# Final Report for Secure Camera Capture System (April 2016)

Christian J. Coffield, Matthew W. Dekoning, Nathan H. Lea, and Kevin T. Seitz, *Student, OSU*

***Abstract*—This report provides an overview of the Secure Camera Capture system produced by Team 6 in Spring 2016's Senior Design 2 class. The project encompassed a three-part hardware and software system, used to capture, securely store, and retrieve photographs taken by an internet-equipped camera on a sixty-second timer. In the report, the project is outlined in both technical and nontechnical terms, the AES encryption method used, the Raspberry Pi microcontroller hardware, and the software used to tie them together. Specifications (both hardware and software) and how they were addressed are detailed, and a guide is provided to reconstruct the project using provided code. Finally, it provides the team's perspective on the final product and possible improvements that could be made on the design.**

## I. INTRODUCTION

The Secure Camera Capture System (SC^2S) is designed to provide an inexpensive, do-it-yourself solution to home or office security, as well as basic timer-triggered photography and storage. It uses a three-stage system to take and encrypt, store, and retrieve photographs captured every minute.

The first stage, comprised of a Raspberry Pi 2 and an equipped camera, automatically takes photos every minute, encrypts them using AES-256, the military standard in security. It then sends them off to a remote server to protect them from tampering or theft before deleting them on the local storage.

The second stage, comprised of a second Raspberry Pi 2 and a large USB external storage device, works as a sort of file storage and organizer system. It takes in pictures sent by the camera device, adds a layer of encryption, and stores them on-disk for later retrieval.

The third and final stage consists of an application programmed for Windows computers and an extended Android app version. This application allows for a user to access the photos on the server remotely from any location, and organizes the photos into a timestamped format for ease of access.

## II. NONTECHNICAL DESCRIPTION

Most of the technology in this project is Raspberry Pi-based. Raspberry Pi is a small, camera-like computer that can be connected to a standard HDMI monitor and operated using standard USB keyboards/mice. This technology is quite prevalent in the world of electrical engineering hobbyists-- many keep at least one around for tinkering in their home.

Because Raspberry Pi is such a widespread technology, it was selected as the primary platform for development. Hopefully, other similarly-skilled engineers that have similar needs can take the code, ideas, and direction contained in this report and the surrounding documentation to recreate and expand on the concept.

Following is a nontechnical description of the three major components in the project: the Camera device, the Server used for photograph storage, and the Client programs used to retrieve data.

### *A. Camera Device: Messrs. Coffield and Seitz*

The Camera device uses a basic Raspberry Pi 2 and the camera custom-designed to work with the Pi; several starter kits for the microcomputer include the camera, and its profile is about as small as can be, making the whole system very compact. It's contained within a 3D-printed case, with the prototype being roughly the size of two stacked cell phones in length, width, and height. The case is flexible and easily wall-mounted, with ports for ethernet connectivity and power on the side and bottom. No special hardware is required to use the device.

After the initial configuration, the Camera device is very, very easy to use. Once plugged into both power and ethernet, it automatically boots itself up and enters the cycle of waiting one minute, taking a picture, sending it over a very secure connection, and repeating. Once the first setup has been done, it never requires user input again; all the device needs to capture photographs and send them to the server is power and an ethernet connection.

One might ask how secure the connection is, since the photographs are being taken and transmitted automatically, without any review from the user. What if they contain sensitive or private information, or otherwise material that one wouldn't want available to the public? The type of security used to connect the Camera device to the Server that stores its images is very, very strong-- military-grade encryption is used, to ensure that sensitive data will never fall into the wrong hands. [2] For more information on the security behind the system at hand, refer to the subsection labeled Encryption Discussion. Additionally, the photos taken are not retained on the device in the traditional sense of storage; only one is kept at a time and only briefly, and it is only kept in a temporary space that is erased immediately upon a loss of power. Because of the above reasons, the photographs taken by the Camera device are delivered safely and securely to their destination without fail. Once an image has been captured, an encrypted connection has been made with the server, and the image has been transmitted, the Camera device's involvement in the system is completed for that iteration.

The device is enclosed in a custom-designed 3D printed case that is easily wall mounted. It features dimensions that comfortably fit a Raspberry Pi and secure the camera into place, and allows access to the power connection and the ethernet connection.

### *B. Server: Mr. Dekoning*

The Server we have designed for this problem is also built on a Raspberry Pi. It works on the same basic principle as any other server that someone using the internet to access a website does; it stores files, and upon request, delivers them.

In terms of hardware, the Server system is no more complicated than the Camera device. It consists strictly of a Pi microcomputer and an externally powered USB hard drive. USB hard drives are a powerful, versatile tool for storage, and they're common and inexpensive enough for a project of this scale. The only setup they require is to be plugged in.

The same kind of security as the Camera is used to protect communication with the outside world. To connect to the Camera, a connection as well-defended as a military line is left waiting for a paired Camera device to send a picture. When the picture is sent, it names it with a timestamp, applies an additional layer of encryption via an on-board AES key, and stores it on the hard drive. Then its interaction with the Camera ceases, and the connection closes. Finally, a new one opens and waits for the Camera to make contact again.

On a separate track from the one communicating with the Camera device, much like a separate tab in a web browser, the Server opens a line that waits for a Client application to connect. The Client, once it connects, will declare whom has initiated the connection to the server, using a login system. If the Client sends the correct username and password, the Server responds by sending it a list of what pictures are stored on the hard drive, organized by timestamp, as well as a session key that gives the Client authenticity. The Client can then request them to be sent, and if they provide a session key matching the one given by the Server, the Server sends the corresponding image. Then the Client can request more images, or close the session. If the session ends, the Server deletes the stored session key.

The real strength of the Server, though, is that it does not need to be anywhere near the Camera or the Client. The real way security is handled is by keeping sensitive information far, far away from the access or recording point. A potential hacker or thief cannot get information from the server if they don't know where it is or can't get to it. A server can be hidden in a safe somewhere, or in another building, even another country. This is the way real, major corporate and government entities protect their information; along with secure connections and potential encryptions, via separation.

### *C. Client Program/Android: Mr. Lea*

Written in C# (for the Microsoft Windows application) and Java (for the Android app), a program is used as an interface to the Server. It's designed to easily and remotely navigate through the Server's file system, and functions very similarly on both of its operating devices, mobile or otherwise. Referred to as the Client, it provides four functions: log-in, registration, navigation, and picture downloading.

Upon initial startup, the Client will request the IP address of the Server it is meant to connect to. Then it will request a username and password from the user, and post these credentials to the Server. If they match with the log-in credentials that the server keeps, the Server responds with an affirmative, and then sends back a session key, followed by a JSON (Javascript Object Notation) file that represents the tree of images currently stored on the Server. The log-in is required every time the program is started, however, to increase security.

The Client takes the JSON file and interprets it into a tree, branching out into Year/Month/Day/Hour with each leaf being an image taken at that minute. This is all that the Client needs to request images from the Server. The user can navigate the tree like they would any standard file directory, entering the 'folder' for the year they would like access to, then the month, day, hour, and click on the minute they would like to request the Server send it.

When the user clicks on a timestamp, the Client posts the picture name to the Server in much the same way it posted the log-in credentials, along with its previously provided key. Assuming the key matches what's saved on the Server, it then sends the image over an HTTPS connection, the same type used for logging into bank accounts, government websites, and other important web connections. The image is base64 encoded. The Client takes the image and displays it on-screen; from there, there is an option to save it permanently to the Client's operating device.

Additionally, at any time, the user can click a button to refresh the tree of images, adding anything new added to the server; this is done by requesting an updated JSON and rebuilding the tree. The user can also re-configure the IP address that the Client addresses as the Server from within the Client itself. Once the IP address has been correctly configured once it does not need to be done again; the Client saves this configuration.

## III. HARDWARE REQUIREMENTS

*1.1 Shall be able to take and upload a picture every minute*

Our camera device uses a crontab on the raspberry pi to execute the script that takes, encrypts, and sends a photograph every minute. For more information on crontabs, please check the references section [3].

*1.2 Shall be able to take picture of at least 640x480px*

Our camera can take pictures of up to 5 megapixels (a resolution of 2592x1544px). In our chosen photo-capture software, we can specify a target resolution to be used for capture. We have chosen 1296x730px, above the requirement of 640x480px.

*1.3 Shall have a method of tamper protection that if stolen, the system will not retain any data beyond the most recent photo captured*

Our device stores only one captured photo at a time. It immediately removes photos upon encryption and sending to storage, so there is no permanent memory of them. In addition, all photos captured are only stored in RAM; if power to the device is severed, there will be no images stored in memory. The camera device also lacks any form of key to access the encrypted server, so a potential hacker could not break into it to gain access to the server.

*1.4 Shall be able to store a year's worth of pictures on the server*

With the current scheme of one photo being saved per minute, there is a space requirement of 525,600 photographs (minutes) in one year. At a filesize of 1.2 MB per picture, the server requires a total space of just over 630 GB. This has been implemented in the form of a USB hard drive, of size 1TB (1024 GB).

*1.5 Shall store the picture securely off board from the camera to protect from thieves*

As stated in the point on tamper protection, our camera device only stores one image at a time-- and that image is stored very briefly. All long term storage is handled by an off-site server, comprised of a second device and hard disk.

## IV. SOFTWARE REQUIREMENTS

*2.1 Shall have a method for the user to retrieve the pictures*

We have implemented a Windows application in C# to retrieve all captured photos in a smooth, metro-style GUI. Additionally, an android application has been developed for this application, and works on Android devices (5.0 or later).

*2.2 Shall be able to retrieve the pictures remotely*

As stated above, there is a client application built for Windows platforms that can be used with any standard Windows OS, 7 or above. This can be placed onto any Windows personal computer and used to access the server from any location, i.e. remotely.

*2.3 Shall use AES 128/256 as the standard for encryption*

The communication method of choice between the camera device and the server uses AES256 encryption. Specifically, it uses a Secure Sockets Layer (SSL) library to establish a connection with the server using a certificate saved on-board.

*2.4 Shall index each image with time and date stamp for ease of management*

As the images are received by the Server Pi, they are saved to the hard drive with a collection of randomly generated letters and numbers. Upon request, they are served using date time properties from when they were saved in the following format: yyyymmddhhmmssr.jpg. The year, month, day, hour, second corresponds to the starting letter, and the r is a random lower case letter to prevent any collisions. These pictures are then placed into a JSON formatted index based on time of arrival for easy client navigation. This index groups all images taken in the same hour together, and has outer layers of year, month, and day. Thus the images are presented to the user grouped within the hour they were taken for easy access.

*2.5 The images shall be accessible from any standard Windows computer (Windows 7 and up)*

Using C#, an object oriented programming language, a Windows application has been developed. This application runs on all Microsoft computers from Windows XP and up, using the .NET framework.

*2.6 Shall securely store the key on board so that any user will not be able to locate the key on the system*

The remote server stores the key in a location on board, but only available if one gains root access to the system. From client side or a web browser it is impossible to obtain this private key.

## V. FEATURES AND SPECIFICATIONS

### *A. Camera Pi: Messrs. Coffield and Seitz*

- Takes one picture every minute
- Stores picture in RAM
- Takes pictures at a resolution of 1296x730
- Crontab integration for minimal processing power consumption
- Secure connection to server for transmitting images using SSL and AES-256 encryption
- 3D-printed enclosure: 3.75”x1.5”x2.6”

### *B. Server Pi: Mr. Dekoning*

- Stores over a year's worth of pictures on a one terabyte hard drive
- Securely interact with the Camera Pi with an AES 256 encrypted connection (openSSL encrypted python thread)
- Securely interact with the Client Program with an AES 256 encrypted connection (nginx server using https)
- Provide safe login and registration methods for the Client Program using SHA-256 hashing and salt methods
- Safely stores the session key in the unreachable ‘/etc’ folder
- Can receive and serve pictures simultaneously
- Instructions to setup a Raspberry Pi to run as the server are on the website
- The Server dynamically names each picture in this format:

YearMonthDayHourMinuteSeccondRandomCharjpg
(4) (2) (2) (2) (2) (2) (1)

- The Server dynamically creates a json index of the pictures it receives.

### *C. Windows Client Program: Mr. Lea*

- Software Requirements:
    - Windows 7/8/10
- Hardware Requirements:
    - Memory:
        - 4GB
    - Hard Disk:
        - 2MB
    - Mouse, Keyboard

### *D. Android Application: Mr. Lea*

- Android 5.0+
- No other dependencies
- Gallery application-compatible (Gallery, Google Photos, etc.)

## VI. DETAILED TECHNICAL DESCRIPTION

### *A. Camera Pi: Messrs. Coffield and Seitz*

Raspberry Pi is a Linux-based device. As such, it has a Linux-like operating system, Raspbian, with all of the standard features one would expect of such a device. One of the most important aspects of Linux systems is the freedom with which they handle scripting-- programming that occurs on the fly, without prior compilation, and allows for easier automation.

This project contains a bash script (.sh file), which, when run, does all of the required work of the Camera device in succession. This was because Raspbian supports a feature known as ‘crontab’. Crontab is used to schedule repetitive tasks to occur on a set timer. In this device, a picture is to be taken every minute, which is a very regular pattern. Behavior like that is perfect for Crontab, which does not lock the device into doing a single task like a simple while-loop would. This means the Pi is still capable of taking other actions, such as modifying the address of the Server that images should be sent to . So, every minute, the Pi automatically runs the script that performs all of the tasks outlined in the flowchart in Appendix A.

Every iteration, the script creates and mounts a temporary directory, or folder. This folder is placed into a folder that is mounted as a temporary filesystem in RAM; if power to the device is lost, the contents of the folder are gone forever. This makes it so that if a thief were to steal the device, the important data (the pictures of said thief) would be gone the moment the camera is unplugged from the wall.

As soon as the directory is created, an image-capture application called raspistill is run, using parameters set by the script to set the resolution of the image, as well as its orientation. This is where the picture is actually taken-- and then stored in the temporary filesystem from above, titled 'toSend.jpg'. The picture is complete and ready to be sent at this stage.

The bash script then calls a secondary script, written in Python. Designated 'sendpicture.py', this is where the image is actually sent. Python is a scripting language natively supported by the Raspberry Pi, and is used for a very variety of tasks ranging from opening websockets to performing diagnostic checks on a system. The Python script has its own set of operations, outlined in the above chart, but they can be summed up as:

1) Create a secure, encrypted SSL socket (SSL is discussed in the Encryption Discussion section)
2) Connect to the Server's IP address on a specific port using the created socket
3) Verify that the SSL certificate on the current device matches the certificate expected by the server
4) Convert the image currently stored in RAM into a string of bytes
5) Send that stream of bytes over the SSL socket to the Server
6) Close the socket and exit the Python script

Once all of that is completed, the picture that was captured is safely on the server and can be deleted from the Pi. This is done by removing the directory that the picture was contained on, and unmounting the temporary file system, clearing its contents from RAM.

With the picture captured, encrypted, and sent to the server, the camera device's work is finished. It proceeds to wait for one minute before repeating the process, and does this continually as long as it is plugged in and has an internet connection. For clarification on the order of and procedures performed during connections between the Camera device and Server, please see the graphic in Appendix D.

Additionally, a case was custom-built in Autodesk Inventor, a Computer Aided Drafting and 3D modeling software. The case is simple to print and assemble. It's rectangular with a space for the Raspberry Pi, with the ethernet port and power port left exposed. The lid has a spot to glue or tape the Pi Camera into place, with the cord for the Camera folding safely within.

### *B. Server Pi: Mr. Dekoning*

The Raspberry Pi functioning as the server uses multiple threads (program flow outlined in detail in Appendix A) to communicate with both the Camera device and the Client application. Because the operating system used by Raspberry Pi devices is Linux based, it included many of the features a standard Linux computer would use; one of them is a technique known as POSIX threading. This type of threading allows for multiple processes to run independently of one another and receive computational servicing pseudo-simultaneously. With this in mind, two threads are opened upon bootup: one to communicate with the camera and receive images (with no feedback, to retain a one-way link and increase security) and one to handle requests from the Client. For more information on POSIX threading, check the references section [4].

The Camera device connects using a completely secure AES256 Secure Sockets Layer (SSL) connection. SSL is used to encrypt connections and ensure security. A socket using SSL is opened in Python, using the open-source library OpenSSL. Images are received through this socket in JPG format and stored away into the attached external hard drive. A running index of what files are on the hard drive is updated, to retain a level of organization for future access. Then the socket is closed, and a new one is opened, waiting for the Camera device to connect once more and send another picture.

Once an image is received, it is immediately run through another layer of encryption. Using a key stored on the Server and only on the Server, all images are put through a hard AES-256 encryption before ever storing it on a hard drive.

Encrypted images are stored on the external hard drive using jpeg format. They are placed in the images directory on the 1 TB hard drive. Additionally, a JavaScript Object Notation (JSON) file is used to assign a tree-like hierarchy to the photos; this JSON file interprets the file locations into real-world date and time stamps. As such, virtual folders are created that sort the images by year, then month, then day, with each image therein sorted chronologically by the minute. The JSON file is stored on the server to be provided to the client upon request. This file, index.json, is what’s used to create the illusion of a year/month/day/hour/minute hierarchy.

The second thread opens up a server using Nginx technology. Nginx is used to serve dynamic HTTPS content (such as images) over the internet. It’s another open source software, under a permissive-free license. Nginx was chosen over other, similar software (like Apache) because of how lightweight it is; because computational costs must be kept low to run on an inexpensive piece of hardware like a Raspberry Pi, it was the ideal choice.

Once the server is established with images and an organizing JSON, it's prepared to accept requests and communicate with the Client software. First, PHP files are used to handle a login using a username and password received as posts from the Client. Once the connection is established and the login credentials have been verified as correct, the server provides the client application with both the current version of the JSON index file so it can build its own version of the hierarchy tree on its operating device and a session key, randomly generated to provide authentication to each connecting Client. From there, the server waits for requests from the client for specific images in the tree, and delivers them upon request—provided the Client can show they have a session key matching one permitted by the server.

### *C. Client Applications: Mr. Lea*

The Client is programmed in C#, using Windows Forms. Windows Forms are the newest and best GUI library for use in the Windows environment, because they are compatible on any Windows computer with the .NET framework installed and are very stable. For the internet transmissions, the Windows Client uses WebClient, an in-built Windows API library that implements many of the necessary functions required to post to, and download data from, the server. This library also takes care of the HTTPS connections, as well as the encryption and decryption that is required for the transmission.

To build an indexing, date-and-timestamped tree from the JSON, a custom JSON parser was written. It uses many different libraries, including regex searching. The JSON tree builds a List of objects that contain the information about each level of the tree. There are objects that represent each level of the tree: The Year Object contains a List of Month objects, the Month object contains a List of Day objects, the Day object contains a list of Hour objects, and the Hour object contains a list of Image objects. The Image object contains information about each image on the server such as the date captured and the name of the image.

This List of Lists is used to build a treeView (expandable tree of clickable links, each link displaying its contents upon click) for display on the Windows Form. On each item that is an image, the image name is assigned to a property on the tree node. This property is used for the request of the pictures on the click of the node. Once the node is clicked, it fetches the image in question from the Server. This is done by posting a command to the Server with an image name to request.

Additionally, the Client has a config file that stores the IP address of the Server it’s meant to connect to. Upon the first boot, the Client will request the IP it’s meant to connect to, and will store it into the config file. This information will not be requested again unless the user wants to reconfigure the Client by clicking through the options menu on the toolbar.

The Client uses posts to give the Server the user’s username and password in much the same way it posts to request image, by passing it some data. This username and password use a system of posts much like the requests for images. It sends the following string:

"username=" + username + "&password=" + password

If the username and password is correct the server will send a zero followed by a session key and the JSON index. When the Client receives the JSON index, it will build a tree of the all the images on the server. This tree is used then to build the TreeView which allows the user to select and images. Once an image is selected, the Client gets its name from the JSON tree and then requests the image from the server by posting the name to serve.php, as well as its session key. Then the server sends back the corresponding, base-64 encoded photo which the Client interprets and displays to the screen. This process is repeated every time a refresh occurs, which is done manually by the user.

Additionally, an Android application has been developed for the project that follows the same principles as the Windows Client. Functionally, it serves the same ends as the Windows client. However, graphically, it has a very different feel, customized for the small touchscreen of a mobile device. The tree is navigated through a series of taps through another tree, and once an hour has been selected, the app loads up a series of thumbnails for each of the images.

These thumbnails are created and stored on the server, and are served to the android client when it selects an hour. This makes it much easier to determine which images are important. Because data and memory are much more precious on a mobile device (compared to a Windows computer), it protects the user from having to cycle through every image in the hour sequentially to see which picture they should click on to enhance.

Both the Windows and Android Clients support saving downloaded images from the server to disc. The Android Client also creates a special folder for saved images and links it with the standard Android Gallery app so the user can view the photos from the Gallery like the rest of their pictures.

For a graphical explanation of the program flow for the Client apps, see Appendix A for a collection of flowcharts.

## VII. RECONSTRUCTION (STEP BY STEP)

### *A. Camera Device: Messrs. Coffield and Seitz*

1. Install the Pi Camera. To do this, press up on the camera connector tab, insert the ribbon cable (making sure that the leads are on the correct side), and press down firmly on the connector tab. For more detailed instructions, check the Pi Camera's instruction manual.
2. Connect the power, ethernet, and HDMI cables, as well as a USB keyboard, so you can begin programming the device. It will automatically boot after being plugged in.
3. Enable the Pi Camera. To do this, type 'sudo raspberry-config' into the command line, and navigate through the GUI to enable the pi camera. It will automatically reboot afterwards.
4. Download the required files. To do this, navigate to the home directory of the Pi by entering 'cd', then type 'git clone https://github.com/SecureCameraCapture/Camera.git' into the command line.
5. Progress through the server setup until step ____.
6. Code in the IP address of your Server. To do this, open up the file named 'sendPicture.py' under the directory /camera. Once the file is open, navigate to line 16 (starting with 'ip = ...') and input the IP address written down from the server setup.
7. If the default log-in user is not the factory user (pi), modify the line in takePicture.sh that calls sendpicture.py to include the proper full path to sendpicture.py.
8. Setup the crontab. To do this, type 'crontab -e' into the command line. If this is your first time using crontab, you will get a message asking which text editor you would like to use-- nano is recommended. Next, navigate to the bottom of the file and write a new line. with the following '* * * * * <file path>', where <file path> is the full path to the shell script, takePicture.sh, that was downloaded in step 4. If the Pi is on default user settings and the git clone was done properly (from the home directory), <file path> would be '/home/pi/camera/takepicture.sh'. This will configure the Pi to automatically run the scripts to take a picture once every minute.
9. Locate the Secure Camera Case STL files.zip file on the project website's Final Design page. (http://nathanlea.com/SecureCameraCapture/project.html). Download it and either send it to be 3D printed at an external source, or print it.
10. Once the case is complete, place the Raspberry Pi inside of it such that the ethernet and power ports align with the corresponding slots in the case.

11. Attach the Pi Camera to the lid of the case by slipping the lens through the corresponding slot on the underside. Secure it with tape or insulating glue.
12. Fold cables for the Pi Camera into the case and place the lid on top of it, clicking it into place.
13. Place the device where pictures should be taken. Face the camera outwards, plug it into an ethernet line with internet access, and plug it into power.
14. Pictures will be taken automatically every minute, so this device's setup is complete!

*B. Server: Mr. Dekoning*

1. Update and Configure the Raspberry Pi
   - A. Boot the Pi (flash the memory card if needed)
   - B. Run *sudo raspi-config* and complete the following
     - I. Change the default password
     - II. Disable boot to desktop
     - III. Use Advanced -> hostname to change the hostname of the Pi - make your own but remember it!
     - IV. Use Advanced -> Memory Split to allocate only 16 MB of RAM for the GPU - this improves performance as it gives the CPU more RAM!
     - V. Enable SSH - once this is configured and you have set up your server, you'll want to disable this
   - C. Download updates to Raspbian by running *sudo apt-get update*, *sudo apt-get updgrade*, and *sudo apt-get dist-upgrade*
2. Create a New User
   - A. Type and run the command *groups*
   - B. Use the list of groups, excluding the group *pi*, to create a new user with this command: sudo useradd -m -G (list of groups seperated by commas and no spaces!) (new username)
   - C. Set the password of this user with the command *sudo passwd (username you just added)*
   - D. Logout of the Pi user using the *logout* command and log back in using the new username and password you just created
   - E. Delete the user 'pi' now using the command *sudo deluser --remove-all-files pi* NOTE: there will be alot of text when you run this, that is fine!
3. Setup the Files
   - A. Create the directories for your webserver to run from using this command *mkdir -r /websites/secure/www*
   - B. Go to the /websites/secure/www directory and run this command to get the scripts that make up the server: *git clone https://github.com/SecureCameraSystem/Server*
   - C. Run *python generateKey.py* and follow the instructions to make the https session key and certificate for your server
   - D. Copy secure.key and secure.csr to /etc
4. Setup and Configure the Server
   - A. Download and install the required software packages with this command *sudo apt-get install nginx php5-fpm php5-curl php5-cli php5-mcrypt php-apc*
   - B. Setup the nginx web server
     - I. Edit the configuration file using *sudo nano /etc/nginx/nginx.conf*
     - II. In the first few lines, change 'worker_processes' from 4 to 2
     - III. Change port 80 to port 443
     - IV. Add this line directing the server to use *secure.key* and *secure.csr* for an https connection

V. Enable the "server_tokens off" line by uncommenting it

VI. Scroll down to the Gzip section and uncomment these lines

a. gzip on;

b. gzip_disable "msie6

c. gzip_min_length 1100;

d. gzip_vary on;

e. gzip_proxied any;

f. gzip_buffers 16 8k;

g. gzip_comp_level 6;

h. gzip_http_version 1.1;

i. gzip_types (leave the list as it is, just uncomment!)

VII. In the "http" block add these four lines to kick denial of service attacks out

a. client_header_timeout 10;

b. client_body_timeout 10;

c. keep_alive_timeout 10 10;

d. send_timeout 10;

VIII. Save and exit nano (CTRL-X, Y) then open another configuration file with the command*sudo nano /etc/nginx/fastcgi_params*

IX. In the second block of text add this line anywhere *fastcgi_param PATH_INFO $fastcgi_path_info*

X. Save that file and exit

C. Configure the PHP settings by opening *sudo nano /etc/php5/fpm/pool.d/www.conf* and uncommenting the lines that say *listen.owner*, or *listen.group*

D. Make the directories your website will be stored in using these commands: *sudo mkdir -p /websites/secure/www*, and *sudo mkdir -p /websites/secure/logs* (NOTE my website is called 'secure', if you want to change that replace 'secure' with your site's name from here on out!)

E. *cd /websites/(your site name)/www* to add an "index.html" file with some basic html of your choosing for testing

F. Add an "index.php" file that contains a basic php script for testing

G. Change the ownership of (your site's) folder with the command *sudo chown www-data:www-data /websites/(your site)* - this allows web users access to some resources (like those web documents you just made to test with)

H. Now configure nginx to serve your file

I. To register your files with the nginx server you must create entries in the "sites-enabled" and "sites-available" nginx directories, so *cd /etc/nginx/sites-available*

II. Register your directories with the "available" part by running *sudo cp default secure*

III. Link the available file to the enabled file with this command *sudo ln -s /etc/nginx/sites-available/(sitename) /etc/nginx/sites-enabled/(sitename)*

IV. Remove the default site with this command *sudo* *rm /etc/nginx/sites-enabled/default*

I. Now fix the /etc/nginx/sites-available/secure file

I. Open this file and scan the document for lines that look similar to these to modify

II. *root /websites/secure/www*

III. *index index.php index.html index.htm*

IV. *server_name (your site).local (your site).com*

V. Add these lines

VI. *error_log /websites/secure/logs/error.log error;*

VII. *access_log /websites/secure/logs/access.log;*

VIII. *location ~[^/].php(/|$){*

IX. *fastcgi_split_path_info 6(.+?.php)(/.*)$;*

*X. if (!-f $document_root$fastcgi_script_name){*

*XI. return 404;*

*XII. }*

*XIII. fastcgi_pass unix:/var/run/php5-fpm.sock;*

*XIV. fastcgi_index index.php;*

*XV. include fastcgi_params;*

XVI. *}*

J. Restart the nginx server with the command *sudo /etc/init.d/nginx relod*

K. Now visit http://secure.local/index.html and http://secure.local/index.php to test the server!

L. Finish the startup script by going back to /etc/init.d and adding these lines to the file "startup" that you previously made: *sudo mount /dev/md0 /websites*, and *sudo /etc/init.d/nginx reload*

M. Finalize those changes by running the command *update-rc.d /etc/startup defaults*

N. You have now setup your Secure Server!

*C. Windows Client Application: Mr. Lea*

1. Download the program's latest build from: https://github.com/SecureCameraCapture/WindowsClient/releases
2. Run the program, Secure.Camera.Capture.Client.exe.
   a. If a warning appears stating that the program is not signed by any certificate, press 'OK'. The program is safe, it just uses a method of connecting to the server directly requires a special permission, and Windows will not be able to identify a source for the program, so it requests user permission to verify it's a safe action.
3. If this is the first time running the Client application, it will request the Server's IP address. This was used previously in the Camera setup, and was obtained in step ___ of the server setup. Enter the IP address and click 'OK'.
4. The Client will then ask to either log in or create an account.
   a. If you have an account already created from previous use, use it to log in.
   b. If you do not have an account, follow the on-screen prompts to create one. They will request a registration number (by default, 123456789101112131415 is the registration number, but this can be changed on the Server.)
5. Once logged in, the main screen is displayed. It shows the tree of images pulled from the Server, as well as a space to display the currently selected image, a config button, a refresh button, a download button, and two arrows.
   a. Clicking on a year in the tree will expand it into months. Clicking a month will expand that into days, and so on, until hours are displayed.

From there, clicking a minute will display the picture taken at that minute.

   b. If a picture is displayed, the computer's arrow keys or the arrows at the bottom of the screen can be used to navigate to the next or previous image.
   c. If a picture is displayed, the download button at the bottom will allow the user to save the image to their hard drive in the same way as any saved file would on a Windows computer.
   d. At any time, the config button can be clicked, which allows for a reconfiguration of the Server's IP address, to connect to a new Server IP.
   e. At any time, the refresh button can be clicked, which flushes out the current tree and retrieves a more up-to-date one from the Server.

6. When finished with the Client application, simply close it using the X in the top right corner.

### *D. Android Client Application: Mr. Lea*

1. Register for the open alpha for this device's app here: https://play.google.com/apps/testing/com.sc3.securecameracaptureclient
2. Download the application to your Android phone (version 5.0 or higher). It can be found at https://play.google.com/store/apps/details?id=com.sc3.securecameracaptureclient
3. Open the application, it will then ask to either log in or create an account.
   a. If you have an account already created from previous use, use it to log in.
   b. If you do not have an account, follow the on-screen prompts to create one. They will request a registration number (by default, 123456789101112131415 is the registration number, but this can be changed on the Server.)
4. If you have not set the IP of the server it will request you do that now. Use the address written down in step ___ of Server setup.
5. Navigate through the tree of timestamps by tapping them. Tapping a year reveals its contained months, and tapping a month reveals its contained days, etcetera. Tapping an hour will open that hour, where images are stored by the minute. Thumbnails will appear as they are downloaded from the Server.
6. Tap on the image you would like to open fullscreen.
7. If you would like to download image, once the image has been loaded, press and hold. It will save to your Gallery for later sending and viewing.
8. When finished, press 'Back' to step backwards through the tree, until the app exits.

## VIII. ENCRYPTION DISCUSSION

### *A. What is AES?*

AES, or Advanced Encryption Standard, is a symmetric block-cipher encryption algorithm. It's a standard used by the United States government for protecting sensitive data. The original version of it was known as the Rijndael cipher, and was one of fifteen cryptography algorithms put forth to be tested and compared for efficiency, security, and ease of implementation in 1999. Since 2002, it has been the national government's standard for security. So, the question is, what is a symmetric block cipher?

A symmetric block cipher first takes a large piece of information, and breaks it into chunks of equal size. For example, we'll say there exists a file that is exactly 1,280 bits in size. AES uses a block-size of 128 bits, so that 1,280 total bits would be broken down into ten segments (blocks) of 128 bits each. A block cipher then uses a secret key (usually some randomly generated string of bits) in an algorithm that modifies each of the blocks. A simple example of an operation like this would be the classic Caesar cipher. Consider a 2-bit key equal to 3. In the Caesar cipher, each block has the key added to its value. So, a block that has a value of 47 would have the key of 3 added to it, resulting in a value of 50.

Once the operation has been performed on all of the blocks, it can be transmitted to its target destination. If an external, malicious party intercepts it, the data is meaningless. Suppose, for example, a message originally read “HELLO WORLD”. By adding 3 to each of the letters, it becomes “KHOOR ZRUOG”. The meaning of the message is lost unless the recipient has the key; if they have the key, they know to subtract 3 from each letter to return it to its original state. That’s where the term ‘symmetric’ comes from; the key used to transform the original data is used to retrieve it once it reaches its end goal. The same key is used to both encrypt, and decrypt.

The operation AES uses is, of course, a much more complex process than an addition. The type of AES used on the devices in this project is AES-256, which is considered strong enough to protect files designated as TOP SECRET by the US government. [5] AES-256 uses a key of 256 bits to perform its transformations on the 128-bit blocks. It performs a process known as ‘hashing’ on each of the blocks, using the key to transform their contents into something new. AES-256, in particular, repeats the hashing process in a total of fourteen cycles. It hashes the original contents, then hashes the hashed version, and again, and again.

### *B. Why AES?*

AES was chosen for this project simply because of how strong it is. The implementation used here is a variant known as AES-256; the number represents the key length, in bits. So, the key could be any one of 2^256 possible options. An interesting study on the security of AES-256 can be found in the references [6], summarized here.

On average, a potential hacker would have to try half of those to brute-force the correct one, so the average number of attempts would be 2^255. That’s roughly 57 quattuorvigintillion trials-- or, in scientific notation, 5.79*10^76. To bruteforce a passcode or key, a typical hacker would use a high-end graphics processing unit-- they just handle repetitive computation faster than a standard CPU. A typical high-end GPU can perform about 2 billion calculations (which means 2 billion key attempts) in a single second. So, suppose the hacker is actually a well-funded hacker collective and can afford to chain together a **billion** of these GPU’s to work together and try to crack the code. That means 2 quintillion key attempts per second. While that sounds really impressive, it’s still just a drop in the bucket compared to how many combinations there truly are.

With the number of seconds that are in a year (31556952), that makes for a grand total of 6.311*10^25 keys per year-- meaning a total of 9.173*10^50 years attempting to crack it just to get through half of the possibilities the key could be. This number is so immense, it needs a comparison for scale. The entire universe has only been in existence for roughly 14 billion years-- 1.4*10^10. It would take roughly 6.55*10^40 times the life of the universe, on average, to crack a single AES key.

On top of the time restriction, the study also details a real life scenario and the power required to perform these operations with a real supercomputer. In short, it would require about a hundred and fifty gigantic nuclear reactors running for the entire time (many, many times the length of the universe!) to power a computer to crack an average AES key.

In summary, AES is the method of choice for encryption simply because of how secure it is. There’s a reason the United States government entrusts their most important and classified information to it, and there’s a reason it became the international standard.

### *C. How is security implemented in the project?*

The bulk of our encryption on the project is done in transit. While pictures are being shifted from the Camera device to the Server, they are at their most vulnerable. A potential thief would have access to the Camera, meaning they could in some way intercept the transmission. So, it needs to be encrypted in its transmission over the internet, to ensure the connection between the Server and Camera are secure. In this project, that is done through an open source piece of software called OpenSSL. This software is commonly used to secure connections between two devices using a ‘certificate’-- a collection of important data items such as a name, organization/affiliation, and location to verify the identity of one device trying to initiate a connection with another. Our Camera has a certificate that matches one on the Server, and when this is proven, a connection is formed.

On this connection, the Server *only* accepts images, and does not do anything with them except store them away. This prevents any malicious software from executing, should an attacker somehow acquire the certificate file found on the Camera device and attempt to place malware on the server disguised as an image. An AES key is generated for each session based on a random number generator, and used by both the Server and the Camera-- the Camera for encrypting, the Server for decrypting.

Then, images are encrypted using that key, and sent over the internet. If someone were to intercept the image, they would receive nothing of value; for an example of what an ecrypted image looks like, please see Appendix D to find a comparison of an image with its encrypted version, courtesy of Aaron Toponce [7]. Because images are only stored in the camera's RAM and are deleted immediately upon sending, they are safe on the camera device-- and with OpenSSL's encryption backing them up, they're safe from capture up until they make it to the server.

The server is supremely secure in that it's remote, separated from the camera and client. It can (and should) be placed somewhere far away from the camera device-- in another building, at the very least, and behind some form of physical security such as a lock or even a safe. The key aspect of the server is that it should not be in reach of a potential attacker, ever. Additionally, an AES key is hard-coded into the Server. Upon image reception, files are immediately encrypted using that key, and stored onto the Server's hard drive as encrypted files. If the Client requests an image, and is verified to be legitimate and secure, the Server unencrypts the image to be sent.

Finally, the Server communicates with the Client. The Server forms a link to the Client through HTTPS (Hypertext Transfer Protocol Secure). HTTPS also uses SSL, similarly to the previous connection. It uses the same encryption method as before when sending files, AES-256. It also has an additional layer of security through RSA, which is used for key switching and transmission. An example diagram can be found in the appendix. This ensures that all communication between all devices is very, very secure. AES is used all the way through getting photographs from the Camera device and to the Client application.

Session keys are implemented in the form of a comma-separated text file, stored onboard the Server. When a Client successfully connects and logs in, a session key is generated for them. This session key is a string of 30 alphanumeric characters that is randomly generated and sent back to the Client after being appended to the text file on-server, and the client stores it. The Client must provide its session key whenever it requests an image. If it does not, or the key does not match any found in the on-server file, the request is denied.

The Client, the last piece in the puzzle, uses HTTPS to connect to the server. HTTPS uses TLS security protocol using AES encryption. A security certificate is downloaded from the Server and the program validates that it is the correct certificate. The certificate then tells the program to use the correct protocol and encryption across the connection. In order to accommodate the self-signed certificate, a custom validation function was developed to allow the client to accept the certificate as safe. The standard WebClient library handles all the encryption and handshaking required for a secure connection.

## IX. CONCLUSION AND AFTERTHOUGHTS

Overall, the project was a success. All requirements were met, and in fact, most were exceeded. Photographs surpass the minimum resolution by a significant margin, photographs are sent every minute, and they are easily retrievable using a Windows client. In fact, there are some features in the project that go far above the requirements posed. On top of the Windows client, an Android application was developed to make the photographs accessible from mobile. Both clients also have additional functionality on top of their initial requirements. Both are capable of not only retrieving and viewing images, but also saving them to their device's memory. The Android application even has extra features like visible thumbnails to make choosing an image even easier.

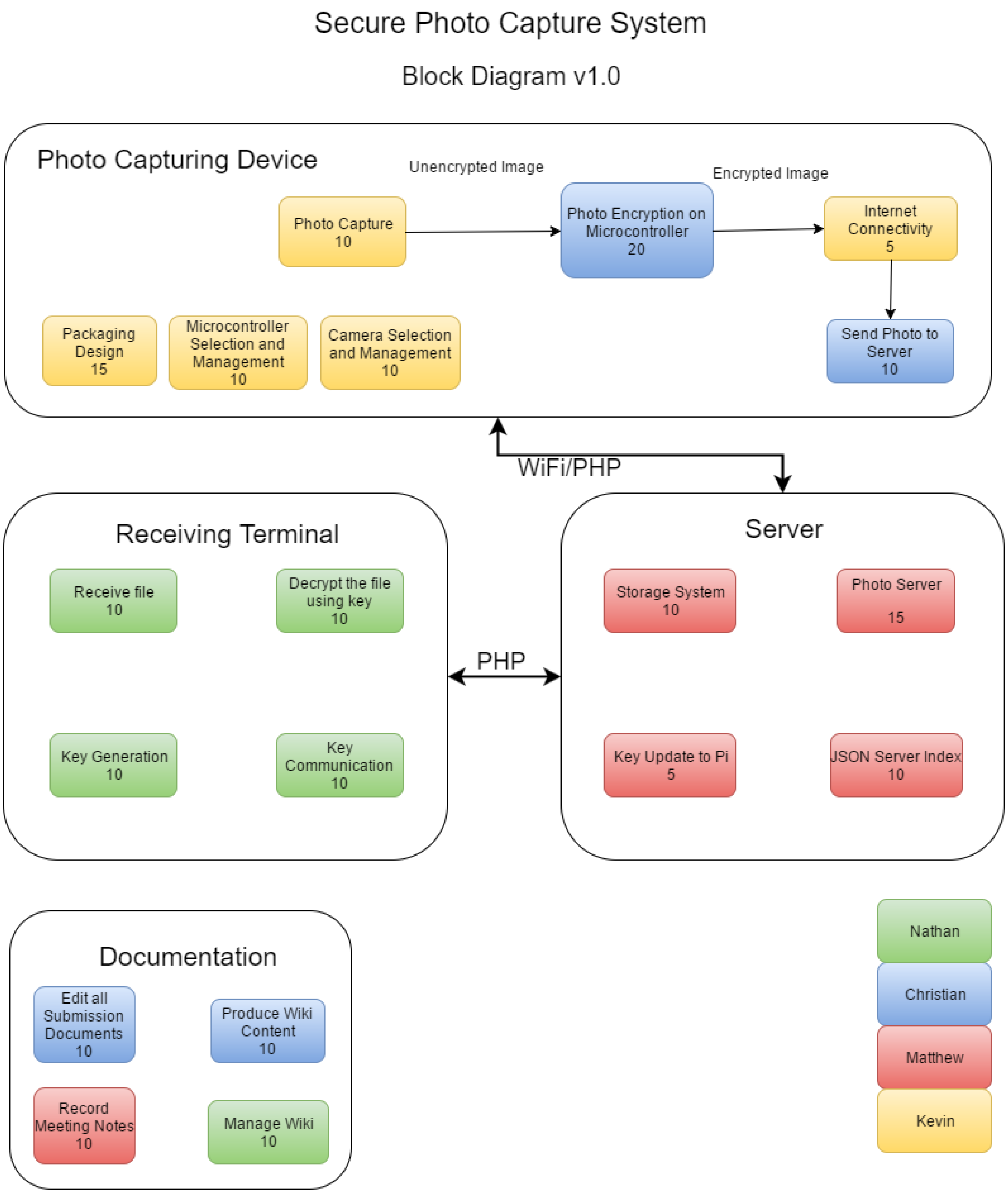
Complications arose throughout development, as should be expected from a project of this size. Originally, the Camera device was intended to have its base on a Raspberry Pi Zero (an inexpensive, weaker Pi) and USB web camera. However, after assembling the device and running test code, the Zero failed to process adequately; image capturing and processing using a USB camera caused the device to frequently crash, so a bump in power was necessary. We switched to a Raspberry Pi 2, which is a common household minicomputer, and ran a string of tests using both the USB camera intended for use with the Zero and a Pi Camera to determine if the camera was the issue. While the USB camera did not crash the Pi 2, its output was much lower resolution and occasionally produced a solid black image. With further research, it was discovered that many USB cameras are not fully compatible with the Pi, because the Pi's operating system is, in essence, a lite version of Linux, and therefore does not have full driver compatibility with all USB devices. In the end, the best option was to use the Pi Camera, which was specially designed to work with the device.

In the early stages of project design, the intent was to design the Server around RAID (redundant array of independent disks). Originally, a string of nine USB flash drives, each 16GB, were to function as the storage bank for the Server. However, this software RAID setup proved to be very unstable and commonly corrupted flash drives and the data on them. We decided to go with a much more reliable 1TB hard drive that has been proven not to fail over long periods of testing and time.

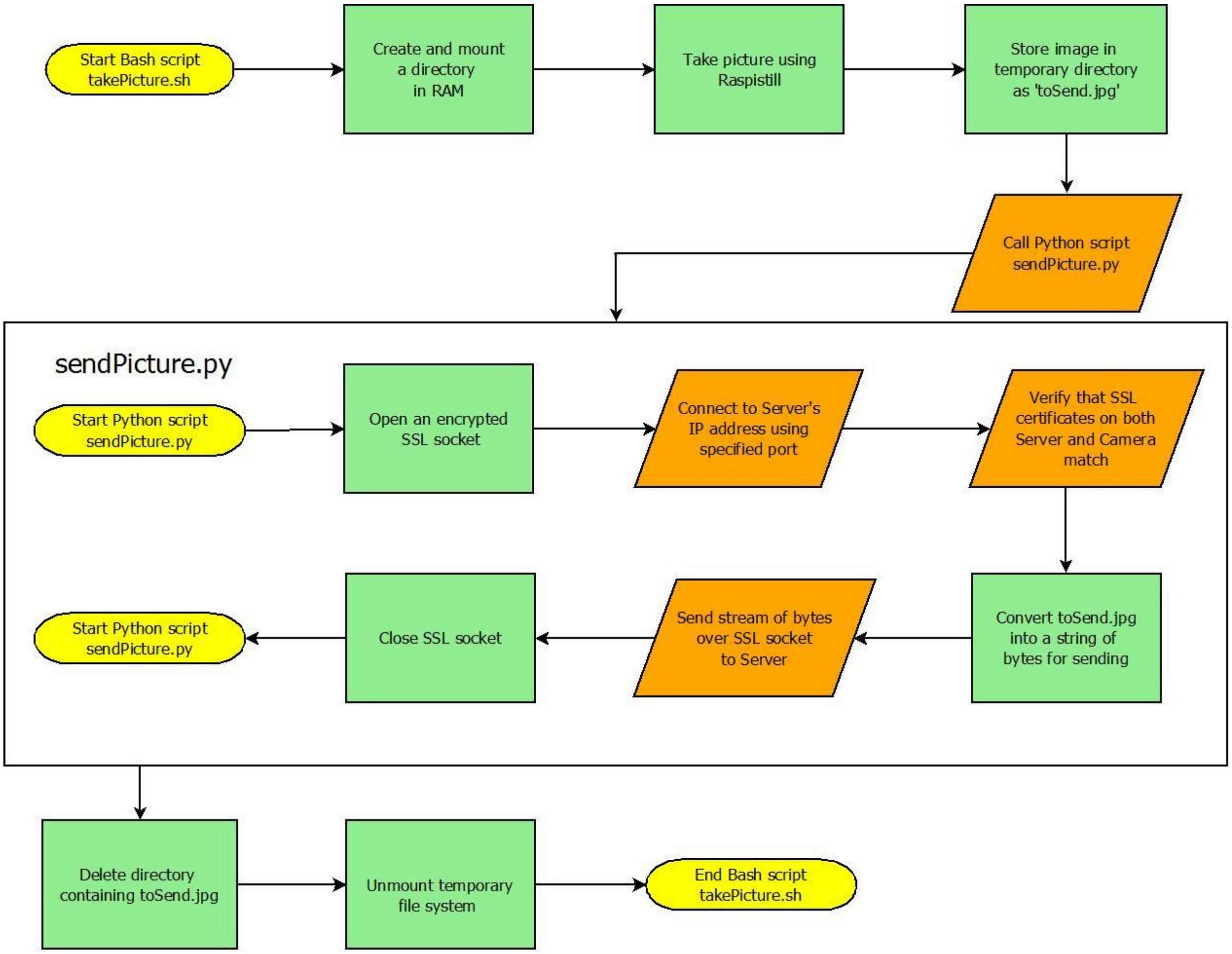
With more time and capital, the team has a variety of additional features they would like to add to the project. The most significant one would be a motion sensor that records a snapshot immediately upon detecting movement. With that, plenty of expansions would be possible—for instance, an alarm that triggers on motion detection and sends the picture to the device's owner. Wireless compatibility using WiFi would be another desirable feature, reducing the number of wires connected to the Camera device. A web application served from the Server to access the images using any standard browser and the login details created in setup would also be a useful addition. Signed certificates would also increase security if this device were to go to market, but those require a domain name and organization, which we deemed outside the scope of the current project.

# Appendix A: Block Diagram and Program Flowcharts

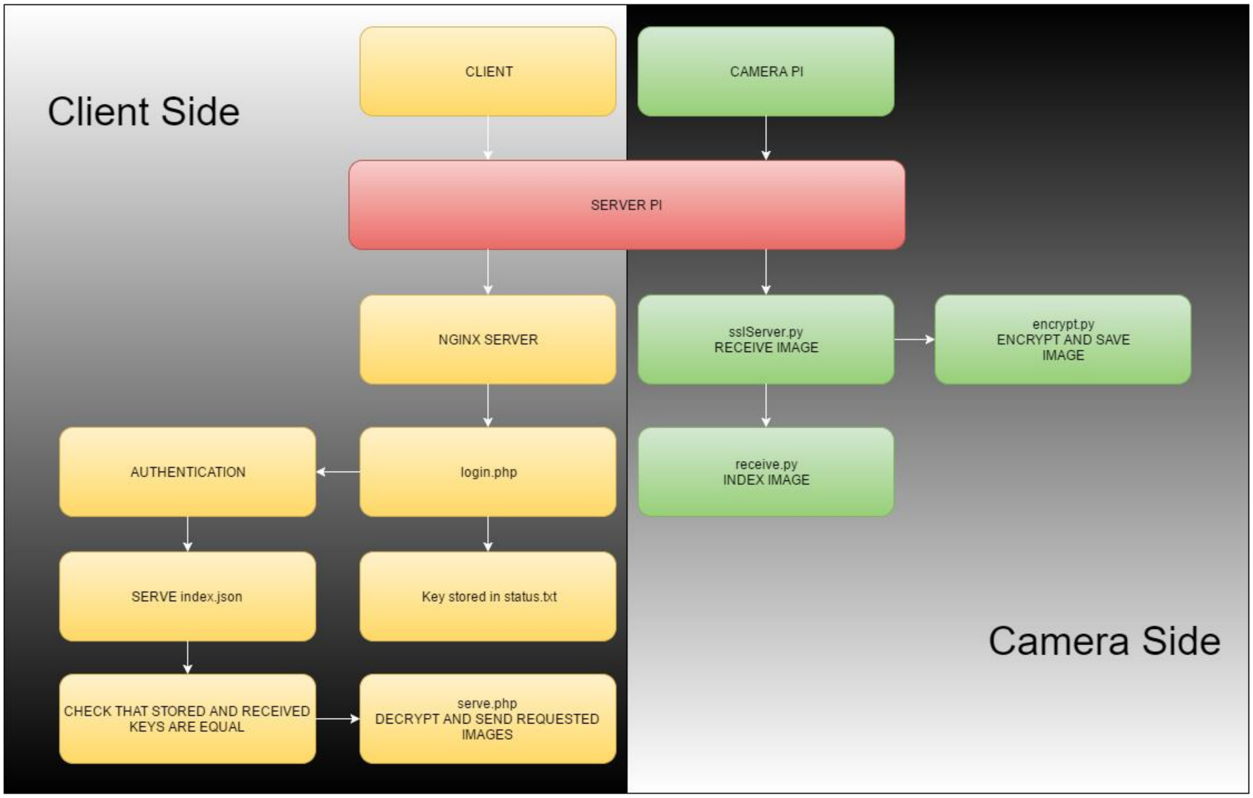
## Block Diagram



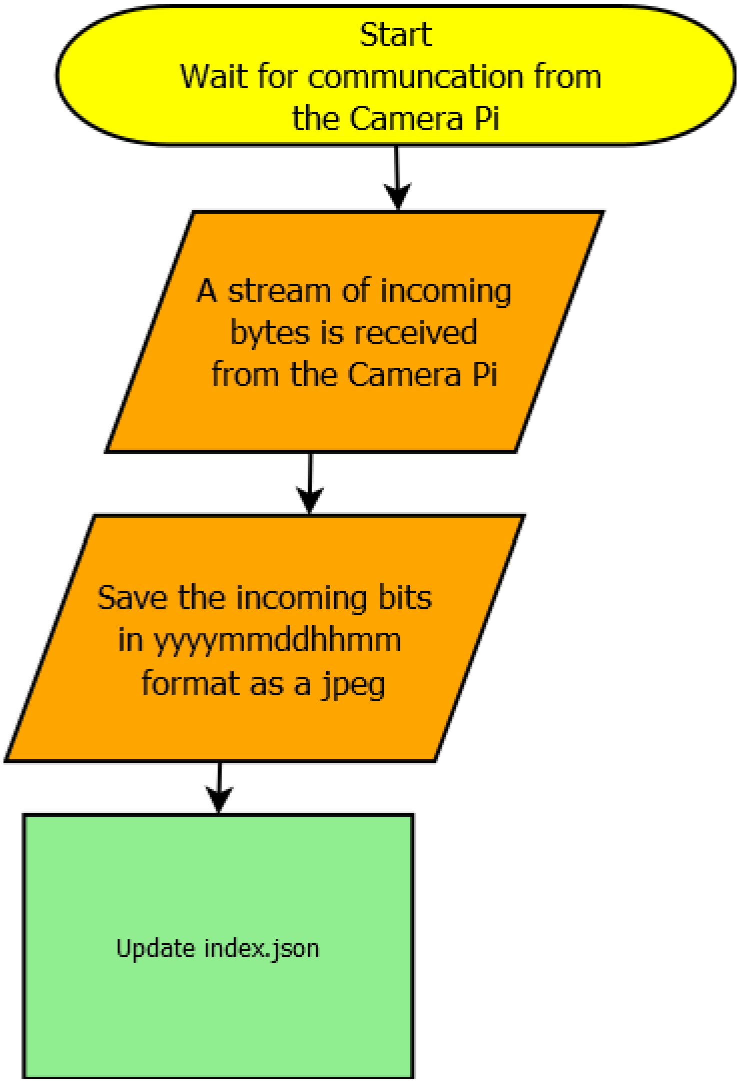
## Camera Device


Start Bash script takePicture.sh
Create and mount a directory in RAM
Take picture using Raspistill
Store image in temporary directory as 'toSend.jpg'
Call Python script sendPicture.py
sendPicture.py
Start Python script sendPicture.py
Open an encrypted SSL socket
Connect to Server's IP address using specified port
Verify that SSL certificates on both Server and Camera match
Convert toSend.jpg into a string of bytes for sending
Send stream of bytes over SSL socket to Server
Close SSL socket
Start Python script sendPicture.py
Delete directory containing toSend.jpg
Unmount temporary file system
End Bash script takePicture.sh

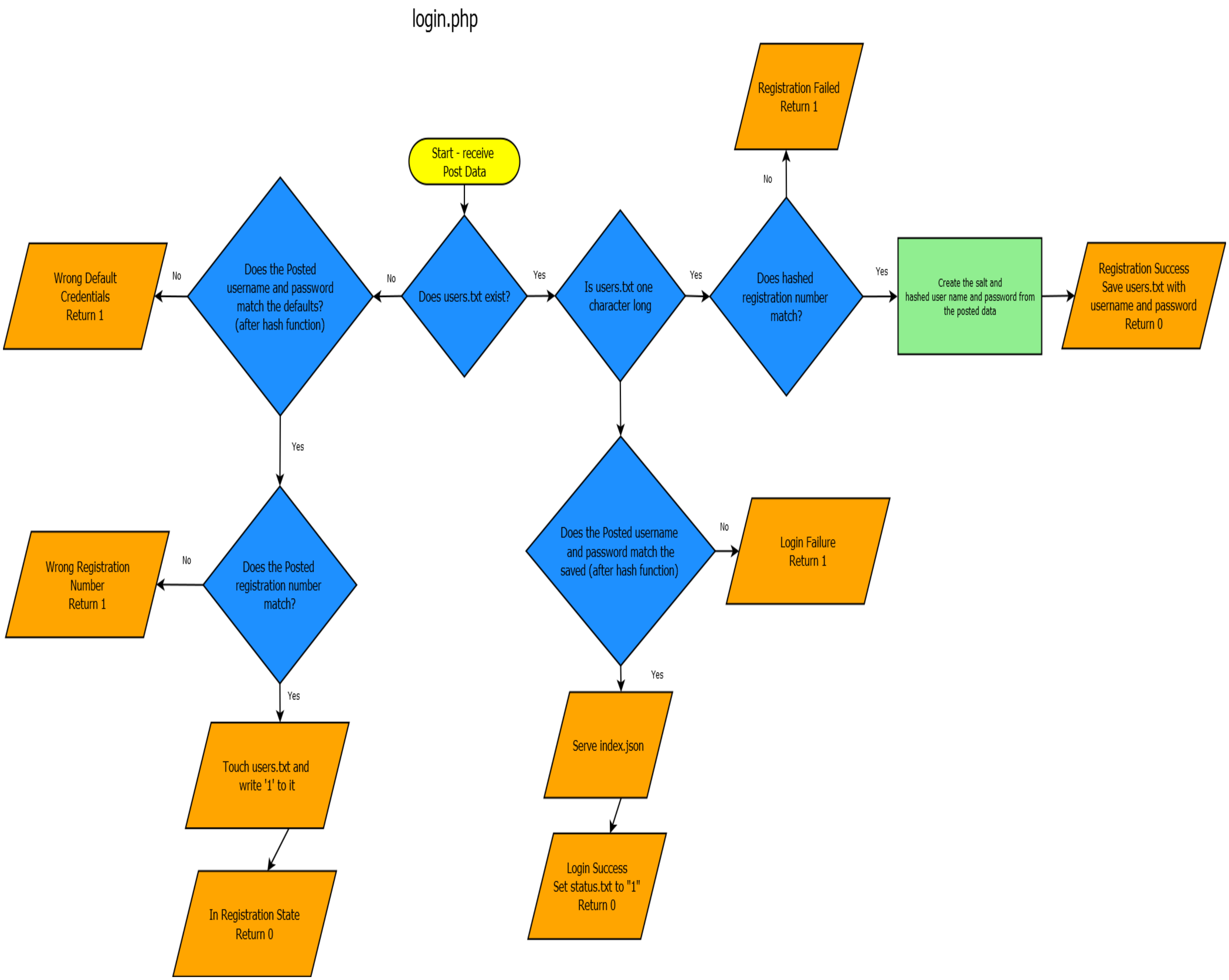
### Server: Overview



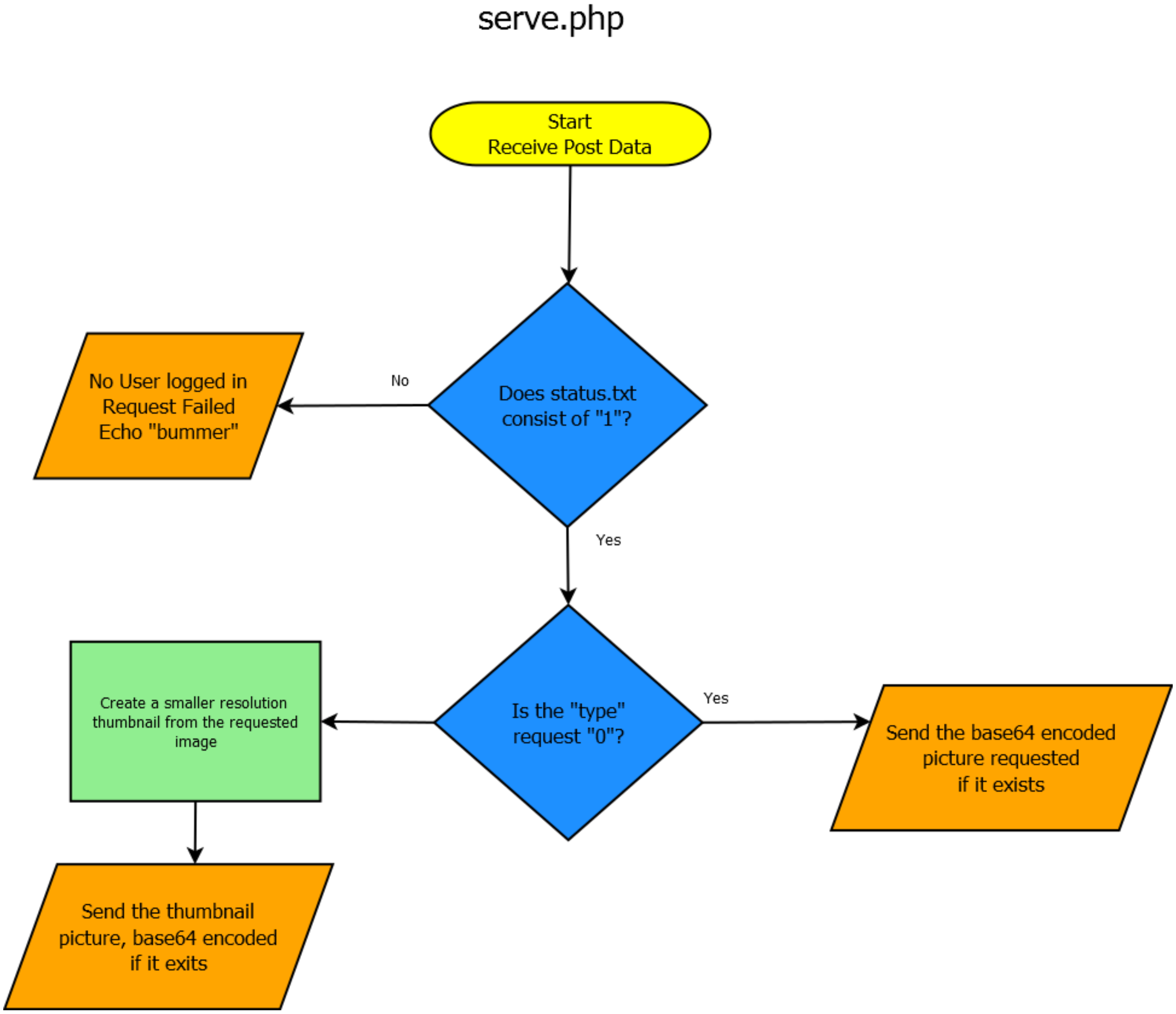
**Server: Camera Thread**


Start
Wait for communcation from the Camera Pi
A stream of incoming bytes is received from the Camera Pi
Save the incoming bits in yyyymmddhhmm format as a jpeg
Update index.json

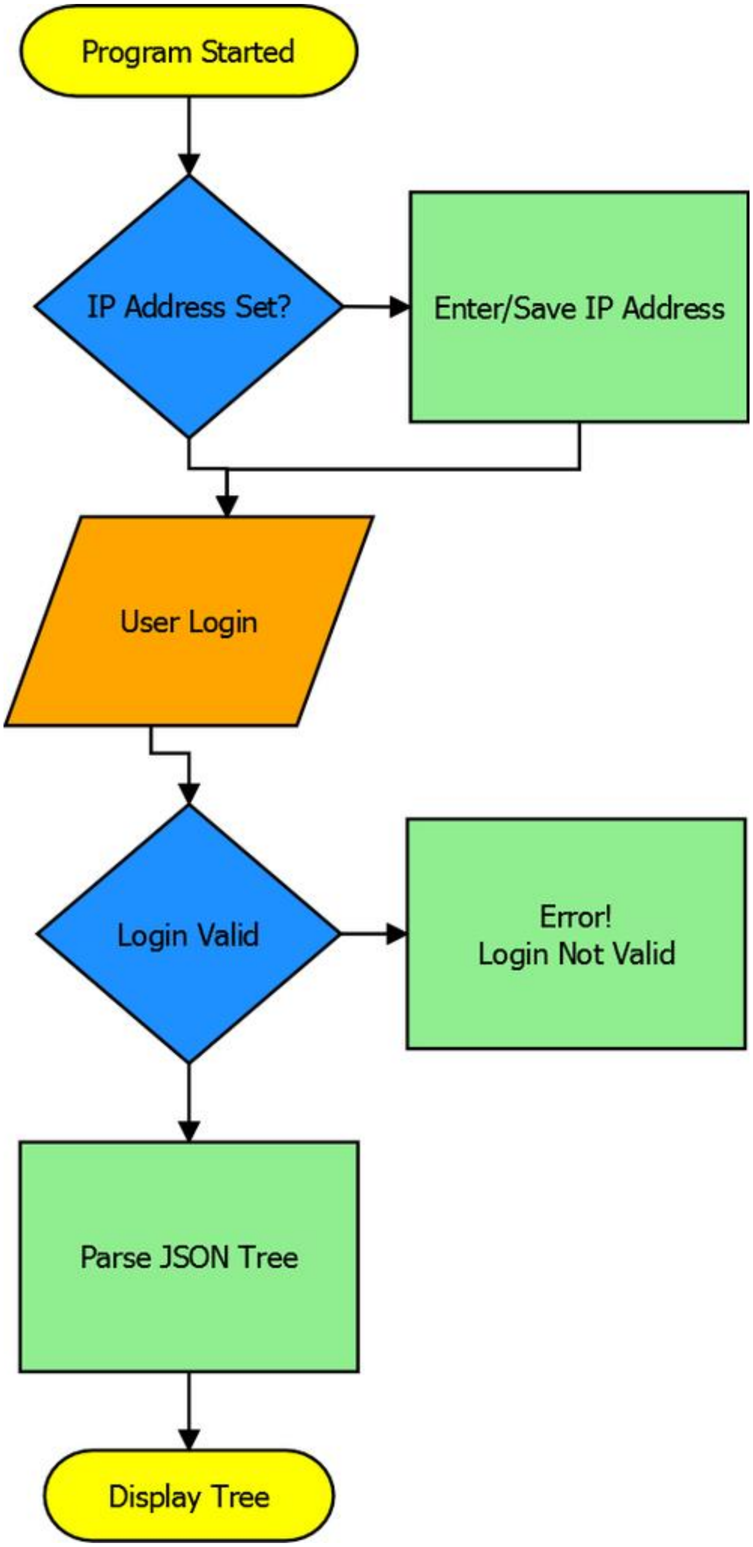
## Server: Login


login.php
Start - receive Post Data
Does users.txt exist?
Yes
Is users.txt one character long
Yes
Does hashed registration number match?
No
Registration Failed Return 1
Yes
Create the salt and hashed user name and password from the posted data
Registration Success Save users.txt with username and password Return 0
Does the Posted username and password match the saved (after hash function)
No
Login Failure Return 1
Yes
Serve index.json
Login Success Set status.txt to "1" Return 0
No
Does the Posted username and password match the defaults? (after hash function)
No
Wrong Default Credentials Return 1
Yes
Does the Posted registration number match?
No
Wrong Registration Number Return 1
Yes
Touch users.txt and write '1' to it
In Registration State Return 0

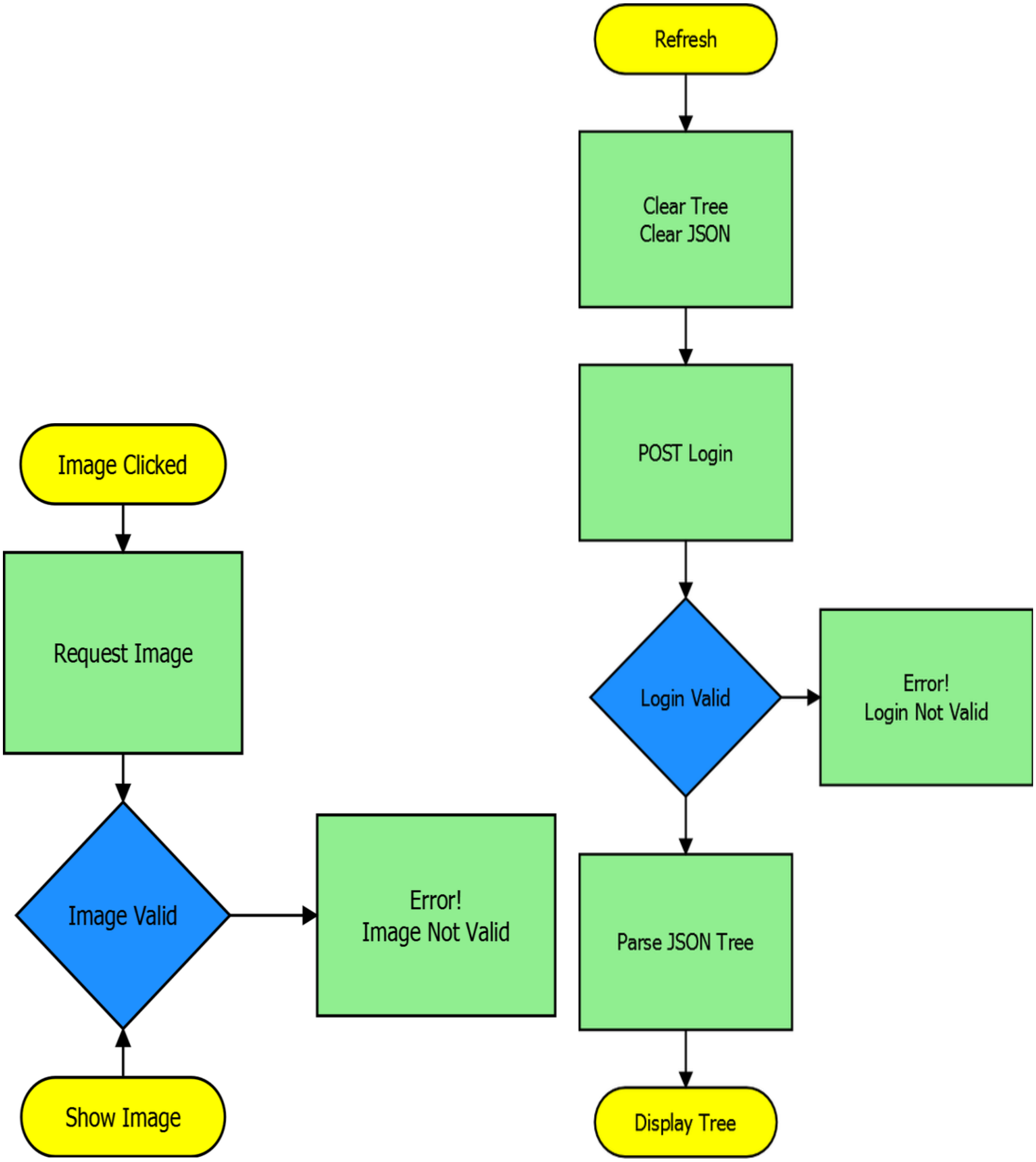
**Server: Serving Images**


serve.php
Start
Receive Post Data
Does status.txt consist of "1"?
No
No User logged in
Request Failed
Echo "bummer"
Yes
Is the "type" request "0"?
Yes
Send the base64 encoded picture requested if it exists
Create a smaller resolution thumbnail from the requested image
Send the thumbnail picture, base64 encoded if it exits

### Client Application: General Operations



### Client Application: Retrieving Images and Refreshing



# Appendix B: Code Repository

### Camera Device: takePicture.sh

```bash
#!/bin/bash

#mount a folder into RAM to store the picture
mkdir -p /home/pi/SecureCameraServer/webcam
sudo mount -t tmpfs tmpfs /home/pi/SecureCameraServer/webcam

#take the picture
raspistill -n -md 5 -o /home/pi/SecureCameraServer/webcam/toSend.jpg

#call sript to send
/usr/bin/python /home/pi/SecureCameraServer/sendpicture.py

#remove the picture from RAM
sudo umount /home/pi/SecureCameraServer/webcam
rm -r /home/pi/SecureCameraServer/webcam

echo "script complete"
#sleep 10
#done
```

### Camera Device: sendpicture.py

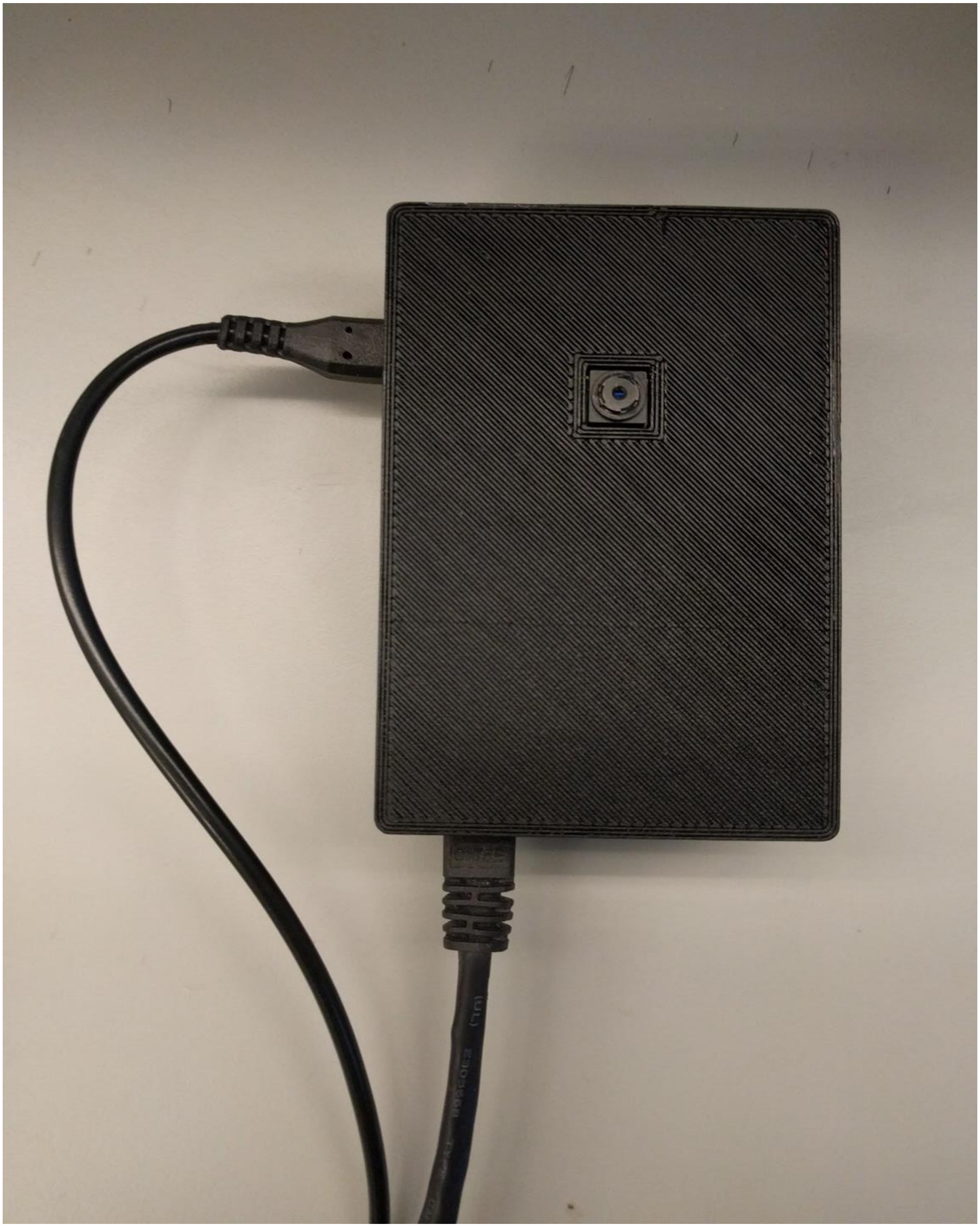
```
import socket, ssl, pprint, sys, picamera
from time import sleep

def get_bytes_from_file(filename):
    return (open(filename, "rb")).read()

if __name__ == '__main__':

   # camera=picamera.PiCamera()
   # camera.start_preview()
   # sleep(5)
   # camera.capture("/home/pi/SecureCameraServer/webcam/toSend.jpg")
   # camera.stop_preview()

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ip = '139.78.71.59'
    port = 10023
    #req cert from server
    ssl_sock = ssl.wrap_socket(s,
ca_certs="/home/pi/SecureCameraServer/server.crt",
cert_reqs=ssl.CERT_REQUIRED)

    ssl_sock.connect((ip, port))

    print repr(ssl_sock.getpeername())
    print ssl_sock.cipher()
    print pprint.pformat(ssl_sock.getpeercert())

    #simple http request-- use httplib in actual
    #ssl_sock.write(get_bytes_from_file(str(sys.argv)))

ssl_sock.write(get_bytes_from_file("/home/pi/SecureCameraServer/webcam/toSend
.jpg"))
    #read a data chunk, not necessarily all returned by server
    #data = ssl_sock.read()

    #note that closing the SSLSocket will also close underlying socket
    ssl_sock.close()
    exit()
```

## Server and Client Applications:

[For all Server and Client-related code, please see the open GitHub repository for this project at https://github.com/SecureCameraCapture . For the sake of brevity, it has been excluded from this report.

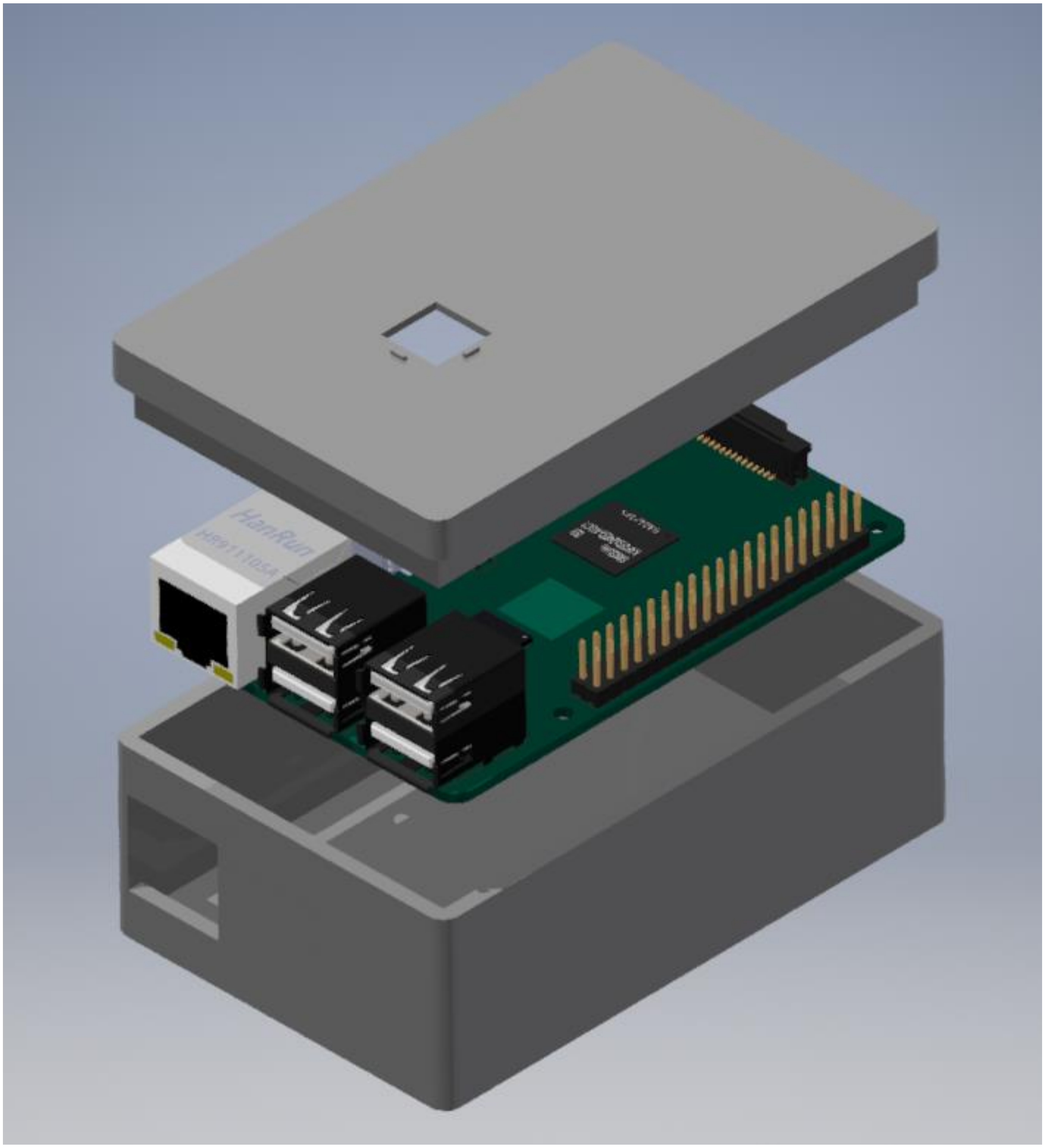
# Appendix C: Photographs and Screenshots of Final Designs

## Camera Device: Photographs

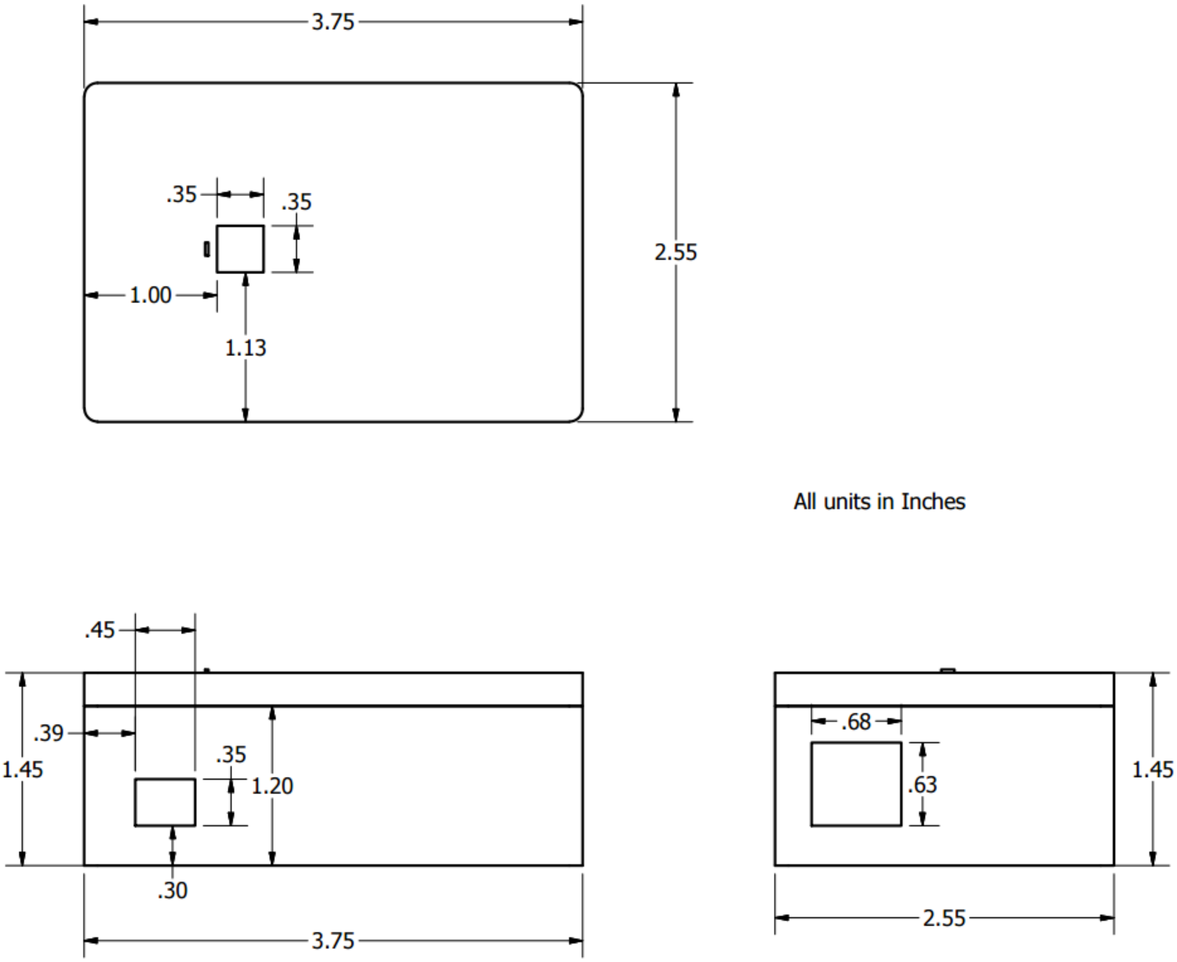


E244054 AWM 2896 80C 30V VW-1
Raspberry Pi 2 Model B V1.1
© Raspberry Pi 2014

**Camera Device: Case 3D Model**

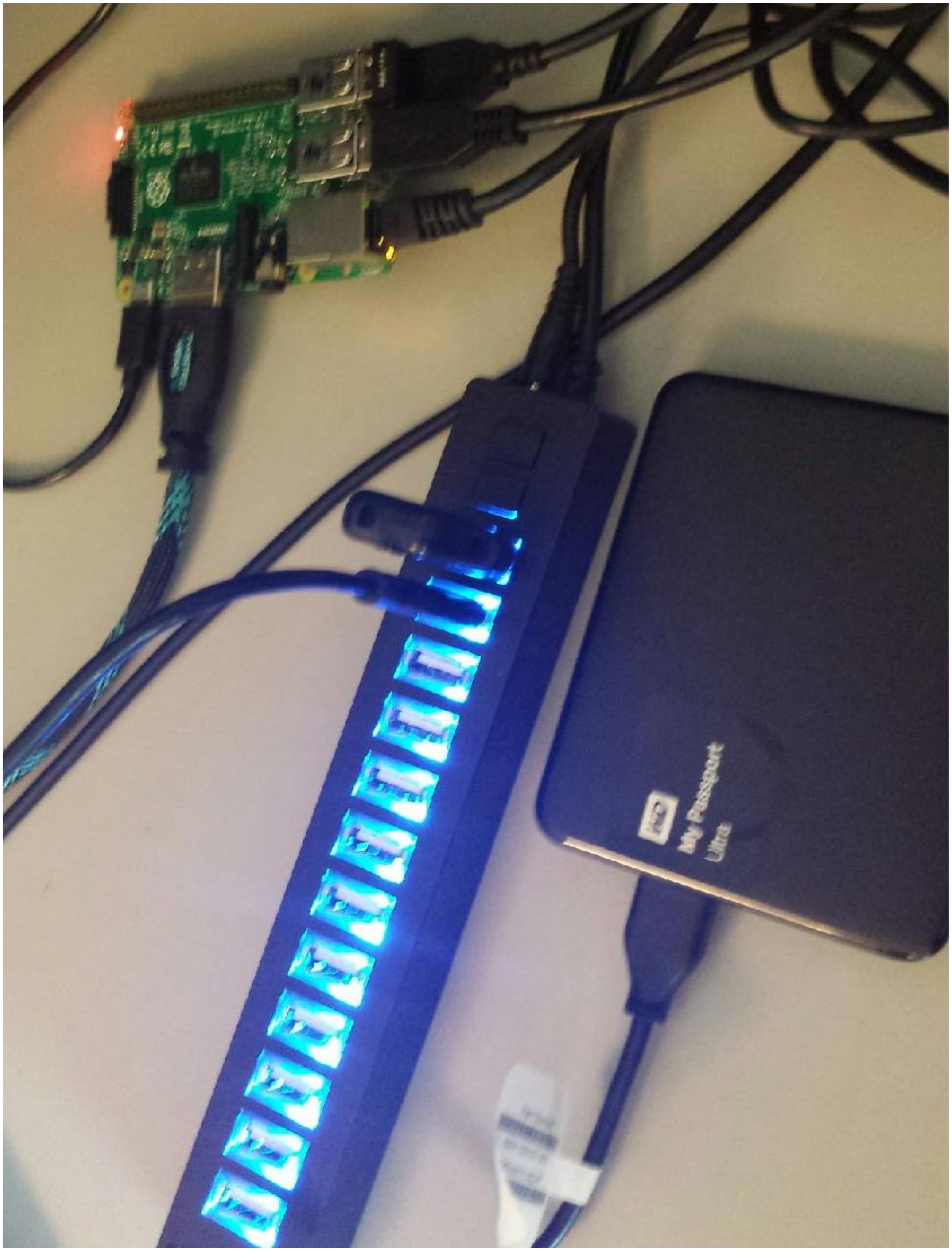


## Camera Device: Case Drawing

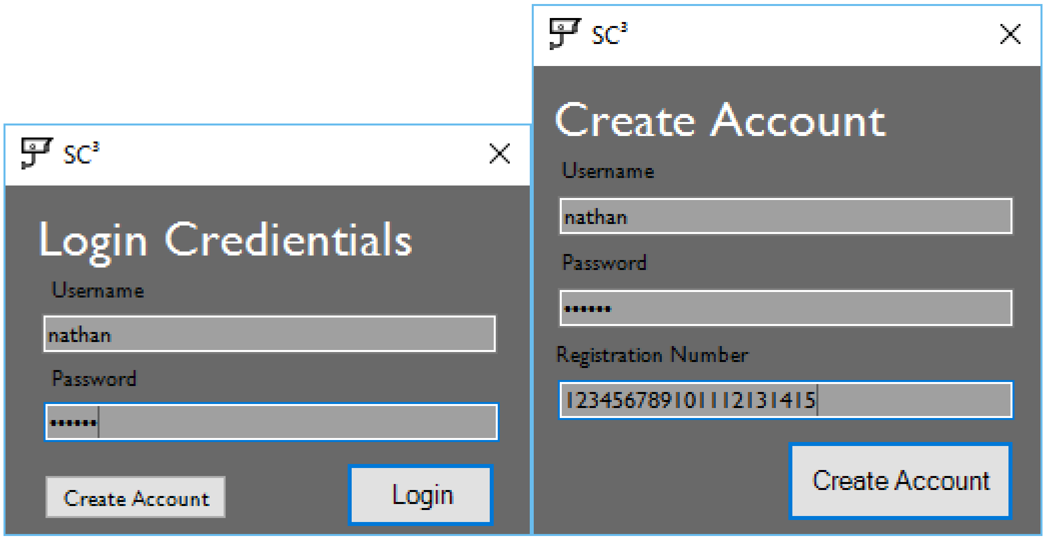


All units in Inches

**Server: Photograph**



## Client Application: Windows Screenshots



SC³

Login Credientials

Username

nathan

Password

••••••

SC³

Create Account

Username

nathan

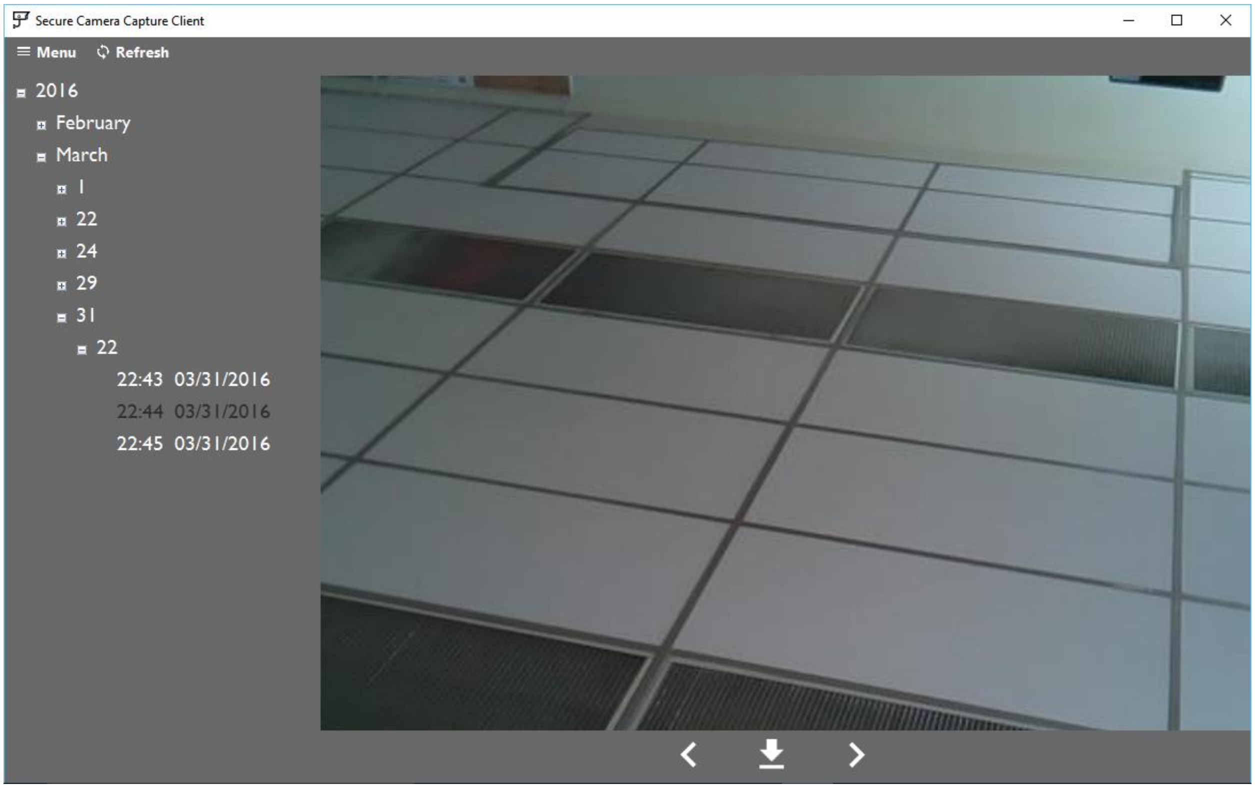
Password

••••••

Registration Number

123456789101112131415

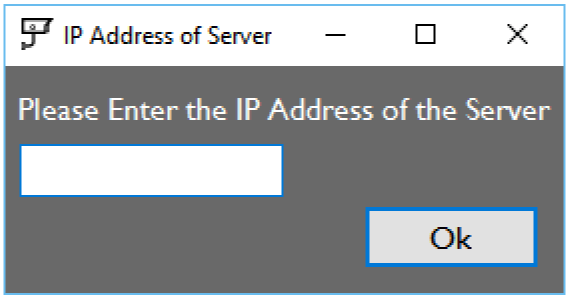
[The Windows Client’s log-in and ‘create account’ screens.]



Secure Camera Capture Client

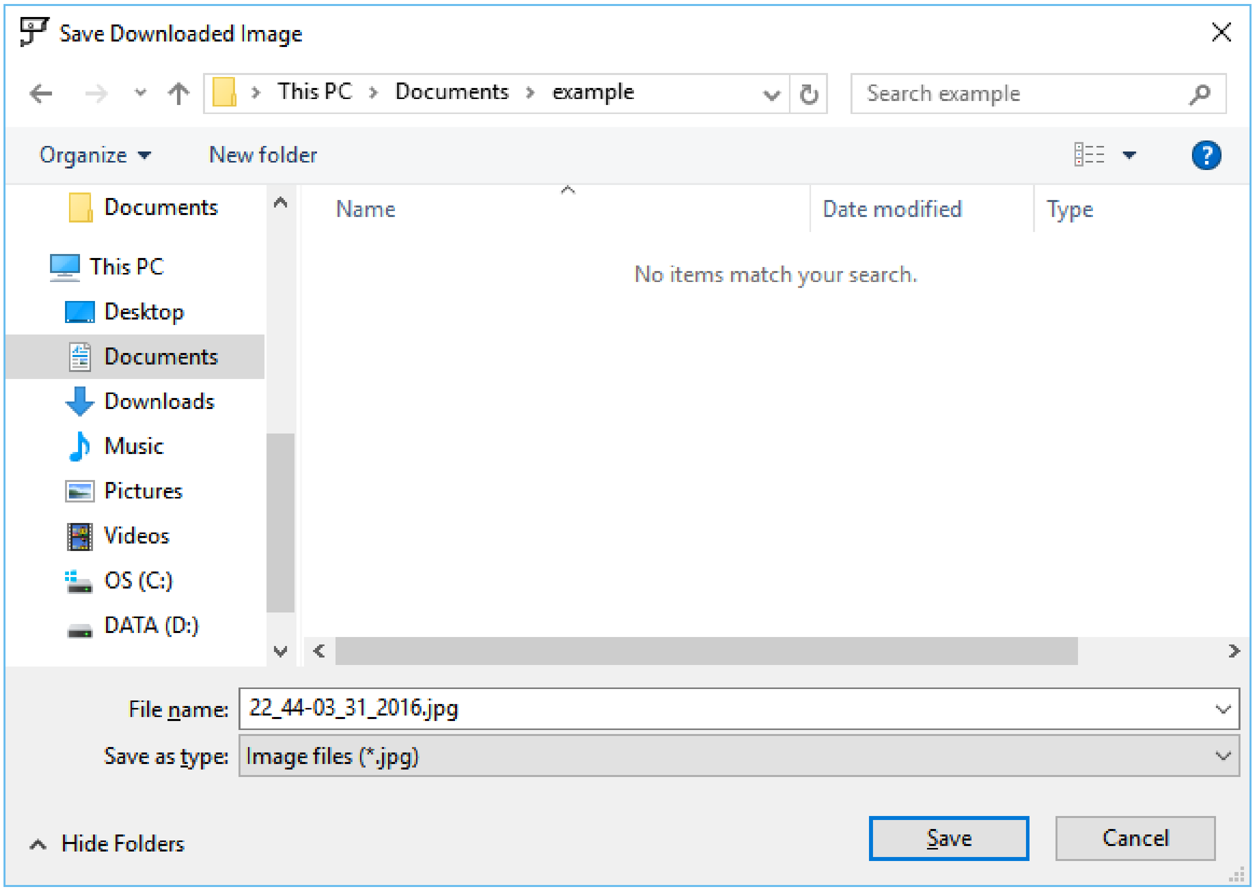
2016
February
March
1
22
24
29
31
22
22:43 03/31/2016
22:44 03/31/2016
22:45 03/31/2016

[A screenshot of the client, showing an image downloaded from the server. In this image, the camera was left pointing upwards, so the ceiling is on display.]



Please Enter the IP Address of the Server

[On initial setup, or when the user chooses to reconfigure the device, this screen is displayed to set the Server's IP address.]



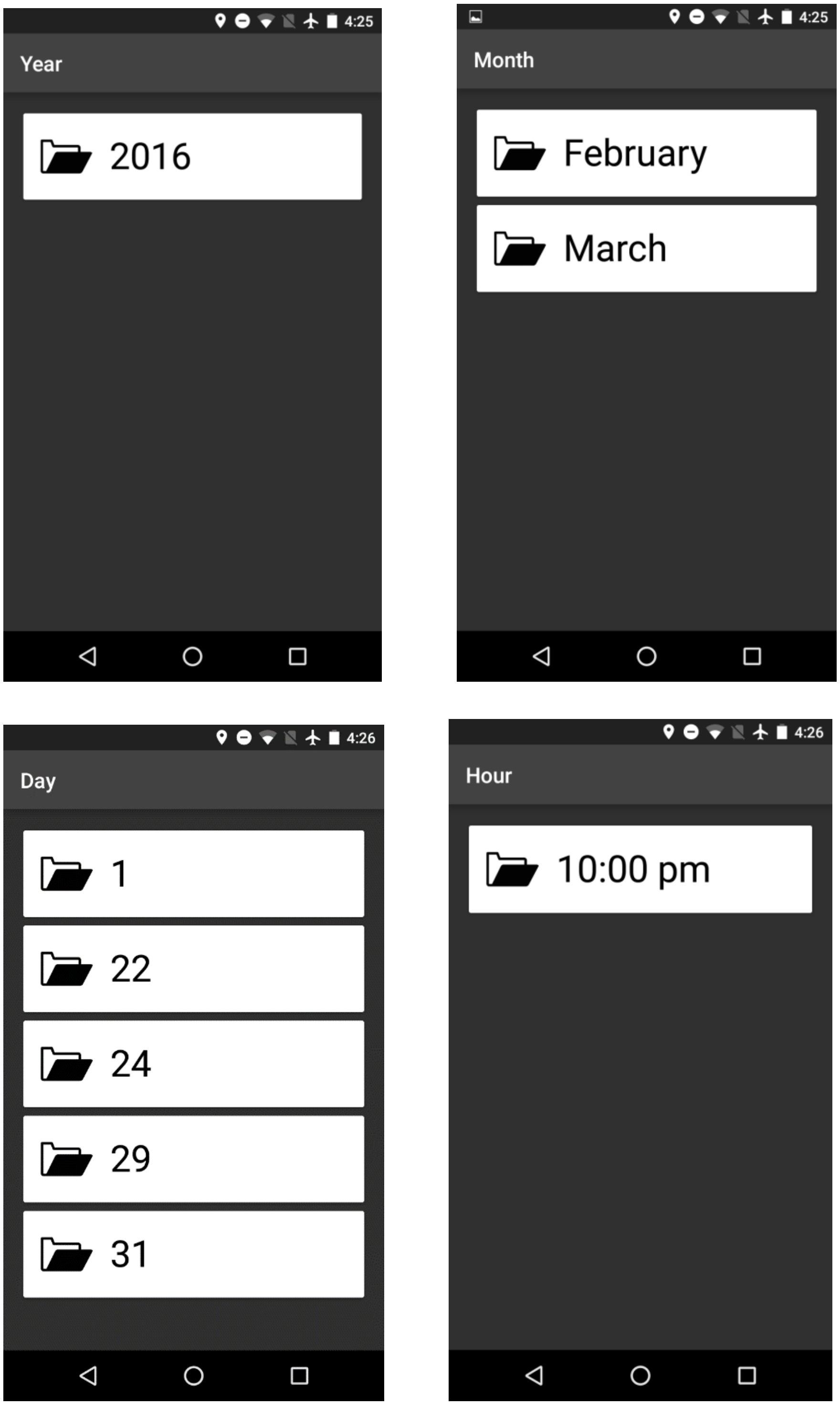
No items match your search.

File name: 22_44-03_31_2016.jpg

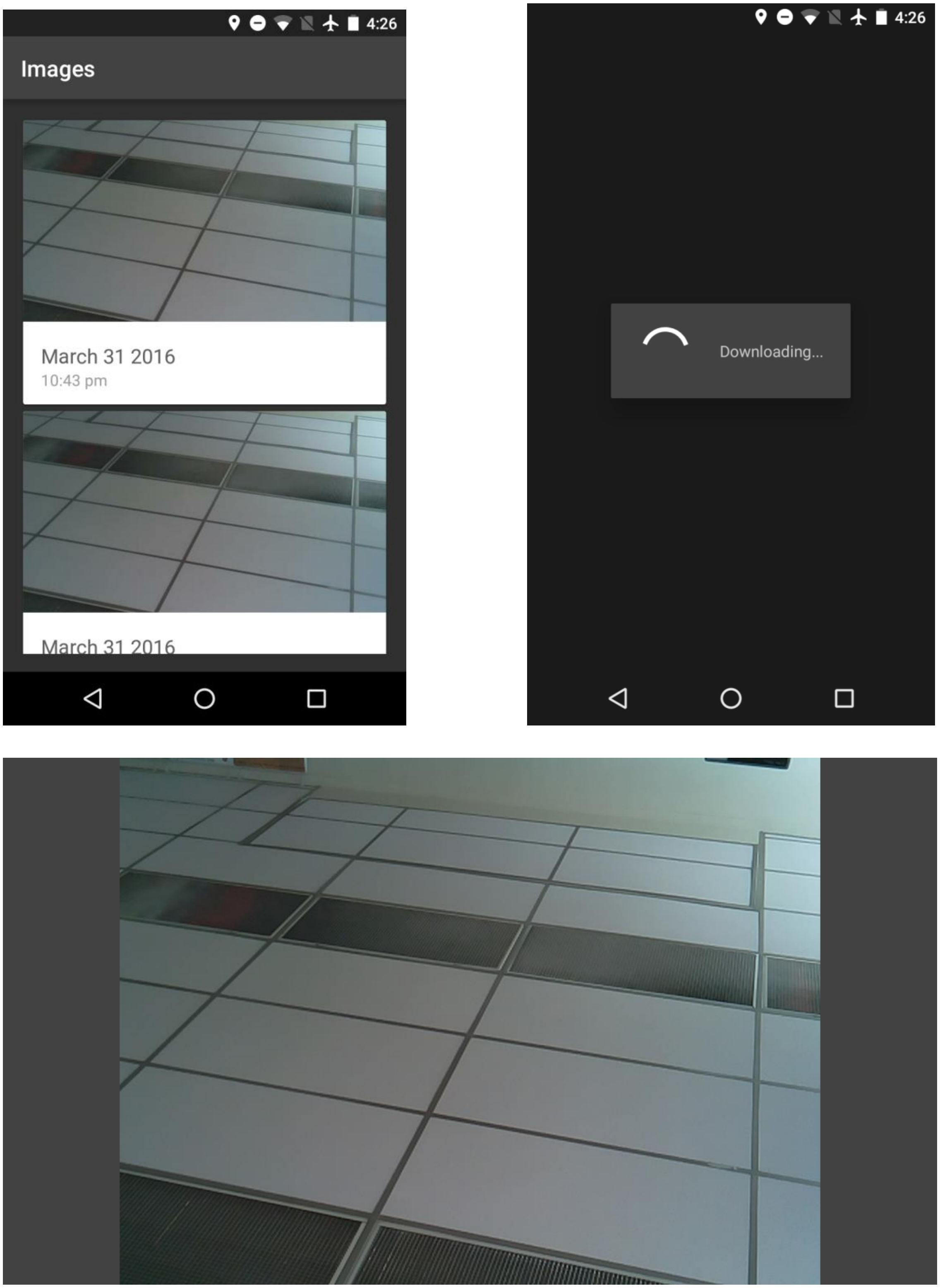
Save as type: Image files (*.jpg)

[When the user clicks the download button at the bottom of an image, a standard Windows save box will appear to save the image wherever the user pleases. By default, the image is named by its timestamp: hour_minute-month_day_year.jpg.]

## Client Application: Android Screenshots



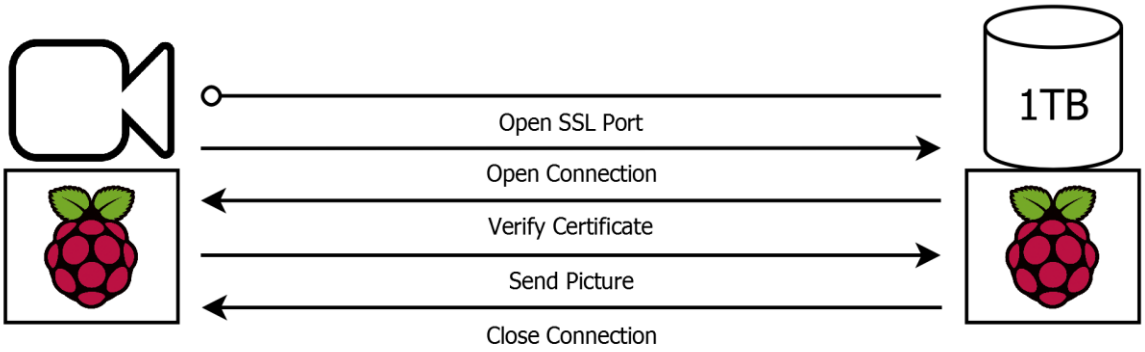
[Screenshots on navigating the tree of images on the android app.]



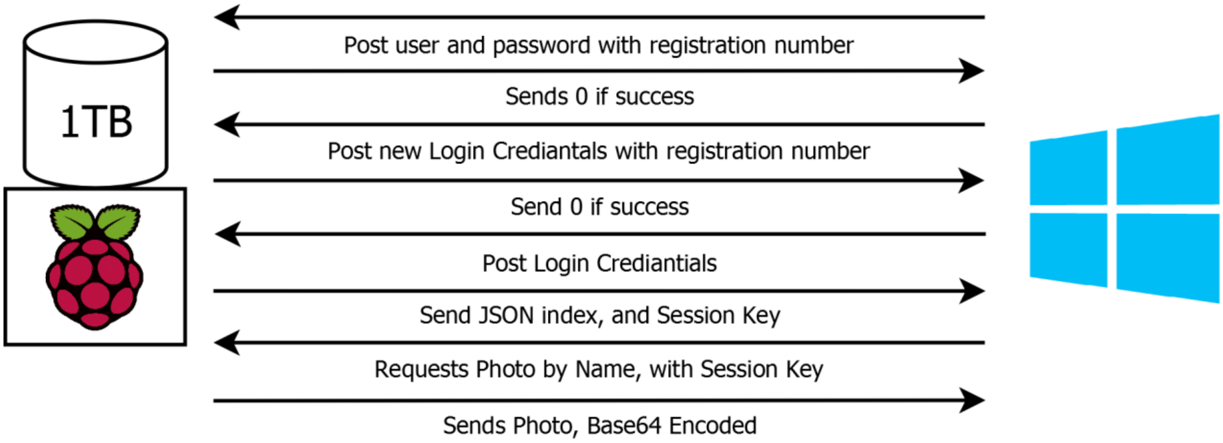
[Screenshots showing the thumbnails displayed when an hour is selected, an image is clicked and loaded, and the resulting full-size image.]

# Appendix D: Background Information

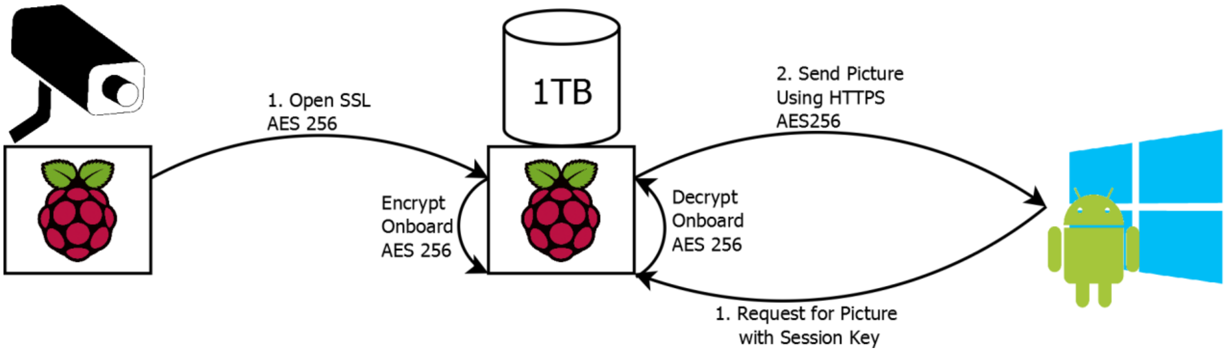
### Connection Diagrams



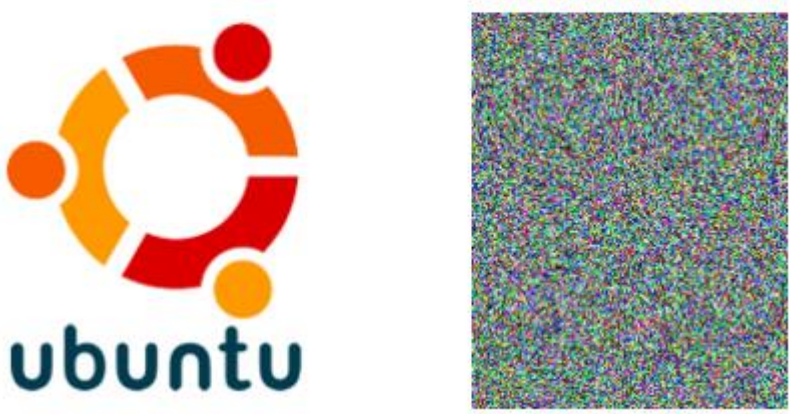
[A diagram outlining the back-and-forth communications between the Camera and Server.]


1TB
Post user and password with registration number
Sends 0 if success
Post new Login Crediantals with registration number
Send 0 if success
Post Login Crediantials
Send JSON index, and Session Key
Requests Photo by Name, with Session Key
Sends Photo, Base64 Encoded

[A diagram of the interaction between the Server and Client application.]



[A diagram of the encryption that occurs on all of the devices in the system. From left to right: Camera device, Server, Client application.]



Plaintext

Encrypted

[An example of what an image looks like pre- and post-encryption.]

## Mathematics for AES time-to-break calculations

*Number of possible key combinations* $= 2^{(number\ of\ bits\ in\ key)}$
*For a key of* 256 *bits, this equals a total of* $2^{256}$ *possible combinations.*

*On average, only half of these would need to be tested to get the correct key.*
*So, it would take, on average,* $2^{255}$ *attempts to guess the correct key.*

*Using a GPU rated at* 2 *gigaflops* (2 *billion calculations per second*) *would still produce a number far too low of use. So, for sake of example, use one billion, working in parallel with one another.*

$$2{,}000{,}000{,}00\ calculations\ per\ second\ per\ GPU * 1{,}000{,}000{,}000\ GPU's$$
$$= 2{,}000{,}000{,}000{,}000{,}000{,}000\ calculations\ per\ second$$

$$1\ year = \ 31556952\ seconds, according\ to\ conversion - metric.org$$
$$total\ calculations\ per\ year = \ 31556952\ seconds\ per\ year * \ 2e18\ calculations\ per\ second$$
$$= \ 6.3113904\ e25\ calculations\ per\ year$$

*total years to guess the correct key, on average, is equal to the number of keys possible divided by the number of keys attempted per year.*

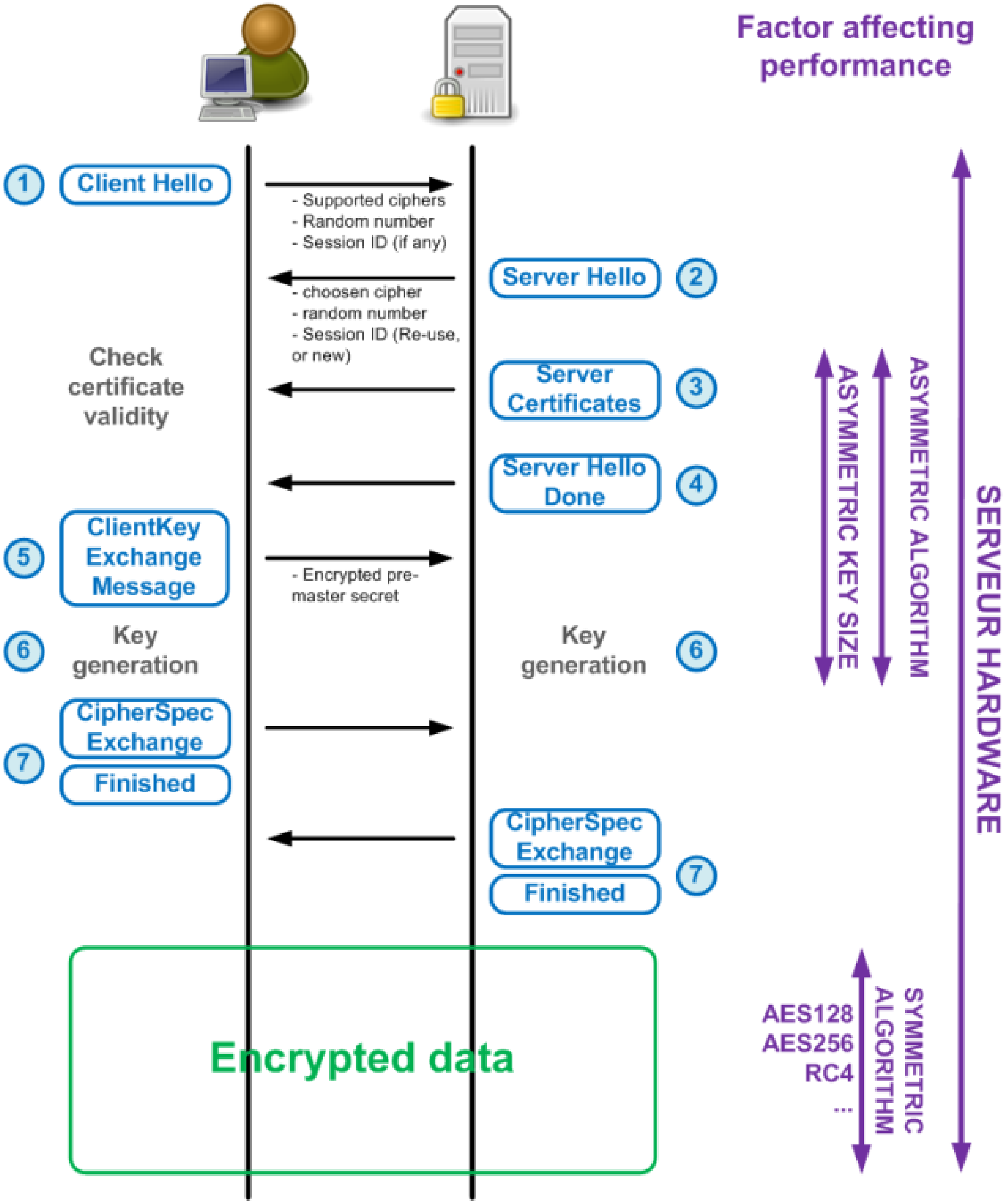
$$= \frac{2^{255}\ calculations}{6.3113904\ e25\ calculations\ per\ year}$$
$$= \ 9.1732e50\ years$$

*For perspective, the universe has existed for approximately* 14 *billion* (1.4*e*10) *years.*

$$\frac{9.1732e50\ years}{1.4e10\ years} = \ 6.55e40$$

*So, it would take a billion high powered GPU's roughly* 6.55*e*40 *times the current life of the universe to crack an AES* 256 *key, on average.*

**Example of RSA used in SSL, credit to alohalb [8]**



# REFERENCES

[1] “Raspberry Pi - Teach, Learn, and Make with Raspberry Pi,” Raspberry Pi Home Comments. [Online]. Available at: https://www.raspberrypi.org/. [Accessed: 09-Apr-2016].

[2] “DASHLANE EXPLAINS: Military Grade Encryption - Dashlane Blog,” Buffer Dashlane Blog, 2015. [Online]. Available at: http://blog.dashlane.com/dashlane-explains-military-grade-encryption/. [Accessed: 09-Apr-2016].

[3] “Scheduling tasks with Cron,” - Raspberry Pi Documentation. [Online]. Available at: https://www.raspberrypi.org/documentation/linux/usage/cron.md. [Accessed: 09-Apr-2016].

[4] D. K. Singh, “Multi-Threading & POSIX Thread APIs,” Multi-Threading & POSIX Thread APIs. [Online]. Available at: http://dipak123.info/htmlslideshow/multithread.html. [Accessed: 09-Apr-2016].

[5] “CNSS Policy No. 15, Fact Sheet No. 1 ,” Fact Sheet, Jun-2003. [Online]. Available at: http://csrc.nist.gov/groups/st/toolkit/documents/aes/cnss15fs.pdf. [Accessed: 09-Apr-2016].

[6] “Time and energy required to brute-force a AES-256 encryption key. ,” reddit, 2014. [Online]. Available at: https://www.reddit.com/r/theydidthemath/comments/1x50xl/time_and_energy_required_to_bruteforce_a_aes256/. [Accessed: 09-Apr-2016].

[7] A. Toponce, “ECB vs CBC Encryption,” pthree, 17-Feb-2012. [Online]. Available at: https://pthree.org/2012/02/17/ecb-vs-cbc-encryption/. [Accessed: 09-Apr-2016].

[8] “Image,” Aloha Load Balancer. [Online]. Available at: https://alohalb.files.wordpress.com/2011/09/ssl_handcheck2.png?w=640&h=763. [Accessed: 09-Apr-2016].