_2's input we must (for reasons not described here):
  - Replace P_2's n inputs with $N=\max(4n,8s)$ new inputs. This reduces the error probability to 2^{-s}.
  - Set each of the n original input values to be the xor of a random set of the new input values.

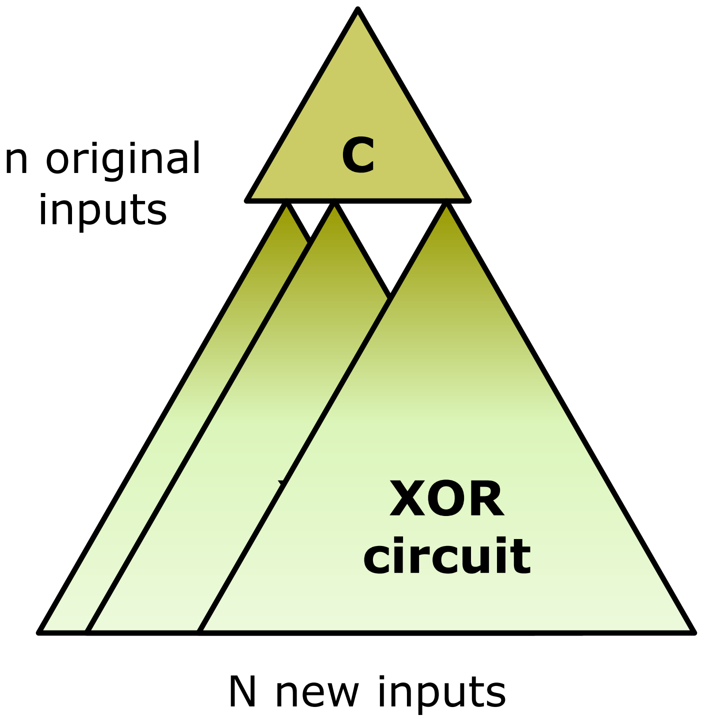
Protecting P_2's inputs

- To protect P_2's input we must (for reasons not described here):
  - Replace P_2's n inputs with $N=\max(4n,8s)$ new inputs. This reduces the error probability to 2^{-s}.
  - Set each of the n original input values to be the xor of a random set of the new input values.



Protecting P_2's inputs

- To protect P_2's input we must (for reasons not described here):
  - Replace P_2's n inputs with $N=\max(4n,8s)$ new inputs. This reduces the error probability to 2^{-s}.
  - Set each of the n original input values to be the xor of a random set of the new input values.
- We set s=40 for 2^{-40} security.



Protecting P_2's inputs

- To protect P_2's input we must (for reasons not described here):
  - Replace P_2's n inputs with N=max(4n,8s) new inputs. This reduces the error probability to 2^{-s}.
  - Set each of the n original input values to be the xor of a random set of the new input values.
- We set s=40 for 2^{-40} security. Therefore

  This might be larger than the original circuit!
  For n=16 input bits get 2560 additional gates!

n	<80	>80
new input bits (N)	320	4n
each original input is xor of	160	2n
# of new xor gates	160n	$2n^2$

Protecting P_2's inputs

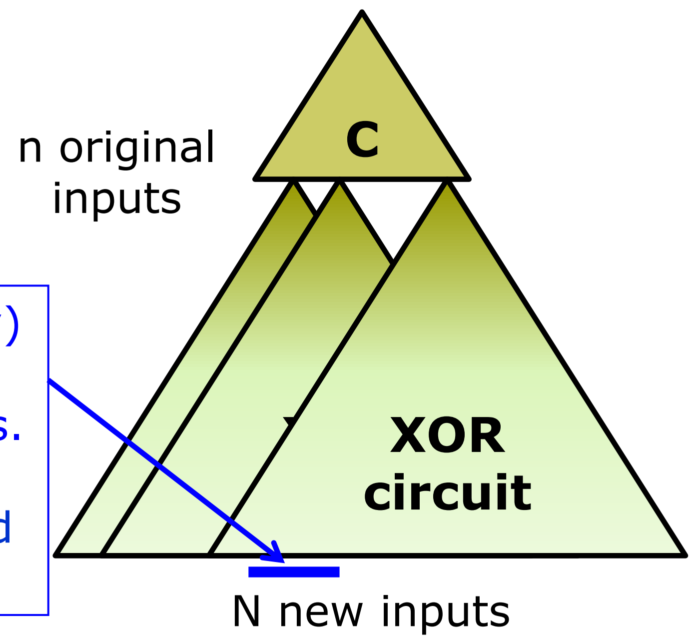
- To protect P_2's input we must (for reasons not described here):
  - Replace P_2's n inputs with N=max(4n,8s) new inputs. This reduces the error probability to 2^{-s}.
  - Set each of the n original input values to be the xor of a random set of the new input values.
- We set s=40 for 2^{-40} security. Therefore

n	<80	>80
new input bits (N)	320	4n
each original input is xor of	160	2n
# of new xor gates	160n	$2n^2$

Luckily, KS08 show how to compute XOR gates for free

Reducing the size of the XOR circuits

- P_2's n input bits must be expanded to N new input bits. Currently use N=max(4n,320).
- It is possible to reduce the size of the XOR circuit (by 60%) by reusing as many gates as possible.



There are likely to be (many) XOR expressions which are used in multiple XOR circuits.

Similar method to structured Gaussian elimination.

Reducing the size of the XOR circuits

- P_2's n input bits must be expanded to N new input bits. Currently use N=max(4n,320).
- It is possible to reduce the size of the XOR circuit (by 60%) by reusing as many gates as possible.
- Actually need a binary [N,n,40] linear code
  - For 2^{-40} security we always need a distance of s=40
  - Would like N/n to be small. Namely the information rate n/N should be large even for **small blocks** (even for, e.g., n=30).
  - Explicit constructions? http://www.codetable.de
  - Randomized constructions?
    - Can achieve N=3n for n=100, N=2n for n=300, etc.

Implementation details

- Implemented in C++
- Elliptic curve routines implemented in assembler
  - Used the standard curve P256 to match AES-128 security level
  - Multiplication of a fixed generator in 1.2 msec

Results for 16bit comparison Wall time, ROM vs. Standard Model

- Stages
- 1: P_1 creating garbled circuits
- 2: OT stage
- 3: transferring the circuits
- 5-6:send decomits
- 7: P_2 checks half the circuits
- 8: P_2 evaluates remaining circuits

Time	Step 1	2	3	4	5	6	7	8	Total
P_1, $s_1 = 160$, $s_2 = 40$									
ROM	74	20	24	0	7	10	0	0	135
Standard	84	20	24	0	7	7	0	0	142
P_2, $s_1 = 160$, $s_2 = 40$									
ROM	74	20	24	0	8	9	35	1	171
Standard	84	20	24	0	7	7	40	2	184
P_1, $s_1 = 240$, $s_2 = 60$									
ROM	159	34	51	0	19	13	0	0	276
Standard	181	35	45	0	18	12	0	0	291
P_2, $s_1 = 240$, $s_2 = 60$									
ROM	159	34	51	0	19	13	78	3	358
Standard	181	35	45	0	18	12	87	5	362

•OT is not the bottleneck.
•ROM time ≈ Standard model time

Looked for an interesting application…

Secure computation of AES
P-Schneider-Smart-Williams

- AES is by design a complex function.
  - Alice has K. Bob learns $AES_k(X)$.
  - Optimized circuit has ~34000 gates.
- Best run times (including circuit construction):
  - Semi-honest: 8 sec. Covert: 100 sec.
  - Malicious: 1150 sec

- This is essentially an OPRF - oblivious pseudo-random function.
  - Implementing this as a circuit in Yao's protocol was suggested before but considered impractical.
  - Has multiple applications [FIPR04, HL08, LLM05, RAFCR09].

Observations

- Most optimizations were based on understanding the protocol and its proof of security
  - XOR for free
  - Coding
  - Used OT protocols which amortize the cost of ZK proofs
  - There is active work on optimizing the current bottlenecks
- Some optimizations are generic
  - Circuit optimization (and the fact we have a compiler)
  - EC based public key crypto
- Surprising observations
  - OT is not the major bottleneck
    - Very efficient implementation of OT.
    - Large circuit; many copies sent and processed.
  - No performance penalty for using standard model compared to random oracle model.

FairplayMP
A System for Secure Multi-Party Computation

Assaf Ben-David

Noam Nisan

Benny Pinkas

ACM CCS 2008

Which MPC protocol to use?

- Wanted to build a full fledged system for secure multi-party computation
- Our high level requirements:
  - We suspected that the number of communication rounds is a major bottleneck
  - Therefore needed a protocol whose # of rounds is constant
  - Wanted to use a Boolean circuit representation of the function (for two good reasons)
- There are many protocols for SMP
  - The BGW protocol efficiently computes arithmetic circuits
  - The BMR (Beaver-Micali-Rogaway) protocol is unique in satisfying all our requirements

Modifying the setting

Theoretical papers assume n symmetric players

- Each player:
  - Has an input
  - Participates in the computation
  - Learns the output
- There is interaction between all players ☹
- Protocol secure if not too many players collude ☹

The model is generalized. Players can be separated into three types.

- Input players (IP)
- Computation players (CP):
  - Emulate the trusted party
  - Interact with each other
  - Protocol is secure if less than half of CPs are corrupt
- Result players (RP) learn the output
- A participant can have several of these roles

The compilation paradigm

- Programs are written in SFDL 2.0
  - An improved version of Fairplay's SFDL, amended to support inputs and outputs from/to multiple parties.

```
program SecondPriceAuction {
   const nBidders = 4;
   type Bid = Int<4>; // enough bits to represent a small bid.
   type WinningBidder = Int<3>; // enough bits to represent a winner
   type SellerOutput = struct{WinningBidder winner, Bid winningPrice};
   type Seller = struct{SellerOutput output}; // Seller has no input
   type BidderOutput = struct{Boolean win, Bid winningPrice};
   type Bidder = struct{Bid input, BidderOutput output};
```

SFDL example: The main function

```
function void main(Seller seller, Bidder[nBidders] bidder) {
   var Bid high = bidder[0].input, Bid second = 0;
   var WinningBidder winner = 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;
   }}}
```

The BMR protocol

- Two random seeds (garbled values) are used for every wire of the Boolean circuit.
- Each seed S_i is a concatenation of n k-bit seeds $s_i^1 \circ s_i^2 \circ \ldots \circ s_i^n$ generated by each of the CPs.
- For each wire, the CPs run a joint coin flip to set a secretly shared random bit λ_w.
- Iff $\lambda_w=0$ then S_0 represents 0, S_1 represents 1. Otherwise their roles are flipped.

The BMR protocol

- The parties compute a 4x1 table for every gate
  - Like in Yao's two-party protocol
  - A table entry for an OR gate is of the form
    - If $\lambda_a \vee \lambda_b = \lambda_c$ then
      - $A_g = g_a^1 \oplus \cdots \oplus g_a^n \oplus g_b^1 \oplus \cdots \oplus g_b^n \oplus s_c^1 \circ \cdots \circ s_c^n \circ 0$
  - Unlike Yao, here the table must be computed by a secure protocol run between the CPs.
  - The BMR paper suggests using any secure protocol to implement this step.
- Finally, given the tables, and seeds of the input values, it is easy to compute the circuit output.

Improvements to the BMR construction

- Computing table entries is the major bottleneck
  - If $\lambda_a \vee \lambda_b = \lambda_c$ then
    - $A_g = g_a^1 \oplus \cdots \oplus g_a^n \oplus g_b^1 \oplus \cdots \oplus g_b^n \oplus s_c^1 \circ \ldots \circ s_c^n \circ 0$
- Change to
  - If $\lambda_a \vee \lambda_b = \lambda_c$ then
    - $A_g = g_a^1 + \cdots + g_a^n + g_b^1 + \cdots + g_b^n + s_c^1 \circ \ldots \circ s_c^n \circ 0$ (addition in a sufficiently large finite field)

How can this step be implemented?

- We replaced
  - If $\lambda_a \vee \lambda_b = \lambda_c$ then
    - $A_g = g_a^1 \oplus \cdots \oplus g_a^n \oplus g_b^1 \oplus \cdots \oplus g_b^n \oplus s_c^1 \circ \cdots \circ s_c^n \circ 0$

  by

    - $A_g = g_a^1 + \cdots + g_a^n + g_b^1 + \cdots + g_b^n + s_c^1 \circ \cdots \circ s_c^n \circ 0$
- Can now use the BGW protocol for this step
- To compute "$g_a^1 + \cdots + g_a^n + g_b^1 + \cdots + g_b^n$" each party i sends shares of g_a^i; sums the shares it receives.
- To compute "$s_c^1 \circ \cdots \circ s_c^n$" party i shifts s_c^i (by $i \cdot k$ bits) and sends shares; sums shares it receives.
- To compute "If $\lambda_a \vee \lambda_b = \lambda_c$" use multiplication to compute $\lambda_a \lambda_b$; use it to get 0/1 result for "$\lambda_a \vee \lambda_b = \lambda_c$"; multiply by "$g_a^1 + \cdots + g_b^n + s_c^1 \circ \cdots \circ 0$".

The improvement to BMR

- Change to
  - If $\lambda_a \vee \lambda_b = \lambda_c$ then $A_g = g_a^1 + \cdots + g_a^n + g_b^1 + \cdots + g_b^n + s_c^1 \circ \ldots \circ s_c^n \circ 0$
- Can now run the BGW protocol.
  - Use 3 multiplications per table entry
- A circuit for the same task (computing one entry in a single gate) has about $\sim 2n^2k$ gates.
  - $n=5$, $k=128$ ⇨ ~ 6400 gates.
- The coin flipping can also be implemented using BGW [DFKNT 05]

The implemented protocol

- FairplayMP is implemented in Java
  - Modular and readable code

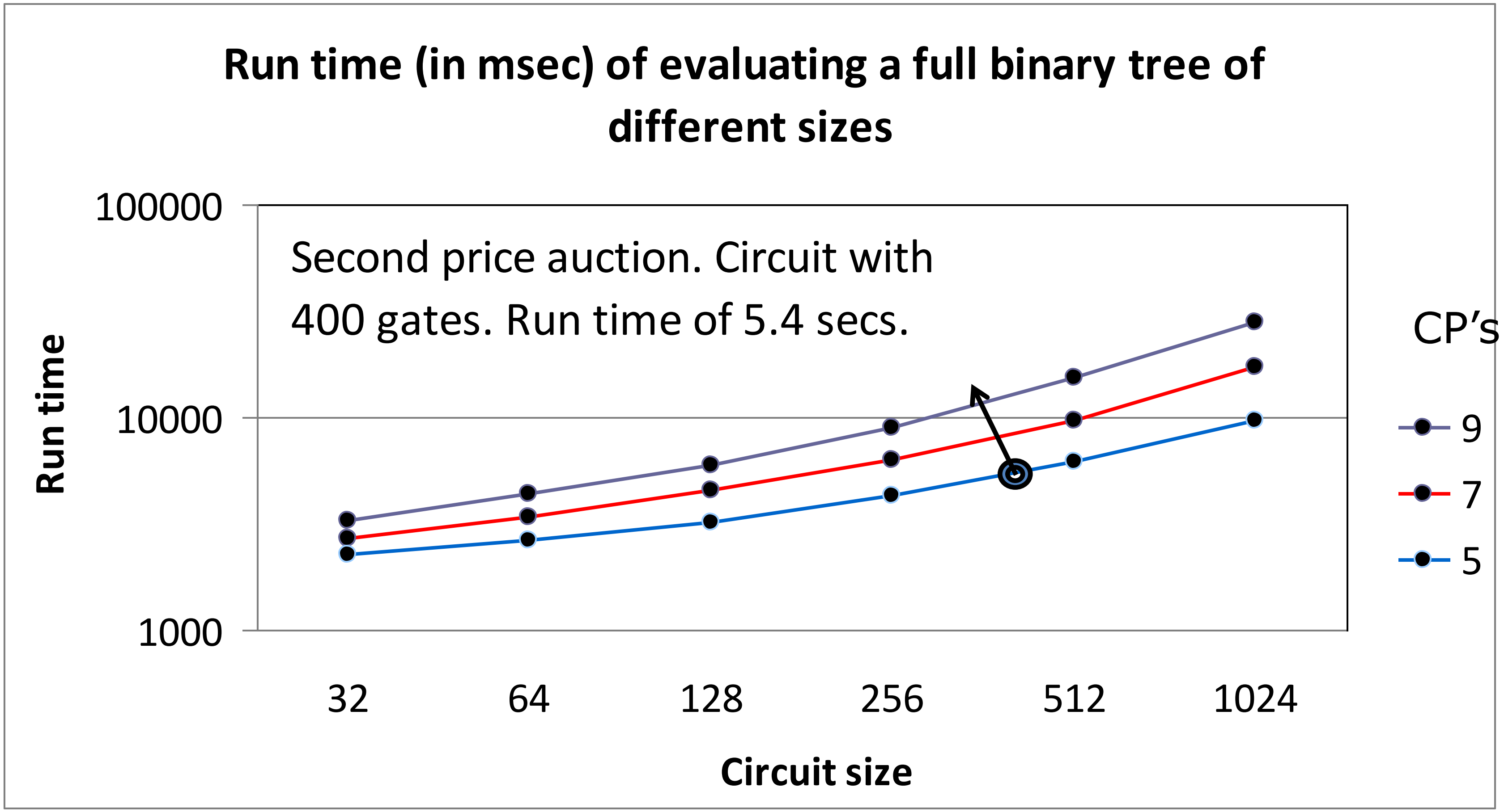
- Five packages (~2000 code lines):
  - circuit – An interface that allows to use different representations of circuits.
  - communication - Basic Client/Server, msg.
  - config – Allows simple configuration via code.
  - players – Implementation of the protocol steps for each of the players (IP, CP, RP).
  - utils – Implementation of BGW and PRG.

Data communication

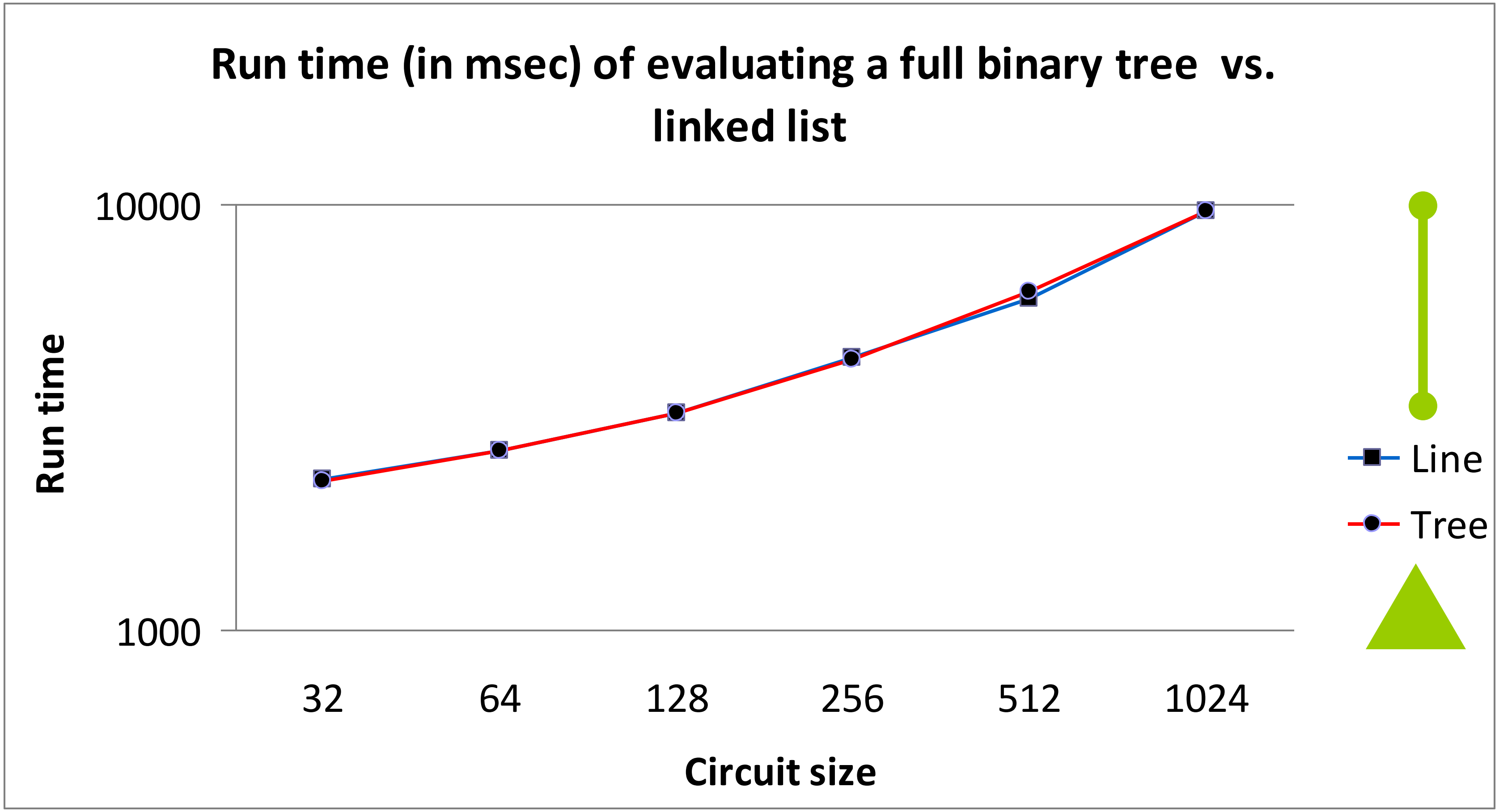
- As in the two-party case, inefficient data communication between the parties can cause major delays.
  - First versions of code handled communication inefficiently.
  - Item wrapping, opening ports, etc.

- Solutions:
  - Handle this very carefully
  - Use Google's `protocolbuffer`

Experiments
The effect of the circuit size



Experiments
The effect of the circuit depth



Conclusions

- FairplayMP
  - First generic system for secure MPC.
  - Many existing MPC protocols, but there are “hidden issues” which make it hard to implement them.
  - Needed to “massage” the BMR protocol.

- Feasibility of MPC systems
  - Semi-honest vs. malicious ☹
  - Random oracle vs. standard model ☺