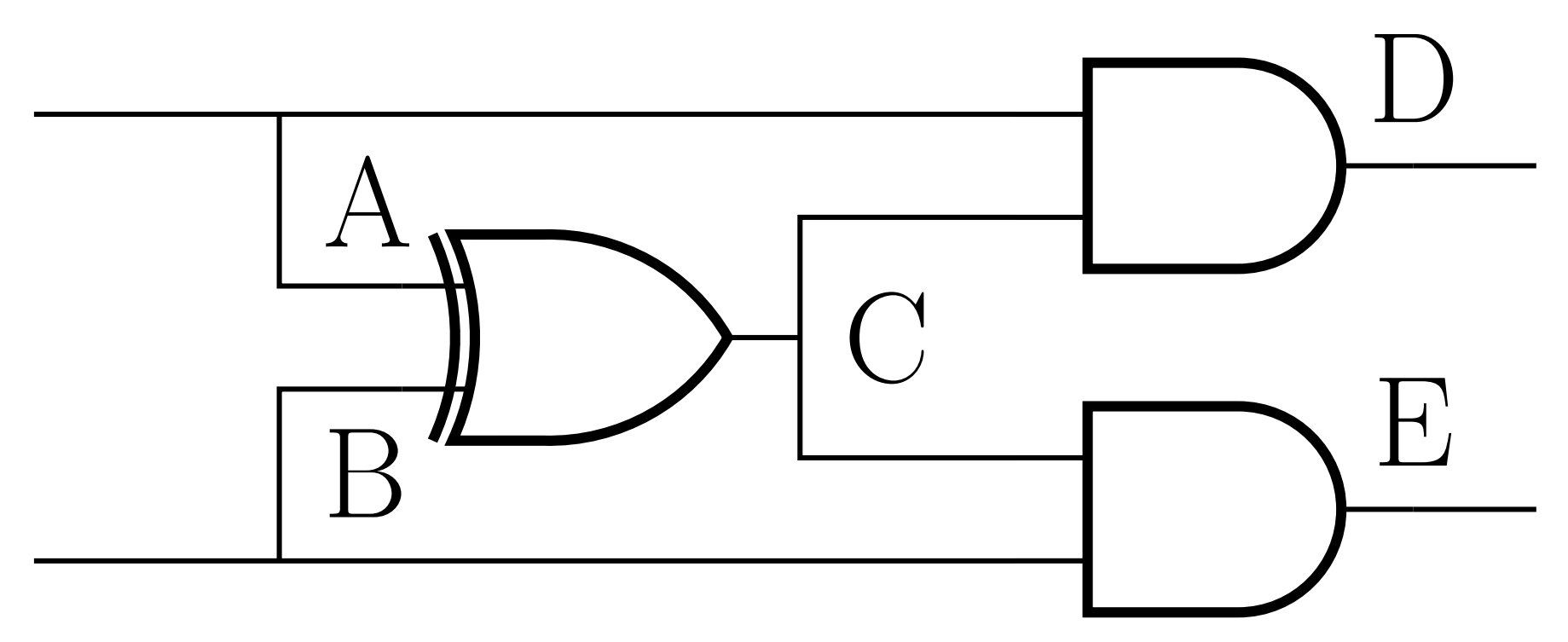
Mobile Cryptographic Coprocessor for Privacy-Preserving Two-Party Computation

Gabriel Kulp, mentored by Dr. Mike Rosulek
Oregon State University School of Electrical Engineering and Computer Science

Abstract

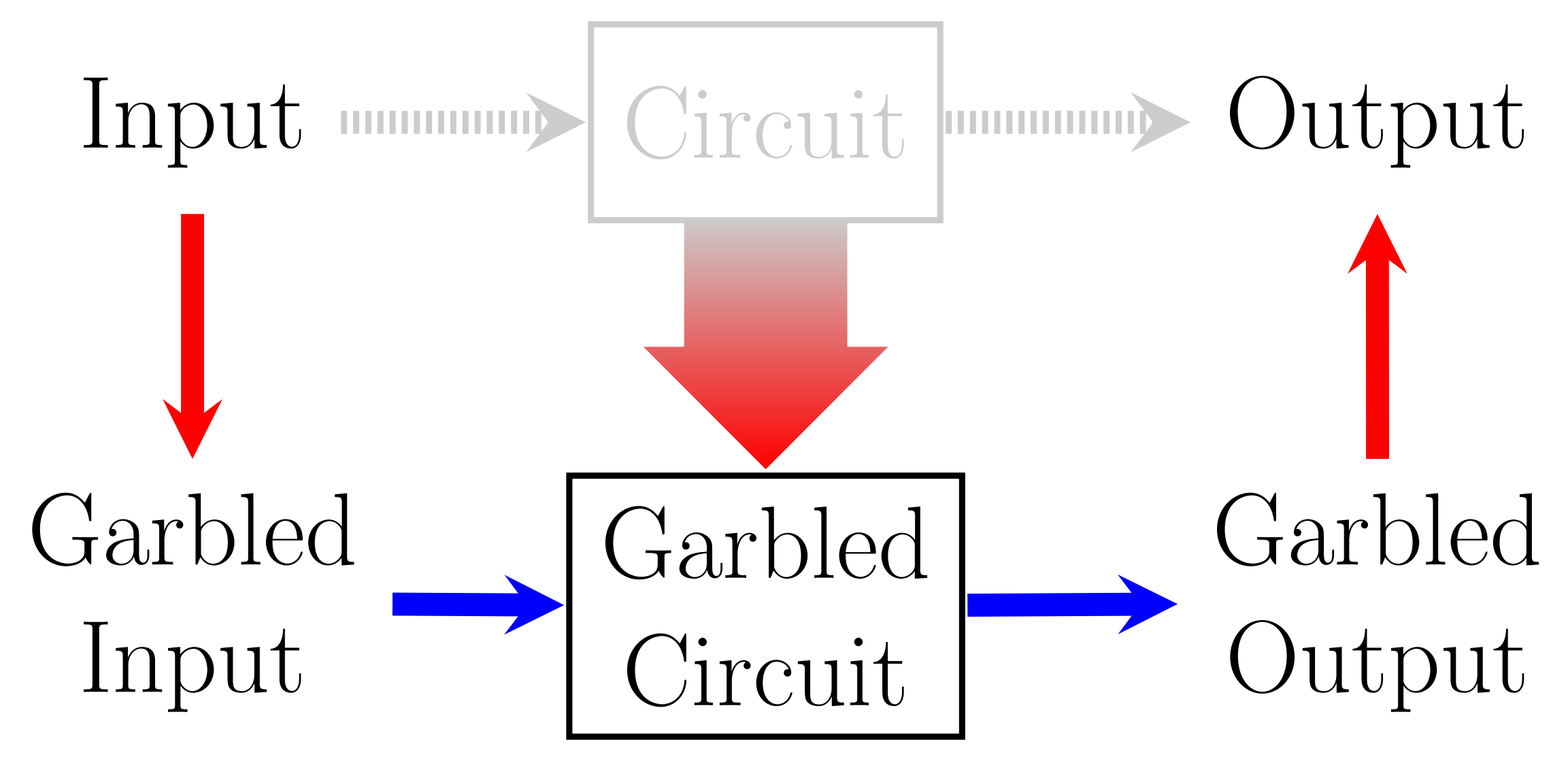
Multi-party computation (MPC) is a field of study focused on devising cryptographic protocols that allow participants to learn the output of some function of their private inputs without trusting a third party to perform the computation. This is usually done at a large scale between data centers, with 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.

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 its 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, like AND and XOR, to describe each gate as a boolean function. Boolean circuits are able to evaluate any mathematical expression, so a system (like mine) capable of processing wires and gates can perform any computation by just providing the desired circuit definition.

Garbled Circuits



A Garbled Circuits protocol allows two parties to supply parts of the input to a Boolean circuit and together compute the output, but without sharing their input with the other party. Rather than following the dotted lines in the picture from input to output, one party (traditionally called Alice, shown here in red) "garbles" the circuit by applying some private transformation to it. She then works with the other party (traditionally called Bob) to convert their inputs into garbled inputs. Since Bob doesn't know the secret transformation, he's the one to evaluate the circuit, albeit blindly, following the blue arrows. Finally, Alice helps Bob transform the garbled output back into a normal output. In my setup, Bob has the FPGA and uses it to more efficiently perform the blind evaluation of the garbled circuit.

Software Approach

Algorithm There are several standard optimizations to garbling and evaluating circuits. I implemented Point-and-Permute, Free-XOR, and Garbled Row Reduction, but not Half Gates (the most recent improvement by my mentor, Dr. Rosulek) or any gadget-wise optimizations. Implementing the remaining state-of-the-art optimization techniques would be a good candidate for future work, since they would further improve evaluation speed.

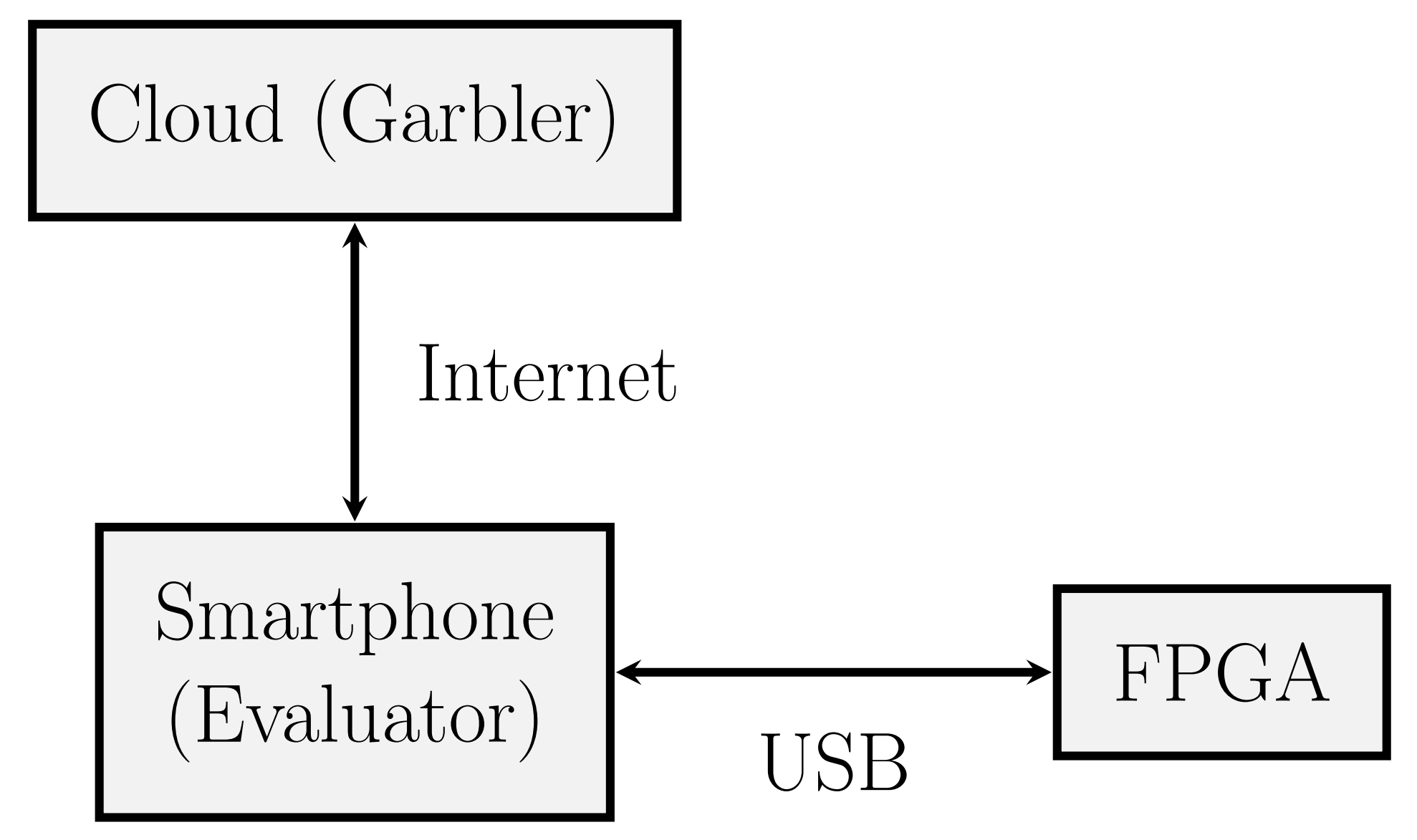
Scalability My implementation uses a scalable streaming approach, wherein the first line of the circuit definition file is interpreted, computed, and transmitted before the next line is even read. This contrasts with an in-memory approach which would read the entire file, perform all computations, then 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 real-world circuits can easily be larger than a typical consumer-oriented computer's memory, streaming is essential to feasibility.

Protocol I send the minimum amount of information for each gate. Since the type and ID are already provided in-order from the circuit definition, I send and receive *only* the AND gate ciphertexts, without padding or metadata, between the garbler and evaluator. The evaluator has a copy of the circuit definition file and uses it to annotate the received ciphertexts before forwarding the complete set of instructions to the FPGA for processing. As soon as the smartphone (garbler) finishes sending data to the FPGA, it has the result ready to return.

Hardware Approach

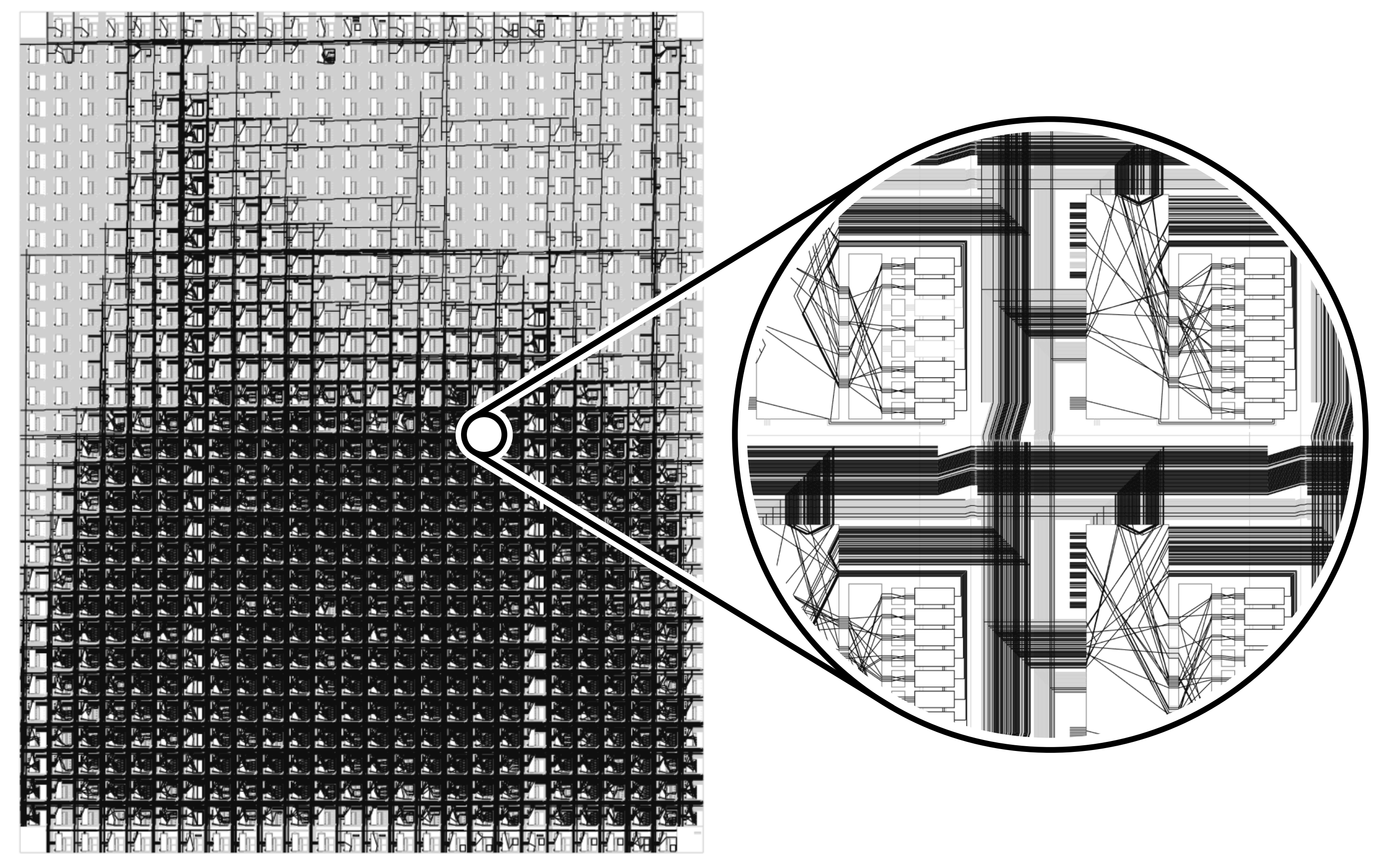
Scalability I used an event-based design inside the FPGA. For example, the memory controller emits a signal when it has finished fetching or storing a value. Right now the design uses on-chip SRAM, which significantly limits the size of circuit that can be safely evaluated. The event-based architecture would make it trivial to integrate a new memory controller for a larger bank of DRAM. Similarly, a faster AES implementation relying on more lookup tables would easily slot into the current design without worrying about tweaking timings or delays in other components.

Limitations My choice of FPGA development platform placed many restrictions on my implementation. For example, I only have room on the chip to provision a block 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.



System topology. *I used my laptop on the local network as the garbler for simplicity, but the system could work with the same code running on a cloud server across the internet.*

FPGA Internals



Place and route output of my hardware design. *The black wires represent the allocated interconnects. The zoomed portion shows the reconfigurable internals of each logic cell.*

FPGA stands for *Field-Programmable Gate Array*. Running my coprocessor on an FPGA allows me to avoid the prohibitively expensive process of manufacturing a "real" single-purpose chip, while still letting me test on hardware instead of simulations alone. 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 above. Zooming in, you can see diagonal wires connecting the bus lines together inside some of these elements. 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.

Results

With the FPGA fully utilized, the calculated **performance improvement is 62x** compared to a software-only system. In my setup, the connection between the FPGA and the smartphone was too slow to sustain this speed, and the real-world performance is 25% slower than without the FPGA. Battery life was not significantly affected when using the FPGA.

For a benchmark, I performed a 64-bit floating point square root where each party supplies half of the representation of the number to take a square root of. While this is not a computation with much use in the real world, the actual computation has little bearing on the per-gate performance of the system. On average, evaluating a gate took 80 clock cycles and waiting for the next gate took 5000 cycles. In a fully-integrated system-on-chip, the connection speed between the smartphone's processor and the FPGA would be far above the bottleneck presented by the network speed.

In conclusion, MPC is a kind of cryptography that allows a whole new class of problems to be solved, but it takes a lot of computing power. MPC could be adopted more readily if there was an easy way to offload that work onto specially-designed hardware. In my project, I conclude that this coprocessor is a feasible architecture to this end with significant performance potential.