CS291D Final Report: a Basic Zerocash Implementation

Gwyneth Allwright, Karl Wang, Dewei Zeng

December 12, 2020

Abstract

In this project, we attempt a basic implementation of Zerocash [1] in Python. Zerocash is a ledger-based digital currency that makes use of zero-knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs) to provide stronger privacy guarantees than currencies such as Bitcoin [2] and Zerocoin [3]. This functionality is provided through a decentralized anonymous payment (DAP) scheme that hides a transaction's origin, destination and amount. We follow [1] to implement the following core functions: `Setup`, `CreateAddress`, `Receive`, `Mint`, `VerifyTransaction`, `Pour`, `KeyGen`, `Prove` and `Verify`, which form the foundations of Zerocash.

Contents

1 Introduction

Data on blockchains such as Bitcoin is public, including the sender, receiver and the amount of money transferred in a payment. While Bitcoin users often utilize different identities to disguise their transactions, it is possible to gain access to both the structure of the transaction graph and the values and dates of transactions. Zerocoin, a cryptographic extension to Bitcoin, aims to introduce better privacy guarantees without requiring new trusted parties, but it still cannot hide the location that money is sent to, as well as the amount of money that is transferred [3]. In addition, it lacks some features of fully-fledged cryptocurrencies, such as payments of exact values.

In order to solve these problems with Bitcoin and Zerocoin, a new digital currency known as Zerocash was devised. Zerocash makes the sender, receiver and amount of money transferred in a payment anonymous, while also improving on the efficiency of Zerocoin [1]. These outcomes are achieved with the help of zk-SNARKs, which are efficient variants of zero-knowledge proofs of knowledge. Zero-knowledge proofs allow the prover of a certain statement to demonstrate that the statement in question is true without revealing additional information about the statement that could result in a compromise of privacy.

One of the primary objectives of this project is to explore zk-SNARKs and their potential applications in the world of blockchains and cryptocurrencies. To achieve this, we use existing zk-SNARK tooling to implement a minimal version of Zerocash in Python with the purpose of gaining a better understanding of Zerocash's theoretical underpinnings. This would be a first step towards demonstrating that zk-SNARKs are a feasible method of enhancing the privacy and performance of transactions on a simple blockchain. Next steps would include benchmarking and comparisons to a blockchain with similar functionality that does not make use of zk-SNARKs.

2 Problem Definition

We wish to understand how to incorporate zk-SNARKs into a basic blockchain in order to improve the blockchain's privacy guarantees. The setup of our scheme must not require any trust beyond a one-time trusted setup of public parameters. The implementation needs to support the minting, merging and splitting of coins without exposing the identities of the users who perform the transactions and the amounts of the currency involved.

3 Solution

The above objectives can be achieved through the combination of zk-SNARKs and a decentralized anonymous payment (DAP) scheme. As part of the Zerocash DAP scheme, we implement the following core functions: `Setup`, `CreateAddress`, `Receive`, `Mint`, `VerifyTransaction` and `Pour` [1]. For the zk-SNARK, we require the additional functions (`KeyGen`, `Prove`, `Verify`) [1]. In the sections that follow, we provide an overview of the DAP scheme, zk-SNARK and their core functions.

3.1 DAP Scheme

3.1.1 Basecoin

The Zerocash system is applied on top of a ledger-based currency (e.g. Bitcoin). This ledger-based currency is referred to as the *basecoin*. All basecoin transactions are recorded in an append-only ledger, which can be accessed by all Zerocash users at all times.

In addition to the basecoin transactions, Zerocash includes two new kinds of transactions — minting and pouring — which will be described later. Mint and pour transactions are also recorded in the basecoin ledger.

3.1.2 Public Parameters

In addition to the ledger, users have access to a set of public parameters. These are part of the one-time trusted setup that takes place before other functions are allowed to execute.

3.1.3 Address Key Pairs

Users may generate as many public and private address key pairs as they desire. The public address keys are published with the purpose of allowing users to make payments among themselves. The secret keys are used for receiving payments.

3.1.4 Coins

Coins are data structures that encapsulate the following information:

- A coin commitment, which is a string that we append to the ledger once the coin is minted.
- A coin value (between 0 and some parameter $v_{\max}$) that specifies the coin denomination in basecoin units.
- A coin serial number, which is a string that uniquely identifies the coin and is used to prevent double-spending.
- A coin address — the public address key of the user who owns the coin.

3.1.5 Coin-Related Data Structures

The Zerocash protocol requires us to maintain the following coin-related information:

- A Merkle tree over coin commitments.
- A list of coin commitments that appear in mint and pour transactions.
- A list of all coin serial numbers that appear in pour transactions.

For efficiency reasons, it is useful to store the latter two lists (which could also be obtained from the ledger) separately.

3.1.6 New Transaction 1: Mint

Mint transactions are used to create coins. At its most basic, a mint transaction can be described as a tuple (cm, v), where cm is the commitment of the minted coin and v is its value. Whenever a coin is minted, this tuple is placed on the ledger.

3.1.7 New Transaction 2: Pour

Pour transactions record the pouring of two input coins into two new output coins (thereby spending the two initial coins). At its most basic, a pour transaction can be described as a tuple (rt, sn_1^{old}, sn_2^{old}, cm_1^{new}, cm_2^{new}, v_{pub}, info), where rt is the root of the Merkle tree over coin commitments, the sn are the serial numbers of the old coins, the cm are the commitments of the new coins, v_{pub} is a coin value and info is an arbitrary string. Pour transactions may also include implementation-specific information.

3.2 zk-SNARK

The zk-SNARK construction consists of a tuple of polynomial-time functions (`KeyGen`, `Prove`, `Verify`). In what follows below, we give a high-level overview of these functions and their properties.

3.2.1 Arithmetic Circuits

For a given field $\mathbb{F}$, an arithmetic circuit C takes as input n field elements $\in \mathbb{F}$ and returns m field elements $\in \mathbb{F}$. We can therefore think of C as a map $\mathbb{F}^n \longrightarrow \mathbb{F}^m$.

In the Zerocash construction, we decompose the circuit input that lives in $\mathbb{F}^n$ into a main input and auxiliary input, where the latter is known as the *witness*. If the dimensions of these two subinputs are u and v respectively, then we can write C: $\mathbb{F}^u \times \mathbb{F}^v \longrightarrow \mathbb{F}^m$.

3.2.2 Circuit Satisfiability

zk-SNARKs can be described in terms of arithmetic circuit satisfiability. The key relationship involved is the following:

$$\text{For a given } X \in \mathbb{F}^u, \ \exists A \in \mathbb{F}^v \text{ such that } C(X, A) = 0^m. \tag{1}$$

The set of all X that satisfy Equation 1 form the set $\mathbb{L}_C$. The statement that a prover would want to demonstrate is that for a given X, we have $X \in \mathbb{L}_C$.

3.2.3 Important zk-SNARK Functions

1. `KeyGen`:

   The function `KeyGen` is used to sample a proving key and a verification key for the zk-SNARK. These keys are both public parameters. Their purpose is to help prove that a certain X is a member of the set $\mathbb{L}_C$.

   `KeyGen` takes as input the security parameter and zk-SNARK circuit C, and returns a key pair.

2. `Prove`:

   The function `Prove` takes as input a proving key, as well as a pair (X, A) — where X represents a main input for the circuit C and A the witness. It returns a proof Π for the statement that $X \in \mathbb{L}_C$.

3. `Verify`:

   The function `Verify` takes as input a verification key, the circuit's main input X and a proof Π. It outputs 1 if there is sufficient evidence that $x \in \mathbb{L}_C$, and 0 otherwise.

3.2.4 zk-SNARK Properties

- *Completeness.* Intuitively, this property means that an honest prover can convince the verifier that $X \in \mathbb{L}_C$. More mathematically, it means that with probability

$$P(\lambda) = 1 - \text{negl}(\lambda), \tag{2}$$

  where λ is the security parameter, the output of `Verify` will be 1 after going invoking `KeyGen` and `Prove` to correctly generate a proof for a circuit input $X \in \mathbb{L}_C$.

- *Succinctness.* This property has two components. First, it means that a proof Π that was generated from `Prove` has $\mathcal{O}(1)$ bits for a given security parameter. Second (again for a fixed security parameter), `Verify` has time complexity $\mathcal{O}(X)$.

- *Proof of knowledge* (intuitive idea). If a proof is verified to be correct, then the prover "knows" (is able to extract) a witness that corresponds to the instance, with certain guarantees around the time complexity of the extraction.

- *Zero knowledge.* The proof does not leak information about the witness.

3.3 Key Zerocash Functions

The DAP scheme described above is implemented by means of a tuple of polynomial-time algorithms (`Setup`, `CreateAddress`, `Mint`, `Pour`, `VerifyTransaction`, `Receive`). In this section, we describe the arguments, outputs and interrelation of these functions.

3.3.1 `Setup`

The purpose of `Setup` is to perform the one-time trusted setup of public parameters. It takes as input a security parameter and produces the following list of public parameters as output:

- $(\text{pk}_{\text{POUR}}, \text{vk}_{\text{POUR}})$: a proving and verification key pair for the zk-SNARK. These are sampled from `KeyGen`.
- pp_{enc}: parameters for the encryption scheme.
- pp_{sig}: parameters for the digital signature scheme.

All three of the above are functions of the provided security parameter.

3.3.2 `CreateAddress`

The purpose of `CreateAddress` is to generate public-private address key pairs for users. It takes as input the public parameters generated by `Setup` and produces a key pair as output.

3.3.3 `Mint`

The purpose of a call to `Mint` is the creation of a coin. It takes as input the public parameters generated by `Setup`, the value of the coin to be minted and the public address key of the coin's owner. It returns a coin data structure for the minted coin, along with the associated mint transaction. The mint transaction is a tuple containing (at minimum) a coin commitment and value.

3.3.4 `Pour`

`Pour` is easily one of the most complicated functions in the Zerocash system. It is used to "pour" two old coins into two new ones, such that the sum of the two old coins equals the sum of the two new ones. The latter sum could potentially an additional value v_{pub} that is publicly spent in the transaction. In this case, the balance equation would be

$$v_1^{\text{old}} + v_2^{\text{old}} = v_1^{\text{new}} + v_2^{\text{new}} + v_{\text{pub}}. \tag{3}$$

The `Pour` operation could have multiple purposes, including switching coin denominations, making public payments and transferring coin ownership.

`Pour` takes the following as input:

- The public parameters generated by `Setup`.
- The root of the Merkle tree over coin commitments.
- The coin data structures for two old coins.
- The secret address keys for the owners of the two old coins.
- The authentications paths for each of the old coin commitments to the root of the Merkle tree.
- Two new coin values.
- The public address keys for the owners of the two new coins.
- A coin value v_{pub}. This is amount that will be *publicly* spent in the pour transaction — for example, to pay a transaction fee or to purchase coins.
- An information string.

One of the primary tasks performed in `Pour` is to prove or disprove the following:

> For the provided Merkle tree root, serial numbers of the old coins and coin commitments for the two new coins, there are two old coins, two news coins and a secret address key such that the following statements hold:
>
> - All four of the coins have the correct commitments.
> - For both of the old coins, the provided secret address key matches the secret address key that is generated from the public address key that forms part of the old coin's data structure.
> - The serial numbers of the old coins are correct.
> - The two commitments of the old coins feature in the provided Merkle tree. This is to ensure that the old coins have been previously minted.
> - The balance equation for the old and new coin values, Equation (3), is preserved.

In order to perform the generation and verification of the proof, we turn to zk-SNARKs. Recall that the circuit C was generated in the `Setup` step. C takes a main input X and witness A that are constructed as follows:

$$X = \left(\text{rt}, \text{sn}_1^{\text{old}}, \text{sn}_2^{\text{old}}, \text{cm}_1^{\text{new}}, \text{cm}_2^{\text{new}}, v_{\text{pub}}, h_{\text{Sig}}, h_1, h_2\right) \tag{4}$$

$$A = \left(\text{path}_1, \text{path}_2, \text{c}_1^{\text{old}}, \text{c}_2^{\text{old}}, \text{addr}_{\text{sk},1}^{\text{old}}, \text{addr}_{\text{sk},2}^{\text{old}}, \text{c}_1^{\text{new}}, \text{c}_2^{\text{new}}\right), \tag{5}$$

where rt is the root of the Merkle tree over coin commitments, the sn are the coin serial numbers, the cm are the coin commitments, v_{pub} is the publicly spent amount in the pour transaction, the h are parameters used to ensure non-malleability, the paths are Merkle tree authentication paths for the two old coins, the c are the coins themselves and the addr are the secret address keys associated with the old coins.

`Pour` also appends its transaction to the ledger. Note that a transaction tuple does not reveal any of the coin values or recipient addresses besides from v_{pub} — it contains only the following information:

$$\left(\text{rt}, \text{sn}_1^{\text{old}}, \text{sn}_2^{\text{old}}, \text{cm}_1^{\text{new}}, \text{cm}_2^{\text{new}}, v_{\text{pub}}, \text{info}, *\right), \tag{6}$$

where $*$ is implementation-specific, but would likely include the zero-knowledge proof Π generated by `Prove`. `Pour` returns the transaction data structure and the two new coins.

3.3.5 `Receive`

`Receive` is used to receive payments. It takes as input the ledger and the address key pair of the recipient. It returns a set of coin objects that have not been spent by the recipient, but that were paid to the recipient through `Pour` transactions. This excludes coins that were minted by the recipient.

3.3.6 `VerifyTransaction`

The purpose of `VerifyTransaction` is to determine whether or not a provided transaction is valid. It takes as input the ledger, the public parameters generated by `Setup` and a mint or pour transaction. It returns 1 if the transaction is valid, and 0 otherwise. In theory, `VerifyTransaction` could be utilized by both individual users and the nodes of the distributed system.

4 Related Work

Although there has been an extensive amount of research related to zk-SNARKs, only a few have made their way into cryptographic tools [4]. Nevertheless, zk-SNARKs are especially useful for blockchains — for reasons other than the mere improvement of privacy guarantees. For example, zk-SNARK proofs are non-interactive, which means that verifiers can check a proof at their leisure, without collaborating with the prover. In addition, zk-SNARKs proofs are concise, which means that they can be verified efficiently. These properties can be used to improve the scalability of blockchains [4].

It should therefore come as no surprise the zk-SNARKs have been used in blockchain systems other than Zerocash. An example of such a system is CODA [5], where one of the main ideas is to bundle up a group of transactions, calculate a zero-knowledge proof for each, and then provide a single proof that can be used to verify them all [4]. There has also been some Zerocash-inspired work that explores integrating zk-SNARKs into Ethereum [6].

In addition to the various blockchain systems that capitalize on zk-SNARKs, there are also variations of zero-knowledge proofs themselves. These include zero-knowledge Succinct Transparent Arguments of Knowledge (zk-STARKs) and bulletproofs.

One of the main advantages of zk-STARKs over zk-SNARKs is the excellent security provided by zk-STARKs — for example, they do not require a trusted setup. However, due to their large proof size, which grows as $\mathcal{O}(\log^2 |C|)$ with respect to the circuit size $|C|$, zk-STARKs are currently not as practical as zk-SNARKs. For the latter, the proof size remains constant as $|C|$ increases [4].

Bulletproofs also do not require a trusted setup. However, their proof size scales as $\mathcal{O}(\log |C|)$, which again makes them (generally) less performant than zk-SNARKs [4].

5 Evaluation

Our attempt to implement a basic version of Zerocash in Python was partially successful. First, we were able to build a basic blockchain from scratch. Second, we made our blockchain distributed through the implementation of a `Node` class, where different nodes could be run on different ports of the same machine. Third, we managed to build most Zerocash functionalities on top of our blockchain. Lastly, we integrated our code with the C++ zk-SNARK library libsnark and constructed the portion of the NP Statement POUR that checks for coin values. However, we struggled to construct the rest of POUR, which was mostly due to the lack of documentation and tooling on zk-SNARKs.

References

[1] Eli Ben-Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *IEEE Symposium on Security and Privacy* (2014), pp. 459–474.

[2] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: `https://bitcoin.org/bitcoin.pdf`.

[3] Ian Miers et al. "Zerocoin: Anonymous Distributed E-Cash from Bitcoin". In: *IEEE Symposium on Security and Privacy* (2013), pp. 397–411.

[4] Alexandre Miranda Pinto. "An Introduction to the Use of zk-SNARKs in Blockchains". In: *Mathematical Research for Blockchain Economy*. Ed. by Panos Pardalos et al. Cham: Springer International Publishing, 2020, pp. 233–249. ISBN: 978-3-030-37110-4.

[5] Joseph Bonneau et al. *Coda: Decentralized Cryptocurrency at Scale*. 2018. URL: `https://eprint.iacr.org/2020/352.pdf`.

[6] Antoine Rondelet and Michal Zajac. *ZETH: On Integrating Zerocash on Ethereum*. 2019. URL: `https://arxiv.org/pdf/1904.00905.pdf`.