Mobile Cryptographic Coprocessor for Privacy-Preserving Two-Party Computation

By
Gabriel Kulp

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented May 28, 2021
Commencement June 2021

AN ABSTRACT OF THE THESIS OF

Gabriel Kulp for the degree of Honors Baccalaureate of Science in Computer Science presented on May 28, 2021. Title:
Mobile Cryptographic Coprocessor for Privacy-Preserving Two-Party Computation

Abstract approved:

Mike Rosulek

Multi-party computation (MPC) is a field of study focused on devising cryptographic protocols that allow participants to learn the output of some function of their private inputs without trusting a third party to perform the computation. This is usually done at a large scale between data centers, with little emphasis on individuals' devices or mobile hardware. In this paper, I present a proof-of-concept implementation of two-party computation on a commodity smartphone paired with a low-power field-programmable gate array (FPGA). I compare the performance and power consumption of the system between a software-only setup and a setup with the FPGA coprocessor used for acceleration. I find a calculated 62× speed improvement assuming a saturated serial connection, and no significant difference in the smartphone's battery life.

Key Words: MPC, Garbled Circuits, FPGA, Cryptography, IceStorm

Corresponding e-mail address: kulpga@oregonstate.edu

Mobile Cryptographic Coprocessor for Privacy-Preserving Two-Party Computation

By
Gabriel Kulp

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented May 28, 2021
Commencement June 2021

Honors Baccalaureate of Science in Computer Science project of Gabriel Kulp presented on May 28, 2021.

APPROVED:

Mike Rosulek, Mentor, representing Computer Science

Vincent Immler, Committee Member, representing Computer Science

Rakesh Bobba, Committee Member, representing Computer Science

Toni Doolen, Dean, Oregon State University Honors College

Gabriel Kulp, Author

Contents

1 Introduction

1.1 Motivation

Multi-party computation (MPC) solves problems that are otherwise impossible without trust. The classic example[1] is that two millionaires at a party want to find out who has more money, but they don't want to divulge how much money they have. They could whisper in the butler's ear, but he might tell someone later or give the wrong answer, so they'd rather not trust anybody. MPC allows them to answer their question without trusting each other or the butler.

A real example from Denmark in 2008 involved the process of finding a fair price for sugar beets after EU market changes: for each potential price, the buyer specifies how much they are willing to buy and the seller specifies how much they are willing to sell. The "market clearing price" is then derived from a computation on these data. While this could have been done via a trusted party, the single Danish sugar beet buyer and the farmer's union were not suitable choices, and hiring a consultant would have been too expensive. The solution was to perform a multi-party computation such that the optimal price could be determined without the buyer or sellers revealing private information.[2]

In the modern cloud-computing model, providers compute on client data using proprietary algorithms. When clients send data to be processed, it is also available to the service provider for logging and analysis. This violation of privacy serves as a building block of the world of commercial IoT, cloud services, and centralized machine learning. MPC provides a cryptographic solution to this problem, removing the need to share data or trust a third party, while still computing the same results.

Support for efficient MPC in mobile devices could open the doors to many other privacy- and security-focused improvements to how our computers communicate. For example, a user could evaluate a pre-trained machine learning model without revealing their data to the service provider and without the service provider revealing their model[3]. This would, for example, allow a user to take a photograph and have it classified by a private algorithm without providing the image in the clear to the owner of the algorithm.

One obstacle to widespread adoption of MPC is the poor efficiency of execution. The protocol's cryptographic overhead slows the computation by several orders of magnitude (see Sec. 2.3). To address this, I propose a coprocessor to accelerate the client's workload. This coprocessor is attached to a smartphone to handle the most power-hungry aspects of the calculation more efficiently than the phone's built-in processor. In theory, this technique could be used to make MPC a common task in the same way that other cryptographic operations (like encrypting and decrypting TLS traffic) are widespread and efficient.

1.2 Scope

In this thesis, I explain Yao's Garbled Circuits, which is an MPC protocol for only two parties. I then implement this algorithm in software, and compare its execution time and power consumption to a second, interoperable implementation in which the smartphone interfaces with a custom coprocessor. It is out of scope to draw comparisons to other implementations or design a practical user interface.

I will not perform hardware modifications to the smartphone, nor design a new device as a stand-in; this limits the throughput of the data link between the smartphone and the coprocessor prototyping device, called an FPGA or "field-programmable gate array."

The coprocessor architecture is restricted by the small and low-power FPGA chip selected for its open-source[4] compatibility. More details on these design decisions are in Sec. 3.4.

2 Background

In this section I'll provide a self-contained explanation of Yao's Garbled Circuits.

Multi-party computation, or MPC, is a kind of cryptographic protocol where several people send some messages back and forth. The point of an MPC protocol is to compute the output of some function on private inputs held by each party, but without those parties needing to share their inputs. If you could trust someone to do the computation and keep everyone's inputs secret, then the calculation would be easy. MPC removes the need to trust other parties.

Multi-party computation can refer to any kind of computation involving any number of parties (greater than one). In this paper, I focus on one protocol under this umbrella, Yao's Garbled Circuits, which performs computations between two parties by evaluating a Boolean circuit.

2.1 Boolean Circuits

A Boolean circuit is a collection of logic gates connected together by wires. Each wire receives a label, with the computation's inputs assigned to the wires on the left. Each gate then compares the labels on its inputs to the gate's own lookup table to determine what label to assign to its output wire on the right. The contents of the lookup table determine how the gate behaves, and we give names to several common sets of table contents. For example, if 0 and 1 are the possible labels for each wire, and correspond to False and True respectively, then Tab. 1 shows an AND gate, an XOR gate, and another AND gate, left-to-right.

Circuits like these can compute any mathematical expression. (For usage in MPC, circuits can't have loops, so any looping algorithm will need its

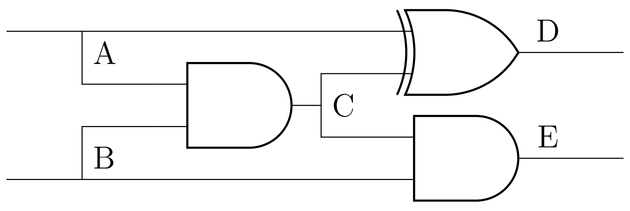

Figure 1: An example Boolean circuit with one XOR gate and two AND gates. The inputs wires are on the left columns and the outputs are on the right column.

loops unrolled first.)

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

A	C	D
0	0	0
0	1	1
1	0	1
1	1	0

C	B	E
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: Truth tables for the gates in Fig. 1. Each column is labeled to match a wire and each row matches an input combination to an output value.

To perform a computation with a circuit, you need to evaluate it. First, assign input labels to the wires on the left. Then, for each gate with all inputs labeled, check its truth table to determine which label to give the output wire. Note that gates are referred to by their output wire. Continue until all of the output wires are labeled, and then interpret those final labels for the result. For example, if every wire is labeled either 0 or 1, then the input to the circuit can be determined by applying the binary representation of a number, one bit at a time, to each wire. Similarly, the output labels can be concatenated as bits of the output number.

The Boolean circuit shown in Fig. 1 is too small to have much practical use. For comparison, even simple math operations like multiplication and square roots can require tens or hundreds of thousands of gates[5].

2.2 Oblivious Transfers

An oblivious transfer is a cryptographic primitive used in garbled circuits. It allows Bob to make a choice about which item to receive from Alice, without Alice knowing which item Bob chose, and without Bob learning anything about the item he didn't choose[6].

You can picture this as Bob sending Alice two empty boxes, labeled 0 and 1, shown in Fig. 2. Alice puts messages in each box, locks them, and sends them back, but Bob only made the key to box 1, a choice he made up-front before sending anything to Alice. Alice doesn't know which message Bob got, since the locks on the boxes look the same to her. Bob doesn't learn anything about the other message since the cryptography involved prevents the possibility of Bob creating two functional keys.

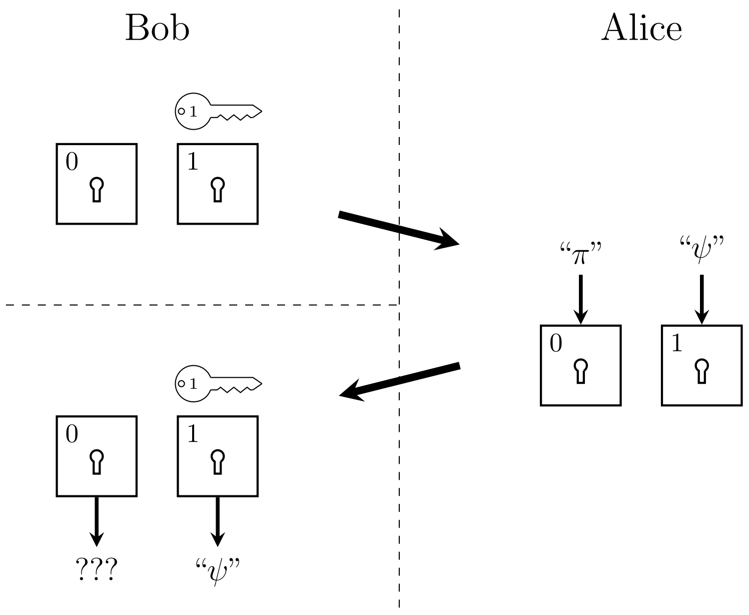

Figure 2: The oblivious transfer, shown with locks and boxes. Bob sends alice two empty boxes, one of which he has already decided he will be unable to open. Alice places different items into each, and sends them back to Bob. Bob opens the box he selected earlier. Alice does not know which item Bob received, and Bob knows nothing about the item he didn't receive.

2.3 Garbled Circuits

Boolean operations are represented by tables, but these tables don't have to map Trues and Falses, and in fact the outputs don't have to match the inputs at all. For example, in Fig. 1 and Tab. 1, wire A could instead have labels of ψ and π while wire B has labels of ϕ and δ. Then gate C (the AND gate on the left) would then have a truth table that dictates the output label when the inputs are π and δ or any other combination. Naturally, one can still evaluate a circuit without knowing which arbitrary labels correspond to True. This forms the basis for garbling.

The goal for Yao's Garbled Circuits is to allow one party to perform some transformation on a Boolean circuit such that the other party can evaluate the circuit blindly. If the garbler, traditionally named Alice, were to evaluate the circuit, she would know how the inputs correspond to Trues

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

A	B	C
ψ	ϕ	$\cap$
ψ	δ	$\cap$
π	ϕ	$\cap$
π	δ	$\Leftrightarrow$

A	B	C
ψ	δ	$\cap$
π	δ	$\Leftrightarrow$
π	ϕ	$\cap$
ψ	ϕ	$\cap$

Table 2: The original truth table from Tab. 1 on the left and partially garbled in the center, and randomly permuted on the right. Each wire's labels have been substituted with an arbitrary pair that correspond to True and False. Note that even in the table on the right, it's possible to determine which arbitrary labels correspond to True versus False.

and Falses, and she is therefore ineligible to evaluate the circuit if the two parties wish to keep their inputs secret. If the evaluator, traditionally named Bob, could determine how the arbitrary labels correspond to True and False values, preserving privacy would be similarly impossible. Instead, we must find a way to completely obscure the function of a gate while still allowing Bob to evaluate it.

Naturally, seeing the truth table would reveal how the arbitrary labels correspond to True and False. Especially since Bob must know the whole circuit definition (for reasons that will become obvious in the next section), he will already know that gate C is an AND gate, and if he received the partially-garbled truth table shown in the center of Tab. 2, then he would immediately know that π, δ, and $\Leftrightarrow$ all correspond to True because of the last row. Even without preserving the order, as shown in the table on the right, the asymmetry of the AND gate still gives away the values hidden under the arbitrary labels. Bob can look at the entire table and count that there are three instances of $\cap$ in the output and only one $\Leftrightarrow$, and conclude that $\Leftrightarrow$ must mean True.

The solution is for Alice to encrypt the output of each individual row of the randomly-permuted and arbitrarily-labeled table, using that row's inputs as keys. Then when Bob wishes to evaluate the circuit beginning with the labels on the left, he combines them to form a key with which he can unlock the correct output for each gate. The rest of the evaluation process proceeds as normal, but each gate evaluation now involves decryption. This way, Bob cannot observe the whole gate truth table like before, and therefore has no clues about the hidden meaning of the arbitrary labels.

At this point, it is easier to proceed with a different metaphor. Consider, instead of truth tables, a collection of four locked boxes, each with two key holes. Here, wire labels become keys such that each wire gets a unique pair of keys where one secretly corresponds to True and one to False. Then to evaluate a gate, you take the two inputs keys (wire labels) and open the only box that they fit in (decrypt the row of the truth table) to reveal the contents of the box, which is another key (the output column of the truth table which contains another wire label).

The process of evaluating the circuit is then Bob unlocking any box he has both keys for, and adding whatever key is inside to his collection as he goes. To begin the process, Alice prepares by making many random pairs of keys and locking them in boxes, and then sends these boxes to Bob. Once Bob has all of the keys that correspond to the inputs to the circuit, he can start unlocking boxes. When he does this, Bob is executing the circuit blindly without learning anything about the values in his computation. The whole garbled circuit is like an opaque machine that Bob drops his keys into, and then he "turns the crank" until the output comes out the other side.

Alice made all the boxes, so she can place the plaintext circuit outputs (0 and 1) in each final box on the right side of the circuit rather than a random key. It's her knowledge of the garbling transformation that's required to convert the answer from random keys and boxes to the result of the calculation.

Careful readers will have noticed a problem, though. How can Bob get the keys that correspond to Alice's inputs and his own inputs to begin the evaluation? Alice can simply send the keys that correspond to her input directly to Bob, since he doesn't know how they correspond to True and False values. For Bob's input, the solution is to use an oblivious transfer as described in the previous section.

Alice generates the true and false keys for every input bit. She can just give Bob the right key for her own input, since Bob doesn't know what it means, but she can't give Bob both keys for each of his inputs or he would be able to speculatively execute the circuit and he might be able to figure out her input. Bob also can't just ask for the key that corresponds to his input, because then Alice would know his input trivially. Instead, Bob requests the keys for each bit of his input with an oblivious transfer and Alice therefore does not know which keys he's receiving. Bob also does not receive any extra keys that would allow him to snoop on other possible circuit evaluation outcomes.

The vertical arrows in Fig. 3 are transformations to and from the garbled

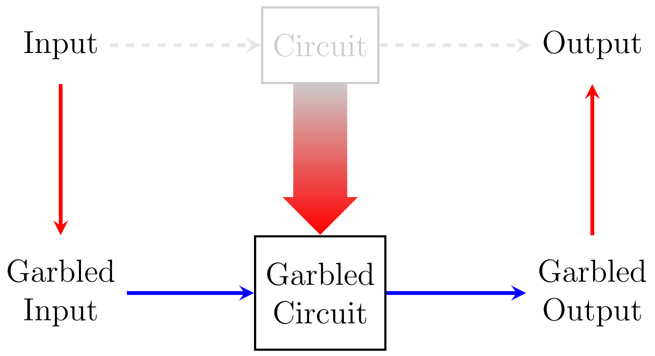

Figure 3: Circuit garbling as a translation. The red arrows show Alice's role and the blue arrows show Bob's. The greyed-out portion shows how a non-garbled circuit would be evaluated.

execution space that Bob works in. The horizontal arrows are computations within this garbled execution space. Since Alice knows about the transformation, she can't be the one to do the execution. Since Bob has all the inputs, he can't learn anything about the transformation. Together, though, they can perform the computation without either of them learning anything about the computation, including each other's inputs, except the final result.

2.4 Yao's Garbled Circuits

This protocol was first described by Andrew Yao in 1986 at that year's IEEE Annual Symposium on Foundations in Computer Science[1]. In the most basic terms, one party "garbles" the circuit by encrypting and transposing the inputs to each logic gate, then sends the stream of garbled gates to a second party, who evaluates the garbled circuit with their own private inputs while learning nothing about the garbler's inputs.

Alice, who garbles the circuit, must get this new representation to Bob. If we assume they both already have the same circuit to begin with, and they agree on a way to sort the gates into a series, then Alice can send the new truth tables that she generated in that order. Further, since the truth tables are permuted randomly anyway, there's no need to send the input columns. This means that Alice only needs to send the encrypted outputs of each gate for each input combination, and Bob knows how to reassemble this information into the complete garbled circuit to evaluate.

This presents an asymmetry in which one party must calculate each gate of the circuit for all inputs and broadcast a high volume of information while the other party performs a fraction of the work and only sends back data about the inputs and outputs. This asymmetry is similar to the current cloud-computing model, where servers in data centers perform difficult calculations while the client does minimal work to request and interpret the result.

There have been several improvements to Yao's original protocol since it was published[7]. For example, in the original protocol, it is unclear how the evaluator knows which row of a truth table to decrypt, since there is no way to observe a key and ciphertext and know that they go together. This leads to "trial decryptions" where valid keys are padded in some recognizable way, and the evaluator attempts to decrypt each row until they find the result with the proper padding. Clearly this is not efficient and improvements are desireable.

2.5 Improvements on the Protocol

Yao did not describe his original protocol with enough specificity to inform the design of a practical implementation. In particular, how does the evaluator know which row of the table to decrypt? One solution is trial decryptions. The garbler chooses keys to end with a bunch of zeros, and the evaluator decrypts until they find the one that ends with zeros. This is bad for performance and the number of ciphertexts to send creates a bottleneck in transmission. Research into improvements has been fruitful, and I detail the most relevant optimizations below.

2.5.1 Permute and Point

This first optimization seeks to avoid trial decryptions by helping Bob know which ciphertext to decrypt given only the keys he already has. The clever solution is to append a 0 or 1 to each key such that matching pairs always have one of each bit (called color bits)[8]. Note that these color bits are arbitrary and do not necessarily correspond to True and False.

Then for each truth table, the rows are permuted randomly as before, but in accordance with the random color bits rather than independently random. This means that when Bob has two keys and would like to decrypt the output of a particular gate, he simply needs to combine the last bit of each key to form the index of ciphertext that those keys decrypt. This is still secure since these color bits do not give Bob any information about which label corresponds to True[8].

In the box analogy, this is like assigning each key a color, and then coloring the locks on the boxes. There won't be any guesswork, because if you have a green and a blue key, you will only try to unlock the box with the green and blue locks, rather than the box with two blue locks.

2.5.2 Row Reduction

To maintain better security and make guessing infeasible, keys and ciphertexts are often chosen to be quite large. In my implementation, I used 128-bit numbers to transmit and store these values. This means that each gate's lookup table takes up $128 * 4 = 512$ bits. Any circuit of real-world use will have many gates; the SHA-256 circuit has over 130000 gates[5], which would mean 8MB of ciphertexts without further optimization, just to hash a single 512-bit block. Hashing a 1GB file would then take around 16TB of ciphertext transfers. Clearly, any way to reduce this number would be beneficial.

In practice, the encryption is performed by hashing the input keys, then using XOR to combine the resulting hash with the output key to form a ciphertext (so the encryption is in the form of a one-time-pad where the pad is generated with a hash). If we choose the first output key such that it matches the hash of the first input keys ("first" meaning after the permutation according to the color bits), then the XOR operation produces a zero, and the first ciphertext in the truth table is zero every time[9].

If the first ciphertext is always zero, then there is no need to transmit it, since Bob can just use the hash of his input keys as the output key if the color bits on the keys indicate that the first ciphertext should be decrypted. This means that instead of transmitting four ciphertexts per gate, we only need to transmit three.

2.5.3 Free XOR

I have so far described pairs of keys or labels as arbitrary (ψ and π for example). The pairs can also have a non-arbitrary relationship to Alice while still appearing random to Bob: Alice can instead choose a random delta, Δ, before she begins garbling, and then for each label A, with a random A_0 for False, she computes $A_1 = A_0 \oplus \Delta$ for the associated True label on the same wire[10]. Since Bob never has both the True and False labels for the same wire, he cannot recover the private delta Δ that Alice chose, and therefore cannot derive the inactive label from the active one in his possession.

Let A and B be input wires to an XOR gate and let C be the output wire. Then Alice can choose the False output label $C_0 = A_0 \oplus B_0$ and the True output label $C_1 = C_0 \oplus \Delta$. It follows that this same output is generated when both inputs are True:

$$A_1 \oplus B_1 = (A_0 \oplus \Delta) \oplus (B_0 \oplus \Delta) = C_0$$

Similarly, when only one input is True, the output is

$$A_0 \oplus B_0 \oplus \Delta = C_0 \oplus \Delta = C_1$$

These choices of C_0 and C_1 mean that the correct output label of an XOR gate can be computed without the hashing function and without sending a ciphertext, since when Bob encounters an XOR gate, he can simply XOR the input labels to produce the output label[10].

The result is that XOR gates become “free” in that neither Alice nor Bob need to perform cryptographic operations to generate or evaluate them, and XOR gates also do not require any transmission of ciphertexts. This means that circuits should be optimized to maximize the number of XOR gates, which now require sending zero ciphertexts, and minimize the usage of AND gates, which require sending three ciphertexts and performing cryptographic hashes.

2.5.4 Beyond

The most recent fundamental improvement at the time of writing is called Half Gates[11], which reduces the data sent from the garbler to the evaluator by short-circuiting AND gates to encode either buffers or inverters depending on each party’s known inputs to that gate. The result is that AND gates only require the transmission of two ciphertexts instead of three. This is also the furthest reduction possible in the number of ciphertexts for an AND gate[11].

Another further improvement is called Garbled Gadgets. While Free XOR allows for free additional modulo 2, Garbled Gadgets extends this to free addition with any modulo[7]. The aim of circuit optimization is then to look for ways to define the computation in terms of additions of any modulo, again with minimal use of AND gates.

2.6 Encryption

For theoretical cryptography, any method of encrypting rows of truth tables would lead to correct and secure MPC. In practice, to make full use of the optimizations mentioned above, encryption is done by using XOR with a one-time-pad to blind the plaintext. This one-time-pad must be indistinguishable from random and deterministically depend on the input wire labels. In practice, these requirements can be met by encrypting the input labels and using that ciphertext as the one-time-pad to generate the ciphertext that goes in the truth table. Since most modern processors include instruction extensions for efficiently computing AES (the Advanced Encryption Standard), this is a reasonable choice for the "hash" function that generates a one-time-pad from the input labels.

The AES algorithm starts with the plaintext as 16 bytes arranged into a 4×4 matrix. The key is also in the form of a 4×4 matrix, and expanded into a series of several such matrices, called round keys. First, the first round key is added to the state array with XOR. Next, the following process occurs nine times:

1. Each byte in the state matrix is substituted with a byte from a lookup table called an S-box.
2. The second row of the matrix is rotated one place to the left, the third row is shifted two places, and the bottom row is shifted three places to the left.
3. The columns are each multiplied by a particular fixed matrix over a Galois field (a particular kind of multiplication).
4. The round key is added to the state matrix with XOR.

After these rounds, the bytes are substituted from the S-box once more and the rows are shifted before adding the final round key to get the encrypted ciphertext (or in our case, the hash output).

This process sounds complicated, but it is well-suited to efficient implementations in both software and hardware, including making extensive use of lookup tables that encode the same operations as listed above.

2.7 Field-Programmable Gate Arrays

An FPGA (Field-Programmable Gate Array) is a kind of chip that implements reconfigurable logic. My coprocessor design runs on an FPGA rather than going through the prohibitively expensive process of manufacturing a "real" single-purpose chip. The "gate array" part of the name means that the chip is made of a grid of general-purpose elements, shown in the zoomed-out view on the left in Fig. 4. This part of the chip is called the fabric.

The "field-programmable" part of the name means that these connections can be reconfigured on-the-fly after the chip has been manufactured and shipped. I used this reprogramming capability to test each iteration of my design. Zooming in, you can see diagonal wires representing a connection between the bus lines and components internal to each element. There are several types of elements distributed across the fabric, including elements specifically for input and output, clock signal generation, digital signal processing, RAM, and basic elements, which contain registers and configurable lookup tables.

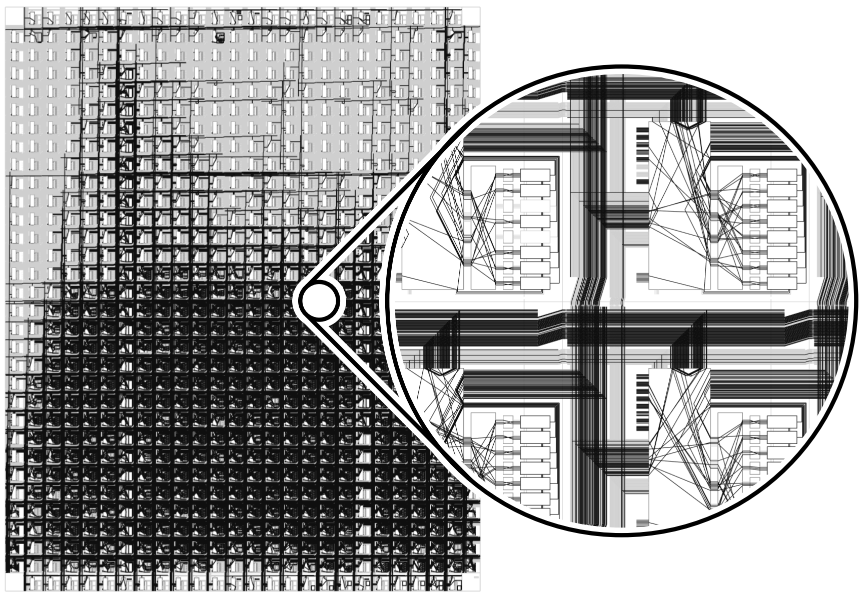

Figure 4: Place and route output of my hardware design. The black wires represent the allocated interconnects. The zoomed portion shows the reconfigurable internals of several logic cells.

HDL stands for "hardware description language" and it's the equivalent of "code" in software programming. Rather than code, a compiler, and an executable, the FPGA build toolchain has HDL, synthesis, place &

route, and the bitstream. I used the Verilog language to define the coprocessor, which is used to define state machines and combinational logic at the register-transfer level of abstraction. Synthesis is the process of turning this description into an abstract hardware definition. Place & route is the process of placing circuit components onto elements of the FPGA fabric and routing connections between these components. Once the final FPGA configuration has been determined, it is serialized into a format that the FPGA chip can read to configure itself[4].

The FPGA configuration is volatile, so the bitstream must be sent to the chip each time the board turns on. The FPGA development board therefore includes a nonvolatile flash memory chip which stores the configuration bitstream and sends it to the FPGA when the board receives power[12].

3 Approach

In this section, I'll describe all of my design choices when planning and implementing this project.

3.1 Algorithm

I first implemented Yao's Garbled Circuits with the permute-and-point optimization. Next, I added the Row Reduction and Free XOR optimizations. I decided not to pursue Half Gates or Garbled Gadgets due to time constraints. Implementing the remaining state-of-the-art optimization techniques would be a good candidate for future work.

I chose to keep the project general-purpose, such that it could execute for any circuit definition within the hardware limits (more on limits in Sec. 3.4). This is in contrast to a "baked-in" approach where I choose one circuit to garble, and hard-code it into the FPGA as an accelerator for only that computation. Algorithmically, this means that the circuit definition needs to be reconsidered dynamically every execution time.

My implementation uses a streaming approach, wherein the first line of the circuit definition file is interpreted, computed, and transmitted before the next line is even read. This is in contrast to an in-memory approach which would read the entire file, perform all computations, then transmit all messages. This means that the size of the circuit is not a concern for the garbler, and the evaluator can simply pass off the network traffic to the FPGA as it comes in. Considering that (useful) circuits can easily be larger than a typical consumer-oriented computer's memory, streaming is essential to scalability.

3.2 Protocol

I embraced the "semi-honest" model, in which the two parties behave correctly, but are "curious" about the information which should be hidden from them. This means that I can define the garbler-evaluator protocol without added security in mind to prevent the parties from acting maliciously. I therefore designed a protocol that makes many assumptions about the other party, like that they run the same version of the software and aim to evaluate the same circuit.

These last two assumptions are fine for development and testing, but any real application would require sending at least the software version and the circuit definition file's hash before starting the transaction. Once both parties have the same definition file, the order of transmission of data structures is the same every time, so there is no need for packet framing and metadata beyond what's inherent in TCP. The parties communicate over a socket that could be local, LAN, or across the internet, and I leave packetizing, buffering, and various transmission guarantees to the TCP/IP stack.

First, the garbler (Alice) listens for incoming connections. Once a socket is established, Alice sends her garbled inputs to Bob (the evaluator). Bob then requests his inputs bit-by-bit through oblivious transfers with Alice. After this setup phase, since the gate types and wire IDs are already provided in-order in the circuit definition, Alice sends Bob *only* the AND gate ciphertexts as she computes them. Finally, once all the gate definitions have been transmitted, Alice sends a hash of each output wire's False label. Bob hashes his own output labels and compares them to the hashes received from Alice. For each wire, if the hash matches, then Bob knows that the output value on that wire is False. If the hash does not match, the output is True. Bob the recombines these True and False values as a binary number to form the output of the computation.

Communication between Bob's smartphone and FPGA is more complicated since it doesn't have access to the circuit definition file. The circuit definition can be considered a set of instructions for the co-processor to execute, and sending these ahead of execution time would add significant design overhead for the storage and access of these instructions. Instead, opcodes (the gate type), operands (the gate ID and input wire IDs), and data (ciphertexts when the gate is an AND gate) are all transmitted over the same serial connection. Each instruction is packed with the gate type first, then the input IDs (or just one ID in the case of a buffer gate), the ciphertexts if present, and finally the gate ID. In addition to the gate type instructions, there are also instructions to set the read/write head, write a wire label, and read a wire label. The writing is used to provide the FPGA with the initial garbled inputs, and the reading occurs at the end to report the calculation result to the smartphone.

3.3 Software

I wrote the garbler and evaluator code in Python. An interpreted language is obviously not ideal for a performance-oriented project, but Python's simplicity proved valuable for debugging and rapid prototyping. With more time or in a following project, I would re-implement the functionality in

Rust or C. I made some design decisions with a language migration in mind, like avoiding libraries beyond the standard library and any esoteric features.

I also made some software design decisions to make the hardware definition steps easier. For example, I learned and re-implemented AES rather than relying on OpenSSL or some other cryptography library. This made the project more of an academic exercise than a typical software engineering process.

3.3.1 Oblivious Transfer

I first implemented the RSA public-key cryptosystem in Python, then used it as a library to implement an oblivious transfer API that uses sockets. The implementation was straightforward.

I originally intended to replace the RSA public-key backend with elliptic curves (ECC), but ultimately prioritized other aspects of the project. This is another area for future improvements, since the ECC cryptosystem allows sending much smaller keys between parties while maintaining the same level of security.

3.3.2 Advanced Encryption Standard

One of my initial goals was to thoroughly understand and then implement the Advanced Encryption Standard (AES). To this end, I did not use a well-audited library like OpenSSL and instead implemented the underlying mathematics directly. In my first design iteration, I computed every operation without lookups with the exception of the S-box. After profiling the performance, I discovered that the Galois field multiplication in the MixColumns step was a tight bottleneck. I replaced the function with a table lookup to improve performance dramatically.

I could have replaced the other round operations with table lookups to reach a "T-tables" implementation, but after implementing this technique in hardware, I decided to leave the software AES functions in their more readable state describing the distinct operations that compose AES rounds.

3.3.3 Garbler

The garbler role is fulfilled by a single-client server. It listens for new connections, performs a fresh Garbled Circuits computation with any client that connects, then returns to waiting idle for the next client. Since my testing and development only ever had one client and one server at a time, there was no need to implement a full FastCGI or WSGI interface. In the cloud-provider model, a scalable server architecture would be more appropriate, but this was not an implementation goal. Calling my libraries from a web server like Flask would make writing a WSGI implementation of the garbler role simple.

3.3.4 Evaluator

I implemented the evaluation logic three ways. First, I write idiomatic Python code in software alone. Next, I wrote an FPGA emulator (also in Python) and modified the evaluation logic to send properly-packed coprocessor instructions to the emulated device. Finally, once I had finished the hardware implementation, I simply swapped the endpoint from the emulated coprocessor to the real one.

In the default configuration, the evaluator script checks for the presence of the coprocessor, and falls back to the software implementation if needed.

3.4 Hardware

I chose the Pine64 Pinephone[13][1] for the mobile smartphone platform since it fully supports a typical Linux environment, allowing me to use the same source code for the client and server[2] software. The Pinephone also includes a USB port accessible to userspace tools in the same way as on a typical laptop or desktop, meaning that the interface to the coprocessor is the same regardless of the platform. I chose the iCEBreaker FPGA development board[12] (with the Lattice iCE40 UP5K FPGA chip[14]) for its low cost, low power consumption, open-source board design, open-source development toolchain[3] (utilities listed in Tab. 3), and accessible community.

I prioritized finding an open-source option because of the importance of public audits for all security-related code. One downside of using the open-source toolchain is that closed-source pre-designed modules (IP cores) cannot realistically be sold or licensed for use. This is not a concern for me because of my goal of allowing total audits and because I do not need the convenience of drop-in solutions like Ethernet controllers or other high-complexity high-performance components that would require the proprietary expertise of the manufacturer.

[1]Allwinner A64 quad-core SoC, 3GB RAM, USB 2.0 over type-C

[2]8-core Intel Xeon E3-1505M CPU, 32GB RAM (evaluator is the bottleneck)

[3]`https://github.com/YosysHQ`

Utility	Executable	Version	Commit
IceStorm	`icepack`, `iceprog`	r777	`c495861`
Yosys	`yosys`	0.9+4052	`0ccc722`
Next Gen P&R	`nextpnr`	r3529	`5a41d20`

Table 3: FPGA toolchain utility versions. "P&R" is an abbreviation of "Place and Route." All tools were built from source from the GitHub commit matching the hash in the right column. These tools can be found on the Yosys Open SYnthesys Suite GitHub page: `github.com/YosysHQ`

3.4.1 Execution

I used an event-based design inside the FPGA rather than a central control unit that issues control signals with fixed timings. For example, the memory controller emits a signal when it has finished fetching a value, and this signal is fed to the next component which uses the fetched value. Sequential and combinational logic considering the current gate type determines which "done" signals are forwarded to "start" signals.

There are only six instructions: set address, read, write, AND, XOR, and BUF. The first three are for direct memory access (DMA) over the serial connection for setting inputs, reading outputs, and debugging. The latter three instructions are garbled gates to evaluate. Each gate instruction begins with the gate type (opcode), then the wire ID for each input. In the case of the AND instruction, $128 * 3 = 384$ bits of ciphertexts follow. All gate instructions end with the gate ID, which is also the wire ID at which to store the gate's output. Since there are so few instructions, and the most common instructions (the gates) each do multiple memory operations, this coprocessor can be considered to have a CISC (complex instruction set computer) architecture.

A high-level overview of the architecture is shown if Fig. 5. The serial signal enters through the SPI module into an encoder/decoder, which breaks each instruction down into its fields and strobes control lines when each field is received. When the wire ID of an input is received, the memory controller fetches the wire label stored at that address. If it is the second wire ID of the gate and it is an XOR gate, then the second fetched wire label is combined with the first with XOR, and the result is held until the gate ID (destination wire label address) is received. The result is then stored at that address.

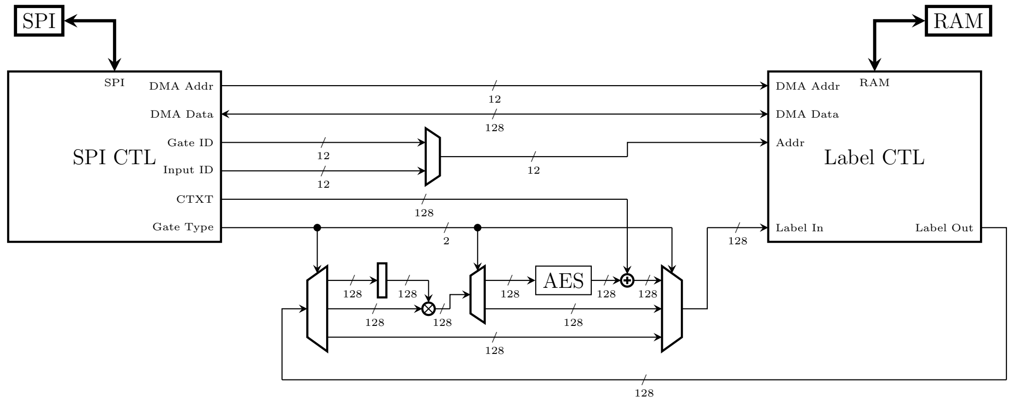

Figure 5: Simplified coprocessor architecture. The main memory-memory logic is implemented in the bottom center. The behavior of the three multiplexers on the bottom over time determine which gate (instruction) is being evaluated.

If the gate type is an AND gate, then the two fetched values are combined to form the input to the AES hash function and the point-and-permute pointer. Once the AES encryption is complete (which occurs while the first ciphertext is being received), the hash is held until the index of the most recently received ciphertext matches the point-and-permute pointer, at which point they are combined with XOR and the result is held until the gate ID arrives. In all cases, the final result is stored as soon as the gate ID arrives.

3.4.2 Pipelines and Memory

The coprocessor has a memory-memory architecture: instructions operate on the contents of memory without any general-purpose registers. Each garbled gate evaluation requires one or two memory reads and one memory write, with two or three address therefore specified within each instruction. Memory operations are done in the order of addresses within the instruction as the instruction is read in from serial byte-wise.

These memory operations are individually pipelined. Memory is used for the AES lookup tables and for the wire label storage. Unfortunately, the very nature of each gate instruction and the AES T-table lookups present load-and-use hazards that require waiting for dependencies and inserting pipeline bubbles. Each AES round requires two memory operations since there is only enough RAM width to support looking up half of the values in the state array at a time, so while these two operations can be pipelined, there is no way to keep the memory pipeline full across rounds. This also means that there would be no speed penalty to performing each round in three lookups instead of two, since the second lookup of one round could take place while the first lookup of the next round enters the pipeline, filling the bubble.

Even combining all four SPRAM blocks to create a wire label array only offers the bit width to accommodate half of a wire label, so memory accesses to the wire label array are also pipelined as two operations.

3.4.3 Limitations

My choice of FPGA development platform placed many restrictions on my implementation. For example, I only have room on the fabric to provision a module capable of performing half of an AES round in a single clock cycle. On a larger FPGA, I would have enough room to fully unroll the algorithm's loops and perform all AES rounds in a single clock cycle. Similarly, I would have more options for fast inputs and outputs.

Right now the design uses on-chip SPRAM for wire label storage, which only has space to safely store 8192 wire labels. It effectively functions as a direct-mapped cache without a larger memory device to back it. This means that any gate in the garbled circuit which requests an input wire with an ID more than several thousand addresses behind the gate ID runs the risk of having the desired wire label overwritten. This issue can be easily detected in software during execution or as a preprocessing/validation step, but the much better solution is to use more RAM.

3.4.4 Scalability

The event-based architecture makes it trivial to integrate a new memory controller for a larger bank of memory. The standard expansion port (PMOD) on the development board makes adding RAM simple, though a new driver would be required on the fabric.

Similarly, a faster AES implementation relying on more lookup tables or an unrolled loop would easily slot into the current design without the concern for tweaking timings or delays in other components.

4 Results

I ran comparative tests between the software-only system and the system with FPGA integration[4]. Since my design goal is a coprocessor for mobile devices, speed and power efficiency were both concerns.

4.1 Speed

Performing a 64-bit integer division with the software-only implementation between Wi-Fi devices on LAN took 7.6 seconds on average. Evaluating the same circuit with the same devices, but with the FPGA attached to the evaluator, took 10.1 seconds on average. My analysis suggests that this slowdown occurred from idle time on the serial line as the evaluator performed byte manipulations and passed the next instruction off to the SPI serial library.

Initial benchmarks of the serial connection indicated the possibility that it would be the bottleneck, even when sending instructions back-to-back. In practice, I added a counter to the coprocessor design to count the number of cycles between the end of one instruction and the start of the next, and found an average of over 5000 cycles spent idle. This number is not useful on its own, but with a calculation of the average number of cycles to process one gate, I can calculate the maximum speed of the coprocessor without concern for the serial or network connections upstream.

Gate type	Cycles	Time at 45MHz	Gates per second
AND	247	5488ns	182000
XOR	26	578ns	1730000

Table 4: Coprocessor performance for each gate type. The right column assumes instructions are sent back-to-back without idle time on the serial connection. `nextpnr` reported a maximum clock speed near 45MHz.

Using a similar cycle counter, I found that XOR gates take 26 cycles on average and AND gates take 247. This time includes receiving the instructions as bits, shifting them into bytes, deserializing into addresses & ciphertexts, fetching from memory, computing AES & waiting to receive the

[4]`https://github.com/gabrielkulp/undergrad-thesis`

correct ciphertext (in the case of an AND gate), and finally storing the computed active wire label. The variability (I provide the *average* cycle time for each gate type) occurs because the serial module and the rest of the coprocessor operate in different clock domains, meaning that the number of coprocessor cycles per received byte depends on the relative frequencies and phases of each clock signal.

In the 64-bit integer division circuit[5] there are 12603 XOR gates and 4285 AND gates. With this ratio, there are an average of 80 cycles per gate. Combining this information with the 5000 cycles between gates, the serial utilization ratio is only 1.6%, meaning that a fully-saturated link would be 62 times faster than what I measured.

This analysis does not consider the potential speed improvements of higher clock speeds. I chose 30MHz for the main clock, but the place and route tool suggested I could go as high as 45MHz without redesigning anything. With redesign work, there are also several pipelines I could improve and processes I could stream instead of batch which would reduce the number of cycles spent processing a single gate and make room for a faster interface to the smartphone.

4.2 Power Efficiency

The iCE40 UP5K FPGA chip is marketed primarily as a low-power choice, with an advertized idle power consumption of 75µA and typical consumption of 1-10mA.[14]. The expectation is therefore little to no impact on battery life from the FPGA itself. Other components, such as the FTDI USB interface chip or the USB daughterboard on the smartphone, could draw more power than the coprocessor itself. (I actually felt more heat coming from the FTDI chip than the FPGA while running tests, but I didn't have a way to properly measure power consumption.)

The simplest way to test power consumption in a "real-world" way is to measure battery life. I modified the garbler and evaluator scripts to print a timestamp and loop once they finish a computation, and then I ran the evaluator script on the smartphone starting at full battery until it died. I ran both tests after leaving the battery plugged in at 100% for several hours to hopefully mitigate any rebound effects of the battery chemistry.

The smartphone lasted about one hour and 40 minutes, with or without the coprocessor attached. This was surprising, but believable considering the low-power nature of most components involved.

I took no explicit measures to decrease power consumption, but several options are available with this chip. The SRAM on the die has a power saving mode which I could enable when waiting for the next gate, and I could drop the clock speed to have the minimum number of clock cycles between each incoming serial byte to still keep up with the stream. Measures like these could counteract the extra power draw that the DRAM expansion would incur. This could be a subject of future research.

5 Conclusion

My vision is to make privacy-preserving computations more common. In the United States, HIPAA and FERPA laws protect healthcare and student records respectively, but these laws prevent analysis that could have great value to society. MPC allows calculations on these data that no single party could ethically or legally compute on their own.

Most academic focus in the field of MPC is on data centers, with many fewer published attempts at enabling low-power devices like smartphones and small FPGAs to participate. Working with these low-power devices has value to the cryptography community: developing in constrained environments inevitably contributes to the less-constrained ones, and the widespread nature of mobile devices offers many opportunities for real-world applications of privacy-preserving cryptography.

Our devices already have efficient hardware support for some cryptographic operations, opening the doors to online banking, secure video calling, and other private transactions. I would like to see mobile devices that can efficiently do MPC with each other or cloud services to open the doors to even more private applications.

This thesis presents a feasible architecture with significant performance potential for low-end mobile devices. With a faster CPU-to-coprocessor connection or the coprocessor instructions implemented as en extension of existing mobile SoC instructions, mobile MPC could be constrained only by the speed of the network connection. Future work promises to be exciting.

References

[1] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, IEEE Computer Society, 1986, pp. 162–167. DOI: `10.1109/SFCS.1986.25`.

[2] P. Bogetoft, D. Lund, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. Nielsen, J. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft, "Secure multiparty computation goes live," vol. 5628, Feb. 2009, pp. 325–343, ISBN: 978-3-642-03548-7. DOI: `10.1007/978-3-642-03549-4_20`.

[3] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski, *Garbled neural networks are practical*, Cryptology ePrint Archive, Report 2019/338, 2019. [Online]. Available: `https://eprint.iacr.org/2019/338`.

[4] C. Wolf and M. Lasser, *Project IceStorm*, `http://bygone.clairexen.net/icestorm/`.

[5] D. Archer, V. A. Abril, S. Lu, P. Maene, N. Mertens, D. Sijacic, and N. Smart, *'Bristol Fashion' MPC circuits*, `https://homes.esat.kuleuven.be/~nsmart/MPC/`.

[6] M. O. Rabin, *How to exchange secrets with oblivious transfer*, Harvard University Technical Report 81 talr@watson.ibm.com 12955 received 21 Jun 2005, 1981. [Online]. Available: `http://eprint.iacr.org/2005/187`.

[7] S. Yakoubov, *A gentle introduction to yao's garbled circuits*, preprint on webpage at `https://web.mit.edu/sonka89/www/papers/2017ygc.pdf`, 2017.

[8] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - secure two-party computation system," in *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, M. Blaze, Ed., USENIX, 2004, pp. 287–302. [Online]. Available: `http://www.usenix.org/publications/library/proceedings/sec04/tech/malkhi.html`.

[9] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," ACM Press, 1999, pp. 129–139.

[10] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz,

Eds., ser. Lecture Notes in Computer Science, vol. 5126, Springer, 2008, pp. 486–498. DOI: `10.1007/978-3-540-70583-3\_40`.

[11] S. Zahur, M. Rosulek, and D. Evans, *Two halves make a whole: Reducing data transfer in garbled circuits using half gates*, `https://eprint.iacr.org/2014/756`, 2014.

[12] P. Esden-Tempski, *iCEBreaker FPGA*, `https://github.com/icebreaker-fpga/icebreaker`, 2017.

[13] Pine64, *Pinephone*, `https://www.pine64.org/pinephone/`.

[14] *iCE40 UltraPlus*, `https://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40UltraPlus`, product page.