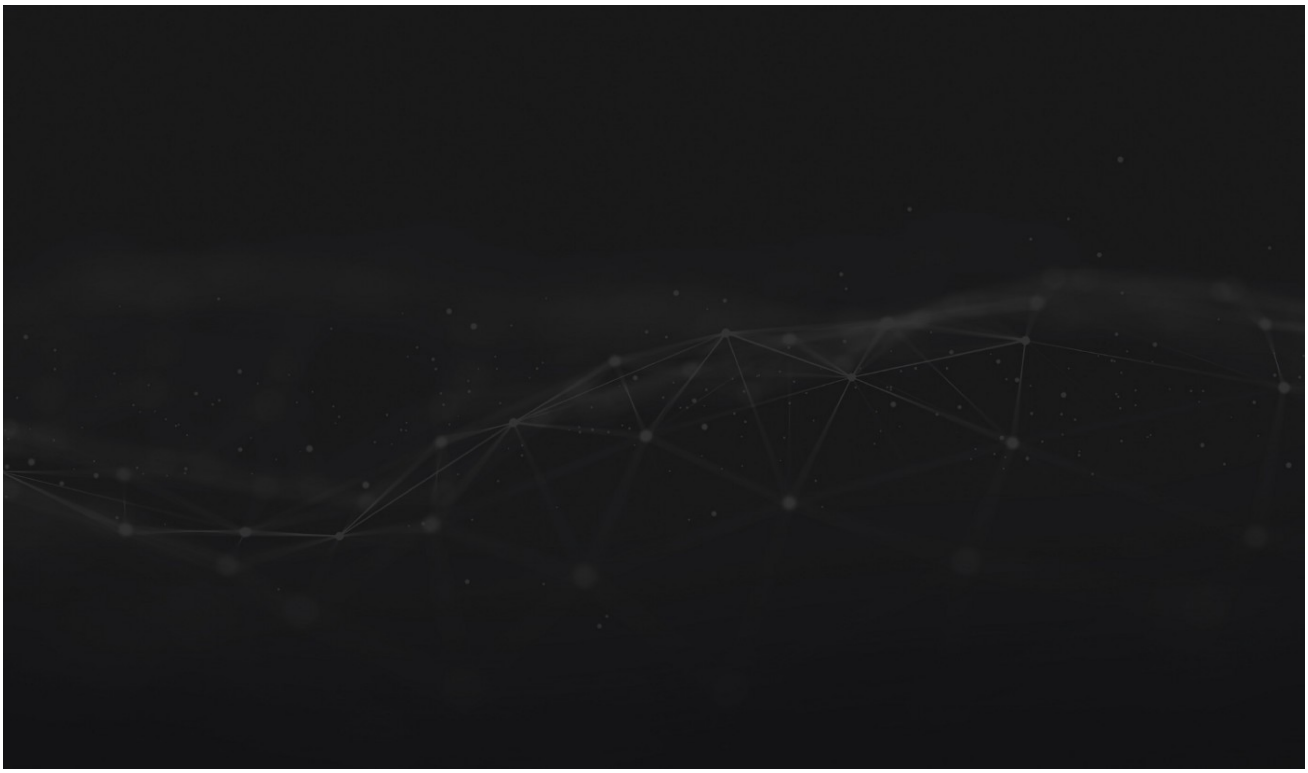
Publ...

Christian Lundkvist Follow

Mar 27, 2017 · 8 min read · Listen

Introduction to zk-SNARKs with Examples



Subscribe to the ConsenSys developer newsletter for more technical guides

In this post we aim to give an overview of zk-SNARKs from a practical viewpoint. We will treat the actual math as a black box but will try to develop some intuitions around how we can use them, and will give a simple application of the recent work on integrating zk-SNARKs in Ethereum.

a *prover* p ing some relation, without revealing the witness to the verifier or anyone else.

We can think of this more concretely as having a program, denoted `C`, taking two inputs: `C(x, w)`. The input `x` is the public input, and `w` is the secret *witness* input. The output of the program is boolean, i.e. either `true` or `false`. The goal then is given a specific public input `x`, prove that the prover knows a secret input `w` such that `C(x,w) == true`.

We are specifically going to discuss *non-interactive* zero knowledge proofs, which means that the proof itself is a blob of data that can be verified without any interaction from the prover.

Example program

Suppose Bob is given a hash `H` of some value, and he wishes to have a proof that Alice knows the value `s` that hashes to `H`. Normally Alice would prove this by giving `s` to Bob, after which Bob would compute the hash and check that it equals `H`.

However, suppose Alice doesn't want to reveal the value `s` to Bob but instead she just wants to prove that she knows the value. She can use a zk-SNARK for this.

We can describe Alice's scenario using the following program, here written as a Javascript function:

```
function C(x, w) {
  return ( sha256(w) == x );
}
```

In other words: the program takes in a public hash `x` and a secret value `w` and returns `true` if the SHA–256 hash of `w` equals `x`.

Translating Alice's problem using the function `C(x,w)` we see that Alice needs to create a proof that she possesses `s` such that `C(H, s) == true` without having to

Definition fits in a tweet :)

A *zk-SNARK* consists of three algorithms `G, P, V` defined as follows:

The *key generator* `G` takes a secret parameter `lambda` and a program `C`, and generates two publicly available keys, a *proving key* `pk`, and a *verification key* `vk`. These keys are public parameters that only need to be generated once for a given program `C`.

The *prover* `P` takes as input the proving key `pk`, a public input `x` and a private witness `w`. The algorithm generates a *proof* `prf = P(pk, x, w)` that the prover knows a witness `w` and that the witness satisfies the program.

The *verifier* `V` computes `V(vk, x, prf)` which returns `true` if the proof is correct, and `false` otherwise. Thus this function returns true if the prover knows a witness `w` satisfying `C(x,w) == true`.

Note here the secret parameter `lambda` used in the generator. This parameter sometimes makes it tricky to use zk-SNARKs in real-world applications. The reason for this is that anyone who knows this parameter can generate fake proofs. Specifically, given any program `C` and public input `x` a person who knows `lambda` can generate a proof `fake_prf` such that `V(vk, x, fake_prf)` evaluates to `true` without knowledge of the secret `w`.

lambda w

A zk-SNARK for our example program

How would Alice and Bob use a zk-SNARK in practice in order for Alice to prove that she knows the secret value in the example above?

First of all, as discussed above we will use a program defined by the following function:

```
function C(x, w) {
  return ( sha256(w) == x );
}
```

The first step is for Bob to run the generator `G` in order to create the proving key `pk` and verification key `vk`. This is done by first randomly generating `lambda` and using that as input:

```
(pk, vk) = G(C, lambda)
```

As discussed above, the parameter `lambda` must be handled with care, since if Alice learns the value of `lambda` she will be able to create fake proofs. Bob will share `pk` and `vk` with Alice.

Alice will now play the role of the prover. She needs to prove that she knows the value `s` that hashes to the known hash `H`. She runs the proving algorithm `P` using the inputs `pk`, `H` and `s` to generate the proof `prf`:

```
prf = P(pk, H, s)
```

Next Alice presents the proof `prf` to Bob who runs the verification function `V(vk,`

Reusable proving and verification keys

In our example above the zk-SNARK cannot be used if Bob wants to prove to Alice that he knows a secret. This is because Alice cannot know that Bob didn't save the `lambda` parameter, and so Bob could plausibly be able to fake proofs.

If a program is useful to many people (like the example of Zcash) a trusted independent group separate from Alice and Bob could run the generator and create the proving key `pk` and verification key `vk` in such a way that no one learns about `lambda`.

Anyone who trusts that the group did not cheat can then use these keys for future interactions.

zk-SNARKs in Ethereum

Developers have already started integrating zk-SNARKs into Ethereum. What does this look like? Concretely, the building blocks of the verification algorithm is added to Ethereum in the form of precompiled contracts. The usage is the following: The generator is run off-chain to produce the proving key and verification key. Any prover can then use the proving key to create a proof, also off-chain. The general verification algorithm can then be run inside a smart contract, using the proof, the verification key and the public input as input parameters. The outcome of the verification algorithm can then be used to trigger other on-chain activity.

Example: Confidential transactions

Photo by Casey Marshall

Here is a simple example of how zk-SNARKs can help with privacy on Ethereum. Suppose we have a simple token contract. Normally a token contract would have at its core a mapping from addresses to balances:

```
mapping (address => uint256) balances;
```

We are going to retain the same basic core, except replace a balance with the hash of a balance:

```
mapping (address => bytes32) balanceHashes;
```

We are not going to hide the sender or receiver of transactions, but we'll be able to hide the balances and sent amounts. This property is sometimes referred to as confidential transactions.


```
balances[fromAddress] >= value
```

Thus what our zk-SNARKs would need to prove is that this holds as well as that the updated hashes matches the updated balances.

The main idea is that the sender will use their starting balance and the transaction value as private inputs, and hashes of starting balance, ending balance and value as public inputs. Similarly the receiver will use starting balance and value as secret inputs and hashes of starting balance, ending balance and value as public inputs.

Below is the program we will use for the sender zk-SNARK, where as before `x` represents the public input, and `w` represents the private input.

```
function senderFunction(x, w) {
  return (
    w.senderBalanceBefore > w.value &&
    sha256(w.value) == x.hashValue &&
    sha256(w.senderBalanceBefore) == x.hashSenderBalanceBefore &&
    sha256(w.senderBalanceBefore - w.value) ==
x.hashSenderBalanceAfter
  )
}
```

The program used by the receiver is below:

```
function receiverFunction(x, w) {
  return (
    sha256(w.value) == x.hashValue &&
    sha256(w.receiverBalanceBefore) ==
x.hashReceiverBalanceBefore &&
    sha256(w.receiverBalanceBefore + w.value) ==
x.hashReceiverBalanceAfter
  )
}
```


proving a erPk , `confTxSenderVk` , `confTxReceiverPk` and `confTxReceiverVk` .

Using the zk-SNARKs in a token contract would look something like this:

```
function transfer(address _to, bytes32 hashValue, bytes32
hashSenderBalanceAfter, bytes32 hashReceiverBalanceAfter, bytes
zkProofSender, bytes zkProofReceiver) {
  bytes32 hashSenderBalanceBefore = balanceHashes[msg.sender];
  bytes32 hashReceiverBalanceBefore = balanceHashes[_to];

  bool senderProofIsCorrect = zksnarkverify(confTxSenderVk,
[hashSenderBalanceBefore, hashSenderBalanceAfter, hashValue],
zkProofSender);

bool receiverProofIsCorrect = zksnarkverify(confTxReceiverVk,
[hashReceiverBalanceBefore, hashReceiverBalanceAfter, hashValue],
zkProofReceiver);

  if(senderProofIsCorrect && receiverProofIsCorrect) {
    balanceHashes[msg.sender] = hashSenderBalanceAfter;
    balanceHashes[_to] = hashReceiverBalanceAfter;
  }
}
```

Thus the only updates on the blockchain are the hashes of the balances and not the balances themselves. However, we can know that all the balances are correctly updated because we can check ourselves that the proof has been verified.

Details

The above confidential transaction scheme is mainly to give a practical example of how one can use zk-SNARKs on Ethereum. In order to create a robust confidential transaction scheme based on this we would need to address a number of issues:

- Users would need to keep track of their balances client-side, and if you lose the balance those tokens are unrecoverable. The balances could perhaps be stored encrypted on-chain with a key derived from the signing key.

- The se[illegible] ild potentially have a system wi[illegible] 3.6K 18 [illegible] proof to initiate the transaction, and the receiver can see on the blockchain that they have a "pending incoming transaction" and can finalize it.