CALIFORNIA STATE UNIVERSITY SAN MARCOS

PROJECT SIGNATURE PAGE

PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

PROJECT TITLE: Partially Homomorphic Encryption Scheme for Real-Time Image Stream

AUTHOR: Angel Ivan Vazquez-Salazar

DATE OF SUCCESSFUL DEFENSE: August 10, 2020

THE PROJECT HAS BEEN ACCEPTED BY THE PROJECT COMMITTEE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE.

| | | |
|---|---|---|
| Ali Ahmadinia, Ph.D.<br>PROJECT COMMITTEE CHAIR | Ali Ahmadinia<br>SIGNATURE | 08/10/2020<br>DATE |
| Xin Ye, Ph.D.<br>PROJECT COMMITTEE MEMBER | Xin Ye<br>SIGNATURE | 08/10/2020<br>DATE |
| PROJECT COMMITTEE MEMBER | SIGNATURE | DATE |

# Partially Homomorphic Encryption Scheme for Real-Time Image Stream

Angel Vazquez-Salazar

California State University, San Marcos

Computer Science Department - Summer 2020

Academic Advisor: Dr. Ali Ahmadinia

Defense Date: Monday, 10 August 2020

Abstract

With over 20 billion internet of things (IoT) devices connected today, having the possibility to implement a low-cost security camera is now an option. Camera vendors can provide software to enable remote viewing anywhere around the world. However, the problem arises in security measure taken. By only providing encrypted services to authenticate a user by login credentials, there still is a vulnerability in raw stream. Newer types of intrusions like spoofing, or man in the middle attacks mean that once they have breached the account, they are able to gain full viewing access of the camera stream. However, by encrypting the stream of these IoT cameras, we have a chance to truly retain privacy and ownership of our data.

With IoT devices being limited with on-board memory and CPU speeds, this can be a challenge. To mitigate and offset any CPU intensive work, we will encrypt the data between our cloudlet device and our edge device by using a partially homomorphic encryption scheme called ”Elgamal encryption”. This CPU offloading allows the cloudlet to transmit streams at much higher framerate. The proposed system will utilize a Raspberry Pi 4, for our cloudlet with our desktop being the edge.

# Table of Contents

## Introduction

Live streaming has become an integral part in modern day society, whether it be via social media or home security applications. The availability of higher bandwidth networks offers superior connection speeds, allowing users to integrate multiple smart camera devices in their home network. As mentioned prior, the trade off with using smart devices is privacy. By connecting multiple smart devices into your network, you offer multiple ways that someone may be able to breach your network through a security flaw in the smart device.

Although there are multiple solutions on the market today to secure your network, there needs to be a way to properly protect the transfer of data within your security system. The ideal scenario to ensure that communication between your cloudlet device and edge will be protected from a breach would be a fully homomorphic encryption.

A fully homomorphic encryption system was first devised in 1978 [1] in order to create a high-level encryption scheme that supports both addition and multiplication operations on cipher texts for a time-shared computer service, similar to how enterprise solutions rely heavily on cloud-based computing and data storage. Sadly, at this time, it was almost impossible to implement being a NP-Hard complexity with the hardness being based on the ideal lattice.

In 2009, Craig Gentry [2] proposed a fully homomorphic encryption scheme off of lattice-based cryptography. In doing so, Gentry was able to make the encryption bootstrappable at every level, reducing the overall noise. In the second and third generation scheme, Gentry and other researchers were able to optimize this scheme further by reducing the noise while increasing efficiency and security. In the future generations, Gentry et al., was able to create a more efficient cryptosystem, to where high powered machines can make these computationally heavy encryption schemes usable.

Although a fully homomorphic encryption (FHE) scheme can be a possible solution for this security application, it is not feasible under current hardware restrictions. The goal of this project was to find a way to apply some form of encryption scheme over image streams while applying object detection algorithms to it. In doing so, we will be able to evaluate the limitations of encryption on a smart device and figure out how we can optimize our current tools to reach a more secure platform for future devices.

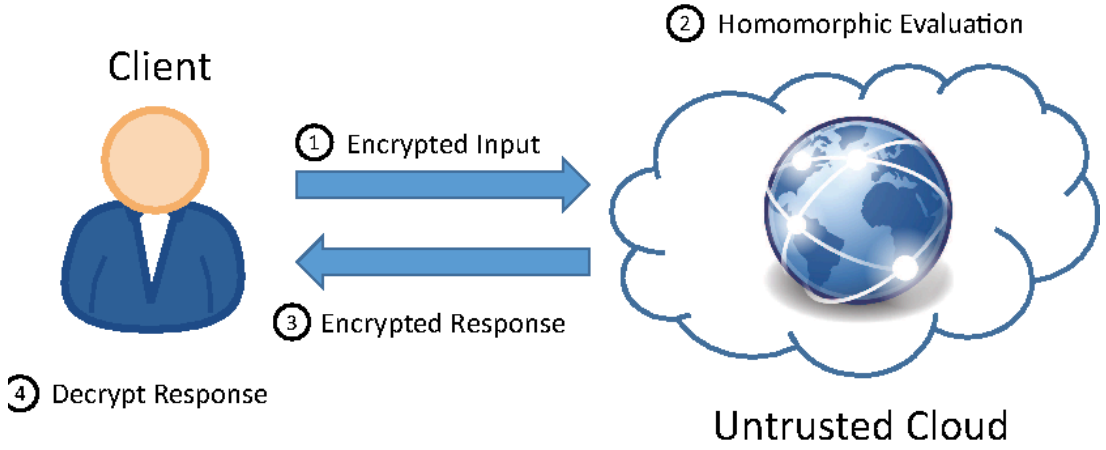


*Figure 1: Basic representation of how FHE works*

IoT Overview

IoT devices follow a form of IEEE 802.15 network protocol, or Zigbee communication, based off of the IEEE 802.15.4 network protocol. These network protocols enforce a maximum throughput while utilizing little to no resources. Everything from a smart watch to a fitness tracker or even smart lights are great examples of how lightweight these devices are in regard to CPU usage. By the IEEE standardization, these devices are able to adhere to the strict guidelines in an effort to make all devices as convenient for the user as possible. For instance, there are motion sensors with battery life extending beyond three or four years yet are able to be connected 24/7 and accessed remotely at the drop of a hat.

However, as local and regional network bandwidth increases and is able to handle higher traffic throughput, the usage of remote security devices has come on the rise. Inexpensive cameras such as Wyze Cam, offer a convenient end-to-end solution for people who desire to monitor their homes or other property at their leisure. For more tech savvy users, a Raspberry Pi, Asus TinkerBoard, or even an Arduino have been solid choices in devising a remote viewing scheme to enable users the option to build out their ideal security system. With predictions showing over 70 billion devices connected by 2025, one thing is clear, there needs to be security measures taken on any web connected device.

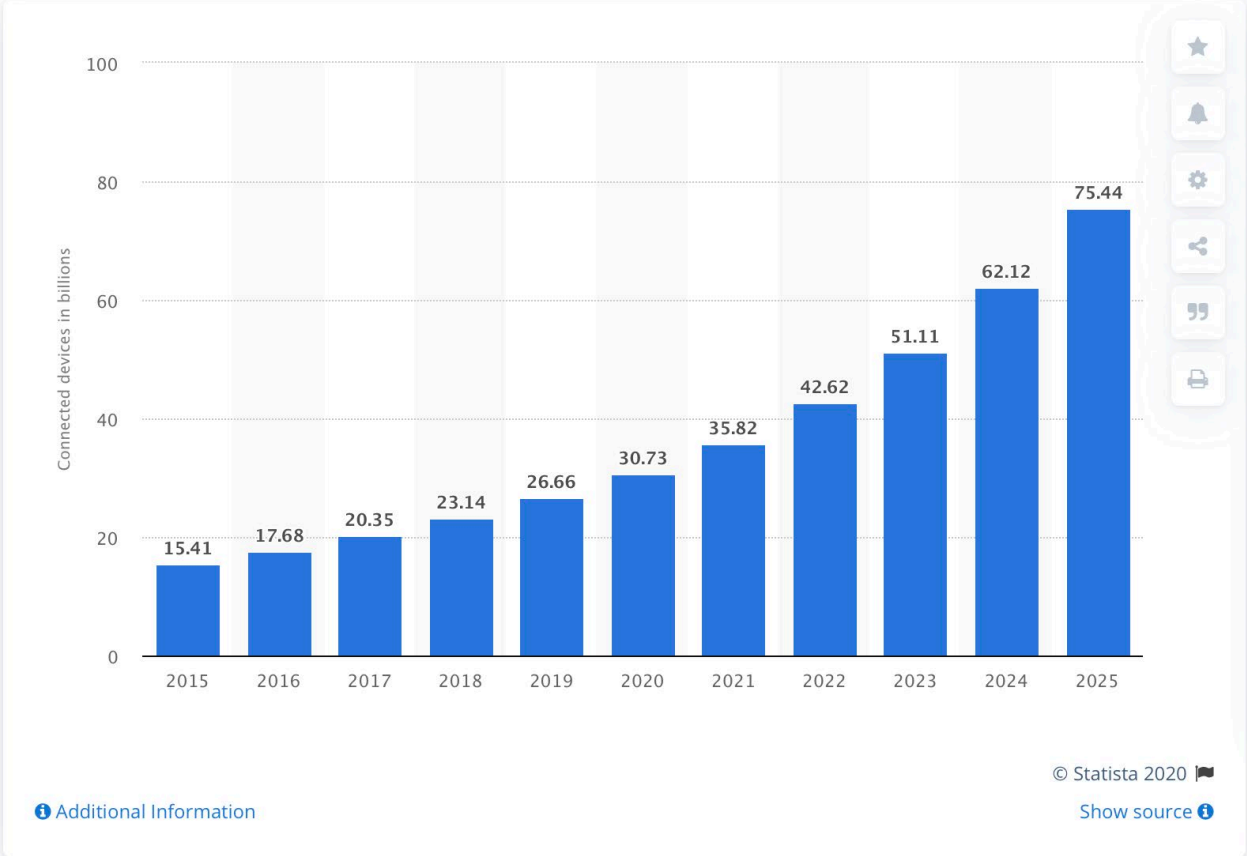


*Figure 2: Statista reports that as much as over 70 billion IoT devices will be connected to the world-wide-web by 2025!*

Some types of proposed intrusion detection systems (IDS) can be as simple as a relay that notifies a user when activity has increased, to using internal sensor temperature indicators to verify if the CPU workload is consistent with the average data usage, to even utilizing hardware sensors that cannot be bypassed via software! While all of these systems are great to utilize, users require a reliable, consistent, and privacy protecting measure.

Implementing a form of security system is difficult since IoT devices have only enough on-board storage space for their internal processes. These security systems are both costly to build and maintain, but also require an abundance of resources to allow real-time notifications. This, on top of having security cameras requiring more bandwidth will limit the total amount of network connected devices as well as reduce the overall upload speed provided by your internet service provider (ISP).

Encryption Overview

The practice of cryptography online has become the standard for privacy protection, although implementations of cryptography have been used in previous centuries. Arguably, the first widely known form of encryption was the Caesar cipher.

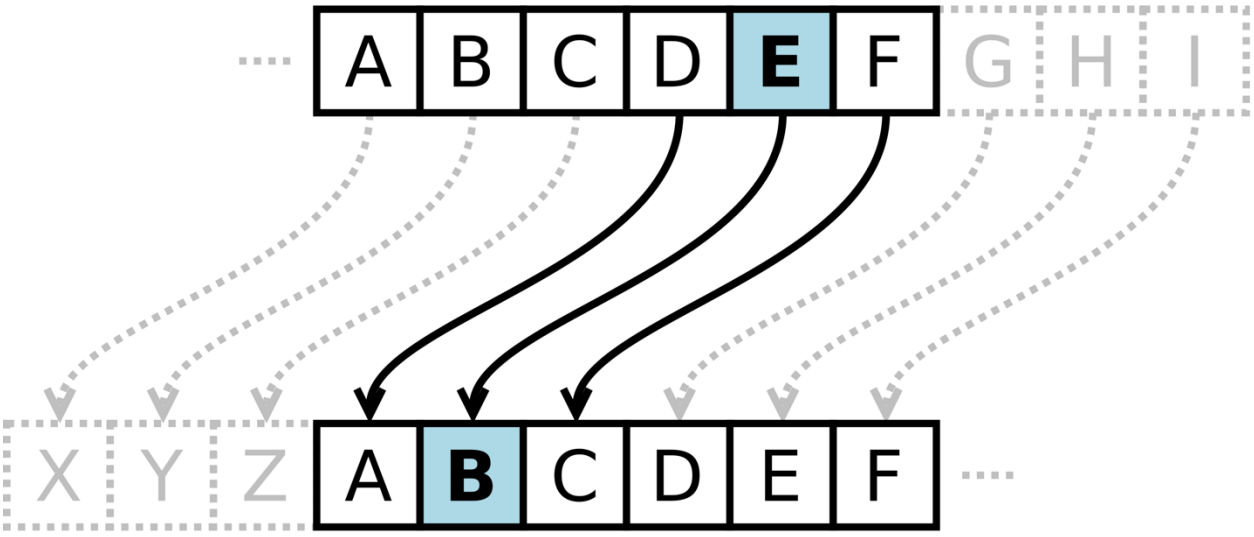


*Figure 3: Notice how the characters shifted by a value of 3?*

This style of encryption used a basic key that shifted the positions of characters either up or down the alphabet. In doing so, a simple text would be rendered useless to the naked eye. However, once the message is passed to a person with the decryption key, or the shift value in this case, the person would be able to decipher the text relatively easily to read.

A second option to solve the key issue is by brute force since the message may have been standard text, allowing the usage of a frequency chart in order to create a platform for decryption.

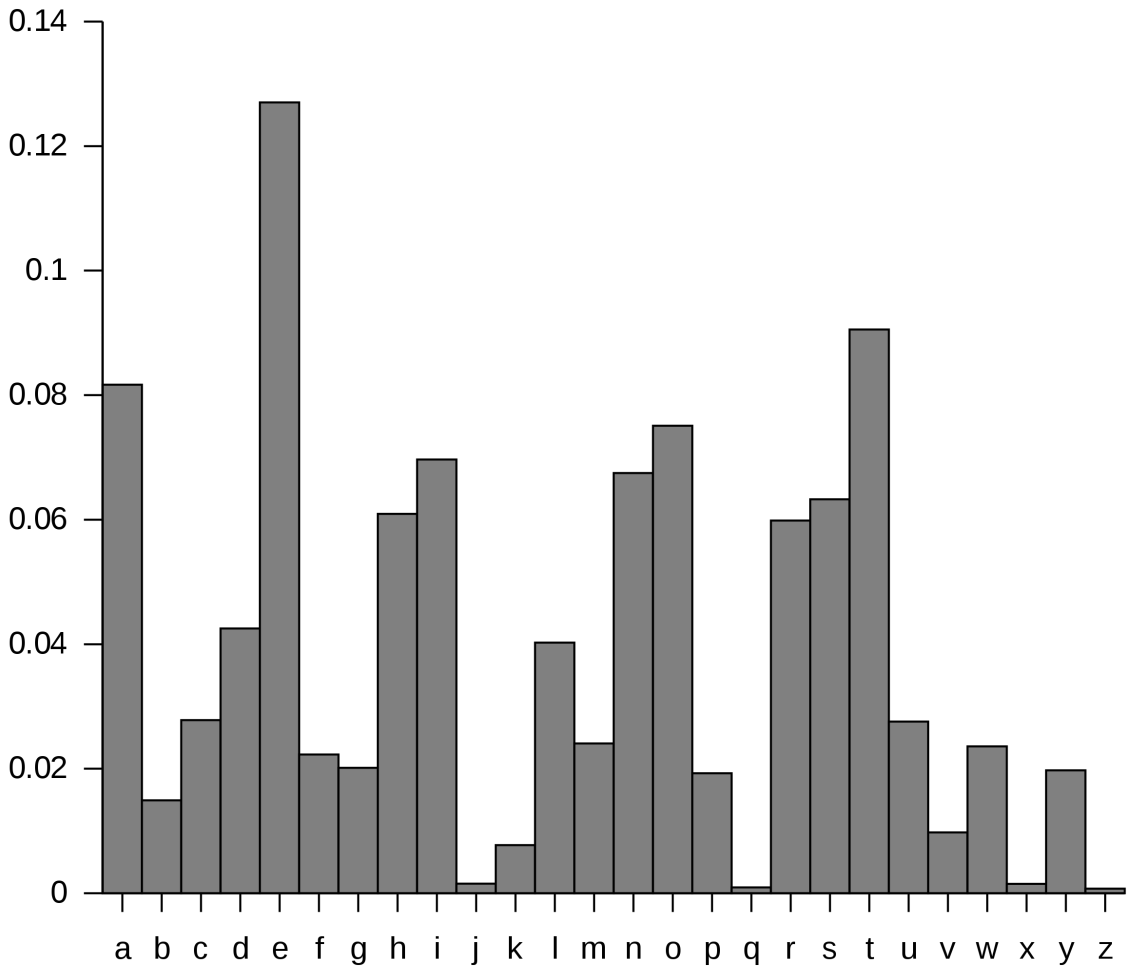


*Figure 4: As you can see, the most prevalent character in the English alphabet is 'e'*

However, in modern security practices, an RSA (Rivest–Shamir–Adleman) [9] key is considered industry standard alongside AES-256 [10]. Although these cryptography schemes are great for security measures for user accounts, a new issue begins, cloud computing.

With cloud computing, data privacy has become the primary focus of users. With big data and internet usage of any type, there will always be a copy of what search result was done or what website a user with IP Address 'X' went on for a designated time frame. This bodes very well for tech giants such as Alphabet's Google and YouTube, the top two most used search engines today. Artificial Intelligence as

well as Machine Learning mean that tech companies are better able to provide accurate statistics on user data, or as some would call it "creating a better user experience for all". For others, this is simply a burden and a violation of rights since they want to protect their privacy in any way possible. As much as big data aides in the development of better analytics, being spied on should not be a concern. This is why the FHE model works so well. It allows users to offload data to cloud servers that do all of the heavy analytical data, without bogging down local servers, yet is able to retain full privacy protection to users involved with data processing.

Project Materials

For the scope of this project, a few things were considered. First was cost of goods. This was not because of budgeting requirements more so finding a capable IoT device to serve as a cloudlet that can process video streams effectively while maintaining a minor overhead for net cost. The second factor in determining hardware was open source repositories. The Google Coral and hardware accelerator that are currently sold are on the bleeding edge of computing devices but is nowhere near the user base of Raspberry Pi [12], Arduino, or Asus TinkerBoard. Another limiting factor is the Google Coral required you use their pre-determined modeling techniques for analysis and has a limited core group that can provide support in the case that the device no longer works as needed. For this reason, the Raspberry Pi 8GB was chosen.

Having a light-weight device that offers a 4-core CPU, 8 GB of RAM, gigabit Ethernet port, as well as USB-C hubs made it the best choice. On top of that, the amount of documentation currently available by users and manufacturers allow you to be flexible in how you approach certain applications. Finally, the benefit of using Python 3.x made this a smart choice for programming.

*Figure 5: A Raspberry Pi Model 4 with an aluminum metal heatsink to disperse heat and keep the device cool.*

This leads to another addition which is the camera for the real-time video stream. Raspberry Pi offers OEM camera modules, as well as having 3rd party vendors make their own camera modules. In this case, the picamera V2 [11] was chosen. The reasoning behind this was to be able to use OEM manufacturer equipment to ensure the best possible outcome, with the least amount of errors. A consideration was made for the picamera V2 NoIR (no infra-red), as well as the V1.3, although for our experimentation, a camera with no infrared capabilities was not required, and the V1.3 is not nearly as equipped as the V2. If this were to be displayed in regions with ultra-low light, NoIR camera modules may provide some improvement.

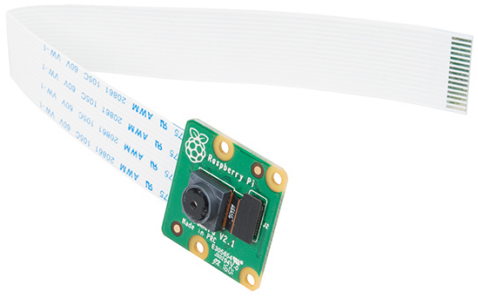

*Figure 6: A Raspberry Pi PiCamera V2 offers up to 1920x1080p resolution at 30 fps.*

Once the hardware requirements were decided, the next portion of the project remained, development. A wide array of programming languages can be used effectively for this project, but the simplicity and effectiveness of Python made it the clear choice. Both AI and ML applications are commonplace for Python, and also being able to hyper-thread and utilize parallelism between multiple cores allow the processing to be as fast as possible. This, on top of Python being a lightweight object-oriented programming language that offers great flexibility between the Raspberry Pi and the host device, an iMac Pro machine.

## Related Works

With such a vast field of IoT, there are a few pieces of work that are core to the development of encryption. However, in the specificity of real-time video stream encryption, there is little to no work that is published. This, on top of the prospect of using an encryption system based off of a fully homomorphic encryption scheme with live-streamed video is non-existent.

Most fully homomorphic encryption schemes are based off of the proposed systems written by Craig Gentry, Ph.D [2]. In his proposal, et al. Gentry utilizes lattice-based cryptography for cipher-text computation. This, alongside his "bootstrapping" method allowed low noise to increase accuracy. With that system, they are focused heavily on data analytics via the cloud for highly regulated industries such as the healthcare industry.

There are similar variants to my proposed work such as "Real-Time Privacy-Preserving Moving Object in the Cloud" by Kuan-Yu Chu [3]. First, this article uses a wired network camera, commonly used for CCTV (closed circuit television) relay, then sending the data to the cloud for object detection. Although this is great, the added difficulty of using a lightweight wireless IoT camera means that there are enhanced security measures that should be in place since it is easier to breach a wireless device over a wired device.

Another great research paper pushing forward in the encrypted video stream is "Private Video Streaming Service Using Leveled Somewhat Homomorphic Encryption" [5]. In this report, they were able to utilize a leveled homomorphic encryption in order to encrypt a video stream service, leading to similar results if the decryption process was not computed correctly.

Moving into the video encoding side, Secure Advanced Video Coding Based on Selective Encryption Algorithms [7] offers the use of a more widely used encoding platform that offers a secure method in which to transfer videos to the end user. By adding another layer going into the intra-prediction mode can offer a *smear* style encryption and a strong compression ratio that makes data transfers easier and quicker.

Dual-Layer Video Encryption using RSA Algorithm [8] is based off of a RSA-themed cryptosystem [9], following which they encode the encrypted video stream using the audio video interleaved (AVI) codec, with a possibility to port to a more common MPEG format. Where they benefit the most is the utilization of dual-layer encryption. In the report, they show that a single layer encryption can mask the frame well, but when a second layer is applied, the frame resembles a television static image.

Another related piece is the widely used Python package, imagezmq, written by Jeff Bass [13]. The utilization of Python websocket programming is key in transferring data between the cloudlet, a Raspberry Pi [12], and the edge device, the desktop workstation. Here, however is where the benefits are increased. This package offers multiple cloudlets (at this time, 12 have been simultaneously tested) to be transmitting data to the workstation.

Some works that stand out in the security driven world of IoT devices can be traced to et al. Gentry, Ph.D. Gentry's [2] dissertation for the first-generation 'Fully Homomorphic Encryption' in 2009 began the revolution of privacy driven encryption for remote processing and storage, such as cloud devices. These cloud servers are able to process data while remaining encrypted, although the host machine retains the key for decryption. As it stands, et al. Gentry [2] has improved ten-fold with his newer third-generation FHE model (GSW) further, by avoiding the relinearization step, taxing CPU speeds dramatically.

Implementation

The basis of this project was utilization of a homomorphic encryption scheme. RSA encryption [9] is a great scheme, predominantly used for username/password protection, along with SHA-based encryption schemes [14]. Elgamal, although, is categorized as a partially homomorphic encryption scheme, utilizing an asymmetric key encryption algorithm.

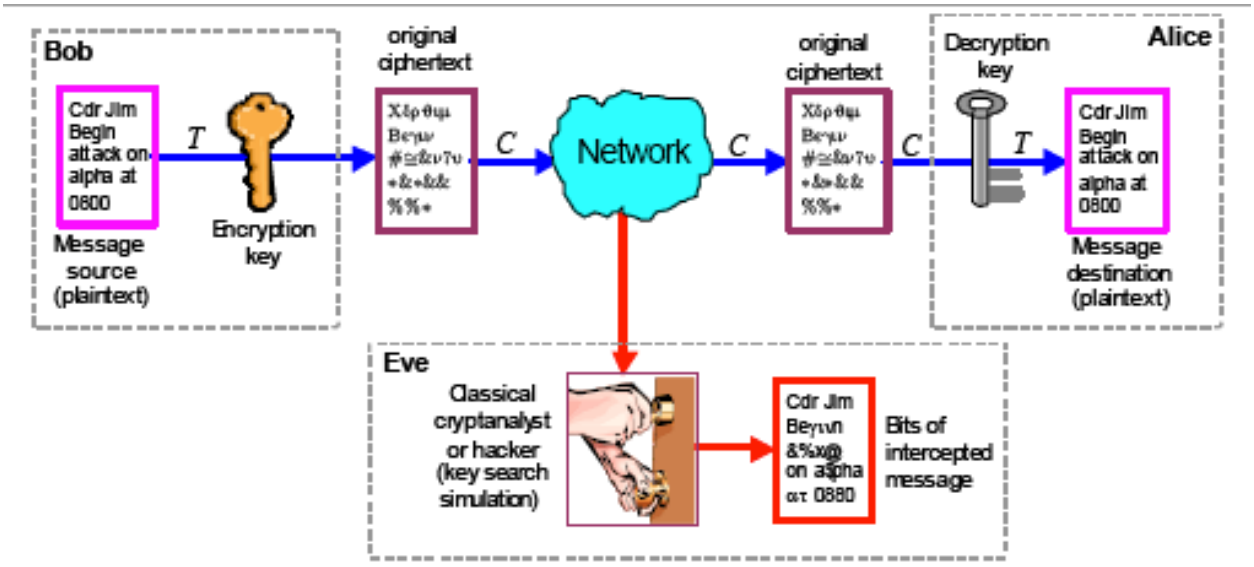


*Figure 7: Illustration of Trivial Cryptography Scheme*

Elgamal encryption is lighter-weight and can be a suited scheme for image encryption. Based off of the Diffie-Hellman key exchange [16], one of the earliest forms of public key exchanges.

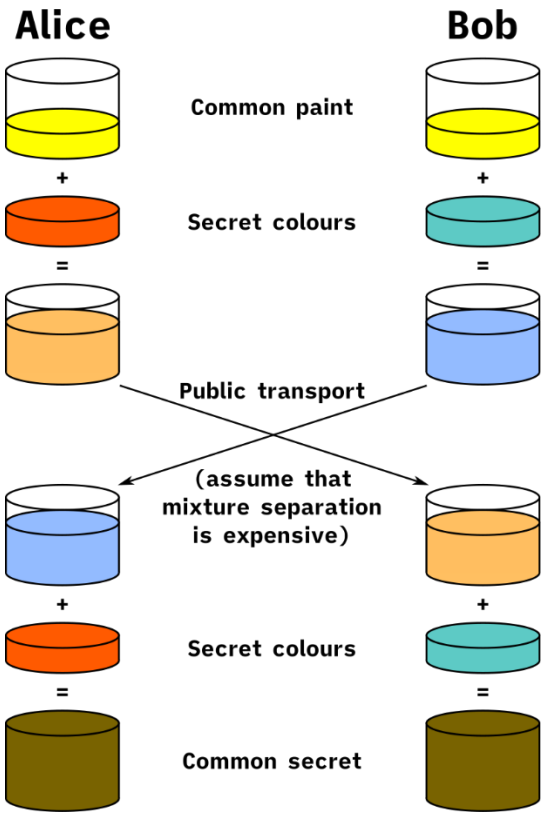


*Figure 8: Illustration of the concept behind Diffie–Hellman key exchange*

$$(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p \quad (1)$$

This key exchange requires a protocol of multiplicative group of integers over a modulo, $p$ and a primitive root modulo $g$.

The preliminary step in creating a *proper* Elgamal Encryption scheme [15] is to begin with key generation.

Elgamal Encryption

## Key Generation

To start off, we need to create a cyclic group $G$ of order $q$.

Following this, we then want to choose a number at random, $x$, inside the set

$$\{0, \dots, q-1\} \quad (2)$$

After $x$ is selected, we then calculate $h$.

$$h = g^x \quad (3)$$

Once $h$ is computed, the cloudlet can then publish this new key, $h$.

Encryption

Once an interaction begins between the cloudlet and the edge, the edge then goes on to follow a similar route of key generation.

To start off, the cloudlet will have made public the following:

$$(G, q, g, h) \quad (4)$$

where *G* is the cyclic group, *q* is the order of *G*, *g* is the generator, *h* being the public key code.

Create a reversible map for message *m*.

Choose a number at random, *y*, inside the set:

$$\{0, \dots, q-1\} \ (5)$$

Compute *s*, referred to as *shared secret*:

$$s = h^y \ (6)$$

Compute $c_1$ or cipher-text 1.

$$c_1 = g^y \ (7)$$

Compute $c_2$, or cipher-text 2.

$$c_2 = m * s \ (8)$$

Once $c_1$ and $c_2$ are calculated, these two are sent to the edge for decryption. Although they pass through public space, there is no longer a concern for any singular user to retrieve a copy of the message since they have no access to the secret key, or even a method in which to receive the keys.

Cloudlet Integration

After the Elgamal encryption scheme has been setup, we then move forward to integrate this to our cloudlet device, a Raspberry Pi 4 [12]. Once the operating system (Raspberry Pi OS) is installed, we then need to verify the modules used, Python 3.7.3, as well as numpy 1.16.2. From here, we install: imutils, imutils.video, math, random, picamera, time, and a few miscellaneous packages.

Utilizing Open CV2 allows us a myriad of supported software that is used daily. By use of OpenCV [17] we can read in singular frames at any given point. Since these all follow the standard *numpy.ndarray* formatting, processing any computations after the fact make it simple. In doing so, we are able to directly multiply the loaded frame shown with $c_2$ in formula 8. This direct multiplication will go across every pixel in the tuple-based frame (our example utilized a 3-tuple image being 640*480*3, being the width, height, and the amount of channels, BGR).

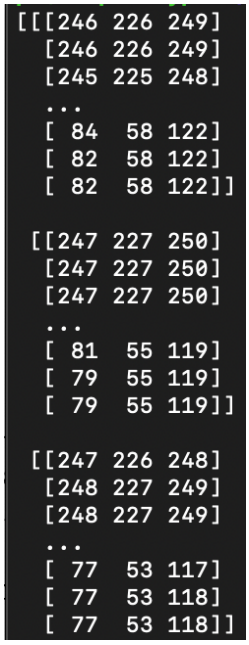


*Figure 9: Frame Printout of image loaded into OpenCV*

Once we compute (8), or $c_2$, these raw values will then become encrypted.

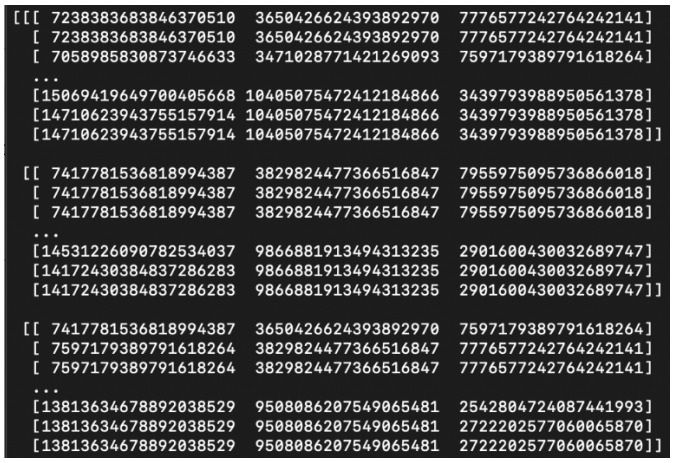


*Figure 10: Frame Printout of encrypted image loaded*

At this stage, we then need to consider the streaming service, in our case a TCP socket was chosen. Local broadcast was no longer considered because the sizing of the encrypted frame shape no longer is a readable file outside of byte code or data processing.

TCP Socket Integration

For the data stream, once encrypted will leave us with a numpy nd.array format that we can *still* process data with, or manipulate in any regard, however, we now need to transfer the encrypted frames *somewhere* for analysis. Since we are no longer concerned with transmitting data that is not encrypted, we utilize a TCP socket in Python for our data transfer.

At the time of this report, there are plentiful ways to integrate a TCP socket for sending frame, as well as many available packages that specifically transfer a numpy nd.array between devices. We chose to send the frame via a custom-made socket.

In order to integrate this socket, a few things are considered. First and foremost, the action of sending the original frame *must* be possible. Secondly, a TCP socket is required versus a UDP socket. This was chosen because we needed to consider that in a UDP environment, there is no possible way to verify the package contents arrived at the location in proper order. By handling this encrypted frame via TCP, we are ensured *always* that the frame is being sent exactly as needed.

Going from here, recall that we have to transmit certain keys to the edge for proper decryption. The first and foremost is the encrypted frame, represented by equation (8). Following this, we also need the first cipher-text (7), the cyclic group key, $q$ and lastly the edges generated key value $g$. These four items must be sent to the edge from the cloudlet in order to compute the decryption properly.

For our project, we decided to implement the encryption key per *socket creation* over per frame. The additional processing and calculations proved moderately difficult for the cloudlet to be able to call for a new private key and public keys. That being said, to facilitate a smooth transfer of data, assuming that these are public keys and do not need their own encryption, we package them as a tuple.

$$all\_keys = c_1, bob\_gen\_g, cyclic\_group\_q \quad (9)$$

Once *all_keys* has been created, we then begin the data processing in order to transmit the tuple. In order to *efficiently* do this, we utilize a Python module, *pickle*.

By using pickle to prepare the data to send by using *pickle.dumps(your_python_object* . The benefit of pickling is to represent the object you choose without having to explicitly *write* the object, thus requiring more processing speeds, and straining the cloudlet more. Also, by utilizing pickle, it converts your python object to a byte-like format, which is exactly what it wanted for the data stream. The next step is to calculate the length of the now pickled object, in order for the edge device to properly calculate the size of the buffer needed to capture every frame. This is important because by being one byte too long or too short will result in a corrupted *and* encrypted file that will no longer be able to be usable.

This calculated length of the pickled encrypted frame will then be pickled itself, then sent the same way as the example shown in equation (12). Once the edge receives the size then loads via *pickle.loads(your_pickled_object)*, you then set this as your buffer size that is looped until the full encrypted frame is received, then un-pickled.

Decryption

In this phase, we will begin decryption of the frames. By this point, we will have received the set of public keys and shared keys that will be used to decrypt the file between the cloudlet and edge, the length (or size) of the encrypted frame, then finally, the encrypted frame itself.

We will now convert the tuple *all_keys* into three separate variables for ease of use. Following this, use the encrypted frame size as the buffer in order to properly receive the encrypted frame, since they are sent in fragments. Lastly, we retrieve the encrypted frame, as mentioned in the section prior.

Now to decrypt the frame, recall during the key generation phase, the cloudlet has *x* as their private key in equation (2). Continuing onward, we compute:

$$s = c_1^x \ (10)$$

Then calculate

$$s^{-1} \qquad (11)$$

Finally computing

$$frame\_dec = c_2 * s^{-1} \ (12)$$

With m finally decrypted, we reach the penultimate step, and that is a type conversion. At equation (12), we are left with the decrypted frame, however the data matrix that builds from it is a float-type object. We can simply type-casting the object as so to revert back to figure 9, a view-able image:

$$frame\_dec\_img = int(frame\_dec) \ (13)$$

At this step we can finally view the decrypted frame! We simply load the *frame_dec_img* into the OpenCV class *cv2.imshow('Decrypted Frame', frame_dec_img)*.

We have successfully been able to use Elgamal Encryption Scheme in order to send the images via TCP socket from the cloudlet to the edge for further processing.

Results

After all is said and done, there are a few constraints in utilizing this methodology that will enable us to smoothly encrypt images later on. Before mentioning the key takeaways from this project as there were many different options as well as criteria that was set in the beginning, we must first show the image results.

*Figure 11: Original (640,480,3) Image Prior To Processing*

As you can see, this image is vibrant in color. This allows us to be able to *clearly* view any differences between this original image compared to the processed (or decrypted) image.

*Figure 12: Decrypted (640,480,3) Image Post-Processing*

As you can see, there seems to be no visual or stark difference between the original pre-processed image, and the post-processed image. This is good news because if we were to further expand on the subject (see "Future Work"), we can rest assure that the image is kept well enough intact that any processing will not be rendered useless.

However, we can view the differences between these two images by independently load each frame into different Python objects and calculate the different by doing the following: *frame_diff = frame_orig - frame_decrypted*. The result from this will mean only the change in pixel values will be stored into *frame_diff.* From here, we can either save the image locally to the cloudlet, or we can simply view the image while the program is operational.

*Figure 13: Difference Between Original & Encrypted Frame*

As you can see, there is some minor distortion happening in the decrypted frame rather than having a pure black image. As et al. Gentry [2] mentioned, there will be inherent noise to the result due to encryption and decryption. However, the noise will be marginal at best, therefore leading to a successfully decrypted image.

That being said, here are the key takeaways mentioned prior:

- For key generation, a maximum value of $10^{20}$ can be utilized without severely stressing the CPU.
- A maximum instance of one encryption key is used per session. This is because generating a key per message (or frame in this case), subsequently multiplying this large random integer to send via socket will overload the CPU of the Edge device.

- A maximum image resolution of (640,480, 3) is to be used. This allows a small enough image that can be encrypted and sent through websocket while still usable for detecting objects (see "Future Work" for more information)
- There is minor distortion to the decrypted frame. Not significant enough to disrupt any processing after the fact.
- Recommended frame rate to not exceed 24. Average human vision cannot distinguish much beyond 24 frames per second, therefore lowering CPU overhead.

Future Work

It is easy to see how much we can build off of this program for future work. As labor intensive as this was to attempt, there are many more possibilities with this project to continue expanding upon. With each bump in the road, being data loss from the TCP socket, or by even having to go around the everlasting problem of the disk space in a Raspberry Pi being hyper limited due to it being a MicroSD card, thus having reduced read/write capabilities make this all the more challenging and being forced to optimize the code at hand. A simple string conversion or using *object.tobytes* could have worked theoretically for the pre-processing of the keys, implicate casting of a pickled object will lead to longer running times and ensuring the Raspberry Pi will not fail after continuous usages.

Subject at hand, the same reason is why we created a tuple object for the keys. In testing of the socket, the edge device runs too fast and will not safely receive the keys if sent separately. This also benefits the cloudlet on the write requirements, because we can create a new pickled object in-line without forcing extra writes.

That being said, the following list is what is considered strongly for future work, as it pertains to the benefit of the overall project.

1. The first improvement that I will work on is the robustness of the web socket. At its current form, it works well enough for this current version, however being able to call back the devices, and possibly even send higher resolution image would be useful for any possible transmission errors that may occur. In the current phase, there are instances which when the encrypted frame is sent, the resolution makes the decrypted image unusable for any post-processing applications.
2. Our second improvement is to include a database, or registry of users that can be stored and updated in the cloud. This way, we can keep and update critical items or faces, and be able to

verify that no duplicate images will be taken and used when needed. In doing so, you can keep a current list of persons to be able to monitor any unwanted activity.

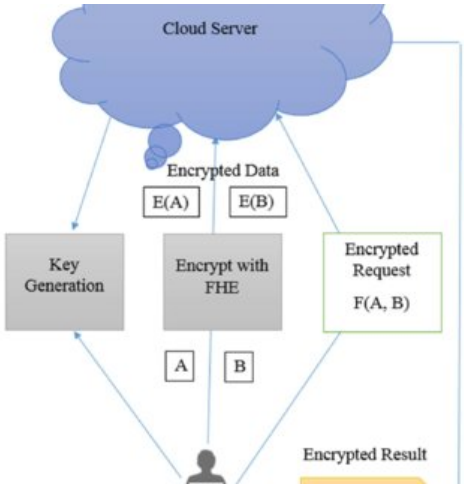


*Figure 14: Overview of how the fully homomorphic encryption scheme is able to securely store data*

3. A third option, expanding on both the first and second point, is to remove static backgrounds. With a call to increase accuracy, we should strive for the highest resolution imagery for our object detection. If the cloudlet devices are fixed and mounted in a static location (never moving or rotating), then there is a possibility that any additional noise to the background can affect both the objection algorithms, as well as the web socket data transfer.
4. A fourth option, although counter-intuitive, is to be able to locally broadcast the frames being encrypted. This step can be crucial into data privacy if the encryption scheme works in a way that is a safe environment to work with.
5. Lastly, the most important topic for Raspberry Pi's would be to find out and implement a true FHE encryption stream. By doing this, you will be able to have an unbreakable cipher text that only the dedicated machine can run with at any given time. Regardless of it being a feasible encryption scheme for such lightweight devices, if there is any remote possibility in implementing such a CPU intensive algorithm, then we have succeeded the goal for the future work.

## Conclusion

In conclusion, although a fully homomorphic scheme is not yet obtainable on such a light weight IoT device, we are able to impose a partially homomorphic encryption scheme to allow private data streams to be encrypted. Spoofing DHCP networks or creating a man-in-the-middle attack is powerless since it only means that the hacker receives the encrypted strings, without any option to decrypt it themselves. Allowing a dedicated Edge device to have the key and be able to decrypt multiple real-time streaming cloudlets, you can rest assured knowing that your data will be safe and view able on the machines of your choice.

Resources


1. Ronald L. Rivest, Len Adleman, Michael L. Dertouzos: On Data Banks And Privacy Homomorphisms
2. Craig Gentry: Fully homomorphic encryption using ideal lattices
3. Real-Time Privacy-Preserving Moving Object Detection in the Cloud
4. Tianhe Gong, Haiping Huang, Pengfei Li, Kai Zhang, Hao Jiang, "A Medical Healthcare System for Privacy Protection Based on IoT", Parallel Architectures Algorithms and Programming (PAAP) 2015 Seventh International Symposium on, pp. 217-222, 2015.
5. Private Video Streaming Service Using Leveled Somewhat Homomorphic Encryption
6. A Fully Private Video on-Demand Service
7. Secure Advanced Video Coding Based on Selective Encryption Algorithms
8. Dual-Layer Video Encryption using RSA Algorithm
9. RSA Cryptosystems
10. Advanced Encryption Standard
11. Raspberry PiCamera V2
12. Raspberry Pi Model 4b 8GM RAM
13. Jeff Bass: ImageZMQ
14. SHA-1 Cryptosystem
15. Elgamal Encryption
16. Diffie-Hellman Key Exchange
17. Open-Source Computer Vision (OpenCV)