CS 261 Notes: Zerocash

Scribe: Lynn Chua

September 19, 2018

1 Introduction

Zerocash [1] is a cryptocurrency which allows users to pay each other directly, without revealing any information about the parties involved or the amount of the transaction. In comparison, Bitcoin does not protect privacy since every transaction reveals the sender, receiver and the payment amount on a public ledger. Although only the public keys are revealed, the graph of all the transactions over time can be correlated with side information to reveal the real identities associated to each public key.

Why is privacy important? Firstly, the payment history can reveal a lot of information, for instance medical information could be used by insurance companies to increase your premiums. Moreover, with Bitcoin the monetary worth of each user is public, and people who are rich with Bitcoin can be targeted physically. In fact, there have been instances where people were attacked physically to acquire their Bitcoin keys. Furthermore, the US government has a regulation that requires financial institutions to safeguard financial data. This demonstrates that financial privacy is indeed recognized to be important, and shows how necessary anonymity is in cryptocurrencies.

2 Preliminaries

Before discussing how Zerocash works, we introduce three cryptographic building blocks that are used in the protocol, namely commitments, key-private public key encryption and zero-knowledge proofs of knowledge. We will not discuss how these cryptographic primitives are constructed (this is discussed in CS276), but we will talk about how to use them. In practice, there are several libraries available, which have already implemented these primitives and these libraries can be used in applications.

2.1 Commitments

A commitment scheme (COMM, check) allows one to commit to a value while keeping it hidden, such that the value cannot be changed after committing to it. One can then reveal the hidden value at a later time, and another party can check that the revealed value is the same as the committed one. A commitment scheme has the following two operations.

1. **Commit**: Given a message z and a secret r, the commitment is $cm = \text{COMM}_r(z)$. COMM can also accept multiple input arguments, where we compute the commitment by concatenating the inputs.
2. **Reveal/open/decommit**: Given cm, r, z check whether z is the committed message, via the operation $\text{check}(cm, r, z) = \text{yes/no}$.

The commitment scheme has to satisfy the following security properties:

1. **Hiding**: Given only the commitment cm, no information about the message z can be deduced.
2. **Binding**: Given a commitment $cm = \text{COMM}_r(z)$, it should be infeasible to find a different message $z' \neq z$ and r' such that $\text{check}(cm, z', r') = \text{yes}$.

How do commitment schemes differ from encryption schemes? Firstly, we note that not all encryption schemes are commitment schemes. For example, consider the one-time pad, where we combine the message with the key using modular addition to get the ciphertext. Although the hiding property is satisfied, it is not binding since it is easy to find a different message and key which give the same ciphertext.

We consider the case of **ElGamal encryption** [3], where we have a cyclic group G with a generator g. Let p be the order of G, we assume that p is a large prime. We denote by $\mathbb{Z}_p$ the finite field of order p.

- **Key generation**: $\text{KeyGen}(pp) = (s, g^s)$ takes public parameters pp and outputs a random secret $s \in \mathbb{Z}_p$ which forms the secret key, and the public key $g^s \in G$.
- **Encryption**: Take as input the public key g^s of the receiver and a message m encoded as an element of G, and encrypt m via $\text{Enc}(g^s, m) = (g^y, mg^{sy})$, where y is chosen randomly from $\mathbb{Z}_p$.
- **Decryption**: Take as input the secret key s of the receiver and the cipertext (c_1, c_2), and decrypt as $\text{Dec}(s, (c_1, c_2)) = c_2/c_1^s$. This recovers the message m since for a ciphertext encrypted under the public key g^s, we have $c_2/c_1^s = mg^{sy}/g^{sy} = m$.

The security of ElGamal encryption holds by the *Decisional Diffie Hellman (DDH)* assumption. This says that the distribution $\{(g^s, g^y, g^{ys})\}$ for s, y randomly and independently chosen from $\mathbb{Z}_p$, is computationally indistinguishable from the distribution $\{(g^s, g^y, g^r)\}$ for s, y, r randomly and independently chosen from $\mathbb{Z}_p$. This assumption implies that given g^s and g^y in the ElGamal encryption scheme, an attacker would not be able to gain any information about g^{sy} and hence would not be able to deduce the message m from its encryption. This also gives the hiding property needed for commitment schemes.

Does ElGamal satisfy the binding property? We consider two possibilities for a commitment scheme based on ElGamal encryption.

1. $\text{COMM}_s(m) = (g^s, g^y, mg^{sy})$. Decommitment reveals s, y, m. A verifier can check that s, y are correct by computing g^s and g^y, so s, y have to be revealed honestly. Hence g^{sy} can also be computed, and used in turn to compute m. Thus we cannot decommit to any other message, so this is a binding commitment.
2. $\text{COMM}_s(m) = (g^y, mg^{sy})$. In this case, decommitment need not reveal the correct values of s, y, m. Suppose for instance that $m = 1$, and let $m' = g^y$, $s' = s - 1$. Then $m' g^{s'y} = g^y g^{(s-1)y} = g^{sy}$, so we can decommit to m' instead. Hence this is not a binding commitment.

This shows that encryption schemes can sometimes be commitment schemes if enough information is provided. In particular, ElGamal can be a good commitment scheme.

2.2 Key-private public key encryption

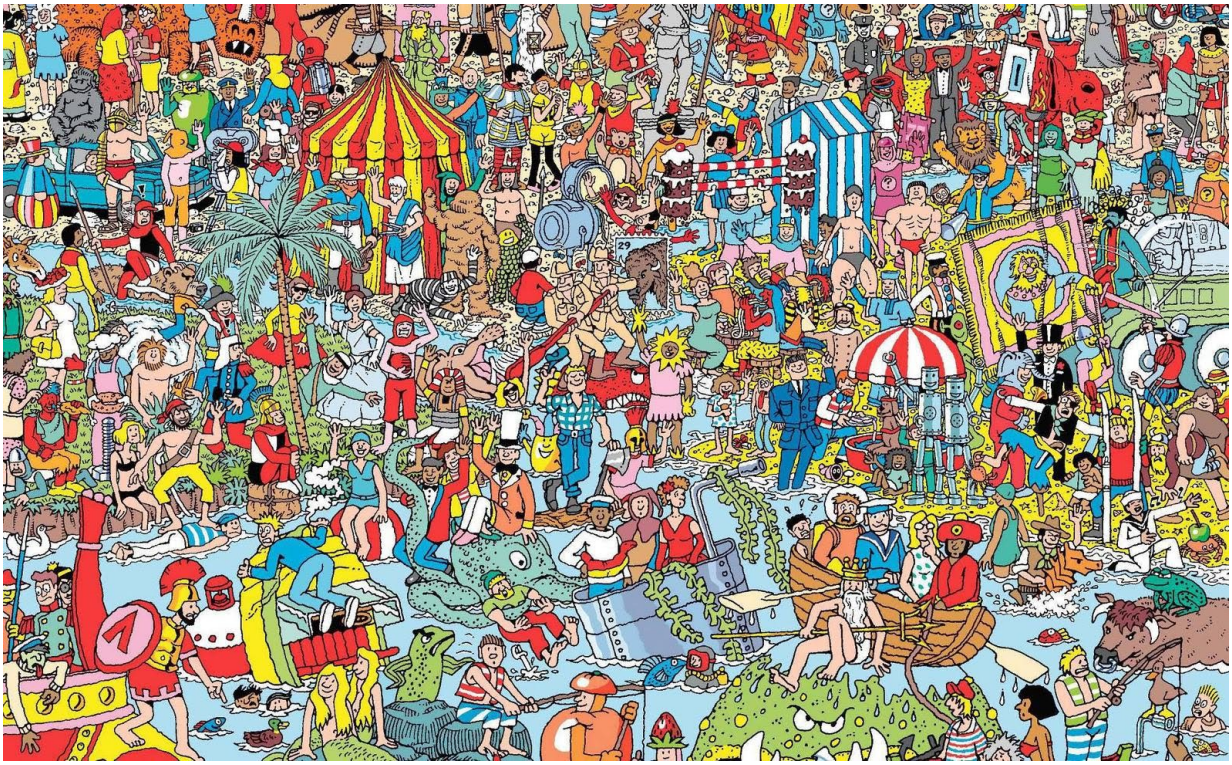
In Zerocash, the goal is to encrypt transactions for a receiver without revealing who the receiver is. Hence we need an encryption scheme such that the ciphertexts hide the public key of the receiver, so an attacker would not be able to learn who the ciphertexts are encrypted for. We call such an encryption scheme a *key-private public key encryption scheme.*

Not all encryption schemes satisfy this property, as their main goal is to hide the message, and the semantic security does not enforce the hiding of the public key. For instance, we can construct an encryption scheme by appending the public key to the ciphertext; this would still satisfy semantic security although it reveals the public key.

Consider the ElGamal encryption scheme again, with $\text{Enc}(pk, m) = (g^y, mg^{sy})$ where the public key is $pk = g^s$. Suppose the attacker knows m and two possible public keys g^{s_1}, g^{s_2}. To figure out who the recipient of the cipertext is, the attacker would have to distinguish between $(g^{y_1}, mg^{y_1 s_1})$ and $(g^{y_2}, mg^{y_2 s_2})$. However, this cannot be done by the Decisional Diffie Hellman assumption. Hence ElGamal is a key-private public key encryption scheme.

2.3 Zero-knowledge proof of knowledge

Suppose we have a big puzzle, and I want to convince you that I know the solution to the puzzle, without revealing any information about the solution. For instance, the puzzle could be a "Where's Waldo" puzzle like this [4]:



The goal is to find where Waldo is in the picture[1], where Waldo looks like this [6]:

I want to convince you that I know where Waldo is, without letting you know anything about the location of Waldo. In other words, I want to give you a *zero-knowledge proof* of the fact that I know where Waldo is. We discuss a few attempts to construct such a zero-knowledge proof.

1. Crop the puzzle to a small part containing Waldo, and blur out the background around Waldo.
   Problem: How can I convince you that this cropped image of Waldo is indeed from the puzzle? I could have simply copied a picture of Waldo from somewhere else.

2. Produce two copies of the puzzle, one with Waldo and one without, such that you cannot tell them apart (since you don't know where Waldo is). I label them and let you pick one of the copies randomly. Then I should be able to tell you the label of the copy that you picked.
   Problem: In this case, I have a 50% chance of being correct even if I don't actually know where Waldo

[1] Clue: Waldo is in the upper right corner!

is, so for accuracy this procedure has to be repeated many times to convince you. Is there a simpler way without any back-and-forth interaction?

3. Take a large piece of paper that can cover the entire puzzle, and make a small slit across it the size of Waldo. Place the puzzle below the paper and shuffle the puzzle under the slit until Waldo shows up on the slit. Then you can look at the slit to see Waldo, while not being able to tell where the slit is located relative to the puzzle.

In general, we have a *prover* who wants to convince a *verifier* that the prover knows a *witness* to a statement, without revealing anything about the witness. In the example above, the witness would be the location of Waldo in the puzzle. There are protocols to create zero-knowledge proofs for a wide range of statements, where a trusted party conducts a one-time setup to give a proving key pk and a verification key vk. The prover and verifier can then use the following operations:

- $\text{Prove}(pk, x, w) = \pi$, where x is a statement that the prover wants to convince the verifier, w is a witness, and π is a proof that reveals nothing about w.
- $\text{Verify}(vk, x, \pi) = \text{yes/no}$. The verifier takes the verification key and the proof, and determines whether the proof is valid or not.

Additionally, it would be desirable for zero-knowledge proofs to be very short and fast to verify, in a non-interactive way. We call such proofs **zkSNARKs** (zero-knowledge Succinct Non-interactive ARguments of Knowledge), and we elaborate on their properties below.

1. **Zero-knowledge**: Proof only reveals if the statement is true or not. In particular, the witness is hidden.
2. **Succinctness**: Proof is very short and fast to verify.
3. **Non-interactivity**: The prover can give a one-time proof to convince the verifier, without the need for any back-and-forth interaction.
4. **Soundness**: "Proofs" of false statements are rejected with high probability.
5. **Proof of knowledge**: The prover knows a witness to the statement, and only true statements have proofs.
6. **Completeness**: Correctly generated proofs of valid statements are always accepted.

zkSNARKs are quite incredible since for instance, a prover can run a computation for a year on a remote server, and send a very short proof for the verifier to check in a few milliseconds that the computation was correct. There has been a lot of research into zkSNARKs, and there are also publicly available implementations for use in practice (for example [5]). We refer the interested reader to Section 2 of [1] for more details and references.

3 Zerocash Protocol

Zerocash is a cryptocurrency that works on another ledger to enable anonymity while still being efficient. For instance, we can add it on top of the Bitcoin ledger, such that users can make transactions that hide the sender, receiver and amount.

The basic intution behind Zerocash is that with each transaction, we will attach a short proof (zkSNARK) that the transaction is valid and contains the information that we want to hide, and anyone should be able to verify this proof quickly. However, there are many details in the actual protocol design. We describe the protocol by starting with a basic variant and adding to it incrementally. In Zerocash, there are two types of

transactions, minting and spending, which we denote by MINT and SPEND respectively. There is also a coin datastructure. We will describe these transactions in each step, together with how the coin datastructure evolves. In what follows, we will use the notation:

- sn: serial number
- cm, k: commitment
- s, r: randomness
- π: zero-knowledge proof
- v: value of coin
- ρ: serial number seed
- pk: public key
- sk: secret key

3.1 Attempt #1: basic serial numbers

$$\text{MINT} : (sn)\,, \qquad \text{SPEND} : (sn)\,, \qquad \text{coin} : (sn)\,.$$

We first define a coin to be a serial number sn. We can mint a coin by broadcasting on the Zerocash ledger that we are minting a coin with serial number sn. We can spend a coin by broadcasting the serial number of the coin that we are spending.

This is a very simple but rather broken construction. The good thing is that double spending cannot happen, since the serial numbers uniquely identify the coins. So one cannot spend the same coin with the same serial number twice without being detected. However, this does not hide any information, since anyone can tell which MINT transaction created the coin in a SPEND transaction. Even worse, anyone can spend the coin since the serial numbers are public. Moreover, each coin has a fixed denomination.

3.2 Attempt #2: coin commitments

$$\text{MINT} : (cm)\,, \qquad \text{SPEND} : (sn, r)\,, \qquad \text{coin} : (cm, sn, r)\,.$$

Like before, each coin has a serial number sn, but now we mint coins by using a commitment to the serial number. More precisely, to mint a coin we first sample a serial number and some randomness r, and we broadcast the coin commitment $cm = \text{COMM}_r(sn)$ on the blockchain. To spend the coin, we reveal sn and r, so that others can compute cm and verify that the spend transaction is valid. The coin datastructure now consists of (cm, sn, r).

The mint transaction no longer reveals the serial number of the minted coin, so only the owner of the coin (who knows the serial number) can spend it. Note that other users can only view the commitment, and because of the hiding property of the commitment scheme, attackers cannot find the serial number. Moreover, the binding property prevents anyone from finding another serial number with the same commitment, so double spending is prevented. However, the spend transaction is still linkable to the mint, and coins still have fixed denominations.

3.3 Attempt #3: zero-knowledge proofs

$$\text{MINT} : (cm)\,, \qquad \text{SPEND} : (sn, \pi)\,, \qquad \text{coin} : (cm, sn, r)\,.$$

Next, we will add a zero-knowledge proof to the spend transaction, so that observers cannot link the spend and mint transactions. The mint transaction remains as before, but in the spend transaction, we reveal the serial number sn and instead of r, we now reveal a proof π that we know r. More precisely, we will add a zero-knowledge proof π that we know (cm, r) such that the following properties hold.

- Existence: cm is in the list of all prior coin commitments on the blockchain.
- Well-formed: $cm = \text{COMM}_r(sn)$.

With this modification, one can no longer link a spend transaction to a previous mint, since the commitment cm is no longer revealed in the spend transaction. Nevertheless, there are still a few problems with this protocol. Namely, we can only use coins with fixed denomination, which may reveal information since one would need to make more transactions to spend higher amounts of coins.

3.4 Attempt #4: variable denominations

$$\text{MINT} : (cm, k, v, r)\,, \qquad \text{SPEND} : (sn, v, \pi)\,, \qquad \text{coin} : (cm, k, v, r, s, sn)\,.$$

We now modify the protocol to allow coins of variable denomination, by using a nested commitment to commit to the value of each coin. As before, for each coin we commit to the serial number and randomness. We will add another commitment on top of this to commit to the coin value. So our coin datastructure will now consist of (cm, k, v, r, s, sn), where $k = \text{COMM}_s(sn)$ is the (inner) commitment to the serial number, and $cm = \text{COMM}_r(v, k)$ is the (outer) commitment to the coin value.

We mint a coin of value v by revealing (cm, k, v, r). This does not reveal any information about the serial number, since s and sn are hidden in the commitment k. But observers will be able to verify the value of the coin by checking that $cm = \text{COMM}_r(v, k)$.

To spend the coin, we reveal (sn, v, π), where π is now a zero-knowledge proof that we know (cm, k, r, s) such that:

- Existence: cm is in the list of all prior coin commitments on the blockchain.
- Well-formed: $cm = \text{COMM}_r(v, k)$ and $k = \text{COMM}_s(sn)$.

We have now made progress by allowing variable denominations. However, the values of the coins are revealed in the transactions. Moreover, a spend transaction with a value v coin can only come after a mint transaction of a value v coin, so we can link spend and mint transactions of the same values.

3.5 Attempt #5: payment addresses

$$\text{MINT} : (cm, k, v, r)\,, \qquad \text{SPEND} : (sn, v, \pi)\,, \qquad \text{coin} : (cm, k, v, r, s, sn, \rho, pk, sk)\,.$$

We now add payment addresses to the protocol. Each coin will have an associated public key pk, secret key sk and a serial number seed ρ. We also fix a pseudorandom function PRF. The public key, which specifies the owner of the coin, is computed as $pk = \text{PRF}_{sk}(0)$, and the serial number is computed as $sn = \text{PRF}_{sk}(\rho)$. We modify the coin commitments to be $cm = \text{COMM}_r(v, k)$ and $k = \text{COMM}_s(pk, \rho)$.

The mint transaction is as before, while in the spend transaction (sn, v, π) we will prove a few more things. Now π is a zero-knowledge proof that we know $(cm, k, r, s, \rho, pk, sk)$ such that:

- Existence: cm is in the list of all prior coin commitments on the blockchain.
- Well-formed: $cm = \text{COMM}_r(v, k)$ and $k = \text{COMM}_s(pk, \rho)$.
- Mine: $pk = \text{PRF}_{sk}(0)$ and $sn = \text{PRF}_{sk}(\rho)$.

Before, the serial number was part of the commitment. Now the serial number is computed using the same secret key from which the public key is derived. Note that π shows that the spender knows the secret key and hence owns the coin, while also showing that the serial number was computed correctly. However, the spend transactions still reveal the value of the coin, and we did not really use the addresses here.

3.6 Attempt #6: mechanism to transfer coins

$$\text{MINT} : (cm, k, v, r)\,, \qquad \text{SPEND} : (sn_A, cm_B, \pi)\,, \qquad \text{coin} : (cm, k, v, r, s, sn, \rho, pk, sk)\,.$$

We now add a mechanism for transferring coins from one party A to another party B. The coin and the mint transaction are the same as before, while we modify the spend transaction as follows. Since each coin has a public key, a sender A can transfer a coin to B by minting a coin with B's public key. But we have to be careful here: since A mints the coin for B, A knows the secrets for the coin. We need to devise the protocol such that A cannot spend a coin that was minted for B. The solution here is that A can mint the coin using only the public key pk_B, while A does not know the secret key sk_B. Since the serial number of the coin is derived from sk_B, A would not be able to spend the coin. Only B can spend the coin since B knows sk_B.

The spend transaction now reveals (sn_A, cm_B, π), where the zero-knowledge proof π shows that A knows $(cm_A, k_A, r_A, s_A, \rho_A, pk_A, sk_A)$ and $(cm_B, k_B, r_B, s_B, \rho_B, pk_B)$ such that:

- Existence: cm_A is in the list of all prior coin commitments on the blockchain.
- Well-formed: $cm_A = \text{COMM}_{r_A}(v_A, k_A)$, $k_A = \text{COMM}_{s_A}(pk_A, \rho_A)$. Similarly, $cm_B = \text{COMM}_{r_B}(v_B, k_B)$, $k_B = \text{COMM}_{s_B}(pk_B, \rho_B)$.
- Mine: $pk_A = \text{PRF}_{sk_A}(0)$ and $sn_A = \text{PRF}_{sk_A}(\rho_A)$.
- Same value: $v_A = v_B$.

Notice that the value of the coin is now hidden. All the spend transaction reveals is the serial number of a coin that is destroyed, and the commitment for a new coin that is created. This enables direct payments between users, while hiding the sender, receiver and the amount.

3.7 Remarks

There are still some additional features in the final Zerocash design, such as joining and splitting coins. This is done using a POUR transaction, which allows to combine coins and split them into desired denominations while preserving anonymity. Moreover, for efficiency the coin commitments are stored in a Merkle tree rather than a list. The protocol is also designed so that transactions are *non-malleable*, which means that adversaries cannot transform transactions on the blockchain into other valid transactions that they could not have made by themselves.

The details of the protocol are in [1], and the talk in [2] also gives a good overview of how the protocol works.

References

[1] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, *Zerocash: Decentralized Anonymous Payments from Bitcoin*, in 2014 IEEE Symposium on Security and Privacy (SP), vol. 00, May 2014, pp. 459–474.

[2] A. Chiesa, *Zerocash: Addressing Bitcoin's privacy problem.* `https://www.youtube.com/watch?v=84Vbj7-i9CI`, 2017.

[3] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, in IEEE Transactions on Information Theory, vol. 31, July 1985, pp. 469–472.

[4] S. Knight, *This AI-powered facial recognition robot zaps the fun from 'Where's Waldo?'*. `https://www.techspot.com/news/75939-ai-powered-facial-recognition-robot-zaps-fun-where.html`, 2018.

[5] S. Lab, *libsnark: a C++ library for zkSNARK proofs.* https://github.com/scipr-lab/libsnark, 2012-2017.

[6] Waldo, *Where's Waldo? Find him in Google Maps.* https://www.blog.google/products/maps/wheres-waldo-find-him-google-maps/, 2018.