Contingent Payment on Blockchain

Naiwei Zheng, Tiantian Gong

May 2, 2019

Abstract

Fair exchange problem has been believed to be impossible without incorporating a trusted third party. However, Blockchain-based permissionless cryptocurrencies by nature can fit into this role of a trusted external party in a trustless way. In this project, we explored the digital goods fair exchange problem in the context of blockchain technology and implemented Zero Knowledge Contingent Payment circuits on Ethereum. zk-SNARKs is employed and adapted to generate zero-knowledge proofs for the data content to be transferred. To limit the computation complexity in the circuits, we utilized MiMC scheme for encryption and Merkle Tree root hash generation. The verification of prover's knowledge of the decryption key is conducted off-chain with the circuits we built, while the key is verified on-chain through Hash Time Locked Contract. We verified the completeness of the protocol, and identified the performance issue from a practical point of view.

1 Introduction

The problem of fair exchange where two parties swap digital goods and neither of them can cheat has been studied for decades. In [8], Cleve demonstrated that the so-called coin flipping by telephone problem cannot be solved with high security, which means fairness is unachievable without a trusted third party. Pagnia and Gärtner [11] presented similar results by relating the strong fair exchange problem to consensus and impossibility results from Fischer, Lynch and Paterson. But Blockchain-based cryptocurrencies like Bitcoin bring about new insights, Zero-Knowledge Contingent Payment (ZKCP) to be more specific, into this problem. In permissionless blockchains, consensus is formed in a distributed trustless manner. But the previous results are not violated since the blockchain technology applied in this setting can in fact be viewed as a form of trustless trusted third party, or honest and transparent verifier and executor.

Consider a specific situation where Alice wants to sell a particular digital goods like a data file, and Bob who knows the Merkle Tree root hash of such file would like to purchase the complete file. But they do not trust each other nor any third party, how can they trade in a safe way that no one can cheat? To solve this problem, they can use a blockchain. Cryptocurrencies allow for conditional or contingent payments through the combination of a hash-locked transaction and a protocol to set things up. Alice could encrypt the data file that Bob is interested in with a random key. Then she constructs a program to output the encrypted data, the hash of the decryption key and the result of running the program. This program is then converted into a zero-knowledge proof to convince Bob of the faithful execution of the program. Bob then forms a locked payment on blockchain requiring Alice's public key and the aforementioned random key. In order to redeem the payment, Alice need to reveal the correct key to Bob.

2 Literature Survey

Theoretically, Andrychowicz *et al.* [2] demonstrated how to use Bitcoin to ensure fairness in two-party secure computation protocols. The main idea is to design a forced financial transfer from one party to the other under certain conditions. For the scheme to work, an improvement of Bitcoin specification was needed. In [3], they further investigated into the malleability of Bitcoin transactions and proposed a malleability-resilient refund transaction that requires no modification to Bitcoin protocol specification. Tramer *et al.* [12] explored Sealed-Glass Proof (SGP) that models computation in trusted hardware systems with unbounded leakage. One of their key observation is that SGPs permit safe verifiable computing in zero-knowledge, as data leakage results only in the prover learning her own secrets.

With respect to implementations, Banasik *et al.* [4] constructed efficient Zero-Knowledge Contingent Payment protocol for a large class of NP-relations which can be used in selling factorization of RSA modulus. Campanelli *et al.* [7] introduced the notion Zero Knowledge Contingent Service Payment (ZKCSP) and implemented their protocols for selling a sudoku solution and proofs of retrievability, where the client pays the server by providing a proof that his data has been correctly stored by the server. They also constructed a SHA256 circuit library with 22,272 AND gates. Delgado-Segura *et al.* [10] proposed a fair exchange protocol with three subprotocols also based on Bitcoin scripts. Their approach does not achieve strong fairness, but the advantage one party can have by deviating from the protocol is bounded. They later [9] used private key locked transactions, which allows the atomic verification of a given private key during Bitcoin script execution.

3 Design Decisions

For the proofs of correct execution of a certain program, Succinct Non-Interactive Arguments of Knowledge (zk-SNARK) protocol, specifically the Pinocchio protocol, is utilized and adapted.

Our program consists of two parts, the low-level part is written in C++ that builds on top of the `libsnark` library to construct circuit gadgets, generate R1CS constraints, transform to QAP, generate proving key and verification key, generate proof for given witness, and verify proof for given primary inputs. We compile it into a dynamic library and expose C functions. Then we write our high-level part in Python, where it loads the low-level part and use C Foreign Function Interface (CFFI) feature to invoke and communicate with the low-level part. The high-level part will implement all the cryptographic modules that generate witness from user inputs. It will also provide rich user interface as a command line application.

During the design phase of the project, we were faced with the problem of choosing a proper encryption algorithm to encrypt the data to be traded in the zk-SNARK circuit. Although AES is the most commonly used encryption algorithm, we found it difficult and expensive to be built into zk-SNARK circuit since AES requires large amounts of table look-up operations while zk-SNARK better supports linear combinations. Converting table look-up to linear combinations can be very expensive and can result in bloated circuit size. Therefore we instead turn to encryption schemes friendlier to linear operations.

Recommended by Prof. Aniket Kate, we decided to apply MiMC, an encryption algorithm with minimum multiplicative complexity [1]. It uses linear combinations on a finite field as the basic block cipher. By applying Miyaguchi-Preneel construction it can also be used as a hash function. We use this hash function in the Merkle Tree construction. For faster calculation of the cipher, we choose G_1's prime order as the modulus for MiMC cipher to avoid modulo operations.

zk-SNARK operates on elliptic curves on a finite field. The choice of the field and the pairing generators affects the range of our variables practical in the circuit. For this reason, we choose the curve `alt_bn128`, which has about 254 bits prime G_1 subgroup order, which would also be the range for all our variables in the circuit.

The proof and verification processes take place off-the-chain, and the on-chain atomic payment is only hash time locked with the decryption key as the preimage. We choose not to perform zk-SNARK verification on chain for compatibility with less powerful blockchain scripting languages like Bitcoin script, where zk-SNARK verification is not currently supported. Since most of them support hash locked transactions with common hash functions, we used SHA265 in our zk-SNARK circuit to commit the encryption key to facilitate on-chain key verification.

However, zk-SNARK has a parameter generation phase, where some random secret values, also known as the "toxic waste" need to be chosen to generate the common reference string (CRS). If the prover knows the toxic waste, he/she can forge a proof without knowing the correct knowledge meant to be proved, and the verifier will accept such fake proof. On the other hand, by the result of Campanelli *et al.* [7], if the toxic waste was chosen by the verifier, he/she can craft it in a way such that the proof generated with such parameters can leak information about the knowledge to be proved, hence violating the zero-knowledge requirement. Therefore, it's vital that we need a solution to let Alice and Bob perform this setup phase in a trustless setting.

The most prominent solution is using Multi-Party Computation (MPC), as practiced by Z.cash to generate their zk-SNARK parameters [5]. However, their circuit is generally fixed and only upgrade when major changes are made. They also want to generate the parameters in such a way that everyone can arguably trust it and use it. This is done by the fact that if anyone participated the MPC ceremony had successfully destroyed the intermediate values, digitally and physically, the toxic waste would never be able to be reconstructed. In another words, unless all the participants colluded, the toxic waste would be considered safe and unknown. This is under the assumption and requirement that everyone must trust at least one participant of the ceremony.

Our use case is somewhat different than Z.cash. First of all, it's impractical to fix our circuit as the file size to be proven varies. We could however, generate several different sized circuits to accommodate our use case, but it's not hard to notice that unlike Z.cash, we only have two entities involved in each trading, and both of them have conflicting interests. It's then reasonable to run MPC between Alice and Bob since they won't collude with each other anyway.

However for the purpose of this project, our interest is not on the MPC part. Thus, in this project we assume the CRS has been generated securely and shared between

Alice and Bob, without actually implementing the MPC protocol.

It's also worth mentioning that the fact that Bob knows the Merkel Tree root hash of the file is also under the assumption that the source of such information is trusted by Bob. But we should consider such trust outside of the trading protocol itself as a specialization of a more general setting. Under such setting, we express an arbitrate verification function $V(x) \to \{0, 1\}$ as a function describing Bob's expected utility. It could indicate the extent to which his needs have been satisfied. It's not practical to compose such functions for all human needs. Therefore we only consider something more specific and programmable, in our case a hash value.

Although we are considering the situation where Bob needs some information and Alice would like to trade this information for something she needs, there could be more general situations like the problem of patent transfer and sublease. Sometimes when Alice have sold patent of some data like a demo of guitar solo, we might want to ensure that Alice actually transferred the patent and the data has been completely removed from any devices she has access to. But this might not be possible without incorporating other forms of third party. The other problem concerns Bob's actions after buying the digital goods. He could sell the goods he purchased to other people. If all buyers have incentives to do so, since digital goods rarely bear wears from friction like virtual goods do, the original seller might be discouraged to participate in such transactions. That's because her expected return from selling the goods is greatly diminished. In extreme conditions, suppose the original price for the data file is x^*. After selling it to Bob, Alice has to adjust the price to compete with Bob. If more buyers are to be considered, the situation for Alice can be worse. This is embedded in the fact that digital goods can be replicated easier than virtual goods.

4 Protocol Details

Assume Alice and Bob had agreed on a MiMC Merkel Tree root hash $h_m = \texttt{MiMC\_Merkle\_Tree\_Root}(m)$ and the size $|m|$ of the data m that Alice would sell to Bob. They also agreed on an amount of cryptocurrency x that Bob will pay Alice for the data. Additionally, they have performed MPC to agree on the same CRS values for the zk-SNARK circuit dynamically generated for the target data size. They both know the proving key pk and the verification key vk. Then the protocol proceed as follows:

1. Alice generates a random key $k \in \mathbb{Z}_p$ where p is the order for our zk-SNARK's G_1 subgroup.
2. Alice generates the SHA256 hash for the key $h_k = \texttt{SHA256}(k)$.
3. Alice encrypts the data with padding $c = E_k(\texttt{Padding}(m))$.
4. Alice generates a proof for the zk-SNARK circuit $pr = Proof_{pk}(Circuit(h_k, c, h_m, k, m) = 1)$.
5. Alice sends h_k, c, pr to Bob.
6. Bob verifies the proof $Verify_{vk}(pr, h_k, c, h_m) = 1$. Terminates protocol if verification fails.
7. Bob publishes Hash Time Locked Contract with x amount of deposit, only Alice can redeem if she reveals the preimage of h_k within certain time period defined by block height.
8. Alice reveals k to the contract with the time period, the contract verifies that $h_k = \texttt{SHA256}(k)$ and sends deposit to Alice's account.
9. Bob retrieve k from the contract and decrypts the data $m = \texttt{Padding}^{-1}(E_k^{-1}(c))$.

5 Results

We have implemented all the circuit modules in Python as a reference for our actual zk-SNARK circuit, and for generating witness. We used it as a chance to measure the performance of these constructs.

The benchmark were performed on an 8 core 2.9 GHz Intel Core i7 machine. Our implementation utilized parallel computation on all the CPU cores. The MiMC encryption and Merkle Tree both achieved about 300 KB/s throughput, but the MiMC decryption can only achieve around 6 KB/s throughput. It is however the expected result, since the decryption performs modular exponentiation with over 252 bits exponent, comparing to only 2 bits exponent ($e = 3$) for encryption and hashing. The most common algorithm used for calculating modular exponentiation has time complexity of $\mathcal{O}(\log e)$ where e is the exponent. Thus, the decryption is expected to be hundreds of magnitude slower than the encryption. It also explains why we chose to prove the correct execution of encryption rather than decryption in our zk-SNARK circuit. If we were proving the decryption, it would cost us hundreds time more constraints to do so, and so does the time it would take for us to generate the proof.

As we can see the MiMC cipher is very slow even on the raw implementation. We would expect even slower run time on the actual zk-SNARK circuit. For this reason, we only generated small test cases of 1KB plaintext data to test our zk-SNARK circuit. The generated circuit for 1KB plaintext has 63820 R1CS constraints. It took 15 seconds to generate the proving key and verification key, where the proving key has 30MB in size. The proof generation took 10 seconds, and verification is instant. We tried to test with 1MB plaintext data but the program hang and crashed. It's because the circuit size and proving key size increase linearly with the data size, and for 1MB data we expect to have about 30GB proving key size, thus the machine quickly ran out of memory.

This performance inefficiency is a crucial obstacle for applying this protocol in practice. While nowadays typical data exchange can range from several megabytes to hundreds of terabytes, this protocol is nowhere near to satisfy the actual application usage.

However, with all the generated test cases, the circuit functioned correctly. Its verification process passed on correct proof and failed on incorrect proof. Therefore, it verifies the completeness of our implementation.

6 Future Research

Due to limited time frame, we only explored the use of zk-SNARK to achieve ZKCP on blockchain. It would be interesting to explore other methods as well. One possibility is to use verifiable decryption, proposed by Camenisch and Shoup [6]. Such that the encryptor can prove to a verifier, if given the correct private key, a ciphertext can be decrypted into some plaintext satisfying certain discrete logarithm relationship. Another ongoing research direction is to utilize the Homomorphic Hiding properties of elliptic curve pairings to construct a similar proof, this paper is expected to publish in late 2019. Finally, by delegating most of the transactions to a Layer 2 micropayment channel, like the Lightning Network, we can utilize the homomorphic property of discrete logarithm itself to construct atomic key revealing and payment mechanism. However, such mechanism will require key length to be the same as data length. There's also no public result of this scheme yet.

7 Conclusion

In this project, we explored the possibility of using zk-SNARK and Hash Time Locked Contract to achieve ZKCP on blockchain. Although it's theoretically possible and we have successfully implemented such solution, it turns out to be impractical due to its expensive computation in the proof generation and decryption process on larger files.

References

[1] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 191–219. Springer, 2016.

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Fair two-party computations via bitcoin deposits. In *International Conference on Financial Cryptography and Data Security*, pages 105–121. Springer, 2014.

[3] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. On the malleability of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pages 1–18. Springer, 2015.

[4] W. Banasik, S. Dziembowski, and D. Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *European Symposium on Research in Computer Security*, pages 261–280. Springer, 2016.

[5] S. Bowe, A. Gabizon, and M. D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *International Conference on Financial Cryptography and Data Security*, pages 64–77. Springer, 2018.

[6] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *Annual International Cryptology Conference*, pages 126–144. Springer, 2003.

[7] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 229–243. ACM, 2017.

[8] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369. ACM, 1986.

[9] S. Delgado-Segura, C. Pérez-Solà, J. Herrera-Joancomartí, and G. Navarro-Arribas. Bitcoin private key locked transactions. *Information Processing Letters*, 140:37–41, 2018.

[10] S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí. A fair protocol for data trading based on bitcoin transactions. *Future Generation Computer Systems*, 2017.

[11] H. Pagnia and F. C. Gärtner. On the impossibility of fair exchange without a trusted third party. Technical report, Technical Report TUD-BS-1999-02, Darmstadt University of Technology , 1999.

[12] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 19–34. IEEE, 2017.

A Source Code

Our source code for the project is available at github.com/rixtox/zk-SNARK-contingent-payment.

Instructions for compiling and running the program is written in the README file.

B Work Distribution

Naiwei Zheng

- Converts MiMC block cipher to stream cipher with padding in Python.
- Converts MiMC two-to-one hash function to arbitrary length Merkle Tree construct in Python.
- Implements top level zk-SNARK circuit in C++ for high-level circuit logic that glues the low-level gadgets together.
- Implements field element to bit vector conversion zk-SNARK gadget in C++ for SHA256 digest verification.
- Implements MiMC encryption zk-SNARK gadget in C++.

Tiantian Gong

- Debugs and fixes MiMC stream cipher in Python.
- Implements MiMC Merkle Tree root hash calculation zk-SNARK gadget in C++.
- Implements hash locked smart contract in Solidity for atomic key revealing and payment on Ethereum.
- Writes tests in JavaScript for the smart contract.
- Deploys smart contract on Rinkeby Ethereum testnet.