CRYPTOCURRENCY AND DECENTRALIZED APPLICATION DEVELOPMENT

REIMPLEMENTING PAPERCUT ON THE ETHEREUM BLOCKCHAIN

Benjamin Brown

Adviser: Michael Linderman

A Thesis
Presented to the Faculty of the Computer Science Department
of Middlebury College
in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Arts

May 2018
ABSTRACT

Bitcoin is a decentralized consensus protocol, based on cryptographic proof instead of trust, that allows nodes in a network to create a currency system. Ethereum is another decentralized consensus protocol based on the blockchain technology of Bitcoin, but the Ethereum network creates a state transition system with arbitrary state. This state transition system allows developers to create decentralized applications in the form of smart contracts. PaperCut is Middlebury’s printing service, which we reimplemented as a decentralized application using Ethereum.
ACKNOWLEDGEMENTS

First, I’d like to thank Professor Linderman for guiding me through the process of writing this thesis. Neither of us knew much about the topic at the beginning of the semester, and he was more than willing to learn with me along the way. I’d also like to thank the rest of the Middlebury Computer Science faculty for their tireless effort to make my and other students’ educations the best they can be. Last, but not least, I’d like to thank my friends and family for their endless support and willingness to let me ramble about cryptocurrency this semester.
# TABLE OF CONTENTS

1 **Introduction**  
   1.1 Motivation .................................................. 1  
   1.2 Assumptions of the Reader ................................. 1  

2 **Bitcoin**  
   2.1 Currency and The Double-Spending Problem .............. 2  
   2.2 Transactions .................................................. 4  
   2.3 Mining and Proof-of-Work .................................. 5  
   2.4 Forks in the Blockchain ................................... 7  
   2.5 Reducing Block Size ......................................... 9  
   2.6 Simplified Payment Verification .......................... 11  
   2.7 Multicoin Transactions .................................... 12  

3 **Ethereum**  
   3.1 Limitations of Bitcoin ..................................... 14  
   3.2 Ethereum Accounts .......................................... 15  
   3.3 Messages and Transactions ................................. 16  
   3.4 Events .......................................................... 20  
   3.5 Developing Decentralized Applications .................. 21  

4 **Recreating PaperCut on the Blockchain**  
   4.1 PaperCut ...................................................... 25  
   4.2 Design Goals and Assumptions ............................. 26  
   4.3 Frameworks ................................................... 27  
   4.4 Application Architecture .................................. 27  
   4.5 Smart Contract API .......................................... 33  
   4.6 Future Work .................................................. 35  

5 **Conclusion** ................................................... 37  

**Bibliography** .................................................. 38
LIST OF TABLES

4.1 Comparison of Centralized and Decentralized PaperCut  . . . . . . . . 28
4.2 State Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.3 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
LIST OF FIGURES

1.1 Price of Bitcoin ................................................. 1

2.1 Currency as a State Transition System ......................... 2
2.2 A Mint’s Transaction History .................................. 3
2.3 The Contents of a Coin ........................................ 4
2.4 A Chain of Blocks ............................................. 5
2.5 Adding a Proof-of-Work to the Chain of Blocks ............... 6
2.6 A Fork in the Blockchain ...................................... 7
2.7 Resolving a Fork ................................................ 8
2.8 Unnecessary Transaction Data .................................. 9
2.9 A Sample Merkle Tree ........................................ 10
2.10 Merkle Trees are Tamper-Evident ............................. 10
2.11 Pruning a Merkle Tree ....................................... 11
2.12 Simplified Payment Verification .............................. 12
2.13 A Multicoin Transaction ..................................... 13

3.1 Ethereum State System ....................................... 14
3.2 Simple Smart Contract ........................................ 18
3.3 Contract Call Example ......................................... 19
3.4 Smart Contract with an Event ................................. 20
3.5 Example JavaScript Callback .................................. 21
3.6 Self-Destructing a Contract .................................. 23

4.1 Printing Interface ............................................... 29
4.2 Buy and Sell Interface ......................................... 30
4.3 Decentralized PaperCut Printing Operation .................. 32
CHAPTER 1
INTRODUCTION

1.1 Motivation

In December of 2017, the price of Bitcoin spiked above $19,500, over 40 times what it was worth a year before. This rise in price generated wild interest from the public and caused many to purchase Bitcoin without actually knowing how cryptocurrency works. Since then, the price dropped to below $7,000, and has since been fluctuating between $7,000 and $11,000 (see Figure 1.1).

The motivation of this thesis is to understand the technical details of cryptocurrency, specifically Bitcoin and Ethereum, and explore how Ethereum can be used to create decentralized applications.

![Price of Bitcoin over the past year](image)

Figure 1.1: Price of Bitcoin

1.2 Assumptions of the Reader

Cryptocurrency relies heavily on cryptography, specifically hash functions, asymmetric key encryption, and digital signatures. So, we assume the reader is familiar with these cryptographic primitives, at least at a high level. Additionally, we explore how Ethereum allows developers to program applications that run in a decentralized manner, so we assume the reader has a moderate amount of programming experience.
CHAPTER 2

BITCOIN

In this chapter, we aim to summarize the most important aspects of blockchain technology by discussing the technical details of Bitcoin. A complete description of Bitcoin is presented in Satoshi Nakamoto’s paper “Bitcoin: A Peer-to-Peer Electronic Cash System,” [4] from which most of the material in this chapter is adapted.

2.1 Currency and The Double-Spending Problem

We define a currency as a system that maintains a mapping from entities to numeric values. Originally, currency systems were based on objects with inherent value (e.g. gold coins), but more recently, we have created currency systems based on the trust of a central authority, such as a bank or mint. For example, a hundred dollar bill, a simple blend of linen and cotton, holds no intrinsic value. But, the value of the hundred dollar bill is backed by the trust we have in the US Government. Similarly, banks and credit card companies act as trusted third parties in the exchange of online currencies in e-commerce. On the other hand, Bitcoin creates an online currency system based on a decentralized network, instead of a trusted third party.

We can model a currency as a state transition system, where the state is the mapping from entities to values and the transitions are monetary transactions. For an example, refer to Figure 2.1. Alice starts with three dollars and Bob starts with one, and then Alice gives one of her coins to Bob, resulting in a new state.

![Figure 2.1: Currency as a State Transition System](image)

For a currency to function properly, it must solve the double-spending problem, which we define as follows: given a unit of currency, how do we ensure that the owner does not spend the same unit twice? In the case of physical currencies, the solution to the double-spending problem is trivial. Assuming that counterfeits cannot be made,
every dollar bill is unique, so two people cannot own the same dollar bill at the same time.

In the case of digital currencies, the solution to the double spending problem is not trivial. Suppose Alice owns one coin (i.e. a unit) of digital currency. Furthermore, suppose this digital currency uses a trusted central authority, known as a mint, to keep track of who owns what coins. We assume that the mint is implemented as a program running on a server. Alice can transfer her coin to Bob by sending a message to the mint with her intent to do so, and Bob would presumably give Alice something else in exchange. Afterward, if Alice were to try to transfer the same coin to Charlie, the mint would recognize the transaction as invalid and reject it. However, Alice could theoretically hack the mint and erase all history of her transaction to Bob. Then, she could transfer the coin to Charlie, and the mint would approve the transaction, thus allowing Alice to double-spend the coin. See Figure 2.2 for a diagram of the two scenarios.

Left: The mint rejects Alice’s second transaction.
Right: Alice hacks the mint, allowing her second transaction to be verified.

Figure 2.2: A Mint’s Transaction History

So, to prevent double-spending of a centralized digital currency, the mint must secure its transaction history. Once the mint has a trustworthy record of the transaction history, it is easy for it to verify whether or not a new transaction is valid. But, securing the transaction history is a difficult problem, so many central authorities (e.g. credit card companies) charge transaction fees to cover their cost of operation.

The goal of Bitcoin, on the other hand, is to create a digital currency system with no central authority. As we will see in the following sections, Bitcoin solves the double-spending problem by creating a decentralized consensus protocol that allows a network to agree upon one order of transaction history using cryptographic proof instead of trust.
2.2 Transactions

The Bitcoin network is composed of a collection of nodes, each identified by a public key. Nodes are able to communicate directly with each other. The purpose of the network is to have all nodes agree upon the monetary value each node holds, and to allow nodes to transfer value to each other. To represent value, the network uses a collection of unique electronic coins, which are defined as a sequence of signed transactions. Each coin is called a Satoshi, and one Bitcoin is worth a hundred million Satoshis.

To transfer a coin, a node initiates a new transaction with the following data:

1. The public key of the recipient
2. The hash of the coin’s previous transaction
3. A digital signature of the above data using the node’s private key

Since the sender includes their digital signature, anyone with the sender’s public key can verify that the sender intended to make the transaction. Furthermore, by including the hash of the coin’s previous transaction, the sender uniquely identifies which coin they are transferring.

As an example, observe Figure 2.3: Alice starts with one Satoshi. Then, she transfers it to Bob (transaction 1), who transfers it to Charlie (transaction 2), who transfers it to David (transaction 3).

A coin consists of a sequence of signed transactions

Figure 2.3: The Contents of a Coin
Once a node has created a new transaction in this manner, it broadcasts the transaction data to all other nodes in the network. Due to network connectivity issues, nodes may receive transactions in different orders. In the next section, we discuss how the nodes in the network agree upon one order.

### 2.3 Mining and Proof-of-Work

In the Bitcoin network, each node stores a complete copy of the system state locally, and updates that state when a transaction is received. To keep track of the order of transactions, each node collects the transactions it receives into discrete blocks. When a block is full, the node hashes all the data therein and includes this hash as part of the next block, forming a chain (see Figure 2.4). This chain of blocks is where the name blockchain comes from.

Each node stores transactions in sequence of blocks, chained together by their hashes

Figure 2.4: A Chain of Blocks

Therefore, each block depends on all the blocks that come before it. Consider Figure 2.4, if the data within Tx0 of Block \( i \) is modified, then the hash of Block \( i \) would change, meaning the hashes of Blocks \( i + 1, i + 2, \ldots \) would all change.

However, this system does not yet provide a way for the nodes to come to consensus. So, each block must also include a nonce such that the block’s hash, when considered as a bit string, has at least \( n \) leading zeros. The task of finding such a nonce takes exponential time as a function of \( n \), and is known as finding a proof-of-work. On the other hand, it takes constant time to verify that a nonce produces a satisfactory hash.
Figure 2.5: Adding a Proof-of-Work to the Chain of Blocks

To keep the network in sync, once a node’s current block is full (the block data may not exceed 1 MB), the node begins to search for a satisfactory nonce for the block. Once a node finds such a nonce, it broadcasts the block data to all other nodes in the network. The other nodes can then easily verify the correctness of the block. To check the validity of a block, a node checks that the block’s hash has at least $n$ leading zeros, that all transactions are correctly signed and not double-spent, and that the previous hash included in the block matches the hash of the previous block stored by the node. Once a node verifies a block, it accepts all transactions of the block as truth, appends the block to its local chain, and starts filling a new block with incoming transactions.

The task of finding a proof-of-work is called mining, and since it is a computationally difficult task, all nodes will finish at different times. Therefore, only one node will be first. So even though nodes may receive slightly different sequences of transactions, the network accepts the sequence from the node which finds a nonce first. Therefore, a Bitcoin transaction is only valid once it has been mined into a block. Furthermore, the parameter $n$ is dynamically adjusted by the network so that each block takes roughly 10 minutes to mine.

To incentivize nodes to verify each other’s transactions and find proofs-of-work, “the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block.” [4, p. 4] Furthermore, since finding an acceptable nonce is probabilistic, all nodes have a chance of being the first to find a proof-of-work for a block, and the more computing power a node has, the more likely it is to be first. In summary, the steps of running the distributed network are as follows [4, p. 3]:

1. New transactions are broadcast to all nodes.
2. Each node collects new transactions into a block.
3. Each node works on finding a difficult proof-of-work for its block.
4. When a node finds a proof-of-work, it broadcasts the block to all nodes, and rewards itself with a new coin.

5. Nodes accept the block only if all transactions in it are valid and not already spent.

6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

### 2.4 Forks in the Blockchain

It is possible for two different valid blocks to be published to the network at roughly the same time, and one block may be accepted by some nodes while the other block is accepted by other nodes. This creates a fork in the blockchain, meaning there are two different versions of transaction history present in the network. For example, in the Figure 2.6, both Block 2a and Block 2b are valid, and can be extended.

In order to keep the network synchronized, nodes always take the longest blockchain as the true version of history. If there is a tie (as depicted in Figure 2.6), then one chain will eventually be extended significantly before the other, thereby breaking the tie. At that point, all nodes working on the shorter chain will switch to the longer chain. For example, in the Figure 2.6, if a proof-of-work is found for a Block 4a long before one is
The tie is broken, all nodes begin working on finding the proof-of-work for Block 5a, and Chain b ‘dies.’

Figure 2.7: Resolving a Fork

found for Block 4b, then all nodes that had been working on finding the proof-of-work for Block 4b will switch to finding the proof-of-work for Block 5a.

With this scheme, as long as a majority of the network’s computing power is controlled by honest nodes, a malicious party cannot modify the order of transactions. However, if an attacker controls a majority of computing power, she may perform a 51% attack and rewrite history in her favor. For example, let’s say Mallory controls a majority of computing power and wants to cheat Vernon, an internet vendor, by ‘buying’ goods for free. Consider Figure 2.6; suppose Mallory transfers one coin to Vernon in Tx21a (highlighted in blue), and waits for Block 2a to be mined. Vernon sees that Mallory has transferred him a coin in Block 2a, so he sends her the goods she is purchasing. Once Mallory receives the goods, however, she immediately creates Block 2b, where instead of transferring the coin to Vernon, she transfers the coin to herself (in Tx21b, highlighted in red). She then directs all of her computing power to extending Chain b, instead of Chain a. If she has more computing power than the rest of the network combined, then she will be able to grow Chain b faster, eventually overtaking and killing Chain a, and nullifying her transfer of a coin to Vernon.

If Mallory waits some time before extending Chain b, then she will have to mine more blocks before overtaking Chain a. Therefore, the deeper in the blockchain a transaction is, the more secure it is. So, to be safe, a vendor may wait until a few additional
blocks have been mined before vending goods.

### 2.5 Reducing Block Size

In the previous sections, we implied that for a node to function in the network, it would have to store the entirety of the blockchain, that is, all the transactions that have ever occurred. However, after a coin’s most recent transaction is old enough, the network does not actually need to know the details of its previous transactions. For example, consider Figure 2.8 (a slightly modified version of Figure 2.3). If the network is currently working on mining Block 100, then the details of Transactions 1 and 2 are no longer important. The only data that the network still needs to store is that David is the recipient of the coin’s latest transaction.

![Diagram of transaction data](image)

*If the network is mining Block 100, then the details of Transactions 1 and 2 are no longer important*

Figure 2.8: Unnecessary Transaction Data

But, a node cannot simply delete this transaction data, because it would no longer be able to produce the correct hash for the containing block, and therefore all subsequent blocks as well. So, the transactions in a block are hashed using a Merkle tree, which is a binary tree where the leaves are data and each internal node is created by hashing its children. Specifically, to calculate a node’s data, we concatenate the hashes of its children, then hash the result. See Figure 2.9 for an example.

![Diagram of Merkle tree](image)
Non-leaf nodes are calculated by hashing their children, and do not need to be stored.

Figure 2.9: A Sample Merkle Tree

Therefore, the root node depends on all the children, making the tree tamper-evident. In other words, given a list of transactions and the root hash of the Merkle tree, we can quickly verify that the root hash indeed came from those transactions. Furthermore, to verify a branch (or the transaction at the leaf of the branch), it suffices to only provide a subsection of the tree. For example, in Figure 2.10, we only need the nodes highlighted in blue to verify the transaction from Alice to Bob, and if that transaction data is modified, it would be evident somewhere up the tree.

Left: if suffices to present only a small number of nodes in a Merkle tree to give proof of the validity of a branch.
Right: any attempt to change any part of the Merkle tree will eventually lead to an inconsistency somewhere up the chain [2]

Figure 2.10: Merkle Trees are Tamper-Evident
When a node first creates a block, it hashes the transactions in a tree and stores the root hash, along with the previous block hash and nonce, in the \textit{block header}. Then, the block stores the header data and all the transactions. But, when the block no longer needs to maintain the details of a particular transaction, it prunes the block’s tree, deleting transaction data and storing internal nodes of the Merkle tree instead (see Figure 2.11). Eventually, it will no longer be necessary to store any transactions in a block. At this point, nodes need only store the root hash and other header data for that block.

For example, suppose Owner 0 starts with one coin and transfers the coin to Owner 1 (Tx0), who transfers it to Owner 2 (Tx1), who transfers it to Owner 3 (Tx2), who finally transfers it to owner 4 (Tx3). And, suppose that these four transactions are hashed into a block (as in the lefthand diagram of Figure 2.11). Then, after a significant amount of time passes without this coin being involved in any transactions, the network no longer needs to know anything about the owners before Owner 4. So nodes can prune Transactions 0 through 2 from the tree (righthand diagram of Figure 2.11).

\section*{2.6 Simplified Payment Verification}

Since all the blocks in the Bitcoin blockchain take up an ever-growing amount of storage (167 GB as of May 13, 2018, and growing roughly 1 GB a week \cite{1}), we’d like a user to be able to verify a transaction without having to store the entirety of the blockchain.
In order to do this, a node may act as a light node and run a protocol known as Simplified Payment Verification (SPV) [2]. Such a node stores only the block headers of the blockchain, or, when needed, queries full nodes to obtain the block headers. Then, to verify a transaction Tx, the node will query full nodes for the branch of the Merkle tree with Tx at the leaf. From this branch, the node can see the details of Tx and check that Tx was accepted by the network.

A light node can verify Tx3 by downloading only Hash01, Hash2, and Tx3, along with the block headers. [4]

![Figure 2.12: Simplified Payment Verification](image)

**2.7 Multicoin Transactions**

In Section 2.2, we implied that one transaction transfers a single Satoshi, and while this is a good model for understanding the details of the Bitcoin protocol, it is not be practical. If it were the case, then if Alice wanted to send Bob $1 worth of Bitcoin, she would have to initiate approximately 10,000 separate transactions (assuming 1 Bitcoin is worth roughly $10,000). Therefore, a transaction must be able to transfer multiple coins.

This is relatively easy to implement, as we can simply provide multiple inputs and outputs to a transaction, where each input and output has a coin value. Typically, we will either have one large input which will be split up, or many smaller inputs which will be combined; and, we will typically have two outputs, one to the recipient, and one returning change to the sender. Then, when a miner verifies a transaction, it must check that the sum of the inputs is at least the sum of the outputs.
Furthermore, if there is any difference between the inputs and outputs, the miner takes this difference as a transaction fee. The purpose is to allow senders to provide an additional incentive for miners, and “once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free” [4, p. 4].

Figure 2.13: A Multicoind Transaction
CHAPTER 3
ETHEREUM

First proposed in 2013 by Vitalik Buterin, Ethereum is a generalization of the Bitcoin protocol that allows for arbitrary system state and user-defined transitions. The goal of Ethereum is to allow developers to create decentralized applications that run on a shared blockchain. In this chapter, we introduce the Ethereum protocol as a conceptual extension of Bitcoin. Unless otherwise noted, the material in this chapter is adapted from the Ethereum white paper [2].

3.1 Limitations of Bitcoin

In Chapter 2, we presented Bitcoin as a decentralized state transition system. In that system, the state is limited to a mapping from public keys to numeric values (representing Bitcoin amounts). The goal of Ethereum is to create a similar blockchain-based network where the system state may contain arbitrary data acted on by arbitrary transitions. To conceptualize this idea, consider Figure 3.1, which depicts a simplified representation of Ethereum state, contrasted with the Bitcoin state system presented in Figure 2.1.

![Figure 3.1: Ethereum State System](image-url)
Instead of simply mapping Alice to 3 and Bob to 1, we can map people to any hex string, which can then be interpreted in an application-specific manner, such as an ASCII string (e.g., “hello world”), an array, a struct, etc. Like Bitcoin, a user can effect a state transition by broadcasting a transaction. But, in Ethereum, transactions may contain arbitrary data that allow for arbitrary state transitions.

Recall that each node in the Bitcoin network independently stores the system state, and when a new block is received, a node must verify each transaction in the block in order, applying the state transition specified by the transaction. In the case of Bitcoin, these transitions are limited to incrementing and decrementing users’ balances. But, in Ethereum, users are able to encode arbitrary transitions into transaction data using a Turing-complete scripting language, and a node verifying a transaction will run this code to update its state. In the follow sections, we describe how Ethereum represents its system state, as well as how transactions are processed by the network.

### 3.2 Ethereum Accounts

The state of the Ethereum system is made up of *accounts*, each identified by a 20-byte address. Like Bitcoin, there is a currency associated with the system, known as *ether*, and each account can hold an amount of ether. But, unlike Bitcoin, accounts have additional fields. All accounts consist of the following:

1. The **nonce**, a counter used to make sure each transaction can only be processed once
2. The account’s current **ether balance**
3. The account’s **contract code** (optional)
4. The account’s **storage** (empty by default)

If an account does not have contract code, then it is called an *externally owned account*, i.e., it is controlled by an external identity (e.g., a person) with a private key. Such an account functions in the same way as a node on the Bitcoin network. On the other hand, if an account does have contract code, then it is called a *smart contract*, and it is controlled exclusively by that contract code. Such an account does not have a private key.

The code within a smart contract is stored as Ethereum Virtual Machine (EVM) bytecode, which is a stack-based assembly language similar to Java Virtual Machine
code. When a contract receives a transaction, its code is activated and has the ability to read from and write to the account’s storage, send and receive ether, and call other smart contract functions. When a node is verifying a transaction sent to a smart contract, it runs the contract’s bytecode to update its state. In the next section, we discuss the details of smart contract capabilities.

As described in the white paper, it is important to “note that ‘contracts’ in Ethereum should not be seen as something that should be ‘fulfilled’ or ‘complied with’; rather, they are more like ‘autonomous agents’ that live inside of the Ethereum execution environment, always executing a specific piece of code when ‘poked’ by a message or transaction, and having direct control over their own ether balance and their own key-value store to keep track of persistent variables.” [2]

### 3.3 Messages and Transactions

Like Bitcoin, transactions are the mechanism through which the state of the system is changed. An externally owned account initiates a transaction in the same way a node in the Bitcoin network does. However, if the transaction is sent to a contract, then the miner, and every node verifying the transaction, must run the contract code to update the state of the system. Since the contract’s code may take some time to run, Ethereum compensates the miner based on the complexity of the computation. Every transaction contains the following elements:

1. The recipient of the message
2. A signature identifying the sender
3. The amount of ether to transfer from the sender to the recipient (possibly none)
4. An optional data field
5. A starting gas value, representing the maximum number of computational steps the transaction execution is allowed to perform
6. A gas price value, representing the fee the sender pays per computational step

The gas price is the amount of ether per computational step the miner receives, and the starting gas is how much ether the sender is willing to pay for the computation. Therefore, the maximum possible transaction fee is calculated as starting gas times gas price. If the miner runs out of gas before the computation is complete, all changes made
by the contract code are reverted, but the gas is still forfeit to the miner as a computational fee. In this way, a user may run any computation on the Ethereum network, so long as they are willing to pay for it. Note that gas price and starting gas are typically specified in \(\text{wei}\), where \(10^{18}\) wei equals 1 ether.

Code running in the Ethereum execution environment has access to the following resources:

1. The **stack**, a last-in-first-out linear data structure
2. **Memory**, an infinitely expandable byte array
3. The contract’s long-term **storage**, a key-value map that, unlike stack and memory, persists between different transactions

A contract is organized into two main sections: a collection of **state variables**, held in the contract’s long-term storage, and a collection of functions containing EVM code. When a user sends a transaction to a contract, they must specify in the data field which contract function to call, as well as provide arguments to that function. Then, the miner runs the function code just like any other assembly language, executing one opcode at a time and incrementing the program counter, stopping when a **STOP** or **RETURN** opcode is reached. Additionally, the miner will halt execution if it runs out of gas.

Messages are similar to transactions, except that they are initiated by the **CALL** opcode of smart contracts. In other words, when a smart contract wishes to call a function of another contract, it does so using a message. Therefore, messages need not be signed, serialized, and broadcast to the network because the miner has the bytecode for all smart contracts in the network. Additionally, messages can be used to send ether to another account from the contract. A message contains the following fields:

1. The sender of the message (implicit)
2. The recipient of the message
3. The amount of ether to transfer alongside the message
4. An optional data field (containing actual parameters for a function call)
5. A start gas value

The start gas value of a message is calculated as the gas remaining after running the computation up to the **CALL** opcode that triggered the message. There is no limit on how many nested function calls a smart contract can make, so the transaction gas is needed to prevent infinite recursion.
Smart contracts are created by special transactions which are sent to the null address (i.e. address 0x0), and the contract address is calculated from the sender’s address and nonce. The data of the transaction becomes the contract bytecode, and the ether in the transaction becomes the contracts starting balance.

As an example, suppose we want to write a smart contract to store a key-value mapping from strings to strings. Then, the contract would have a state variable to store the mapping, and `read` and `write` functions to manipulate the mapping. In Figure 3.2, we give an implementation of this contract written in Solidity, which is a JavaScript-like language that compiles to EVM code. [7]

```
contract DataStorage {

    // state variable
    mapping (string => string) data;

    function write(string key, string value) public {
        data[key] = value;
    }

    function read(string key) public view returns(string) {
        return data[key];
    }

}
```

Figure 3.2: Simple Smart Contract

Now, we can use transactions sent to this contract to change the state of the system. Consider, for example, Figure 3.3: The DataStorage contract is located at account 0xdef..., and Alice, who owns the account at 0xabc..., would like to write the mapping “hello” → “world” to DataStorage’s data variable. So, she initiates a transaction to DataStorage where the transaction data is a binary encoding of the function call `write('hello', 'world')`. Note that in State, Alice’s ether balance has decreased slightly due to the transaction fee she paid to the miner.
On the other hand, if Alice simply wants to read from the key-value store, she would not need to initiate a transaction because she is not altering the state of the blockchain. Therefore, Alice could simply query the network for the most recent state of the blockchain, then on her own machine, run the `read(string key)` function. Note that in Solidity, such a non-modifying function is marked with the “view” keyword. In this example, the `read` function is simple, but it is possible to have view functions that are more complex. However, those functions would still only be run by an interested user on their local machine, not by every node verifying a transaction.

In summary, the transaction verification algorithm is as follows:

1. Check if the transaction is well-formed (i.e. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender’s account. If not, return an error.
2. Calculate the transaction fee as starting gas times gas price, and determine the sending address from the signature. Subtract the fee from the sender’s account balance and increment the sender’s nonce. If there is not enough balance to spend, return an error.
3. Initialize as variable `gas` as the amount of starting gas, and subtract a certain quantity of gas per byte in the transaction.
4. Transfer the transaction value from the sender’s account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract’s code, subtracting gas price from `gas` for every computational step, either until completion or until `gas = 0`. 

![Figure 3.3: Contract Call Example](image-url)
5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner’s account.

6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner. [2]

### 3.4 Events

In addition to the capabilities discussed in the previous section, smart contracts can also log arbitrary data to the blockchain. In Solidity, this functionality is exposed as *Events*. The contract defines an event using the `event` keyword and specifies an event name and formal parameters. Then, the event can be called like a function throughout the contract code. Whenever an event is called, the contract logs the arguments to the blockchain, making the data publicly available. Furthermore, up to three parameters can be marked with the `indexed` keyword, which allows users to search and filter events based on these parameters.

Users can listen for events in a frontend application by polling the blockchain and running a callback whenever an event is logged. For example, suppose we want to write a contract that allows users to broadcast textual messages to each other. To keep the messages in order, we need a `messageCounter` state variable, and to broadcast the messages, we need a `Message` event. Figure 3.4 gives a Solidity smart contract that implements this functionality.

```solidity
class BroadcastMessage {
  uint256 messageNumber = 0;

  event Message(uint256 indexed messageNumber, string message);

  function broadcast(string message) public {
    Message(messageNumber, message);
    messageNumber += 1;
  }
}
```

Figure 3.4: Smart Contract with an Event

In the next chapter, we will discuss how a webpage can attach a JavaScript callback to listen for a particular event. But, for now, suppose that a group of users have a web-
page where the JavaScript function in Figure 3.5 is called whenever the BroadcastMessage contract emits a `Message` event, and the arguments of the `Message` event are passed as parameters to the callback.

```javascript
function onMessageReceived(count, message) {
    console.log('Message number:', count);
    console.log('Message:', message);
}
```

Figure 3.5: Example JavaScript Callback

Then, a user can broadcast a message by initiating a transaction to the contract, and all other users can listen for the message. It is important to note that unlike running a function locally to query the state of a contract, logging an event costs gas because it changes the state of the blockchain.

### 3.5 Developing Decentralized Applications

As we have seen Sections 3.2 through 3.4, smart contracts have the ability to store data, run functions, and communicate with external users. Furthermore, smart contracts have access to their ether holdings, transaction and message data, and the header data of the block in which the contract call is mined. Specifically, a contract has access to the block nonce, which provides a source of randomness in contract execution, which may be important for certain applications (e.g. gambling).

Developers can abstract many of the blockchain details, and simply use a smart contract as a server for a decentralized application. For example, suppose we want to create a decentralized voting application. We would write a smart contract with a `vote` function, which would record the vote of the sender (and make sure each sender only votes once). We would also write a user interface that allows the user to log in with their Ethereum private key and broadcast a transaction to our smart contract. The advantage of using a smart contract in this scenario is that it ensures the votes are counted fairly because they are publicly broadcast. Indeed, every node in the network verifies every block, and part of those blocks are the voting transactions, so all blocks will agree on the result of the votes.

Additionally, since smart contracts also have the ability to hold money in the form of ether, they naturally have many financial applications. For example, suppose three people wish to combine their funds and hold them in escrow. They wish to allow one
person to withdraw a maximum of 0.5% of funds per day, two people together to withdraw a maximum of 10% of funds per day, and all three together to withdraw as much as they would like. Without a smart contract, these people would have to entrust the money with a third party, potentially paying a steep transaction fee or taking on the risk of allowing the third party to invest their money at its discretion. Instead, they could create a smart contract to hold the funds and encode these rules in the contract code. The transaction fees would be minimal, as they are based on computational cost, not a percentage of funds held, and the money would be safe because the contract would only let them withdraw it as specified. Furthermore, they would not have to take the risk of allowing another party to handle their money.

Once a contract is deployed, its code cannot be changed. This immutability protects the integrity of a smart contract because a user knows exactly what a contract will do before they interact with it, and this interaction will never change. For example, consider the voting application from earlier in this section. If we could change the contract code halfway through a vote, then we could make it so that some votes counted double and sway the outcome of the vote, thereby destroying the integrity of the contract.

Contract immutability, however, puts restrictions on developers. If a developer deploys a contract, but later finds a bug in their code, they have no way to push an update or patch to fix the bug. Some bugs are small enough that the developer can work around them in the rest of their application, but some bugs are too big for such a solution. For this reason, there is an EVM opcode that causes a contract to self-destruct and send all of its funds to a specified address. Once a contract has self-destructed, it can no longer receive transactions. A developer can include a contract function that kills the contract and sends all funds back to the himself. For example, in the Solidity contract of Figure 3.6, the contract constructor specifies the transaction sender as the owner, and will only self destruct when the owner calls the `selfDestruct` function. Note that `tx` and `msg` are global variables that provide access to transaction and message information, respectively, and `selfdestruct` is a built-in function that kills the contract and sends its funds to the given address.
However, allowing contracts to self-destruct presents another set of challenges. For example, suppose ContractA acts as a library for ContractB by providing utility functions, and ContractA self destructs. Then ContractB may become unusable, and there may be no way to recover the ether that it holds. In fact, this is exactly what happened in November of 2017 to the Ethereum smart contract development company, Parity. Parity had a multi-signature wallet contract (similar to the multi-party escrow example described before) which relied on a library contract. Allegedly, a user accidentally made himself the owner of the library contract, then caused it to self destruct. The wallet contract, which held 513,774.16 ether belonging to 587 wallets, then became unusable, and the ether was inaccessible [8]. At the time, one ether was worth around $450, meaning over 230 million dollars was lost.

Another challenge of decentralized app development is that, by design, all information stored on the blockchain is public. Even if a contract does not provide a function to easily access a certain state variable, the entire state of the contract is available. Therefore, a user could acquire the state of contract’s account, and deduce the value of the variable directly from the state. In other words, there are no secrets on the blockchain, and if a developer wishes to store private information in a smart contract, then they must encrypt that information before sending it in a transaction. Previously, we described how a smart contracts can act as a server for an application, but this restriction limits some smart contract capabilities. For example, many centralized application servers store (the hash of) user passwords in a database, but this is not feasible with a smart contract.

Lastly, both Bitcoin and Ethereum create extra risk for users because they have to
keep track of their private keys. If a user loses their private key, then they have no way to access the funds that private key controlled.

Because of all these challenges, it is often easier to develop a centralized application with a traditional server. However, as we’ve seen in a few examples, some applications can benefit from using a smart contract. For such applications, the cost of increased development complexity is worth the pay-off of having a trustless, decentralized server.
CHAPTER 4
RECREATING PAPERCUT ON THE BLOCKCHAIN

In this chapter, we describe our implementation of a prototype decentralized application. PaperCut is Middlebury’s printing service, and we reimplemented it as a decentralized application on the Ethereum blockchain. It is important to keep in mind that this is purely a proof-of-concept, and, as we’ll see, there are many reasons why PaperCut should stay a centralized service. However, this example illustrates many of the challenges of working with smart contracts. The entire codebase, with instructions on how to run the project in development, is available at https://github.com/BenWBrown/PaperCutBlockchain.

4.1 PaperCut

PaperCut is the webservice that Middlebury students use to print documents. Each semester, each student receives $30 of PaperCut ‘money,’ which they can spend exclusively on printing. From a user’s perspective, the printing process is:

1. Login to PaperCut website with Middlebury credentials
2. Upload a document to be printed
3. Walk to printer
4. Login to printer with Middlebury credentials and release documents

The primary purpose of PaperCut is to create accountability for student printing. If students were able to print an unlimited amount of documents (including expensive documents such as posters), then there would be no limit to the amount of money the school could spend on printing. Instead, the school allocates a portion of their budget every year towards printer supplies and ensures they do not go over budget by making students use PaperCut. Furthermore, the reason PaperCut makes the user login to the printer before printing is to prevent users from accidentally sending documents to the incorrect printer.

In addition to $30 a semester, students receive $12 each Winter Term, and students receive double the allocation during their senior year. Since letter-sized grayscale documents cost approximately $0.05 per page to print, many students have more than enough PaperCut money, and some even graduate with over $300 in their PaperCut account. If a student does run out of PaperCut money, they can buy more from the college, but there does not currently exist a way for students to exchange PaperCut money with each other.
4.2 Design Goals and Assumptions

In our reimplementation of PaperCut, we keep many of the design assumptions from the current PaperCut system. Namely, we assume that the college wants to allocate some money to every student each semester, that printers are connected to the internet, and that users must be physically present at a printer before printing. However, since we use an Ethereum smart contract, we assume that users are identified by Ethereum accounts instead of their Middlebury credentials.

Furthermore, we assume that if we were to deploy the application, it would be deployed to the global Ethereum blockchain. It would be possible to create a Middlebury-only Ethereum blockchain by forming a separate network of machines on the Middlebury campus running the Ethereum protocol. However, students would probably not want to run their personal computers to mine blocks because the Ethereum with which they would be rewarded would only hold value in the Middlebury network. Therefore, the college would control a large majority of computing power on the network. At which point, running a decentralized application holds no advantage over running a college-controlled centralized application.

Our design goals, in order of importance, are listed below.

1. Use a smart contract to store and manipulate the PaperCut balance of all students. PaperCut currently stores transaction history for each user on a centralized server. Therefore, the user must trust PaperCut to correctly and securely maintain their transaction history. We instead use a smart contract to make transaction history fully auditable and verifiable, and allow data to be securely stored on the Ethereum blockchain.

2. Mimic the PaperCut user experience as closely as possible. The current PaperCut user experience is relatively simple, and the system effectively limits the college’s printing costs. Since we have similar design assumptions, we aim to create a similar user experience.

3. Ensure user privacy and security. As described in Section 3.5, there are no secrets on the blockchain. Thus, we aim to keep users’ files private from each other by not sending file data in blockchain transactions. Furthermore, we must ensure that a user can only print their own files, and no one else’s.

4. Allow students to buy and sell PaperCut money from each other. This is a feature that many students wish existed in the current PaperCut, and a smart contract would allow students to trade PaperCut money in a trustless manner.
5. Minimize transaction cost. Using a smart contract implies paying transaction fees, so we aim to minimize these costs. However, since we are only implementing a prototype and not deploying to the main Ethereum network, this goal has lower priority.

4.3 Frameworks

Below is a brief description of the main frameworks used in our implementation.

1. React.js is a JavaScript library for creating responsive web interfaces. We used React to create our frontend. [6]

2. Express.js is a framework for developing web servers in JavaScript. We used Express to create a server that mimicked a printer in the PaperCut network. [3]

3. Truffle is a framework for developing smart contracts on the Ethereum blockchain. It includes a built-in Solidity compiler, as well as scripts that automatically deploy smart contracts to the blockchain. Additionally, Truffle includes a tool called Ganache, which allows the user to run a personal, development blockchain at a port on their machine. Therefore, when a developer is still testing a smart contract, they can deploy it to their development blockchain for free, instead of to the real Ethereum blockchain. Truffle also provides a JavaScript API to interact with smart contracts, which we used for the printer Express server. [10]

4. Web3.js is a JavaScript API for interacting with smart contracts. It provides functions to initiate transactions, query blockchain state, and listen for smart contract events. We used Web3 in the frontend to communicate with the smart contract. [11]

5. Npm is a JavaScript package manager that we used to install and maintain external JavaScript dependencies. [5]

4.4 Application Architecture

As mentioned in the previous section, we have three layers in our application:

1. A user interface, implemented as a React app. The interface has a basic login page, where the user logs in using their Ethereum private key, and a home page
with two tabs. The first tab is where the user can initiate a print job, and the second tab is where students can buy and sell PaperCut money in an open market. Screenshots of the two tabs are shown in Figures 4.1 and 4.2

2. A printer implemented as an Express server. Since the application is a proof-of-concept, we don’t use an actual printer, nor do we deal with actual files. Instead, our ‘files’ are simply strings, and we ‘print’ them by logging them to the command line of the Express server. Because the printer is set up as a server, we do not currently have a user interface to interact with it, so all interactions are done through the React app. Furthermore, since the printer communicates with the smart contract, it needs access to an Ethereum account.

3. A smart contract that stores and controls user balances. The smart contract is implemented in Solidity and stores a mapping from addresses to integers user PaperCut balances. Additionally, when a user prints a document, the contract acts as an intermediary between the user and the printer, only deducting a user’s balance after a file has been printed.

The user interaction is similar to the current PaperCut, as depicted in Table 4.1.

Table 4.1: Comparison of Centralized and Decentralized PaperCut

<table>
<thead>
<tr>
<th>Step</th>
<th>Centralized PaperCut</th>
<th>Decentralized PaperCut</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Login to PaperCut website with Middlebury credentials</td>
<td>Login to PaperCut application with Ethereum private key</td>
</tr>
<tr>
<td>2</td>
<td>Upload a document to be printed</td>
<td>Upload a document to be printed</td>
</tr>
<tr>
<td>3</td>
<td>N/A</td>
<td>Receive a file-specific one-time code for the document</td>
</tr>
<tr>
<td>4</td>
<td>Walk to printer</td>
<td>Walk to printer</td>
</tr>
<tr>
<td>5</td>
<td>Login to printer with Middlebury credentials</td>
<td>Enter one-time code into printer</td>
</tr>
<tr>
<td>6</td>
<td>Release document to be printed</td>
<td>Document is printed automatically</td>
</tr>
</tbody>
</table>

The infrastructure to support this interaction is more involved. Observe Figure 4.3. The three layers of the application are across the top, and the steps needed to print a single file are in chronological order from top to bottom. These steps are as follows:
Papercut Client

Address: 0x627306090abaB3A6e1400e9345bc60c78a8BEf57
Ethereum balance: 99.78870302
PaperCut Balance: 384.90
Withheld Balance: 0.10

Print  Buy and Sell

I_L0v3_2_c0d3.txt  Initiate Print

HelloWorld.txt
Status: Ready to print with code: 00
Cost: 0.05

homework25.java
Status: Ready to print with code: 5d
Cost: 0.05

*** The below button mimics physically entering the OTC into the printer.

One-Time Print Code  Print!
Logout

Figure 4.1: Printing Interface
Papercut Client

Address: 0x627306090abaB3A6e1400e9345bC60c78a8B3Ef57
Ethereum balance: 100.78810028
Papercut Balance: 0.50
Withheld Balance: 4.50

Print  Buy and Sell

Papercut Money: 4.5
Price: 0.01 ETH
Buy

Papercut Money: 380
Price: 2 ETH
Buy

PaperCut Money to Sell

<table>
<thead>
<tr>
<th>Price (in ETH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.50</td>
</tr>
</tbody>
</table>

Price (in ETH)

| 0.01            |

Make Offer  Refresh

Figure 4.2: Buy and Sell Interface
1. The user sends the file and their public key to the printer server via an HTTPS request.

2. The printer calculates the price to print the document, based on page length, page size, color, etc.

3. The printer responds to the user’s request with the price, thereby acknowledging that it has received the file.

4. The printer and the user both send the hash of the file to the smart contract, and the printer also sends the price.

5. Assuming no errors have occurred, the hash from the user and the printer match, and the smart contract then withholds the printing price from the user’s PaperCut balance, preventing them from attempting to overspend. If the user wishes to cancel the printing job, then they would get their withheld balance back.

6. The smart contract broadcasts approval of the print job via a Solidity event.

7. The user application simply displays approval, and waits for a one-time code to be generated.

8. The printer generates a cryptographically-secure random one-time code (OTC) and encrypts it with the user’s public key. This one-time code will be used to print the document.

9. The printer sends the encrypted one-time code to the smart contract.

10. The smart contract passes the encrypted one-time code to the user via a Solidity event.

11. The user decrypts the one-time code with their private key.

12. The user physically walks to the printer, and enters the one-time code into the printer’s interface.

13. The printer sends the file hash to the smart contract to ensure that the user has not attempted to cancel the print-job in the meantime.

14. If the user has not canceled the print-job, the smart contract withdraws money from their withheld balance.

15. The smart contract sends approval back to the printer via a Solidity event.

16. The printer prints the document.
Event
Physical
HTTPS
Transaction
Communication Type

Figure 4.3: Decentralized PaperCut Printing Operation
The reason the user sends only the hash of the file to contract is twofold. First, if the user sent the entire file, then since there are no secrets on the blockchain, anyone could see what files the user is printing. Second, even if the user encrypts the file, using the entire file would still require large transactions and storing large amounts of data on the blockchain, both of which are expensive. Instead, the hash gives the user, smart contract, and printer a way to uniquely and securely refer to a file without having to pass around the entirety of the file data. Therefore, the file hash is passed as a parameter in all the transactions and events in Figure 4.3.

The one-time code functions similarly to logging in to the printer with Middlebury credentials, but it does not require the printer to store a database of user passwords. To make using a one-time code secure, the code must be randomly generated and encrypted with the user’s public key before it is broadcast on the blockchain.

Allowing users to buy and sell PaperCut money is a natural extension of the smart contract. In the following steps, we have two users, referred to as the Seller and the Buyer.

1. The Seller broadcasts a transaction to the smart contract, making an offer to sell \( X \) amount of PaperCut money for \( Y \) ether.
2. The smart contract ensures that the Seller has at least \( X \) amount of PaperCut money, and withholds that amount from the Seller.
3. The Buyer broadcasts a transaction to the smart contract, taking the offer to buy \( X \) amount of PaperCut money for \( Y \) ether, and transferring \( Y \) ether to the smart contract.
4. The smart contract deducts \( X \) PaperCut money from the Seller, increments the Buyer’s PaperCut balance by the same amount, and sends \( Y \) ether to Seller.

### 4.5 Smart Contract API

In this section, we document all the state variables and public functions of the PaperCut smart contract in Tables 4.2 and 4.3.
<table>
<thead>
<tr>
<th>Type and Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapping (address =&gt; uint256) balances</td>
<td>Stores the current available balance of all users.</td>
</tr>
<tr>
<td>mapping (address =&gt; uint256) withheldMoney</td>
<td>Stores the withheld balance of all users.</td>
</tr>
<tr>
<td>mapping (address =&gt; mapping(uint256 =&gt; bool)) userUnapprovedFiles</td>
<td>For each user, stores a hash-set of file hashes the user has requested to print, but that have not been approved yet.</td>
</tr>
<tr>
<td>mapping (address =&gt; mapping(uint256 =&gt; bool)) userApprovedFiles</td>
<td>For each user, stores a hash-set of file hashes that have been approved to print.</td>
</tr>
<tr>
<td>mapping (uint256 =&gt; uint256) filecosts</td>
<td>Maps a file hash to the cost of printing the file.</td>
</tr>
<tr>
<td>uint256[4][] offers</td>
<td>Stores a (dynamic-sized) array of (length 4) arrays. Each length 4 array represents an offer, containing a unique offer number, the address of the seller, the PaperCut amount offered, and the ether amount requested.</td>
</tr>
<tr>
<td>uint256 nextOffer</td>
<td>The number used to create the unique offer number of each offer. It is incremented every time an offer is created.</td>
</tr>
<tr>
<td>address owner</td>
<td>The address that created the contract and initialized in the contract constructor.</td>
</tr>
</tbody>
</table>

\[1\] It would be much more natural to store offer data in a struct, but Web3 does not yet support logging a struct in an event. So, we use a fixed-size array as a workaround.
### Table 4.3: Functions

<table>
<thead>
<tr>
<th>Function Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Papercut()</td>
<td>Constructor that initializes the owner of the contract. For development purposes, it also initializes some contracts with a starting amount of PaperCut balance.</td>
</tr>
<tr>
<td>userAddValue()</td>
<td>Adds PaperCut balance to the sender of the transaction based on how much ether is sent.</td>
</tr>
<tr>
<td>adminCashOut()</td>
<td>Sends the contract funds to the owner of the contract.</td>
</tr>
<tr>
<td>userPrintRequest(uint256 filehash)</td>
<td>User requests a file to be printed.</td>
</tr>
<tr>
<td>userCancelPrint(uint256 filehash)</td>
<td>Cancels a print-job.</td>
</tr>
<tr>
<td>printerPrintRequest(address user, uint256 filehash, uint256 cost)</td>
<td>Printer sends a file hash that the user has requested to be printed.</td>
</tr>
<tr>
<td>printerAnnouceCode(address user, uint256 filehash, uint256 iv, uint256 pk1, uint256 pk2, uint256 pk3, uint256 pk4, uint256 ciphertext, uint256 mac)</td>
<td>Printer announces the encrypted one-time code. The one-time code is encrypted using elliptic-curve public-key cryptography, so the printer must send the initial value (iv), public key (pk1-pk4), ciphertext, and message authentication code (mac) for the user to decrypt the data.</td>
</tr>
<tr>
<td>printerAnnouceFilePrinted(address user, uint256 filehash)</td>
<td>Printer asks permission to print a file, and the smart contract deducts the user’s funds.</td>
</tr>
<tr>
<td>getBalance(address addr)</td>
<td>Gets the PaperCut balance for a specified account.</td>
</tr>
<tr>
<td>getOffers()</td>
<td>Returns the array of current offers.</td>
</tr>
<tr>
<td>makeOffer(uint256 pcAmount, uint256 ethAmount)</td>
<td>Allows a user to make an offer to sell PaperCut money.</td>
</tr>
</tbody>
</table>

#### 4.6 Future Work

Our implementation provides a proof-of-concept for a decentralized version of PaperCut; however, there are still many problems to solve before the application is ready for deployment. Most obviously, we would need access to a printer API that allowed us to print from actual printers. Then, we would have to extend the user application to upload
real files and connect the Express server to the printer API. Furthermore, since there are many printers spread across Middlebury’s campus, we would have to redesign the infrastructure to deal with multiple printer servers, instead of sending data to just one.

Another aspect of the application that would require improvement is the login process. Currently, the user must login by copying and pasting their Ethereum private key. A better solution would be to encrypt the private key with the user’s password and store the encrypted key on disk. Then, the user could login by entering their password and having the application decrypt their private key for them.

For development purposes, we used one-time codes that were one byte long encoded as hex strings, as pictured in Figure 4.1. However, in order for a one-time code to be secure, we would have to increase its length. Since each one-time code would only exist for a short amount of time (less than 24 hours), we could still use relatively short codes. For example, if we used codes that were five bytes long, there would be over one trillion possible codes, and each code would be 10 digits long. Furthermore, to decrease the length of the code, we could represent it as a base-32 string.

In our current implementation, we have no way for an admin to manually increase users’ PaperCut balance (e.g. $30 a semester). Additionally, the smart contract functions that are meant to be called by the printer do not check that the transaction is actually coming from a printer. So, we would have to expand the capabilities of the admin, allowing them to deposit money into user accounts and maintain a list of trusted printer addresses.

Lastly, we would have to address the issue of transaction costs. In our current implementation, both the user and printer have to pay fees (in ether) for every transaction they make. So, if a user requests to print a file, but then later cancels that request, the user and printer both lose some ether to transaction fees. One solution would be to have the smart contract reimburse the user and printer the cost of these transactions in ether, and charge the user’s PaperCut balance. With this solution, the user would still have to pay for cancelling a print-job, but at least they would not have to simultaneously maintain an ether balance and a PaperCut balance.
CHAPTER 5
CONCLUSION

In this thesis, we’ve seen that Bitcoin is a decentralized consensus protocol that allows a peer-to-peer network to agree upon an ordering of transactions. This network is then able to implement a currency system without a central authority. Part of the power of Bitcoin, and cryptocurrencies in general, is that they can be used by anyone with an internet connection, regardless of political boundaries. We believe that since Bitcoin is controlled by a majority of computing power, instead of some central authority, it is a natural embodiment of democratic ideals. On the other hand, we must acknowledge that Bitcoin is often used for illicit and immoral activities, such as black-market trading and computer ransomware. Furthermore, the Bitcoin network now consumes more energy per day than the entire country of Ireland [9]. So, it is unclear whether the societal impact of Bitcoin has been more positive or negative.

Additionally, we’ve seen that Ethereum is also a decentralized consensus protocol that uses the same blockchain technology as Bitcoin. But, the Ethereum network may store arbitrary state, as opposed to just a currency system. This arbitrary state allows developers to create decentralized applications in the form of smart contracts. Ethereum was first implemented in the fall of 2015, and it did not gain much public visibility until its price spiked in late 2017. Compare this situation to 1992, three years after the invention of the World Wide Web [12]. At the time, the world had no conception of what sort of applications would be built on the World Wide Web 26 years later. Smart contract development is in a similar stage of infancy now as the World Wide Web 26 years ago. Though it is not a universally held opinion, it is our belief that in another 26 years, people will be developing decentralized applications the likes of which developers today have never imagined.
BIBLIOGRAPHY


