CryptoLock

"Secure Door Access using Blockchain Technologies"

Eduardo Portet and Laura Salinas
EC500 J1 - Connected World
Professor Babak Kia

Project Overview

In an interconnected world, security becomes increasingly difficult to find in our daily lives. While there are various methods to secure different aspects of our lives, we often find that these new technologies incur loopholes which lend themselves to hacking, tampering, and an overall loss of personal information.

Enter blockchain technology, a revolutionary way to think about transactions occurring in the digital and physical world. A simplified model for what blockchain does is as such: a transaction is first requested by someone and then sent to a series of connected computers known as nodes in order to be validated. Validity in the blockchain world can come from a variety of sources like cryptocurrency, contracts or records. Once verified this transaction becomes a “block” on the chain of other such transactions and remains unalterable. One of the key properties of this technology is that it stores blocks across its network that are identical in the information they hold. In doing so, the blockchain cannot be controlled by any one person and has no one point of failure.

Our motivation in using blockchain technology is thus to create an absolutely secure way to autonomously open and close doors through the use of tradeable digital tokens. By incorporating this technology we’ll be able to provide confidentiality, availability, and reliability when it comes to scenarios that require the use of a door to gain access. Arguably, the place we want to feel most secure is in our own homes, but if successful this project could be extended for use on any type of system utilizing locking mechanisms. By leveraging the power of the blockchain we can create a system that locks and unlocks only as transactions on the network are validated by measures a user implements. This type of technology can mitigate break-ins using physical lock picking measures, tampering of automatic locking systems, NFC systems, and use of expensive smart locking devices. A system capable of doing this would be operating on a cryptographically secure, decentralized, and tamper-proof network.

This project draws upon content from areas like hardware security, authentication, API’s and decentralized computing.

Technical Approach

This project can be divided into two sections one for software and one for hardware. The software aspect of the project establishes the connection to the Blockchain and controls the logic of the project. The hardware aspect transforms the accessibility and advantage of the Blockchain into a functional product for the end user.

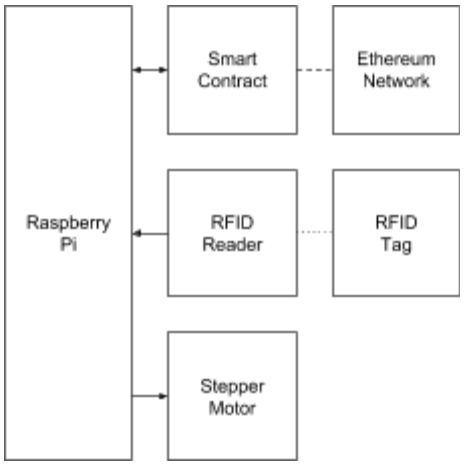

Figure 1. Design Overview of Components

Ethereum Network

The Blockchain protocol used for this project is Ethereum. Ethereum provides straightforward accessibility through APIs and a flexible platform to interact with the Blockchain using SmartContracts to trigger events and access information.

The software aspect of this project was built closely following the implementation of Project Oaken's ACORNS into the Ethereum Network. First, we needed to install the Ethereum blockchain on the connected device. We used a Raspberry Pi 3 for its preferred price range and convenient processing power to install the blockchain. In order to decrease the amount of time and storage data required to install the chain, we used the Ethereum test network Rinkeby. Once the chain is installed on the RasPi, we can create a node of the Ethereum Network using the Geth RPC client. This client allows us to host an Ethereum network using our device. Then we can interact with the network and the SmartContract using the *Web3.js* library.

SmartContracts

SmartContracts are a crucial part of this project since we use them to authenticate users. A SmartContract can be created using the high-level language, Solidity, and you can initiate transactions, declare variables or functions that can later be executed using *Web3.js*. The functions and variables have a privacy property that determines if they are accessible outside the SmartContract and through API calls. Once, a SmartContract is created the code is immutable and variables can only change the value through the use of existing functions.

In this project the SmartContract, named Gatekeeper, allowed us to interact with the Ethereum Network. The contract has two public facing functions. The main function is called `check`, it takes two strings as parameters, and returns true if the first parameter is a valid identification. It then changes the valid string to the second parameter and updates the blockchain through a transaction. The other function is called `getValidData` which is used for debugging the application as it returns the valid string that would make `check` return true.

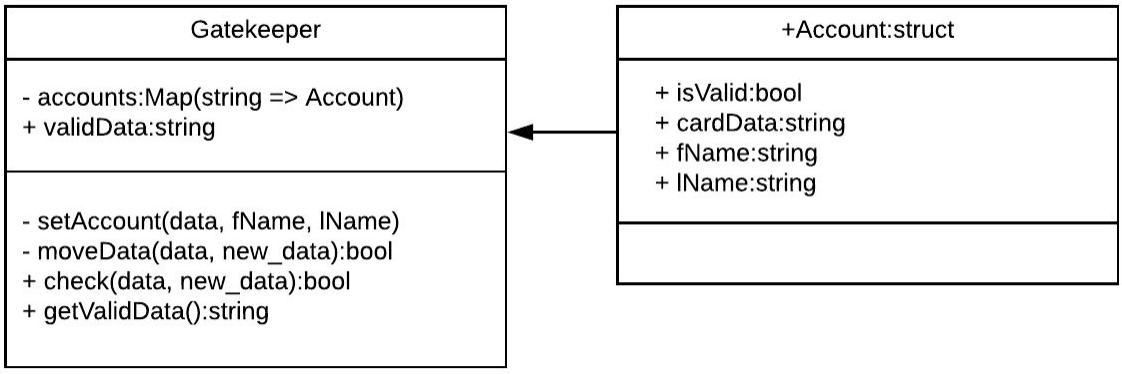

Figure 2. SmartContract UML Class Diagram

Node.js Code

This is the heart of the project as it combines all the components. This Node.js file hosts the *Web3.js* library that communicates with the Geth client node. This allows us to authenticate using the RasPi's Ethereum account and make function calls to the Gatekeeper contract. This code also keeps track of the states of the LEDs to help the user identify the status of the transaction, and execute the motor open and close Python scripts as needed. Once the program is executed it will enter a loop that waits for the next available tag in range.

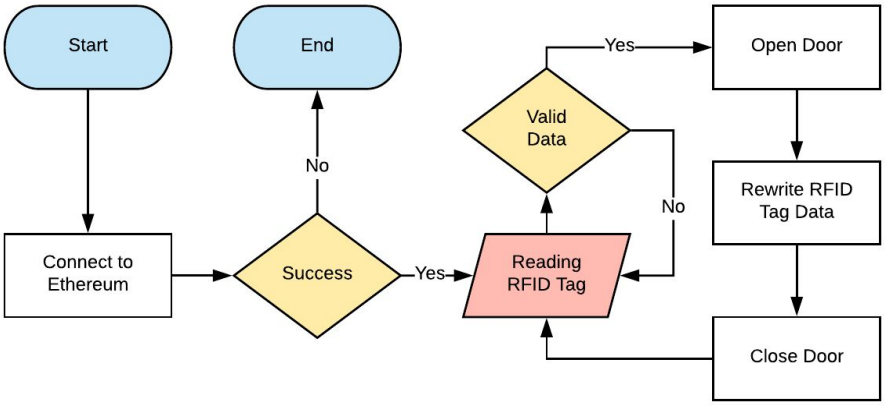

Figure 3. Gatekeeper.js Flowchart

On startup, the program initializes a connection with the Geth RPC node and authenticates to the Ethereum Network using the device's Wallet ID, if successful the program will then light the red LED and wait for a tag scan. On a tag scan, the data read from the tags will be used to determine whether or not a transaction can be initialized. At this point, we call the `check` function from the SmartContract and pass the data from the card and a random hashed value. If the card data was valid, the contract will then replace the valid data to the new hash and write that new hash to the RFID tag for future usability. Then we can call the Python scripts that would open and close the door, and wait for the next tag.

RFID Reader

The RFID reader functionality is dependent on two node libraries, the pn532 library and the ndef library. These libraries allow an easy interface to manipulate and exchange data between the tag and the reader. However, these libraries also limit the diversity of tag types that it can interact with; these libraries only work with NTAG215 tags. Any other type of tag will show up as undefined since the library doesn't know how to interpret the format in which data is stored. As such, the project has three different ways to display the interaction with the RFID reader. The first scenario is where the tag that was read is not in the format of NTAG215 and was flagged as an unidentified or corrupted scan. The second scenario is an invalid scan, this is the result of a scanned tag holding data that was deemed invalid by the SmartContract. Finally, we have the success scenario, where the tag was read flawlessly and its contents were authorized by the SmartContract resulting in the doors being opened.

SHA-3

In order to add a layer of security to this project, we decided to encrypt the information on the tags such that if a tag were to be misplaced or compromised, a bad actor would not be able to read its values and trigger the smart contract gaining access to a door. This encryption was accomplished through the use of the sha3 module within the *web3.js* library. The *web3.js* library is a collection of modules which contain specific functionality for the ethereum ecosystem.

To understand how the sha3 module works we have to first understand how the Secure Hash Algorithm 3 works. This algorithm is the latest release from the National Institute of Standards and Technology (NIST), and works by implementing the cryptographic family called Keccak. Keccak is based on an approach aptly named “sponge construction” which uses random permutations to “absorb” any input data and return or “squeeze” any output data. When data is absorbed data blocks are XOR’ed and stored in subsets and then transformed using a permutation function. In the squeeze stage, data blocks are read from this same subset and alternated with the transformation function.

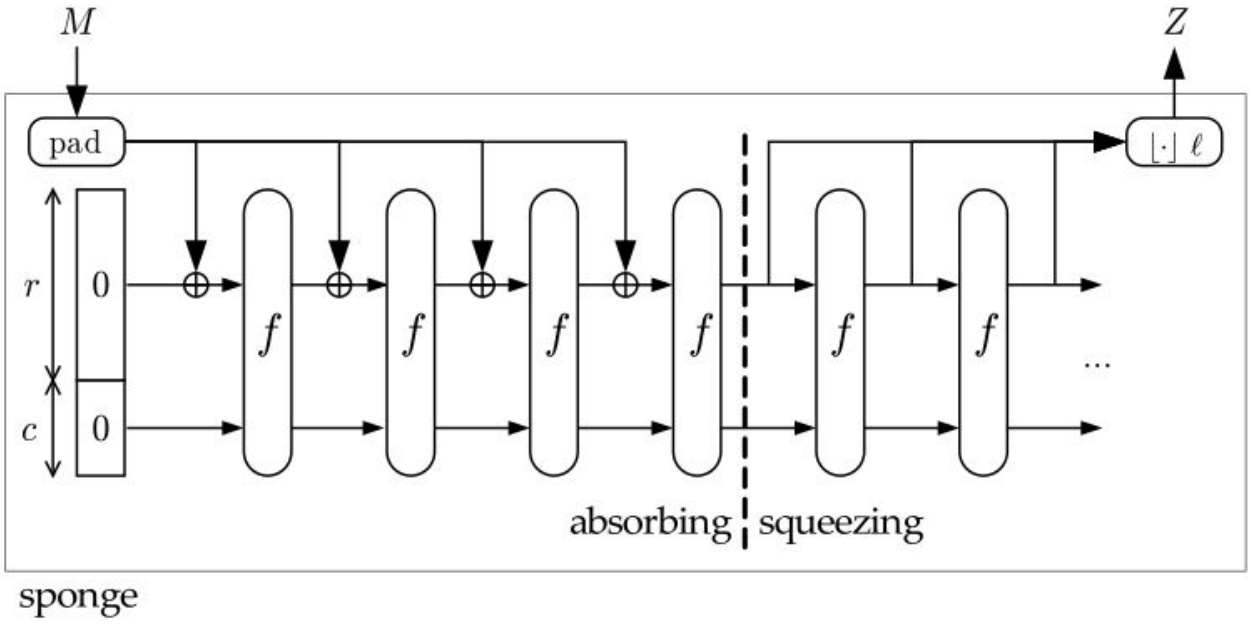

Figure 4: SHA-3 spronge construction visualization (*www.keccak.team)*

The encryption that is placed into the NTAG215 tag is handled by `var new_data = web3.sha3(web3.eth.blockNumber.toString()).slice(-4).toString();` in the gatekeep.js code. Once a tag is tapped and the action is validated through SmartContracts, we use the current block number and save it as a string, then we take the last 4 characters of this

string and hash them saving it as new_data. This new hashed value is then written back to the tag and updated in Ethereum, encryption is then complete.

Motor Mechanism

Since the motivation behind this project was to model the secure opening/closing of a door, we wanted to have a physical representation of this motion being triggered by smart contract interaction on the blockchain.

When deciding between physical mechanisms we thought of choosing between either a Sparkfun 400steps/rev stepper motor or the Adafruit 12VDC lock style solenoid. The decision between the two basically came down to how readily we could set up and integrate the mechanism with the Raspberry Pi. The solenoid needed an additional transistor and diode as far as wiring and would need much more current.

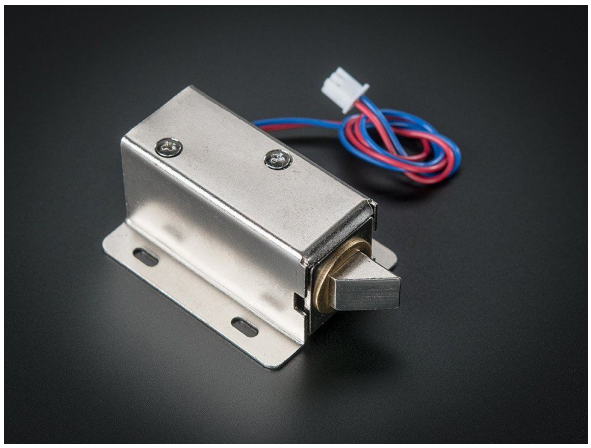

Figure 4. Locking solenoid

Figure 5. 400 steps/rev stepper motor

Set up for the stepper motor was fairly straightforward once pinout diagrams were laid out and reference voltage was set. The first step in setting everything up was to figure out which of the 4 wires from the motor corresponded to the coils that were in phase with each other. The motor we acquired for this project was a 4 wire bipolar stepper motor which is 2 phase and thus has 2 groups of coils. The motor driver alternates polarity to the coils in order to turn the rotor.

To figure out which pair of wires belonged to a coil we used a multimeter to test the ends of wire pairs. If there was reading other than 0 ohms between the wires then they were not part of the same coil. For our motor, the red and blue wires were one coil and the green and black wires were the other. In order to drive the motor and not require

excessively complex circuitry, we added the Elegoo Stepstick A4988 module to control the changing in polarity for the rotor turns. Once that was established we found the datasheet for the module and *Figure 6* represents a basic pinout diagram for 4 wire bipolar motors with the motor driver and a microcontroller.

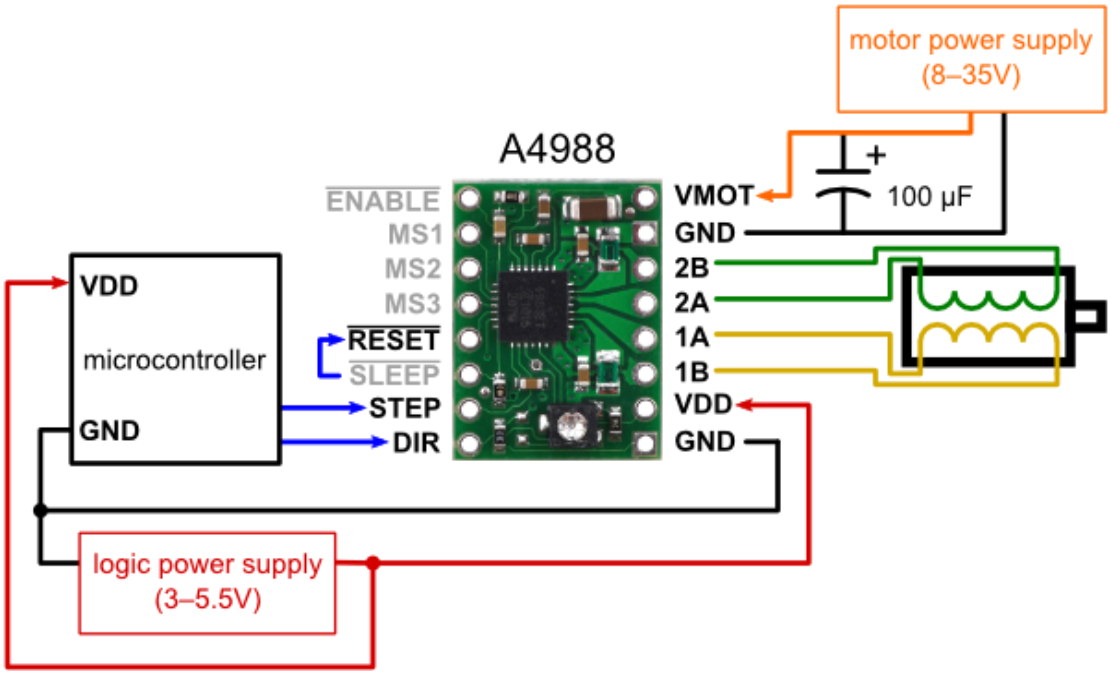

Figure 6. Motor driver general pinout diagram

For the setup with a Raspberry Pi, the DIR pin is connected to GPIO 20 and the STEP pin is connected to GPIO 21. VDD from the driver is connected to the 5V GPIO pin on the Pi.

To control the motor we used two Python scripts, one that would rotate the rotor clockwise 360° and “open” the door, and another one that would trigger 1 second later that would rotate counterclockwise and “close” the door. *Figure 7* shows the code for the open door script. We initialize GPIO pins 20 and 21 for step and direction, we set CCW low such that the rotor turns clockwise. The steps per revolution are hardcoded as 400 given that this is the default for this motor. A for loop then counts up to the step_count and stops. Each cycle toggles the STEP pin high for .001 seconds.

In the second Python script, the only change is that CCW is set high such that the door returns to its initial position. Step_count and the delay remain the same so that the action does not cause the motor to skip steps and create jerky motions.

```
from time import sleep
import RPi.GPIO as GPIO

DIR  = 20   # Direction GPIO Pin
STEP = 21   # Step GPIO Pin
CCW  = 1    # 1 = Counterclockwise Rotation
            # 0 = Clockwise Rotation
SPR  = 400  # 400 Steps per Revolution

GPIO.setmode(GPIO.BCM)
GPIO.setup(DIR, GPIO.OUT)
GPIO.setup(STEP, GPIO.OUT)
GPIO.output(DIR, CCW)

step_count = SPR
delay = 0.001 # keep above 001 #smaller turns faster
for x in range(step_count):
    GPIO.output(STEP, GPIO.HIGH)
    sleep(delay)
    GPIO.output(STEP, GPIO.LOW)
    sleep(delay)

GPIO.cleanup()
```

Figure 7. Open door source code

Potential Additions & Discarded Implementations

- Added: SHA-3 hashing for RFID tags
- Added: Update hashed tag with Ethereum node in addition to local node
- Not implemented: QR codes

Conclusion

Overall this project was a great way to solidify the concepts we learned in class. We were able to explore cryptography, hardware security, authentication, and decentralized computing. Working on a project with the revolutionary technology that is the blockchain, we were able to create a deeper understanding for ourselves of this dense topic. Thanks to the recommendation by Professor Kia to use Project Oaken, we were also able to create local, low-cost connected devices which work with blockchain technology. Putting together the different hardware and software components on this project helped to remind us of some programming concepts we had forgotten as well as old circuitry knowledge we had not employed in some time. This was a fun system to design and implement and we hope that with our archived repo future enthusiasts can pick up and develop the program further.

Libraries and Documentation

Our GitHub repository can be found at https://github.com/BUConnectedWorld/Group3 along with documentation about how to set up a Raspberry Pi to run this program.

Helpful Website Resources

https://howtomechatronics.com/tutorials/arduino/how-to-control-stepper-motor-with-a4988-driver-and-arduino/

https://pgaleone.eu/raspberry/ethereum/archlinux/2017/09/06/ethereum-node-raspberri-pi-3/

http://raspnode.com/diyEthereumGeth.html

https://www.oakeninnovations.com/