Learn Hashing, Binary, and Proof-of-Work with MicroProver (Code Companion #2)

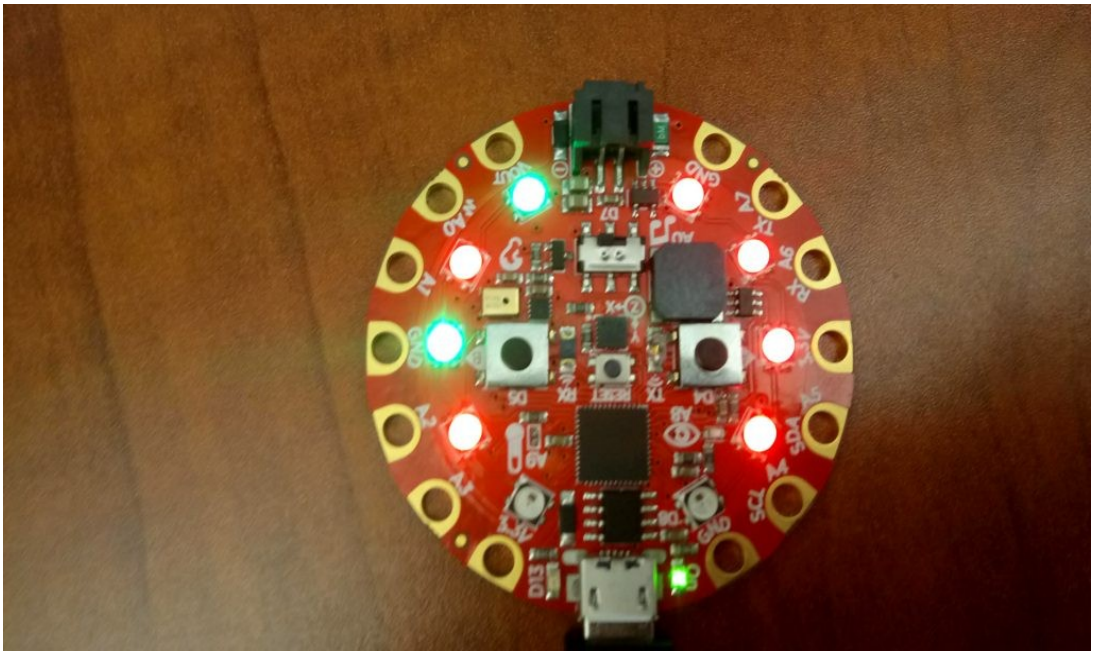
Author	josh
Date	2019-08-28 11:00:08

Note: This article focuses on the development of MicroProver. See my slides for the full BTC2019 talk

Overview

Proof-of-work is a Bitcoin and blockchain topic of vital importance, as it allows transactions to occur without trusting an intermediary. However, understanding this concept also requires some computer science background. One needs to know about hashing algorithms, binary numbers, and a bit of probability to "get" proof-of-work.

I wanted to do a better job of explaining the concept of proof-of-work to individuals without a computer science background - so I came up with the idea of *visualizing* hashing and binary numbers with a cool little microcontroller I received at PyCon 2019. This code companion will dive into the development of MicroProver, and how I turned this project into a session at the 2019 Blockchain Training Conference!



MicroProver (running on the Adafruit Circuit Playground Express) displays a final "block hash"

How MicroProver Works

Toy Hashing Algorithm

The first thing needed to make simulated proof-of-work operate is a *hashing algorithm*. Bitcoin uses the cryptographically secure SHA-256 for its mining operations, among other things. However, these cryptographic algorithms are not readily available on microcontroller platforms such as the Circuit Playground. It took extensive effort to get cryptographic primitives working for my offline address generation project, and required a more powerful line of processor than the CPX's M0.

However, for the purposes of this project, a cryptographically secure hash algorithm is not needed! This project is designed for visualization and learning, and has no security requirements. So in order to create the 8 bit hashes I wanted, I simply used the simplest form of hash function one might use for creating a basic hash table. All of the code in this article can be found in src/core/MicroProver.py

```
    # Settings for "cryptography"
    self.HASH_MOD = 256

...

    # Return a really simple 8 bit hash
    # This is for educational purposes, so we don't need a
    # cryptographically secure hash, we just need one that works
    def hash_8bit(self, data):
        hash8 = data % self.HASH_MOD

        return hash8
```

For an 8 bit hash, we use the 8 bit number space that contains 256 total possible numbers (0-255). This algorithm takes the modulus of the data and 256, giving us a reasonable hash for our purposes.

Binary Representation

The other important concept needed to address proof-of-work is understanding binary numbers. In daily life, most of us are used to base 10, and working in other bases is a rather foreign concept. In order to make this easier to understanding in the context of proof-of-work, I decided to use the built in CPX LED lights, with *red* representing 0 and *green* representing 1.

The code takes the data (the hash output) and converts it from a raw number to an array of binary values, either True/False (used by the LED function) or 1/0 for string representation. The data is turned into binary using a technique called *bitmasking*, where one single bit of a byte is isolated using the & binary operator.

```
    # Convert a byte of data (8 bit hash, etc.) into
    # an array of bits represented by True (1) and False (0)
    # (with default mode bool)
    # Specify optional mode "bit" for 0/1 representation
```

```
def byte_to_bitarr(self, byte, mode="bool"):

    # Define some bitmasks for each spot in the byte
    # We'll create the array using bitmasking
    masks = [ 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 ]
    byte = int(byte)

    bitarr = []
    for i in range(0, 8):
        if mode == "bit":
            masked = byte & masks[i]
            bit = "1" if masked > 0 else "0"
        else:
            bit = bool(byte & masks[i])
        bitarr.append(bit)

    return bitarr
```

The LED display function then takes that array and lights up individual LEDs on the board to represent each bit:

```
# This function displays an LED representation of a byte
# It lights up 8 LEDs on the Playground Express board
# Green represents a 1 bit
# Red represents a 0 bit
def display_byte_led(self, byte):

    bitarr = self.byte_to_bitarr(byte)
    for i in range(0, 8):
        if bitarr[i]:
            color = self.GREEN
        else:
            color = self.RED

        # Load the pixels from 1 - 9 so they
        # fill evenly on each side of the board
        cpx.pixels[i + 1] = color
    cpx.pixels.show()
```

Tying It Together for Proof-of-Work

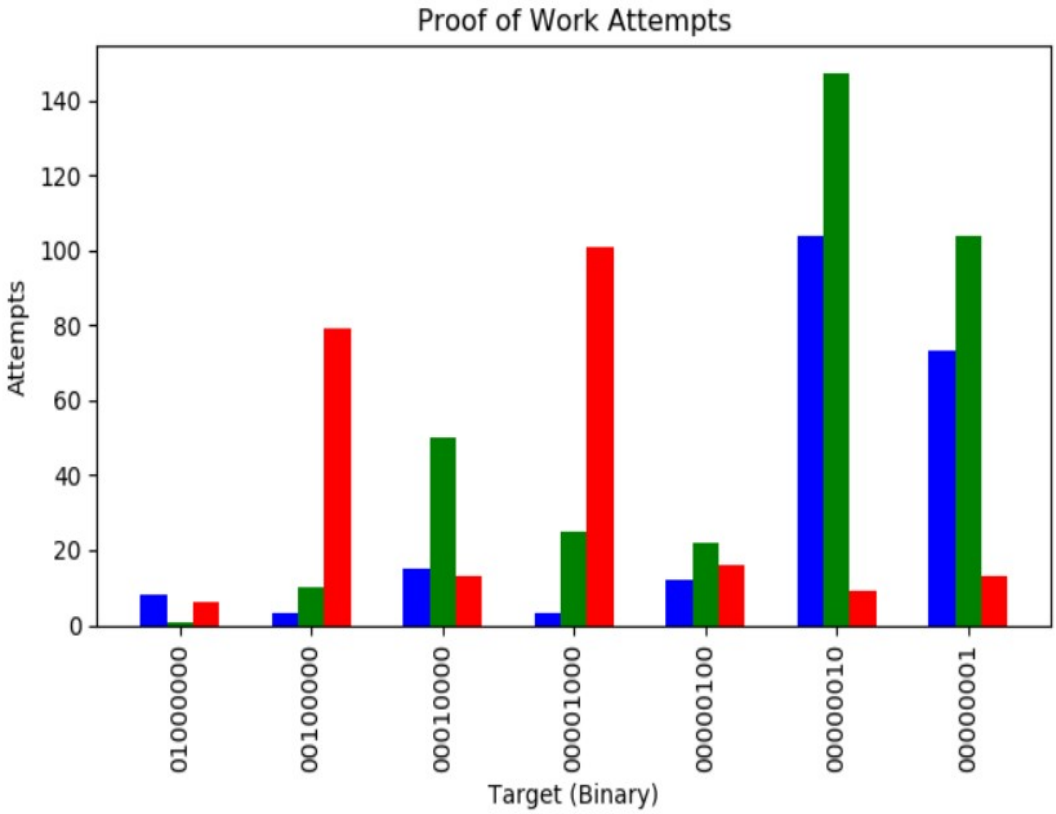
Using these nifty little hash concepts, binary operators, and LEDs, it becomes bit easier to visualize the binary number comparisons needed by proof of work. Proof-of-work requires comparing a *hash output* to a *difficulty target*, and both can be thought of as binary numbers. For this toy visualization, we use small 8 bit numbers.

For example, say we have a difficulty target of 00100000. This value (32 in decimal) has two *leading zeroes* represented as an 8 bit, unsigned integer. Therefore, the final hash output must have at least 3 *leading zeroes* to be less than the target. For example, 00010100 is a valid block hash. Because the probability of finding this "block hash" decreases as the difficulty target decreases numerically, we "proved work" by doing a lot of guesses to get a solution.

In MicroProver, the user gets to program the difficulty level from 1-7, with the level being the number of leading zeroes in the target. This way, it is easy for the user to visualize the algorithm as attempts and the final solution are displayed on the board. They can look for the red "leading zeroes" as the algorithm works, and get a better feel for how difficulty affects the computing power/time needed to prove work.

Extra Visual Help - Data Visualization Script

As an addition to the board itself, I created a data visualization script in src/dataviz/graph_pow.py. This script takes an optional log generated by the CPX simulation and graphs targets/vs. attempts, breaking down how difficulty affects the probability of finding a solution:



Proof of Work difficulty target vs attempts to find a solution. It's probability!

Proof-of-Work, Made Accessible

When I got the Circuit Playground, I knew I had to use it to create an interesting chaintuts project! After getting oriented with programming basic utilities on the board, I decided its features would be great for an educational assistant.

Fortunately, I have had the opportunity to take this project further and teach at the 2019 Blockchain Training Conference in Denver, CO! For this session, I'll be breaking down proof-of-work and using MicroProver for interactive simulations and data visualization.

By using LEDs to visualize hashing, binary numbers, and a bit of probability, we can make understanding this critical blockchain security topic more accessible to those without a computer science background. And the more folks that understand decentralization and trustless software, the more we can drive adoption of these technologies.