Implementing zk-SNARKs in Spartan-Gold

Aaron Smith
CS 168 Blockchain and Cryptocurrencies
San Jose State University
December 14, 2020

1. Introduction

A major problem with bitcoin is the lack of anonymity. Even if a wallet address is never used in more than one transaction, there are still ways to trace the movement of bitcoin back to individual users. Various solutions have been proposed to fix this, but the one that this project focuses on is the use of zk-SNARKs. A zk-SNARK is a Zero-Knowledge Succinct Non-Interactive Argument of Knowledge. This is a type of zero-knowledge proof that can be used to hide transaction details (e.g. addresses and amounts) on the blockchain while still providing a way to validate those transactions. The original inspiration for this project was found in Zcash [1]. Zcash is a bitcoin fork originally released in 2016, and detailed in the Zerocash paper [2]. Zcash is the most prominent usage example of zk-SNARKs. It maintains anonymity by hiding the addresses and amounts involved in transactions, yet transactions are still validated and published to the blockchain.

This project extends the class's cryptocurrency Spartan-Gold [3], written in Javascript, in order to provide anonymity. The open source projects snarkjs [4] and Circom [5] provide the zk-SNARK functionality that is used to extend Spartan-Gold. Transactions are made anonymous by eliminating the need to tie user addresses (or any user identifiable information for that matter) to transactions or units of currency.

2. Brief Explanation of zk-SNARKs

The way zk-SNARKs work in relation to this project is actually quite complicated and difficult to understand. The first step is to create a function that will verify a transaction. This function needs to be created in such a way that it can be encoded in a mathematical representation in the form of polynomials. In order to prove that a transaction is valid, a prover must prove that they know and can solve such a polynomial

without revealing the solved polynomial itself (hence zero-knowledge). Homomorphic encryption and elliptic curve pairings are used to evaluate the polynomial "blindly", i.e. neither the prover nor the verifier know which point on the polynomial is being evaluated (which prevents cheating). This process is a one time job (non-interactive) so everyone on the network can easily verify (succinct) without extra interaction with the prover. In order to be non-interactive, encrypted public parameters (a common reference string that includes a prover and a verifier key) are created and used by everyone in the proving/verification process. The original parameters (sort of like a seed) used to create the reference string must be destroyed or else someone can use them to cheat the system (the creation of the reference string is in itself a complicated process). The final result is that a prover can prove a transaction is valid with extremely high probability while not revealing the contents of the transaction [6].

3. Snarkjs

The snarkjs open source library is a Javascript and Pure Web Assembly implementation of zkSNARKs. When paired with Circom, it can be used to create zkSNARK proofs and verify them. The setup of snarkjs involves going through what is called a Powers of Tau Ceremony. This is just a way to create the necessary public parameters involved in zkSNARKs. In the real world, like in the case of Zcash, these public parameters need to be completely random, generated by multiple participants, and the original "seed" needs to be discarded, all for security reasons. In this project, the public parameters were generated on a single machine and didn't go through any extra steps to make things more secure. Thus, this implementation isn't necessarily cryptographically secure, but works as a proof of concept.

The project repository [7] already contains the generated files from the snarkjs setup process, along with the compiled Circom verification function. These files (circuit_final.zkey, verification_key.json, verifier.wasm) are all that are necessary to use snarkjs in this project. The following code snippet demonstrates how snarkjs is used along with these files to create a proof and to verify the proof.

```
// Generate proof
let input = {cm1: correctHash, cm2: badHash, sn: sn, r: r, index: 0};
let proof = await snarkjs.groth16.fullProve(input, "verifier.wasm", "circuit_final.zkey");

// Verify proof
let vKey = JSON.parse(fs.readFileSync("verification_key.json"));
let res = await snarkjs.groth16.verify(vKey, proof.publicSignals, proof.proof);
if (res) {
  console.log("Verification OK");
} else {
  console.log("Invalid proof");
}
```

4. Circom

Circom is a language that is used to write arithmetic circuits to be used by snarkjs. Visually, circom looks similar to a normal programming language, but it has some key differences that make things difficult. To make a transaction verification function, the function must be thought of in terms of a circuit. This circuit has inputs and outputs, and the transaction is verified using constraints. All constraints must be written as a quadratic expression, and no constraints can depend on values that are unknown at compile time. These two points make creating arithmetic circuits rather difficult and confusing, which is why the one used in this project is very simple. Observe the circom transaction verification circuit (note the symbols <== and === are used to check a constraint):

```
template Verify() {
    var HASH_LENGTH = 256;
    signal input cm1[HASH_LENGTH];
    signal input cm2[HASH_LENGTH];
    signal input sn[HASH_LENGTH];
    signal private input r[HASH_LENGTH];
    signal private input index;
    signal switcher[256];

    // First compute sha256(r||sn)
    component hash = Sha256(HASH_LENGTH * 2);
    for (var i = 0; i < HASH_LENGTH; i++) {
        hash.in[i] <== r[i];
        hash.in[i + HASH_LENGTH] <== sn[i];
    }

    // Now compare the hash to one of the cmlist inputs, specified by index
    for (var i = 0; i < HASH_LENGTH; i++) {
        switcher[i] <== (cm2[i] * index);
        hash.out[i] === (cm1[i] * (1 - index)) + switcher[i];
    }
}
```

5. Implementation Overview

The currency of this project is made up of individual coins which are called ZksnarkCoins. A ZksnarkCoin is the smallest unit of this currency, and the only unit for that matter. It is made up of three 256-bit values: cm, r, and sn. The values r and sn are

two randomly generated numbers, and the value cm is equal to the hash of r concatenated with sn, i.e. cm = sha256(r||sn). The owner of a coin knows all three values; a coin can be spent if all three values are known (though technically r and sn are the only values that need to be known, since cm can be calculated from them). When a coin is first minted, its hash value cm is added to a ledger that is stored on the blockchain. This ledger is a list of all valid coins, it can be thought of as similar to a list of UTXOs.

A user can spend their coin by posting a transaction that contains the value sn and a zkSNARK proof that essentially proves: "I know r, such that the value cm = sha256(r||sn) appears on the ledger of valid coins". Note that this transaction never reveals the values cm nor r, but it does sn. Since cm is never revealed, no one else can look at the ledger and see which coin is being spent. Since r is never revealed, no one else can compute cm, since it would require reversing a sha256 hash. The reason sn is revealed is so that it too can be added to the blockchain in a different ledger containing the sn of already spent coins; this ledger is used to prevent double spending.

When a transaction is validated, i.e. a spender creates a valid proof for a valid coin, a new coin is minted and the old is spent. Thus, all transactions (except coinbase transactions) involve turning an old coin into a new coin. So if a user u1 wants to transfer a coin to user u2, then u1 first sends the coin's values (cm, r, sn) to u2. The user u2 then takes the coin and mints it into a new one. By creating a new coin, u2 validates that the coin sent from u1 is valid. It also prevents u1 from trying to spend the coin since it will have been invalidated in the minting of a new coin. The user u2 is now the only user that knows the values for the new valid coin.

The actual Circom verification circuit (shown in section 4), took a shortcut from the above stated proof. To create a Circom circuit that checks if the cm of the coin being spent appears on the ledger is quite difficult. So instead of taking the whole ledger as input to the circuit, the circuit takes in two cm values on the ledger. One of the cm values is the correct one for the coin being spent, the other is a random cm from the ledger. By doing this, the verification circuit is much simpler, while still preventing others from seeing exactly which coin is being spent. Others will still have a 50% chance (or more) at guessing which coin is being spent. In the real world, this might be viewed as a serious shortcoming, but this tradeoff was considered adequate for the sake of this project.

The Circom verification circuit (shown in section 4) takes in five parameters: cm1, cm2, sn, r, index. The inputs cm1, cm2, and sn, are known as public signals. This means that anyone can see the values of these inputs, but they cannot see the other values: r and index. These values are hidden, but the zkSNARK is used to prove that the spender knows these hidden values without revealing those values. The reason cm1 and cm2 are public is so that a verifier can verify that those cm values appear on the ledger as a valid coin. The value sn is public to prevent double spending. The value index is used in the circuit to determine which cm (cm1 or cm2) is to be evaluated. That is, if index = 0, then the circuit simply verifies that cm1 = sha256(r||sn). If index = 1, then the circuit verifies cm2 = sha256(r||sn). In either case, all the circuit does is verify that the prover knows the hidden parameter r that makes the value cm = sha256(r||sn) appear on the ledger. It is important that exactly which cm is being spent is hidden, that way others can't attempt to track a coin's existence between users.

An astute reader might think of a few problems with this scheme. This implementation is intentionally simplistic, but it operates as a proof of concept. The actual Zerocash protocol extends this scheme quite a bit to offer more features. In Zerocash, for example, the ledger of valid cms is represented as a merkle tree. The Zerocash verification circuit therefore proves that the coin cm exists in the merkle tree, which enables the protocol to scale efficiently and to have more security. In this project a list is used instead, since implementing a merkle tree in Circom is no simple task (Circomlib has a merkle tree package but seems buggy and is very difficult to use [8]). The verification circuit instead doesn't even read from the whole ledger, but is rather given two cm values to choose from. These are not strong guarantees. Another thing that Zerocash does is it gives coins amount values, i.e. there are different units of currency. In this implementation, there is only one unit and one value, which is one ZksnarkCoin. In Zerocash, multiple inputs and outputs can be involved in transactions, and the transfer of currency between users is done differently, among other differences [2]. The decisions made for this project were intentional to prevent the implementation from becoming overly complicated.

6. Overview of Classes

a. ZksnarkCoin: Used to represent a single unit of currency.
   i. Coins are made up of three 256 bit values: cm, r, sn. A ZksnarkCoin is simply a placeholder for all three values. The value cm is the hash of r concatenated with sn, i.e. cm = sha256(r||sn). The values r and sn are two random values, where r is kept private by the coin owner and sn is a public parameter used in a zk-snark proof.

b. ZksnarkTransaction: Represents a basic transaction.
   i. Transactions are much different then in Spartan-Gold. A transaction simply turns an old coin into a new coin. In the process, the old coin is invalidated (i.e. cashed out, preventing it from being spent again) and a new coin is minted in its place.
   ii. A ZksnarkTransaction contains the values involved in a transaction, namely a snarkjs proof and the cm of the new coin minted.
   iii. A ZksnarkTransaction object isn't used in coinbase transactions. Instead, in coinbase transactions, coins are directly added to the blockchain by miners. This is necessary because, unlike in the ZksnarkTransaction class, there is no old coin from which the new coin will be minted.

c. ZksnarkBlock: Represents a block on the blockchain.
   i. A ZksnarkBlock keeps track of valid coins, already spent coins, transactions, and coinbase transactions.
   ii. Provides two significant methods: addTransaction() and verifyTransaction(). The method verifyTransaction() is used to validate a snarkjs proof. This is an asynchronous method. It's separated from addTransaction() because the verification process takes some time and by the time the transaction is verified it's possible the blockchain has already moved on to a future block. Once a transaction is verified, a miner then adds it to the current block using addTransaction(). This method simply adds the transaction, the new coin cm, and the old coin sn, to the blockchain.

d. ZksnarkBlockchain: Holds configuration information and utility methods.

i. Because of the very different blocks and transactions used in this project, some functionality had to be changed here on top of Spartan-Gold. Most importantly, new deserialization functions had to be defined because when blocks, transactions, and coins are sent over the network they become serialized.

e. ZksnarkClient: A client/user of the currency.
   i. A number of methods had to be overridden from Spartan-Gold to obtain correct client functionality. The method postTransaction() simply takes a valid coin and sends it to another client. The method receiveTransaction() is the inverse of postTransaction(), it's called when another client sends a coin to the current client. This method takes the coin and creates a snarkjs proof to mint a new coin from the old one. It then broadcasts the proof and the new coin's cm (i.e. a ZksnarkTransaction), which the miner then takes and adds it to the blockchain after it's validated.
f. ZksnarkMiner: A ZksnarkClient that is also a miner for the blockchain.
   i. ZksnarkMiner extends from ZksnarkClient, which means many methods from Spartan-Gold's Miner class had to be copied over to this class. The most notable difference is addTransaction(), which upon receiving a new transaction from a client, proceeds to verify the transaction and then adds it to the current block.
g. ZksnarkUtils: Contains utility functions.
   i. Utility functions were created to be used by different classes in this project. The function bufferToBitArray() converts an object of type Buffer to an array of bits, which is provided as input to the snarkjs proof. The function parsePublicSignals() is used to read a snarkjs proof's public signals and return the public parameters as Buffer's. The other functions: random256BitNumber(), createNewCoin(), and listContains(), are self explanatory.

7. Conclusion

The original goal of this project was to use zkSNARKs to extend Spartan-Gold to provide anonymity by hiding implementation details. This goal was successfully obtained through the use of the Snarkjs and Circom libraries. In this project, senders and receivers (i.e. their addresses) of the currency are no longer tied to the currency itself nor to the blockchain. The blockchain does not keep track of balances, as was the case in Spartan-Gold. Instead, each client keeps track of their own balances, since each client alone can know how many valid coins they hold. The blockchain doesn't record any client addresses either, nor does it specify which coins are spent and which unspent. The only time a client's address is used in this project is when a client sends a coin to another client via their address (the coin is not broadcasted). The only people that know the two parties involved in a transaction are the two parties themselves. There is no identifiable information published to the blockchain.

As mentioned in the original project proposal, a secondary and optional goal was to implement a system similar to Zcash's two address types (z-addresses and t-addresses). In Zcash, z-addresses are private, and t-addresses are transparent. Currency can be sent between any combination of the two addresses, but only transactions between two z-addresses are 100% anonymous. This project only implements something similar to z-addresses. Furthering this project to include non-anonymous transactions would've been possible but also not trivial. This would've involved creating new types of transactions, and extending the client, miner, and block classes to take these transactions into consideration. Furthermore, this feature may have involved redefining the current transaction design and it's accompanying Circom circuit. While this might not sound too difficult, debugging this project does take a considerable toll, and implementing this was considered too costly in time to produce.

Therefore, the main goal of the proposed project was realized while the optional goal was not. This project was very time consuming and challenging, but it was also very eye opening and educational. It was an interesting and rewarding project that explored the anonymity possibilities of cryptocurrencies.

References

[1] Overview of Zcash. https://z.cash/technology/

[2] The original Zerocash paper that Zcash is based on.
http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf

[3] The Spartan-Gold repository. https://github.com/taustin/spartan-gold

[4] Javascript zk-SNARK implementation repository. https://github.com/iden3/snarkjs

[5] Circom (circuit building) repository. https://github.com/iden3/circom

[6] Explanation of how zk-SNARKs work.
http://petkus.info/papers/WhyAndHowZkSnarkWorks.pdf

[7] The repository containing source code for this project.
https://github.com/aaronsjsu/Cryptocurrency-zkSNARK-Project

[8] Circomlib, a repository containing useful Circom packages (such as sha256).
https://github.com/iden3/circomlib/