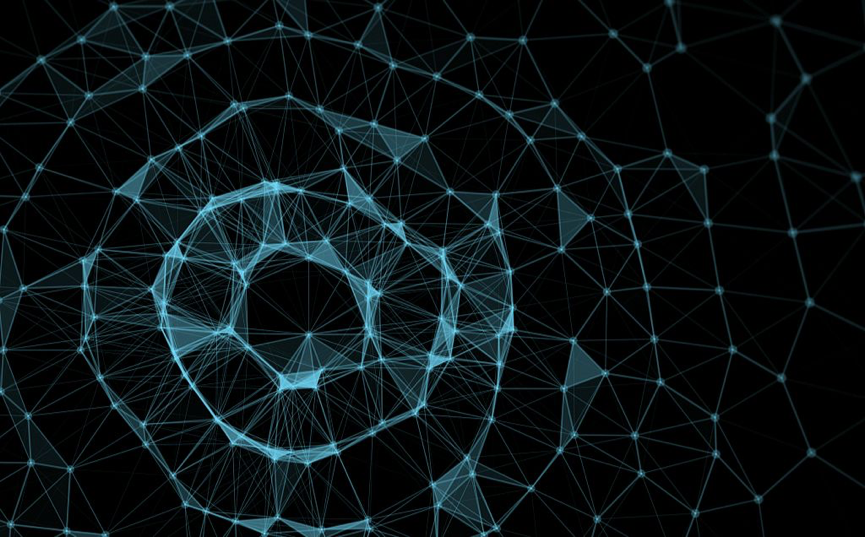
**[An Update on Integrating Zcash on Ethereum (ZoE)](https://blog.ethereum.org/2017/01/19/update-integrating-zcash-ethereum/)**

By [Christian Reitwiessner](https://blog.ethereum.org/author/christian_r/)



Members of the Ethereum R&D team and the Zcash Company are collaborating on a research project addressing the combination of programmability and privacy in blockchains. This joint post is being concurrently posted on the [Zcash blog](https://z.cash/blog/zcash-eth.html), and is coauthored by Ariel Gabizon (Zcash) and Christian Reitwiessner (Ethereum).

Ethereum’s flexible smart contract interface enables a large variety of applications, many of which have probably not yet been conceived. The possibilities grow considerably when adding the capacity for privacy. Imagine, for example, an election or auction conducted on the blockchain via a smart contract such that the results can be verified by any observer of the blockchain, but the individual votes or bids are not revealed. Another possible scenario may involve selective disclosure where users would have the ability to prove they are in a certain city without disclosing their exact location. The key to adding such capabilities to Ethereum is zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) – precisely the cryptographic engine underlying Zcash.

One of the goals of the Zcash company, codenamed [Project Alchemy](https://z.cash/blog/project-alchemy.html), is to enable a direct decentralized exchange between Ethereum and Zcash. Connecting these two blockchains and technologies, one focusing on programmability and the other on privacy, is a natural way to facilitate the development of applications requiring both.



As part of the Zcash/Ethereum technical collaboration, Ariel Gabizon from Zcash visited Christian Reitwiessner from the Ethereum hub at Berlin a few weeks ago. The highlight of the visit is a proof of concept implementation of a zk-SNARK verifier written in Solidity, based on pre-compiled Ethereum contracts implemented for the Ethereum C++ client. This work complements [Baby ZoE](https://z.cash/blog/zksnarks-in-ethereum.html) , where a zk-SNARK precompiled contract was written for Parity (the Ethereum Rust client). The updates we’ve made involved adding tiny cryptographic primitives (elliptic curve multiplication, addition and pairing) and implementing the rest in Solidity, all of which allows for a greater flexibility and enables using a variety of zk-SNARK constructions without requiring a hard fork. Details will be shared as they are available later. We tested the new code by successfully verifying a real privacy-preserving Zcash transaction on a testnet of the Ethereum blockchain.

The verification took only 42 milliseconds, which shows that such precompiled contracts can be added, and the gas costs for using them can be made to be quite affordable.

### What can be done with such a system

The Zcash system can be reused on Ethereum to create shielded custom tokens. Such tokens already allow many applications like voting, (see below) or simple blind auctions where participants make bids without the knowledge of the amounts bid by others.

If you want to try compiling the proof of concept, you can use the following commands. If you need help, see <https://gitter.im/ethereum/privacy-tech>

git clone https:*//github.com/scipr-lab/libsnark.git*

cd libsnark

sudo PREFIX=/usr/local make NO\_PROCPS=1 NO\_GTEST=1 NO\_DOCS=1 \

CURVE=ALT\_BN128 \

   FEATUREFLAGS="-DBINARY\_OUTPUT=1 -DMONTGOMERY\_OUTPUT=1 \

-DNO\_PT\_COMPRESSION=1" \

   lib install

cd ..

git clone --recursive -b snark https:*//github.com/ethereum/cpp-ethereum.git*

cd cpp-ethereum

./scripts/install\_deps.sh && cmake . -DEVMJIT=0 -DETHASHCL=0 && make eth

cd ..

git clone --recursive -b snarks https:*//github.com/ethereum/solidity.git*

cd solidity

./scripts/install\_deps.sh && cmake . && make soltest

cd ..

./cpp-ethereum/eth/eth --test -d /tmp/test

*# And on a second terminal:*

./solidity/test/soltest -t "\*/snark" -- --ipcpath   /tmp/test/geth.ipc  --show-messages

We also discussed various aspects of integrating zk-SNARKs into the Ethereum blockchain, upon which we now expand.

### Deciding what precompiled contracts to define

Recall that a SNARK is a short proof of some property, and what is needed for adding the privacy features to the Ethereum blockchain are clients that have the ability to verify such a proof.

In all recent constructions, the verification procedure consisted solely of operations on elliptic curves. Specifically, the verifier requires scalar multiplication and addition on an elliptic curve group, and would also require a heavier operation called a bilinear pairing.

As mentioned [here](https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell/), implementing these operations directly in the EVM is too costly. Thus, we would want to implement pre-compiled contracts that perform these operations. Now, the question debated is: what level of generality should these pre-compiled contracts aim for.

The security level of the SNARK corresponds to the parameters of the curve. Roughly, the larger the curve order is, and the larger something called the embedding degree is, and the more secure the SNARK based on this curve is. On the other hand, the larger these quantities are, naturally the more costly the operations on the corresponding curve are. Thus, a contract designer using SNARKs may wish to choose these parameters according to their own desired efficiency/security tradeoff. This tradeoff is one reason for implementing a pre-compiled contract with a high level of generality, where the contract designer can choose from a large family of curves. We indeed began by aiming for a high level of generality, where the description of the curve is given as part of the input to the contract. In such a case, a smart contract would be able to perform addition in any elliptic curve group.

A complication with this approach is assigning gas cost to the operation. You must assess, merely from the description of the curve, and with no access to a specific implementation, how expensive a group operation on that curve would be in the worst case. A somewhat less general approach is to allow all curves from a given family. We noticed that when working with the Barreto-Naehrig (BN) family of curves, one can assess roughly how expensive the pairing operation will be, given the curve parameters, as all such curves support a specific kind of optimal Ate pairing. Here’s a [sketch](https://drive.google.com/file/d/0BwDmGb8qpc8RMEhBMlR5VHE3bEU/view?usp=sharing) of how such a precompile would work and how the gas cost would be computed.

We learned a lot from this debate, but ultimately, decided to “keep it simple” for this proof of concept: we chose to implement contracts for the specific curve currently used by Zcash. We did this by using wrappers of the corresponding functions in the [libsnark](https://github.com/scipr-lab/libsnark) library, which is also used by Zcash.

Note that we could have simply used a wrapper for the entire SNARK verification function currently used by Zcash, as was done in the above mentioned Baby ZoE project. However, the advantage of explicitly defining elliptic curve operations is enabling using a wide variety of SNARK constructions which, again, all have a verifier working by some combination of the three previously mentioned elliptic curve operations.

### Reusing the Zcash setup for new anonymous tokens and other applications

As you may have heard, using SNARKs requires a [complex setup phase](https://z.cash/blog/the-design-of-the-ceremony.html) in which the so-called public parameters of the system are constructed. The fact that these public parameters need to be generated in a secure way every time we want to use a SNARK for a particular circuit significantly, hinders the usability of SNARKs. Simplifying this setup phase is an important goal that we have given thought to, but haven’t had any success in so far.

The good news is that someone desiring to issue a token supporting privacy-preserving transactions can simply reuse the public parameters that have already been securely generated by Zcash. It can be reused because the circuit used to verify privacy-preserving transactions is not inherently tied to one currency or blockchain. Rather, one of its explicit inputs is the root of a Merkle tree that contains all the valid notes of the currency. Thus, this input can be changed according to the currency one wishes to work with. Moreover, if it is easy to start a new anonymous token. You can already accomplish many tasks that do not look like tokens at first glance. For example, suppose we wish to conduct an anonymous election to choose a preferred option amongst two. We can issue an anonymous custom token for the vote, and send one coin to each voting party. Since there is no “mining”, it will not be possible to generate tokens any other way. Now each party sends their coin to one of two addresses according to their vote. The address with a larger final balance corresponds to the election result.

### Other applications

A non-token-based system that is fairly simple to build and allows for “selective disclosure” follows. You can, for example, post an encrypted message in regular intervals, containing your physical location to the blockchain (perhaps with other people’s signatures to prevent spoofing). If you use a different key for each message, you can reveal your location only at a certain time by publishing the key. However, with zk-SNARKs you can additionally prove that you were in a certain area without revealing exactly where you were. Inside the zk-SNARK, you decrypt your location and check that it is inside the area. Because of the zero-knowledge property, everyone can verify that check, but nobody will be able to retrieve your actual location.

### The work ahead

Achieving the mentioned functionalities – creating anonymous tokens and verifying Zcash transactions on the Ethereum blockchain, will require implementing other elements used by Zcash in Solidity.

For the first functionality, we must have an implementation of tasks performed by nodes on the Zcash network such as updating the note commitment tree.

For the second functionality, we need an implementation of the equihash proof of work algorithm used by Zcash in Solidity. Otherwise, transactions can be verified as valid in themselves, but we do not know whether the transaction was actually integrated into the Zcash blockchain.

Fortunately, such an implementation was [written](https://github.com/ConsenSys/Project-Alchemy); however, its efficiency needs to be improved in order to be used in practical applications.

**Acknowledgement**: We thank Sean Bowe for technical assistance. We also thank Sean and Vitalik Buterin for helpful comments, and Ming Chan for editing.

Source: <https://blog.ethereum.org/2017/01/19/update-integrating-zcash-ethereum/>