The Cost of Private Computation: Homomorphic Encryption for Low SWaP Platforms

Daniel Achee
Jennifer Quintana
2020.03.20
EE209AS Embedded Systems Cybersecurity

This project explores the landscape of private, secure data analytics. If a user wants to analyze data produced by an IoT device without giving away that data to an untrusted third party, they have several options. One is to perform the analysis locally, which may be impractical given the resource constraints of their IoT device. An alternative is that they can encrypt the data with homomorphic encryption, send it to another more powerful device which can perform the analysis directly on the encrypted data, and receive the results, only decrypted on their local device. We investigated the cost of private computation on a low SWaP device, a series of AWS EC2 cloud computing instances, and a client-server model incorporating a resource-constrained platform and a more high-performance platform.

1.0 Introduction & History

Homomorphic encryption is an encryption scheme that allows the processing of data in its encrypted form without requiring access to the key to decrypt. To preserve privacy during data processing on a remote server, homomorphic encryption allows for data operations directly [1]. The first fully homomorphic encryption scheme was proposed in 2009 by Craig Gentry in a collaboration between Stanford University and IBM Watson [2]. As cloud computing and constellation-style communication hardware, such as the SpaceX Starlink small satellite constellations, become more available, the need for secure communication and processing among nodes becomes increasingly important. Processing capabilities on small size, weight, and power (SWaP) devices have made advancements, especially in the system on a chip (SoC) and field programmable gate array (FPGA) technical development areas. This means more resources are available on an end device for processing purposes and therefore enhances the ability to enact more sophisticated security and privacy measures among end devices and remote computing services. Homomorphic encryption is one of the emerging tools to aid in the enhancement of privacy features in computing and secure communications.

2.0 Homomorphic Encryption Basics

Homomorphic encryption provides secure data processing directly on the cipher text, whose computation results are also encrypted. The resulting data analysis provides results which match the equivalent analysis performed on the plaintext data [3] and the structure of the homomorphic data is preserved [4]. A variety of partially and fully homomorphic encryption schemes are available for research and analysis. Fully homomorphic encryption (FHE) schemes allow for evaluation of arbitrarily complex programs on encrypted data [5], whereas partially homomorphic encryption (PHE) schemes are

homomorphic with respect to only one type of data operation, such as addition or multiplication. PHE is more computationally efficient than FHE, but trades some security for that efficiency [1].

Various implementations of homomorphic algorithms have been developed using public asymmetric key systems such as RSA, ElGamal, Paillier algorithms [1]. Some of these different encryption schemes that employ public asymmetric key systems include Brakerski-Gentry-Vaikuntanathan (BGV), Brakerski-Fan-Vaikuntanathan (BFV), Enhanced Homomorphic Cryptosystem (EHC), Algebra Homomorphic Encryption (ElGamal-based) (AHEE), and Non-interactive Exponential Homomorphic Encryption Scheme (NEHE) [1].

2.1 Types of Homomorphic Encryption

BGV and BFV are asymmetric encryption schemes to encrypt a data stream bit-wise. They use lattice-based cryptography, which has shown some resistance to quantum computing used for forced data decryption [6]. BGV employs ring learning with errors (RLWE), which is difficult to solve even for quantum computers [6]. These schemes are fully homomorphic and have two subprocesses – one dealing with integer vectors linked to the decisional LWE and one dealing with integer polynomials linked to RLWE [7]. The polynomial version of the BGV and BFV encryption schemes are more computationally efficient and more widely explored in terms of applied research systems using FHE [7].

In 2016, Cheon-Kim-Kim-Song developed the CKKS fully homomorphic encryption scheme. CKKS supports approximate arithmetics over complex numbers and exploits ring isomorphism, which is a different implementation strategy than BFV. The CKKS encryption scheme consists of encryption, decryption, addition and multiplication, and rescaling [4]. The security of CKKS encryption, like BFV, is based on the hardness assumption of RLWE, so the fundamental security properties are similar for both BFV and CKKS [7].

2.2 Encryption Libraries

Two widely used homomorphic encryption libraries for research are the Simple Encrypted Arithmetic Library (SEAL), developed with Microsoft Research and MIT, and the Homomorphic Encryption Library (HElib), developed by IBM. Both libraries are available for use on Github. Both libraries are implemented in C/C++ without external dependencies [8].

HElib is developed based on the BFV polynomial encryption scheme. BFV in this implementation contains arrays of polynomials, at least two polynomials per array. These grow in size with each multiplication operation until a relinearization is performed [8]. In the implementation, homomorphic encryption additions are generated by computing a component-wise sum of these arrays [8].

SEAL supports both BFV and CKKS implementation methods. The SEAL implementation of BFV differs from the standard BFV mathematical approach because the ciphertexts are described by a tuple representation – a coordinate representation – of the polynomial [8]. SEAL's support for CKKS is also implemented, and the example functions are provided to support both homomorphic encryption methods. Their results are compared in our analysis.

3.0 Experiment Technical Approach

3.1 Experiment Goals

Our investigation of the implementation of homomorphic encryption for low-SWaP devices had three main goals:

1. Analyze and implement the SEAL homomorphic encryption library on a series of low-SWaP devices, including processor-only solutions, FPGA-only solutions, and hybrid SoC solutions.
2. Implement the SEAL homomorphic encryption library on a series of Amazon WebService Enhanced Cloud Computing (AWS EC2) services to benchmark functional performance and memory requirements.
3. Implement a client-server model using a homomorphic encryption scheme between a low-SWaP device and a cloud computing service instantiation.

The team had some experience with low-SWaP devices for space applications and wanted to explore the feasibility of homomorphic implementation in as small a platform as we could reasonably accomplish. We investigated a series of embedded devices, listed in Table 1.

Table 1: Devices examined for implementation of homomorphic encryption scheme.

Item No.	Manufacturer	Device Description	Platform Type
1	Intel	DE10 Nano SoC (Cyclone V)	SoC (ARM + FPGA)
2	Intel	DE0 Nano (Cyclone IV)	FPGA
3	Xilinx	Arty A7 (Zynq 7020)	SoC
4	Texas Instruments	MSP430 LaunchPad	Microcontroller
5	Raspberry Pi	RPi 4	ARM Processor

These platforms were selected as being low-SWaP options, defined as small enough and low power enough to fit on a 1U cubesat, which is 10cm x 10cm x 11cm. Other SoCs, such as the Cyclone 10 or Stratix 10 Intel boards were examined but determined as too large and had a power consumption greater than 5-10W. They were also prohibitively expensive for the purpose of this project, each board on the order of $10,000.

For the Amazon Webservices Elastic Cloud Computing goal, we implemented the SEAL libraries on the following instances, describe in Table 2.

Table 2: AWS EC2 Instances used for implementation of homomorphic encryption scheme.

Item No.	Instance Name	Instance Type	CPU Cores	Memory
1	t1_micro	General Purpose	1	0.5 GB
2	c5n_large	Computation Optimized	2	4 GB
3	t2_micro	General Purpose	1	1 GB
4	r5a_large	Memory Optimized	2	16 GB

These instances were selected to represent a mix of resource-constrained and high-performance platforms. The t1_micro instance was examined and later disregarded because it did not have the computing power to run the polynomial degrees of interest. The t2_instance represents the resource-

constrained platform, whereas the c5n_large and r5a_large instances represent high-performance computation and memory optimized platforms, respectively.

These instances were also used to benchmark our client-server model, for the third project goal. For a client-server model, one can theoretically separate the computational requirements, to allow for a small SWaP platform to use a homomorphic encryption scheme for enhanced privacy and security. We benchmarked our client-server implementation using the modified SEAL libraries against the baseline data collected during the implementation of the AWS EC2 instances, with all computation running on a single platform instance. The implementation and results are discussed in the next section.

3.2 Experiment Implementation

3.2.1 Low SWaP Platform Implementation

After examining potential low SWaP platforms, we first selected the DE10 Nano SoC, a development kit with a hybrid ARM processor and FPGA Cyclone V chip, manufactured by Intel, shown in Figure 1.

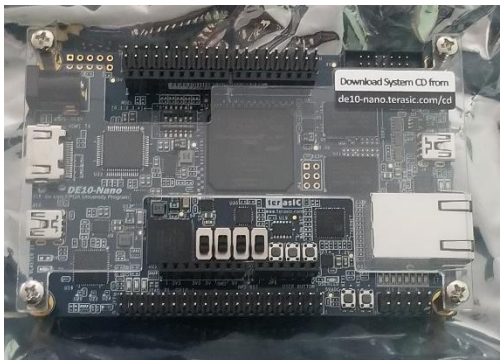

Figure 1: Intel DE10 Nano SoC, dual-core Cortex A9, 32-bit ARM processor, Cylcone V FPGA with 110,000 Programmable Logic Elements (PLEs).

The link to the product page is in 7.0 Appendix A: Product & Application Links. We used a Yocto Poky Linux OS booted from a 32GB µSD card slot on the board. The development board also had 1GB DDR3 memory accessible by the processor and FPGA fabric. Our interface to the board was Ethernet. A top level functional block diagram is shown in Figure 2.

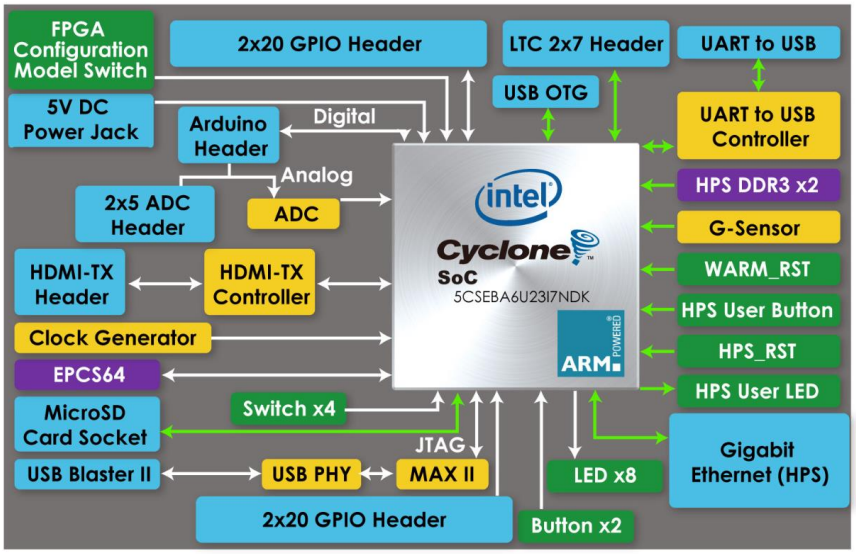

Figure 2: Top level block diagram of the DE10 Nano SoC development kit.

We first attempted to instantiate the SEAL libraries using the Linux kernel running on the ARM Cortex A9 processor. However, we quickly learned that many of the SEAL library functions are not compatible with a 32-bit instruction set, so our processor was unable to compile and run any meaningful data. We then attempted to port the C-based SEAL library to HDL using the MATLAB and Simulink DSP and embedded processing toolboxes. This instantiation exceeded the PLE capacity of the FPGA gate fabric, so we determined that this first processor selection, chosen for its known usage by the European Space Agency for LEO orbits and US cubesats and small space payload platforms, was insufficient for homomorphic encryption schemes. For reference, both the SEAL and HElib libraries require 64-bit processor capabilities.

We switched then to an alternate, a Raspberry Pi 4, with the 4GB RAM option. The board used for this experiment is shown in Figure 3.

Figure 3: Raspberry Pi 4, 4GB RAM, Cortex A78 quad-core 64-bit ARM processor.

The Raspberry Pi 4 with 4GB RAM has a 64-bit ARM Cortex A78 quad core 64-bit processor, which was determined to be sufficient for running the majority of functions of the SEAL libraries. A top level diagram of the Raspberry Pi 4 development kit is shown in Figure 4.

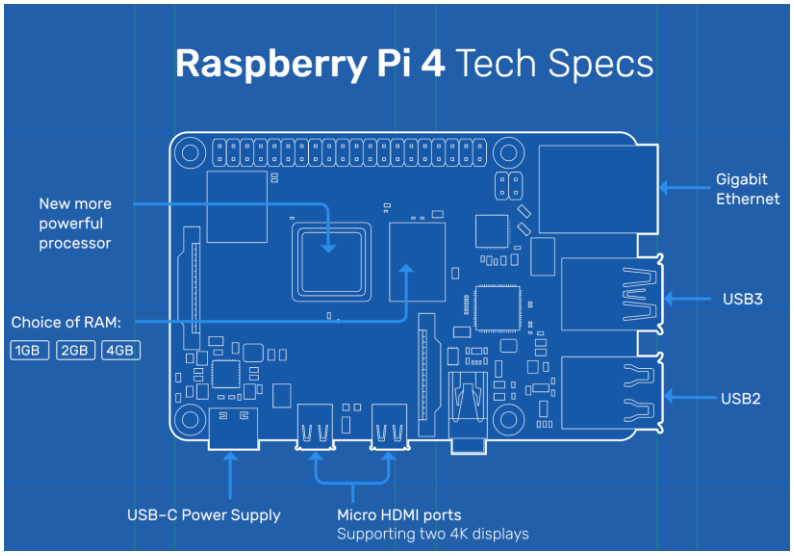

Figure 4: Top level block diagram of basic features for the Raspberry Pi 4 development kit.

The OS was loaded onto a µSD card, using a modified Raspbian image provided at the link in 7.0 Appendix A: Product & Application Links. We opened an SSH terminal using PuTTY to communicate with the Raspberry Pi.

The SEAL library functions were run on the Raspberry Pi 4 successfully, with data presented in 4.1 Low SWaP Results. Full functionality for the homomorphic encryption functions, including key generation, encryption, addition and multiplication functions on the ciphertext, and decryption, were successfully performed on the Raspberry Pi 4 for polynomial degrees up to and including a polynomial degree of 16384.

3.2.2 Amazon Webservices Elastic Cloud Computing Implementation

For objective 2, we implemented a series of AWC EC2 instances to compare a general purpose, computation optimized and memory optimized high-performance computing options for hosting homomorphic encryption schemes. A view of the AWS EC2 dashboard is shown in Figure 5.

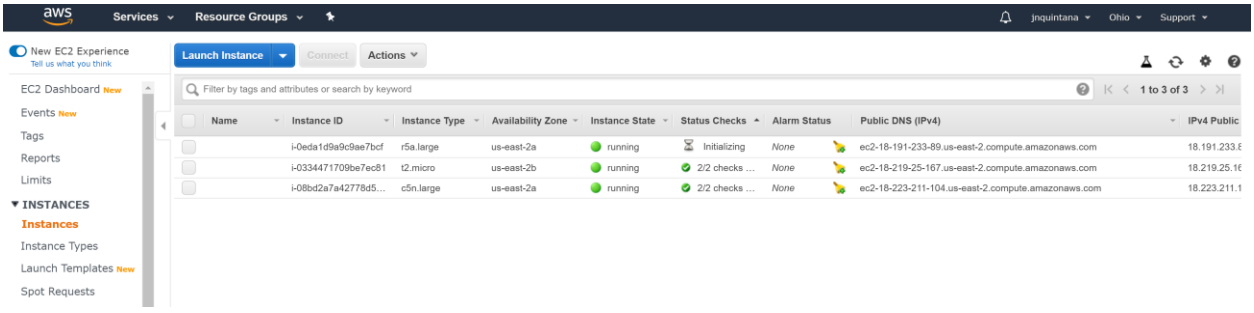

Figure 5: AWS EC2 dashboard with multiple instances used in the implementation and characterization of the homomorphic encryption schemes provided by SEAL. AWS EC2 account shown is a screenshot from J. Quintana.

We instantiated the instances using an Ubuntu 18.04 server image and accessed them via SSH from our local machines. We installed the SEAL libraries and performed all benchmarking via the SSH session.

3.2.3 Client-Server Implementation

After benchmarking the performance both with the low SWaP embedded device in objective one and the cloud computing service instances in objective two, we sought to improve the performance of the homomorphic encryption library by distributing the computational requirements between a resource-constrained client and a high-performance server. We added a series of options to the SEAL libraries and used the c5n_large to test the client-server model, compared to the benchmarks obtained under objective two. The added performance test options are shown in Figure 6.

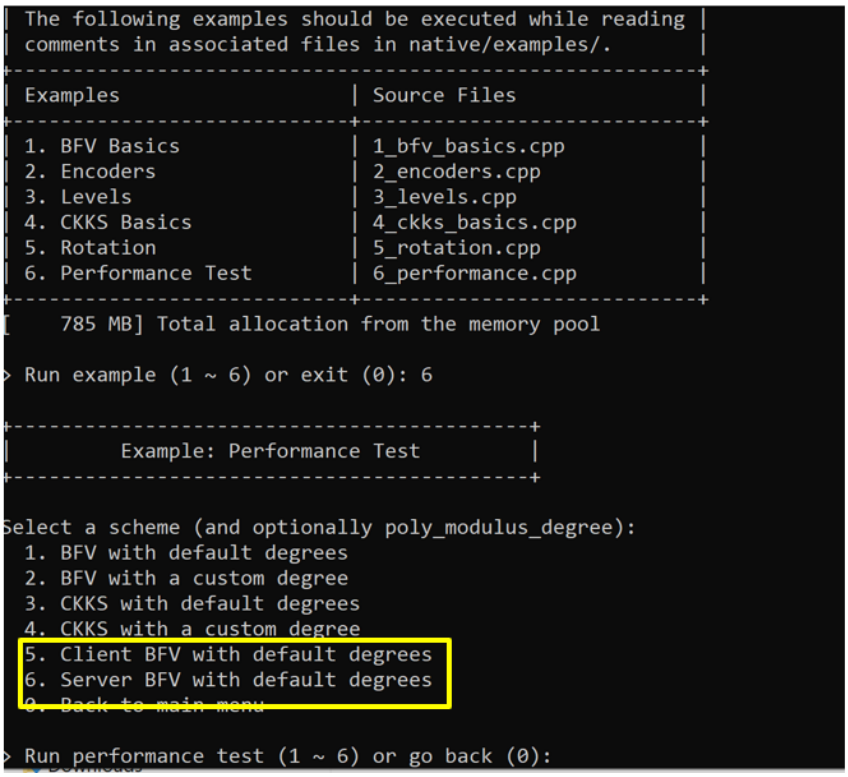

Figure 6: Client-server implementation benchmarking options, in SEAL Performance Test menu.

The client-server implementation had limited success over WiFi with the Raspberry Pi to AWS EC2 instance, so much of the benchmarking was performed between AWS EC2 instances. Improvements on this implementation are suggested in 5.0 Summary & Future Work.

4.0 Experiment Results

Metrics used to evaluated the performance of the different instantiations of the SEAL library and its modified client-server model included average computation time for various functions computed during the performance tests for different selected polynomial degrees. We also looked at CPU usage percent using the basic Linux diagnostic tools, but this measurement, though averaged over multiple test cycles, is crude and should not be used as a determination in our success criteria.

Performance metrics for objectives one and two were obtained by J. Quintana. Performance metrics for objective three were obtained by D. Achee. All results were obtained using an SSH connection to each platform. Any differences resulting from using two computers to obtain data were determined negligible. D. Achee also obtained data for the AWS EC2 instances, which correlated with the averages obtained by J. Quintana for the same instance type.

4.1 Low SWaP Results

The Raspberry Pi instantiation was able to successfully run all performance tests, with the exception of those run for the polynomial degree option 32,768 (2^{15}). The performance tests were run for both BFV and CKKS encryption schemes provided in the SEAL library functions. The computation time in milliseconds for each function instrumented is shown as a function of polynomial degree choice in Table 3 for the BFV encryption scheme implementation and in Table 4 for the CKKS encryption scheme implementation.

Table 3: Computation time [ms] of instrumented homomorphic encryption functions for BFV encryption schemes provided by the SEAL library, computed using the Raspberry Pi 4.

#	Parameter	SEAL BFV Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	0.233	0.466	0.984	2.092	4.435
2	Average Unbatch	0.186	0.344	0.740	1.569	3.373
3	Average Encrypt	3.329	6.524	22.071	64.670	219.878
4	Average Decrypt	1.566	2.417	7.137	26.089	100.465
5	Average Add	0.025	0.036	0.116	0.498	2.047
6	Average Multiply	11.460	22.102	74.159	279.802	1,146.392
7	Average Multiply Plain	1.006	2.076	8.810	37.527	158.410
8	Average Square	8.252	16.237	55.181	209.815	867.853
9	Average Relinearize	N/A	N/A	16.476	72.648	396.322
10	Average Rotate 1 step	N/A	N/A	16.613	73.975	400.704
11	Average Rotate Random	N/A	N/A	53.823	341.561	1,765.455
12	Average Rotate Columns	N/A	N/A	16.624	73.992	400.718
13	CPU Usage [%]	5.0 %	10.6%	87.7%	99.7%	100%

Table 4: Computation time [ms] of instrumented homomorphic encryption functions for CKKS encryption schemes provided by the SEAL library, computed using the Raspberry Pi 4.

#	Parameter	SEAL CKKS Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	1.801	2.969	7.514	20.257	61.416
2	Average Unbatch	2.594	4.250	14.735	57.825	257.827
3	Average Encrypt	3.016	4.455	21.417	66.505	232.468
4	Average Decrypt	0.148	0.215	0.842	3.325	13.085
5	Average Add	0.027	0.036	0.114	0.446	1.998
6	Average Multiply	0.427	0.729	2.951	12.627	48.889
7	Average Multiply Plain	0.186	0.316	1.243	4.921	19.506
8	Average Square	0.302	0.517	2.137	9.173	36.203
9	Average Relinearize	N/A	N/A	16.323	71.417	391.350
	Average Rescale	N/A	N/A	6.235	30.999	140.706
10	Average Rotate 1 step	N/A	N/A	16.734	73.160	399.765
11	Average Rotate Random	N/A	N/A	57.636	279.600	1,790.253
12	Average Complex Conjugate	N/A	N/A	16.684	73.036	398.881
13	CPU [%]	7.0%	11.6%	81.1%	98.3%	100%

As we would expect, the Raspberry Pi 4 computation time increased exponentially with increasing polynomial degree used. Note that for some low polynomial cases (1024 and 2048), some functions are not computed.

For the functions add, multiply (ciphertext), multiply (plaintext), and square, the computation time for the BFV and CKKS implementations are shown in Figure 7.

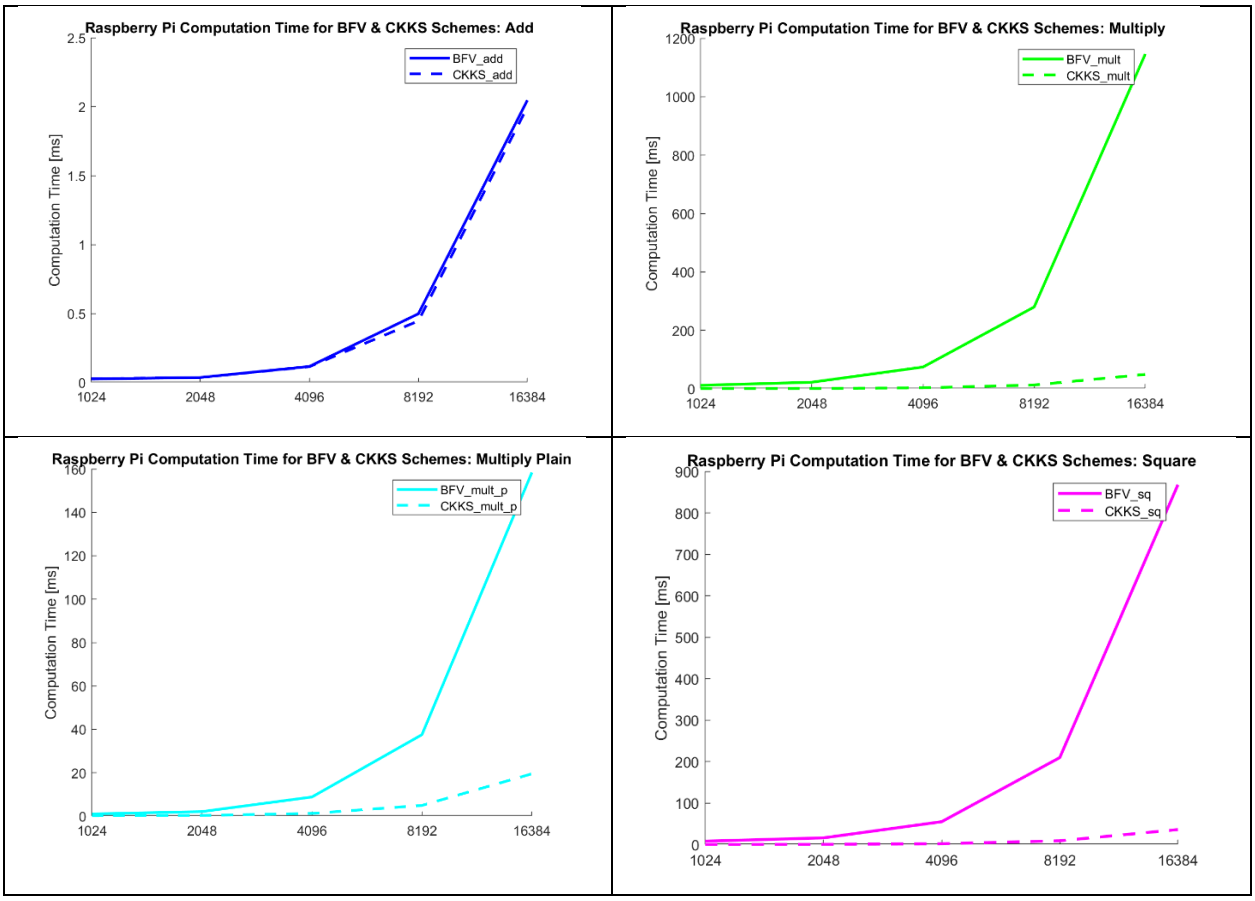

Figure 7: BFV and CKKS encryption scheme computation times for Raspberry Pi 4 low-SWaP implementations.

The BFV and CKKS computation time for the addition functions track closely to one another, but the multiplication and squaring functions for CKKS complete with much less time than those equivalent functions for BFV. The BFV functions used in the SEAL libraries use a tuple of the polynomial which differs from the textbook implementation and may contribute to this difference. It should be noted that the computation time for the ciphertext multiplication function (green) exceeds 1s in the BFV case, which can cause unacceptable delays for computation on large volumes of data. These results provide motivation for our attempts in objective three to reduce the computational costs on the client side.

4.2 AWS EC2 Results

The AWS EC2 instances (t2_micro, c5n_large, r5a_large) have similar performance characteristics to one another for both BFV and CKKS scheme implementations. A table of average computation results for all three instances with both BFV and CKKS implementations are shown in Table 5 through Table 10.

Table 5: Computation time [ms] of instrumented homomorphic encryption functions for BFV encryption schemes provided by the SEAL library, computed using the t2_micro AWS EC2 instance.

#	Parameter	SEAL BFV Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	0.024	0.050	0.129	0.273	0.539
2	Average Unbatch	0.026	0.048	0.105	0.214	0.453
3	Average Encrypt	0.537	1.034	3.034	8.262	26.371
4	Average Decrypt	0.118	0.234	0.764	2.671	11.488
5	Average Add	0.005	0.008	0.034	0.112	0.430
6	Average Multiply	1.103	2.180	7.349	27.757	116.834
7	Average Multiply Plain	0.113	0.234	0.931	4.047	19.624
8	Average Square	0.754	1.509	5.038	20.058	82.345
9	Average Relinearize	N/A	N/A	1.496	7.331	42.251
10	Average Rotate 1 step	N/A	N/A	1.549	7.354	43.365
11	Average Rotate Random	N/A	N/A	6.213	31.027	205.262
12	Average Rotate Columns	N/A	N/A	1.493	7.360	42.124
13	CPU Usage [%]	1%	2.3%	16.6%	51.8%	89.7%

Table 6: Computation time [ms] of instrumented homomorphic encryption functions for CKKS encryption schemes provided by the SEAL library, computed using the t2_micro AWS EC2 instance.

#	Parameter	SEAL CKKS Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	0.496	1.055	2.467	6.197	17.045
2	Average Unbatch	0.558	1.182	3.273	11.141	46.447
3	Average Encrypt	0.508	0.950	3.376	9.462	31.022
4	Average Decrypt	0.015	0.026	0.112	0.405	1.705
5	Average Add	0.006	0.009	0.030	0.105	0.418
6	Average Multiply	0.053	0.088	0.345	1.364	5.432
7	Average Multiply Plain	0.014	0.023	0.091	0.375	1.453
8	Average Square	0.035	0.064	0.251	1.015	4.191
9	Average Relinearize	N/A	N/A	1.530	7.811	43.702
	Average Rescale	N/A	N/A	0.733	3.516	15.965
10	Average Rotate 1 step	N/A	N/A	1.750	8.344	47.054
11	Average Rotate Random	N/A	N/A	7.205	31.284	227.630
12	Average Complex Conjugate	N/A	N/A	1.757	8.244	46.406
13	CPU [%]	1.0%	1.7%	15.0%	46.2%	100%

Table 7: Computation time [ms] of instrumented homomorphic encryption functions for BFV encryption schemes provided by the SEAL library, computed using the c5n_large AWS EC2 instance.

#	Parameter	SEAL BFV Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	0.025	0.054	0.105	0.213	0.425
2	Average Unbatch	0.023	0.039	0.083	0.168	0.352
3	Average Encrypt	0.421	0.786	2.274	6.227	19.964
4	Average Decrypt	0.082	0.165	0.541	1.920	7.527
5	Average Add	0.004	0.006	0.018	0.073	0.309
6	Average Multiply	0.747	1.571	5.255	20.001	81.804
7	Average Multiply Plain	0.081	0.169	0.712	2.999	12.707
8	Average Square	0.522	1.071	3.612	13.791	57.450
9	Average Relinearize	N/A	N/A	1.078	5.273	31.282
10	Average Rotate 1 step	N/A	N/A	1.083	5.310	31.056
11	Average Rotate Random	N/A	N/A	4.467	20.178	136.762
12	Average Rotate Columns	N/A	N/A	1.082	5.291	30.989
13	CPU Usage [%]	1.0%	1.7%	12.0%	36.7%	100%

Table 8: Computation time [ms] of instrumented homomorphic encryption functions for CKKS encryption schemes provided by the SEAL library, computed using the c5n_large AWS EC2 instance.

#	Parameter	SEAL CKKS Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	0.364	0.788	1.802	4.487	12.292
2	Average Unbatch	0.390	0.831	2.325	8.039	33.520
3	Average Encrypt	0.422	0.792	2.643	7.334	23.595
4	Average Decrypt	0.009	0.016	0.063	0.245	1.041
5	Average Add	0.004	0.006	0.017	0.068	0.284
6	Average Multiply	0.024	0.048	0.179	0.841	3.900
7	Average Multiply Plain	0.009	0.016	0.060	0.239	0.950
8	Average Square	0.016	0.032	0.120	0.552	2.763
9	Average Relinearize	N/A	N/A	1.097	5.357	31.338
	Average Rescale	N/A	N/A	0.519	2.574	11.756
10	Average Rotate 1 step	N/A	N/A	1.280	6.155	34.147
11	Average Rotate Random	N/A	N/A	4.458	28.515	145.624
12	Average Complex Conjugate	N/A	N/A	1.275	6.045	33.749
13	CPU [%]	0.7%	1.3%	11.0%	41.3%	100%

Table 9: Computation time [ms] of instrumented homomorphic encryption functions for BFV encryption schemes provided by the SEAL library, computed using the r5a_large AWS EC2 instance.

#	Parameter	SEAL BFV Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	0.056	0.056	0.124	0.273	0.513
2	Average Unbatch	0.022	0.046	0.091	0.196	0.451
3	Average Encrypt	0.482	0.902	2.604	7.491	24.381
4	Average Decrypt	0.102	0.186	0.632	2.374	9.425
5	Average Add	0.005	0.007	0.020	0.083	0.327
6	Average Multiply	1.002	1.986	6.581	25.472	103.115
7	Average Multiply Plain	0.102	0.206	0.930	3.827	16.475
8	Average Square	0.676	1.390	4.609	18.096	74.667
9	Average Relinearize	N/A	N/A	1.270	6.631	39.109
10	Average Rotate 1 step	N/A	N/A	1.322	6.585	39.271
11	Average Rotate Random	N/A	N/A	4.299	24.564	162.075
12	Average Rotate Columns	N/A	N/A	1.287	6.708	39.196
13	CPU Usage [%]	1.7%	2.3%	14.0%	63.1%	98%

Table 10: Computation time [ms] of instrumented homomorphic encryption functions for CKKS encryption schemes provided by the SEAL library, computed using the r5a_large AWS EC2 instance.

#	Parameter	SEAL CKKS Degree Computation Time [ms]				
Degree		1024	2048	4096	8192	16384
1	Average Batch	0.479	1.080	2.227	5.365	14.436
2	Average Unbatch	0.536	1.102	3250	10.403	44.064
3	Average Encrypt	0.481	0.855	3.145	8.735	28.172
4	Average Decrypt	0.012	0.022	0.091	0.302	1.229
5	Average Add	0.004	0.007	0.022	0.075	0.311
6	Average Multiply	0.033	0.066	0.268	0.984	4.578
7	Average Multiply Plain	0.013	0.024	0.100	0.351	1.500
8	Average Square	0.022	0.045	0.179	0.677	3.103
9	Average Relinearize	N/A	N/A	1.449	6.599	38.316
	Average Rescale	N/A	N/A	0.667	3.164	14.578
10	Average Rotate 1 step	N/A	N/A	1.623	7.275	41.515
11	Average Rotate Random	N/A	N/A	5.934	28.260	181.675
12	Average Complex Conjugate	N/A	N/A	1.579	7.215	41.180
13	CPU [%]	0.7%	2%	13.3%	48.7%	93.4%

The add, multiply (ciphertext), multiply (plaintext), and square functions are compared for the t2_micro resource-constrained instance, which is most comparable to a small SWaP platform, and the c5n_large and r5a_large optimized instances. A graphical comparison of the results is shown in .

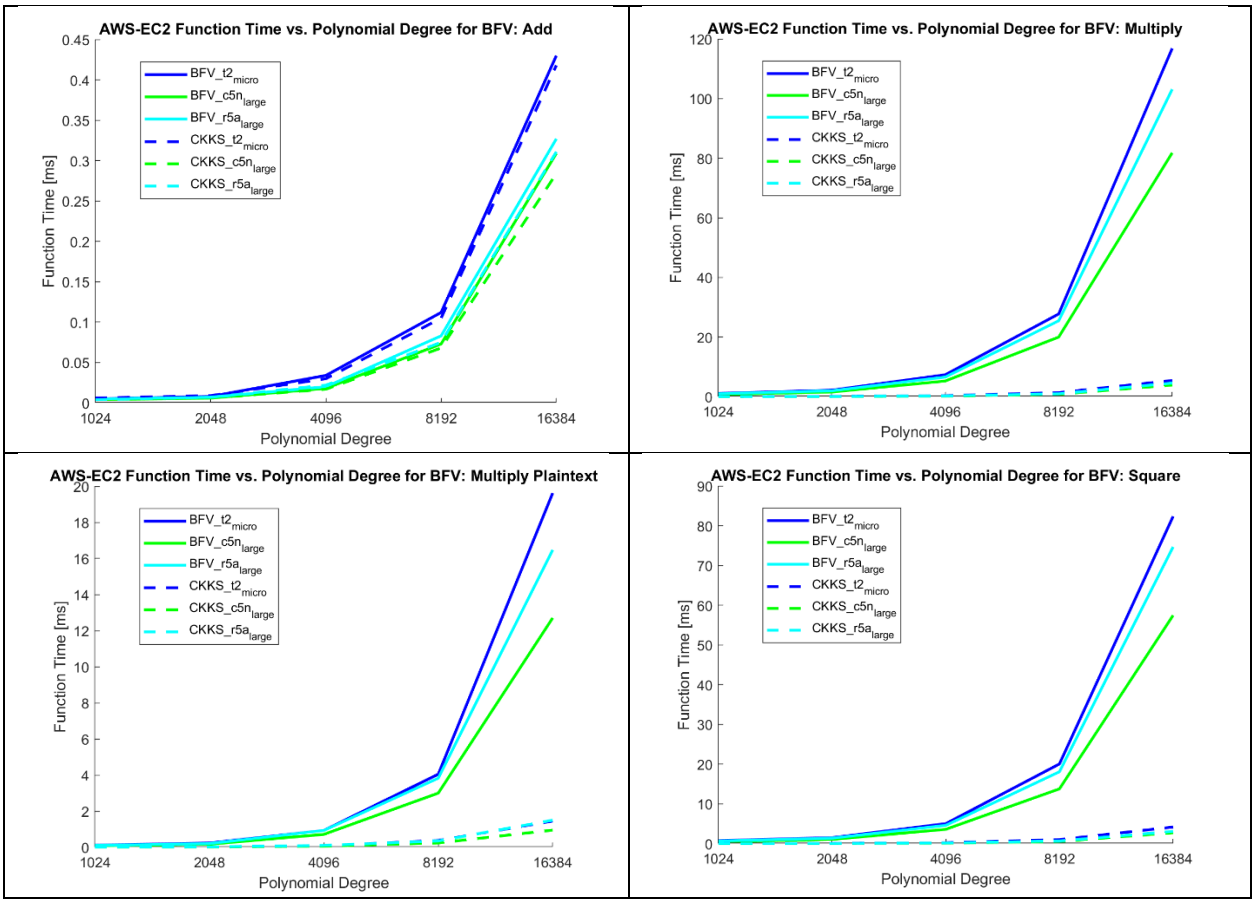

Figure 8: BFV and CKKS encryption scheme computation times for AWS EC2 instances.

Similar to the low SWaP Raspberry Pi results, the AWS EC2 instances track well to one another for the add function, but the CKKS implementation of multiply (ciphertext and plaintext) and square functions are more computationally efficient than their BFV equivalents. Even though the t2_micro instance is significantly more resource constrained than the high-performance optimized c5n_large and r5a_large instances, their computation times are closely grouped.

The AWS EC2 instance computation times are overlaid with the Raspberry Pi 4 computation times from objective 1. These are shown in Figure 9.

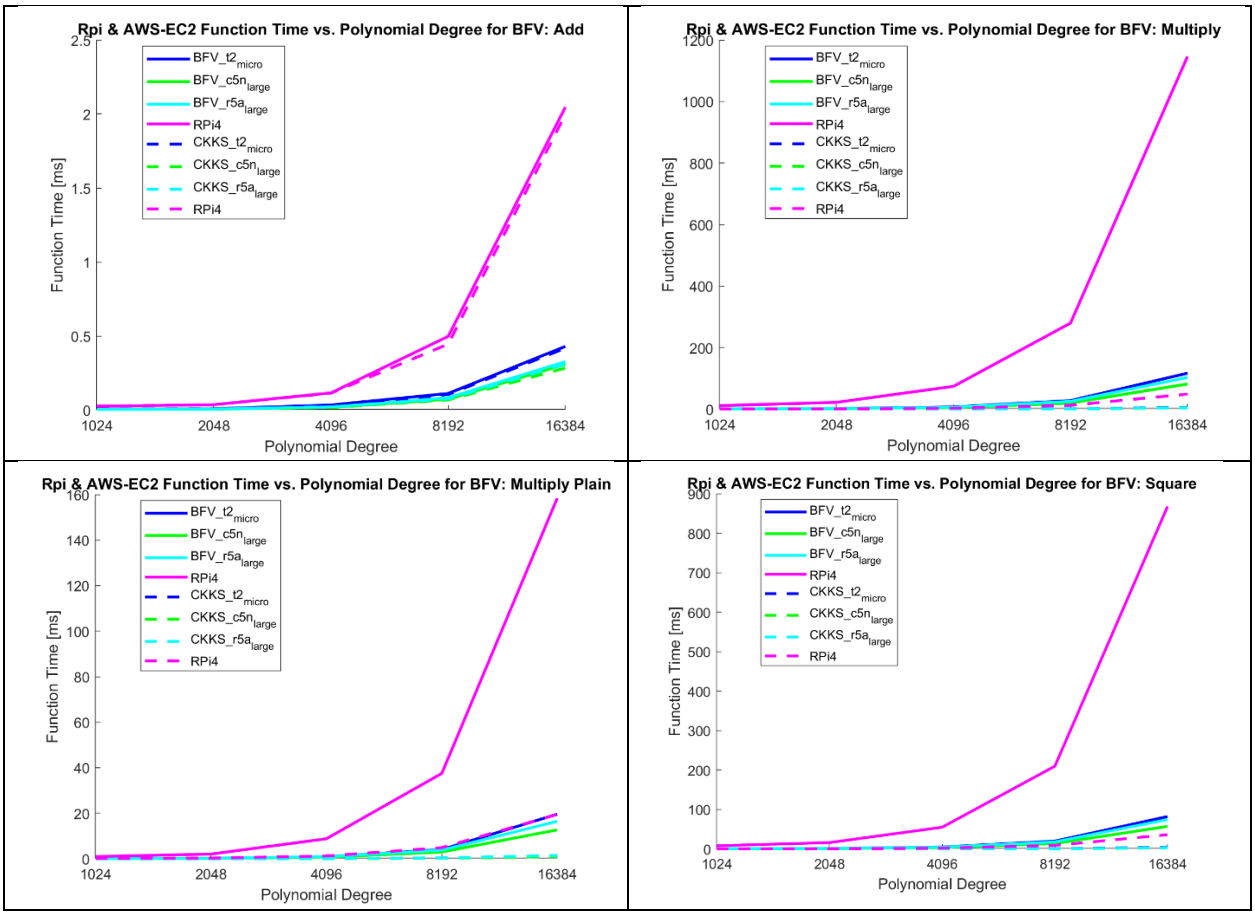

Figure 9: BFV and CKKS encryption scheme computation times for Raspberry Pi 4 and AWS EC2 instances.

The Raspberry Pi's implementation of the BFV encryption scheme, takes significantly longer for higher polynomial degrees than the AWS EC2 instances for the equivalent computation. For the CKKS implementation, the multiplication and square functions for both AWS EC2 and Raspberry Pi implementations track closely to one another. For the add function, both the BFV and CKKS implementations on the Raspberry Pi are more computationally costly than the equivalent computation on the AWS EC2 instances.

Figure 9's graphical comparison of the Raspberry Pi and AWS EC2 implementations helps to inform our implementation of the client-server model. We postulate that we can improve the most by optimizing the BFV scheme for the client-server model because the disparity in computation times between the Raspberry Pi 4 and the high-performance AWS EC2 instances is greatest for the BFV scheme for all four functions analyzed.

4.3 Client-Server Implementation Results

The client server implementation was performed on the BFV encryption scheme, based on our analysis from objective 2. A screenshot of the console view for the updated client-server computation is shown in Figure 10.

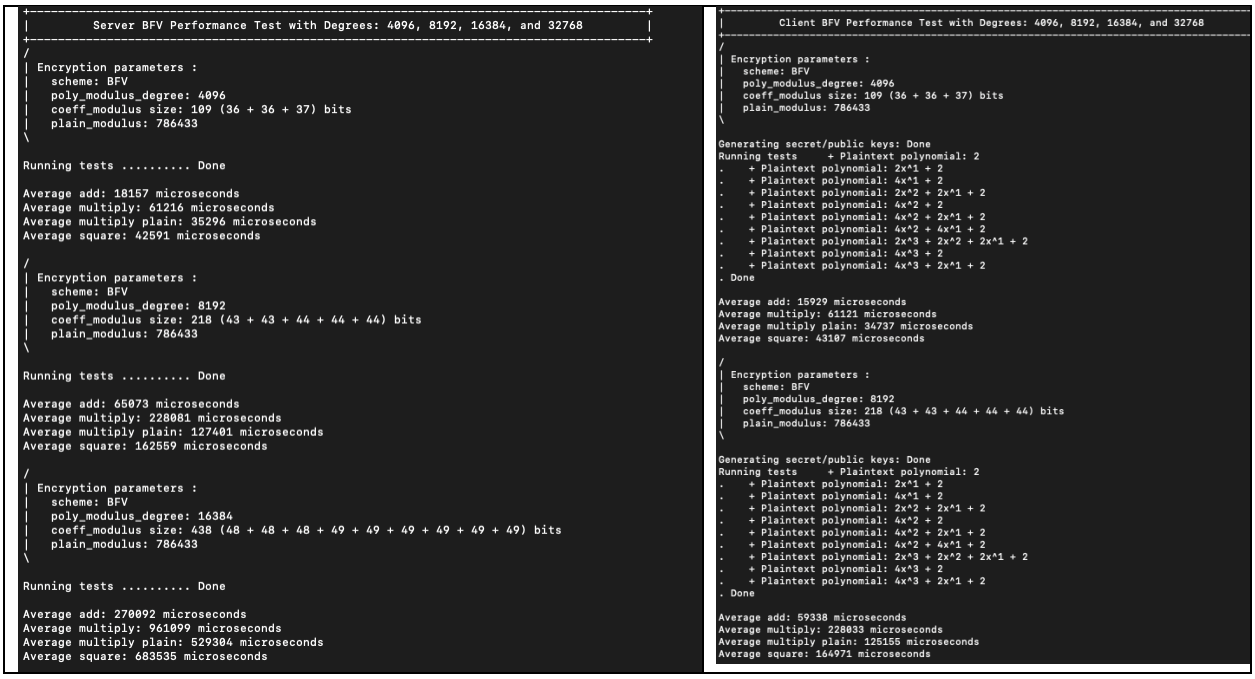

Figure 10: Console view of client server implementation, taken with c5n_large AWS EC2 instance.

For the client-server implementation, the code was instrumented in the same manner for computation time measurements. We included the network traffic time in this calculation because in terms of total cost of the system, network traffic time must be considered when being compared to an entirely local computation. Our results were different than expected, when compared to the baseline local computation. We expected a similar trend in computation time versus polynomial degree, simply shifted vertically to account for the distributed computation between the client and server. However, what we saw was an increased computation time contribution from the sharing of information between client and server, which was greater than the local computation time for low polynomial degree. We think the contribution of the serialized and transmitted data exceeds the computational time for the function itself. These results are shown in Figure 11.

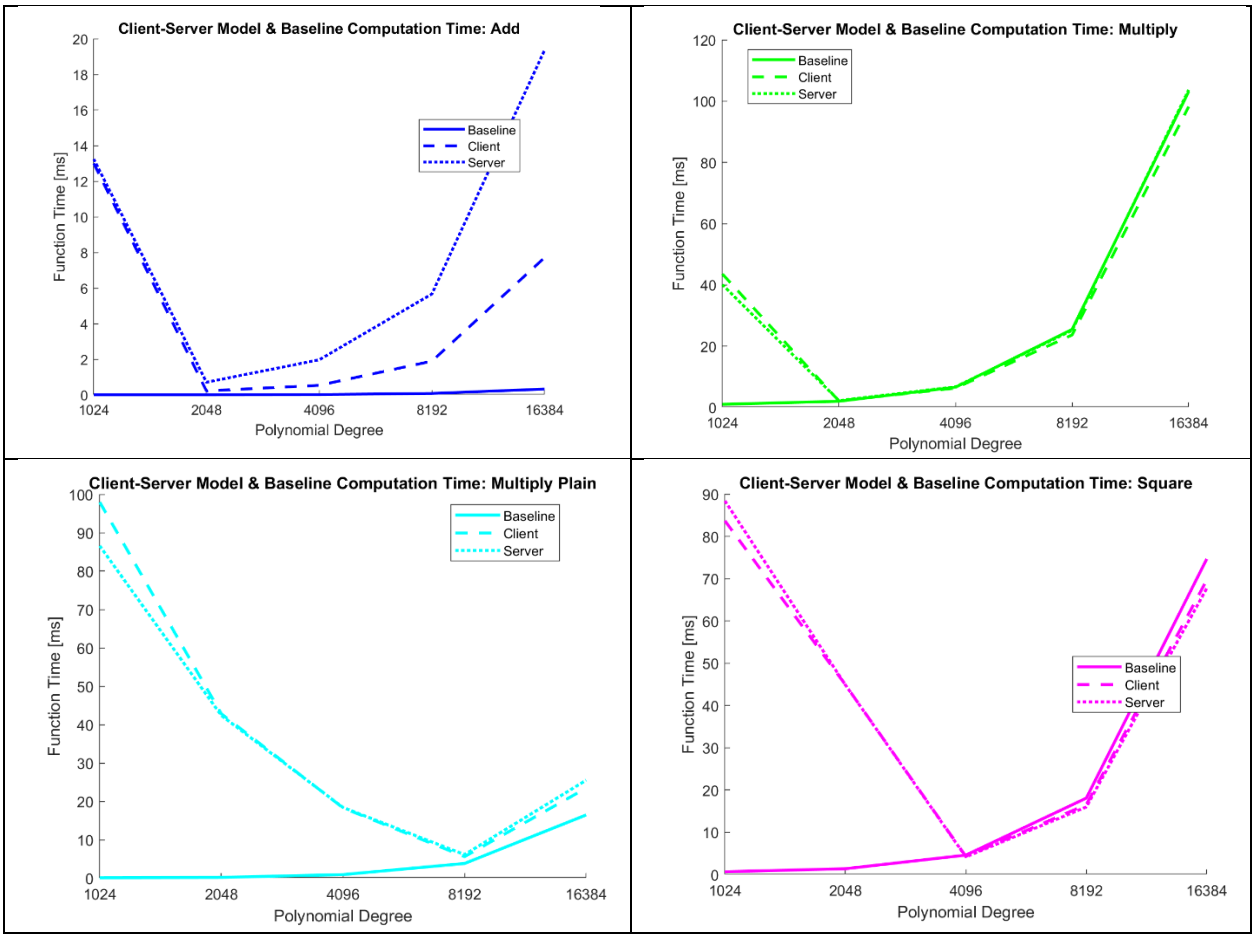

Figure 11: Baseline vs. client-server model computation time for add, multiply (ciphertext), multiply (plaintext), and square functions using the modified SEAL library functions.

We conclude that the serialization and transmission of the data contributes to the total computation time for each function in a manner than exceeds the benefit of distributing the computation between platforms. Future work to improve the transfer of data between platforms may reduce this cost, so that the client-server model yields improved results. The size of the object in bytes for each polynomial degree is shown in Table 11.

Table 11: Object size vs. polynomial degree.

Polynomial Degree	1024	2048	4096	8192	16384
Object Size [bytes]	16,489	32,873	131,177	524,393	2,097,257

The client-server model unfortunately does not show improvement over the baseline local computation method. Future work to optimize the SEAL library as well as a client-server model may in the future make that model more feasible. However, at this time, it is not a viable alternate solution to the baseline local computational model.

4.4 Related Work

Some groups have performed similar low SWaP analyses and implementations of homomorphic encryption with the motivation to improve the security of IoT devices. One group performing related

work is led by A. Prasitsupparote from the Security Fundamentals Laboratory in Japan [9]. Their experiment compared the HElib and SEAL libraries to implement a fully homomorphic encryption scheme on a privacy preserving protocol for wearable devices in healthcare systems, shown in Figure 12 [9].

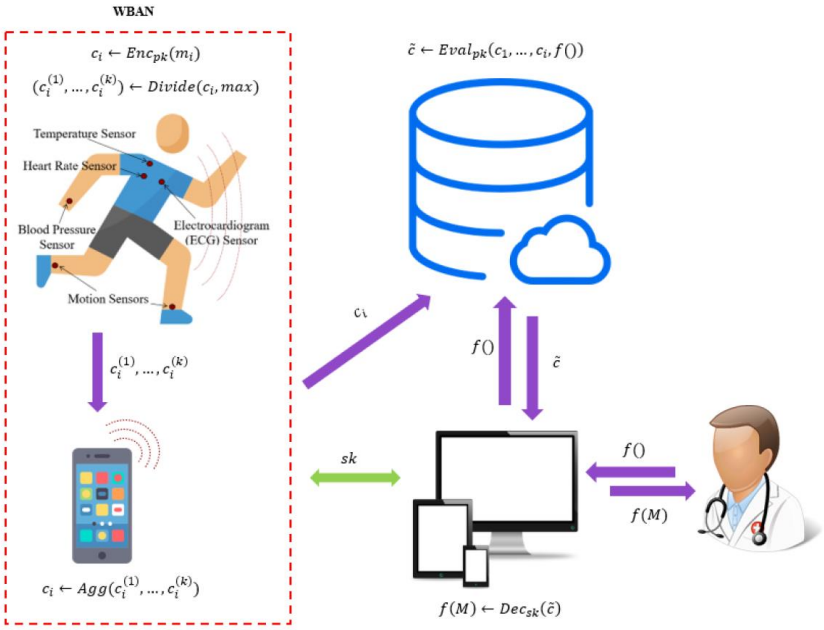

Figure 12: Wireless Body Area Network implementation of homomorphic encryption for IoT device security and protection of personal identifying information related to healthcare applications and health tracking [9].

The implementation by this group was performed on a PC and a Raspberry Pi combination. The number of homomorphic operations were limited by function, to account for the limited processing resources on the Raspberry Pi [9]. They separated and measured the computation and communication costs using OMNET++ and Castalia to compare the HElib and SEAL capabilities [9]. Computational delays were larger for the SEAL library than HElib, but SEAL provided more homomorphic computational capabilities [9]. The implementation results for the Raspberry Pi computation times were comparable to our results. The authors noted that delay times for this implementation was not practical for real-time bidirectional communication [9].

In addition to research work, there are solicitations for solutions to cybersecurity problems that can include homomorphic encryption. Broad Area Announcements (BAAs) for FY2020 – FY2024 posted include:

- Capabilities for Cyber Resiliency
- Foundations of Trusted Computational Information Systems
- Cyber Warfare Detachment

This is an active area of research, and with improved computational capability on embedded devices, may provide a practical solution for data security.

5.0 Summary & Future Work

Our experiment contained three objectives to investigate and characterize the implementation of a homomorphic encryption scheme on a small SWaP platform, on a cloud computing service, and via a client-server module, the latter used to investigate the potential gains of splitting the computation cost between platforms. We determined that the BFV implementation using the Microsoft SEAL library was more efficient on the AWS EC2 instances than on the Raspberry Pi, however, for all but the addition function, the Raspberry Pi implementation was comparable to the AWS EC2 implementations. We implemented a client-server model of the SEAL library and determined that when the computational and communication times are analyzed together, the total computation cost exceeds that of the equivalent computation performed on only the local instance.

Future work includes the optimization and improvement of the communication protocol between the client and server as well as potential optimization of memory allocation of the SEAL library itself. The advancement of computational capability in small SWaP devices has made homomorphic encryption a possibility for wearable and other IoT or small embedded devices.

6.0 References

[1] P.V. Parmar, et. Al., “Survey of Various Homomorphic Encryption Algorithms and Schemes”, *International Journal of Computer Applications*, (0975-8887), Vol. 81, No. 8, April 2014.

[2] C. Gentry, “Fully Homomorphic Encryption Using Ideal Lattices”, *Stanford University, IBM Watson*, 2009.

[3] K. Lauter, et. Al., “Can Homomorphic Encryption be Practical?” *Microsoft Research*, *Eindhoven University of Technology, University of Toronto*, 2011.

[4] N. Downlin, et. Al., “Manual for Using Homomorphic Encryption for Bioinformatics”, *Microsoft Research, Princeton University*, accessed Jan 2020.

[5] Z. Brakerski and V. Vaikuntanathan, “Efficient Fully Homomorphic Encryption from (Standard) LWE”, *SIAM Journal on Computing*, Vol. 43, No. 2, 831-871, 2014.

[6] J. Howe, et. Al., “On Practical Discrete Gaussian Samplers for Lattice-Based Cryptography”, *IEEE Transactions on Computers*, Vol. 67, No. 3, March 2018.

[7] S. Fau, et. Al., “Towards practical program execution over fully homomorphic encryption schemes”, *Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2013.

[8] K. Laine, et. Al., “Simple Encrypted Arithmetic Library 2.3.1”, *Microsoft Research*, 2018.

[9] A. Prasitsupparote, et. Al., “Implementation and Analysis of Fully Homomorphic Encryption in Wearable Devices”, *Proceedings of 4th International Conference on Information Security and Digital Forensics, ISDF2018*, Greece, 2018.

7.0 Appendix A: Product & Application Links

DE10 Nano SoC Development Kit

https://www.amazon.com/Terasic-Technologies-P0496-DE10-Nano-Kit/dp/B07B89YHSB/ref=sr_1_fkmr1_1?dchild=1&keywords=de10+nano+sock&qid=1584739601&sr=8-1-fkmr1

CanaKit Raspberry Pi 4 Starter Kit – 4GB RAM

https://www.amazon.com/CanaKit-Raspberry-4GB-Starter-Kit/dp/B07V5JTMV9/ref=sr_1_3?dchild=1&keywords=raspberry+pi+4+canakit&qid=1584739700&sr=8-3

Raspbian Image

https://www.raspberrypi.org/downloads/

8.0 Appendix B: Instructions to Run SEAL Library Implementation

8.1 Installing Microsoft SEAL

Follow the instructions on https://github.com/Microsoft/SEAL to download the source code and install microsoft SEAL for your respective platform

8.2 Integrating Our Code

Let seal_location be the location of the SEAL library on your system

5.0 Download our github repo
6.0 $ cd client_server_performance_test
7.0 $ cp 6_performance.cpp seal_location/native/examples
8.0 $ cp examples.h seal_location/native/examples
9.0 $ cp examples.cpp seal_location/native/examples
10.0 $ cd seal_location/native/examples
11.0 $ make
12.0 $ cd ../bin

8.3 Running the Server Performance Test

It is important to always start the server before the client or else the client will report "Error connection refused".

$ ./sealexamples [portnumber]

where portnumber is the port that you want to the server to use.

Then select option 6 from both the first and second menu. It will hang until you start the client test.

Running the Client Performance Test

$ ./sealexamples [hostname] [portnumber]

where hostname is the IP address or DNS name of the server and server is the port used by the server

Currently the implementation is most stable when both the client and server are run using the same host. Therefore, please specify localhost as the hostname. Then select option 6 from the first menu and option 5 from the second menu.

8.4 Running Other Performance Tests

$ ./sealexamples

To run any of the other 10 performance tests you don't need to any command line arguments. Please select the respective test using the menu presented after starting the code.

9.0 Appendix C: List of Associated Files

README.txt
209AS_FinalPresentation.pptx
Client-Server Model Software Folder
Console Data Logs (text files)

https://github.com/daniel-achee/ece209-project.git

10.0 Project Timeline

- January 28th: Boards received from Amazon, initial firmware image compiled.
- February 4th: Generate preliminary design (unencrypted data sample run, metric data baseline established).
- February 18th: Midterm Presentation.
- February 20th: Received and Configured Raspberry Pi 4
- February 25th: Generate final design (contains both homomorphic encyrption analysis tools and plaintext tools). Obtain data on metrics for comparison (see next section)
- March 4th: Results comparison, generate final report