CSCI1515 Capstone: Optimisations to Yao's Garbled Circuits

Maya Fleischer
Johannesburg, South Africa

1 PROJECT OVERVIEW

In this final project for CSCI 1515: Applied Cryptography, I worked with Lucy Gramley on implementing various optimisations for Yao's Garbed Circuits. Implementing Yaos' Garbled Circuits was a project completed by all students individually earlier in the course, but we chose to continue and expand on our work. We implemented 4 different optimisations, namely Point and Permute, Garbled Row Reduction, freeXOR, and half gates.

Yao's garbled circuits allow two parties to solve a boolean circuit without revealing the intermediate values or the other party's inputs. This protocol works with two parties, the garbler and the evaluator. The garbler parses the circuit, generates garbled gates (consisting of distinct ciphertexts) and labels corresponding to the garblers input, and encrypts the values (when necessary) before sending these off to the evaluator. The evaluator uses oblivious transfer to retrieve the gates and labels, evaluate them and decrypt (when necessary) and then send them back to the garbler where the garbler can then reveal the final output to the evaluator.

This protocol is important because it allows two parties to securely compute a function. For the Yaos project we implemented this without considering efficiency or optimizations. For our final project, we built on our initial construction of Yao's garbled circuits and extended this basic implementation and introduced four different optimizations that improve the efficiency and cut down on the computations required for the Yao's garbled circuit protocol. These are point and permute, garbled row reduction (from 4 cipher-texts to 3), freeXOR, and half-gates. We have set it up so that these four optimizations can work together and are compatible with one another (although we should note that half gates supersede garbled row reduction so the implementation is slightly different depending on if half gates are in use or not). We will describe our implementation and the efficiency benefits of these four optimizations below as well as discuss their compatibility with one another.

2 OPTIMISATIONS

In this project, we implemented 4 optimisations to Yao's Garbled Circuit. Each of these increased the computation and communication cost of the protocol. In large problems, this will make a significant different in the cost of running the protocol.
We will outline each optimisation including the difference in performance, the logic, and the steps of implementation.

Point and permute:

Explanation:
This first optimisation is the foundation for the other optimisations. It introduces the notion of a **select bit**. The select bit is the first bit of each wire label. For each of the wires, the zero label and one label will have complementary select bits.
These are used to coordinate between the garbler and the evaluator about which ciphertext to decrypt, without exposing any information about the true value of the wire.
Specifically, in this case, we order the ciphertexts based on the select bits of the inputs used to create the ciphertext. The evaluator and garbler 'agree' on an order (based on the select bits). The evaluator therefore does not have to try each of the ciphertexts and do a verification step. It can simply look at the select bits of the input wires, and select the appropriate ciphertext.

Performance:
This reduces the computation cost of the evaluator to 1|c|.

Implementation:

- *garbler*: Update the way that we generate the labels. Instead of randomly generating each value, we generated the values in pairs, ensuring that one had a 0 select bit and the other had a 1 select bit. We set these labels to be the zero and one labels for a certain wire. The 0 and 1 select bits were random and did not correspond to the zero and one labels.
  We also updated the label length from 16 to 32 bits.
  When sending the values to the evaluator we looked at the select bits of the two inputs. We ordered the ciphertexts based on the select bits of these inputs. The order (that the garbler and evaluator agreed upon) was 00, 01, 10, 11.
  We add the ciphertexts to the garbled gate in this order and do not shuffle them.
  This is so that the evaluator can identify which ciphertext to decrypt.
- *evaluator*: In the step where we evaluate the gates, we previously tried to decrypt each ciphertext and had a verification function that identified the correct output. Now, we use the select bits of the inputs to identify the appropriate ciphertext.
  For example, if input A had select bit 1 and input B had select bit 1, we would decrypt the 4th ciphertext at that garbled gate. This way we ensure the correct output and do not have to try the decryption multiple times.[YAK]

Garbled Row Reduction (GRR3):

Explanation:
This optimisation removes one of the ciphertexts, lowering the total from 4 to 3. Based on the ordering from above (based on the select bits and the order given), the 'top' (00) ciphertext is removed. The value of this ciphertext is known by both the garbler and the evaluator to be 0. The output wires of the gate being garbled need to be updated to correspond with the ciphertext of value 0. This step happens in the garbler, and the evaluator simply treats the 00 ciphertext as 0 and continues as normal. Because the ciphertext is 0, it does not need to be calculated or sent from the garbler to the

evaluator.

Performance: This optimisation reduces the computation and communication cost for the garbler from 4|c| to 3|c|.

Implementation:

- *garbler*: We remove the computation of the first ciphertext. However, we need to set the output wire values such that the ciphertext truly would have a value of 0. Because our encryption algorithm is the hash of the two inputs XORed with the output, this means we set the output wire value (for the first ciphertext) to be the hash of the two inputs. To explain this better, say the first cipher text is (A0, B1) and outputs to C0. We would need to update the value of C0 to be hash(A0, B1).
  We have to ensure that point and permute will still work, and so we also have to alter the other output wire value. We do so by setting the select bit of the other output wire to be the opposite of the select bit for the first. In the example given above this would entail looking at the select bit of C0, and setting the select bit of C1 to be complementary.
- *evaluator*: Previously, we indexed into the ciphertexts of the garbled gate based on the select bits of the inputs. For all combinations of select bits besides (0,0), we do the same - noting that we have to decrease the index by 1 (because of the missing first ciphertext). For the (0,0) case, we know that the output will be equal to the hash of the two inputs. We can set our output value to hash(lhs, rhs), or can decrypt as we do for the other ciphertexts, creating a ciphertext of value 0 to use in the computation. [YAK]

freeXOR:

Explanation
The freeXOR optimization allows for the computation of XOR gates entirely for free, hence the name. This is done by adjusting the relationship between the 0 and 1 wires and introducing a random label R. This allows us to compute W1 using W0 and R and where W0 is computed using the lhs and rhs wires. Introducing the random label allows us to reduece computation.

Performance:
Eliminated the computation cost. We went from 3|c| in garbler and 1|c| in evaluator (with just grr3 implemented) to free computation with freeXOR.

Implementation:

- *garbler*: We first adjusted our generate label to work with freeXOR and grr3. For freeXOR to work, the first bit of the random label R needs to be 1 so we set it so that if the label was being generated for R, the first bit would be 1 and otherwise, the first bit is 0. We then use the following construction:
  Initialize the random label (R).
  Populate the zeros and ones labels based on W1 = W0 XOR R.
  Then in our generate gates when we reach an XOR gate we:
  Set the zeros output (Z_zero) to be the zero label left hand wire XOR the zero label right hand wire.
  Set the ones output (Z_ones) to be Z_zero XOR R.

  With this construction, we do not need to add these gates to our garbled table meaning we will need to account for this when we later evaluate.
- *evaluator*: In evaluator, we simply XOR the lhs and rhs wires. This becomes the final output of the garbled wire. We also have to account for the fact that XOR gates were not added to our garbled table. This means that when we index into the garbled table with NOT or AND gates, we will need to subtract the number of freeXOR gates. [YAK]

Half Gates

Explanation:
This optimisation works only for the AND gates in the circuit. It works by breaking the circuit into two smaller AND gates (each called a half gate). There is a garbler half gates and an evaluator half gate. The idea is that if one of the inputs if known by either the garbler or the evaluator, then then the gate can be evaluated using only two ciphertexts. If this happens for each of the half gates, it will be a total of 4 ciphertexts. However, we use row reduction for each of the half gates, lowering the number of ciphertexts from 4 to 2.
Each AND gate is broken down into two smaller gates, such that the overall output is $c = (v_i \wedge b) \oplus (v_i \wedge (v_j \oplus b))$
Garbler half gate: $v_i \wedge b$
Evaluator half gate: $v_i \wedge (v_j \oplus b)$
As seen above, the output of the gate is found by XORing the two half gates. This happens on the garbler side (to set up the output wire values to be correct), and on the evaluator side (to evaluate the correct output).

Performance:
This reduces the computation and communication costs.

Implementation:
We referenced multiple papers, but our implementation was ultimately based on Two Halves Make a Whole. We followed the protocol outlined there closely.

- *garbler*: Here we generate the ciphertexts for both gates. Because we use row reduction, we set the "output" of the first half gate to be equal to the hash of A. This means we only have to send one ciphertext for each half gate. We generate both the garbler and evaluator gates, and send these ciphertexts to the evaluator. We also set the output wires for the overall AND gate to be the XOR of these two values.
- *evaluator*: In the evaluator, we select the appropriate operations based on the select bits of the inputs. Because we used point and permute, we know that the 0 bit inputs will always correspond with the first ciphertext (not present because of row reduction). We solve the evaluator and garbler half gates and we xor the result to get our final output. [ZRE15] [EKR]

3 VERIFICATION OF RESULTS

We had already implemented a working version of Yao's Garbled Circuits without any of these optimisations. We had 6 example circuits that we ran with our original version of the protocol.
After adding each optimisation, we tested the same circuits, ensuring that we got the same outputs. We tested different combinations of our optimisations and with any configuration, the output of the circuits was the same. We are confident that our optimisations work and that they follow the procedures as outlined in the papers we looked at.

4 LIBRARIES

We used the **CryptoPP** library, as well as the standard C++ library.

5 BIBLIOGRAPHY

[EKR] https://securecomputation.org/docs/pragmaticmpc.pdf

[YAK] https://web.mit.edu/sonka89/www/papers/2017ygc.pdf

[ZRE15] https://eprint.iacr.org/2014/756.pdf