CredaCash™ Transaction Protocol and "Zero Knowledge Proofs"

Creda Software, Inc.

CredaCash™ uses "zero knowledge proofs" to allow a spender to prove the validity of a transaction while keeping private the sources of funds, the destination addresses, and the transaction amounts. The zero knowledge proof allows the network and any person using the network to verify the validity of a transaction without accessing the private data.

Terminology

Payments are made in the form of "billets". Each billet consists of a value and a serial number. Every transaction output creates a billet, and a billet is then spent by using it as the input to a subsequent transaction. The sender of a billet is called the "Payor", and the recipient, the "Payee".

Transaction Protocol

The transaction protocol operates as follows:

1. The Payee generates a 256 bit random or pseudo-random spend_secret.

2. The Payee selects a value for the billet property enforce_spendspec_hash. This property must be either zero or one. If it equals one, then "spendspec_hashed", a property of the transaction used to spend the billet, must match the value chosen in the next step. If it equals zero, spendspec_hashed in the spend transaction will be ignored. Zero is a safe choice because it will have no effect on the ability to spend the billet. If one is chosen, see the next step for the conditions that must be satisfied in order to spend the billet.

3. If the value of enforce_spendspec_hash equals one, the Payee must select a set of public signing keys and a script that will be evaluated to determine the validity of the

signatures included in the transaction used to spend this billet. These values, formatted into a single string, are called the spendspec. The Payee must then compute

spendspec_hashed = stdhash256(spendspec)

In the current version of the CredaCash system, this feature is not implemented, and if enforce_spendspec_hash equals one, then spendspec_hashed must always be set to zero in order for the billet to be spent. When this feature is eventually implemented, the system will not allow a spendspec to be used that hashes to zero, since the value of zero is reserved to represent "no spendspec". The specific algorithms that will be used for the signatures, the script and the hash will be determined in the future when this feature is implemented.

4. The Payee computes

destination = zkhash(enforce_spendspec_hash, enforce_spendspec_hash * spendspec_hashed, zkhash(spend_secret))

where zkhash is a hash algorithm designed to have all the properties of a cryptographic hash (one-way, collision-free, pseudo-random) while capable of being efficiently verified in a zero knowledge proof. The details of the zkhash algorithm are provided below. Note that for the highest levels of security, the spend_secret can be generated on an air-gapped host and only zkhash(spend_secret) transferred to the wallet, so no computer on the network will be able to spend the billet. This could also be accomplished in the future, when spendspec's are implemented, by generating public signing keys from a private key stored on an air-gapped host.

5. The Payee chooses a sequence_type parameter with a value of "0", "1" or "9". This is an advisory value, not enforced by the protocol, with the following meaning:

"0" – The Payee requests the Payor to always use payment_number = 0 for all billets sent to this destination.

"1" – The Payee requests the Payor to use payment_number = 0 for the first billet sent to this destination, payment_number = 1 for the second billet sent to this destination, payment_number = 2 for the third billet, etc.

"9" – The Payee requests the Payor generate a random 128 bit payment_number each time a billet is sent to this destination.

Sequence_type's "0" and "1" are suitable for single-use destinations that are only provided to one Payor, while sequence_type "9" is suitable for destinations that will be provided to multiple unrelated persons, for example, a donation or payment destination that will be posted on the public internet.

6. The Payee encodes the destination and sequence_type values into a "payspec". The payspec encoding is generally a Base-58+ encoding intended to allow easy transmission via email, text message, the internet, etc. It is an encoding, not an encryption, and can be decoded by anyone. The details of this encoding are described in the Transaction API Reference Manual.

7. The Payee sends the payspec to the Payor (the billet sender). For privacy, payspec's containing sequence_type's "0" and "1" should be sent to one person only through a secure (authenticated and encrypted) channel. Only payspec's with sequence_type "9" are suitable for sending to more than one unrelated person, posting to the public internet, or sending through an authenticated but non-private channel.

8. The Payor decodes the payspec to extract the destination, sequence_type, and, if present, the requested_amount.

9. The Payor chooses the amount of the payment.

10. A transaction must have one or more input billets with sufficient value that the sum of the input billet values = output billet values + the witness donation.

11. The Payor locates one or more billets that it can spend with sufficient value to cover the outputs.

12. The Payor queries the system to obtain the root hash of the Merkle tree containing all commitments contained in all indelible blocks in the blockchain. "Recent" means a Merkle root value from the last 48 hours, although using the most recent value possible is recommended in order to allow time for the transaction to clear. See below for more details on the composition of the Merkle tree.

13. Along with the root hash, the Payor obtains the level of the last block placed into the blockchain when the Merkle root was computed, and the system parameters:

output_value_minimum
output_value_maximum
input_value_maximum

The system parameters and the Merkle root are all associated with the blockchain level, i.e., for a given blockchain level, there is only one Merkle root and set of minimum and maximum values. This allows the Payor to send only the blockchain level with the transaction, and from the blockchain level, the system will determine the Merkle root and system parameters that are required for the transaction.

14. Input billets may be specified either by publishing the billet's commitment in the transaction or by providing a Merkle path to the commitment as a hidden input to the zero knowledge proof. Publishing a commitment in the transaction compromises privacy because it identifies the billet used, linking the new transaction to the prior transaction. It can be used however when the wallet does not wish to look up the billet's Merkle path and is not concerned about the loss of privacy, or when the input billet comes from a very recent transaction that has not yet been incorporated into an indelible block. In the future, it may also potentially be used to spend a billet that has an expired commitment and is no longer in the Merkle tree.

15. For each input billet for which the commitment will not be published in the transaction, the Payor looks up the billet's commitment number (its location in the Merkle tree) and the hash inputs along the path from the commitment to the Merkle root. There is only one Merkle root input for the entire transaction, so the Merkle paths for all inputs billets must be computed for the same tree state and lead to that single root value.

16. For each input billet, the Payor computes

serial_number = zkhash(commitment, spend_secret)

17. A transaction may contain multiple outputs, each sent to the same or different destinations which might belong to the same or different persons. One of the output billets would commonly be used by the Payor to return "change" back to itself.

18. If the amount of the payment exceeds output_value_maximum, the Payor divides the amount into multiple output billets, each of which is greater than or equal output_value_min and to less than or equal to output_value_maximum.

19. If the sequence_type in the payspec sent by the Payee is "0", the Payor selects payment_number = 0 for all billets sent to this destination. If the sequence_type is "1", the Payor selects payment_number = 0 for the first billet sent to this destination, payment_number = 1 for the second billet sent to this destination, etc. If the sequence_type is "9", the Payor selects a random 128 bit payment_number for each billet sent to this address.

20. For each billet to be sent, the Payor computes

address = zkhash(destination, payment_number)

21. For each billet to be sent, the Payor selects a 64-bit value for the billet and computes

value_enc = value ^ zkhash64(destination, payment_number)

where ^ is the bitwise exclusive-or operator.

22. Using the Merkle root retrieved above, the Payor sets

commitment_iv = lower_128_bits(merkle_root)

23. For each output billet in the transaction, the Payor computes:

commitment = zkhash(destination, payment_number, value, commitment_iv)

24. The Payor selects an amount for a donation to the witness who incorporates this transaction into a block. The formula to compute the suggested minimum donation is given in the Wallet Developer's Guide. If the donation is less than the suggested minimum, the transaction may not be forwarded by one or more relays, and it may or may not be incorporated by a witness into a block. On the test network, a transaction may also have a negative donation, allowing test billets to be created using a transaction with no inputs, one or more outputs, and a negative donation that offsets the output values.

25. If necessary, the Payor adjusts the amount of "change" to itself in the transaction in order to satisfy the equation

sum(input billet values) = sum(output billet values) + witness donation

26. The Payor selects one of the system's zero knowledge proof keys that is large enough to accommodate the number of transaction inputs and outputs. Generally, the Payor would select the smallest possible key, in order to minimize the memory and CPU time needed to compute the zero knowledge proof, but any larger key can also be used.

27. The Payor constructs a zero knowledge proof as follows:

- Public inputs for the entire transaction:

Merkle root
output_value_minimum
output_value_maximum
input_value_maximum
witness_donation

- Public inputs for each input billet:

serial_number
spendspec_hash, which eventually will be computed from the spendspec that may be included with an input billet, but for now is always zero
if the billet's commitment is published in the transaction:
the commitment

- Hidden inputs for each input billet:

spend_secret
enforce_spendspec
payment_number
value
commitment_iv
if the billet's commitment is not published in the transaction:
the commitment
the commitment number (location in the Merkle tree)
the hash inputs from the commitment to the Merkle root

- Public inputs for each output billet:

address
value_enc
the commitment

- Hidden inputs for each output billet:

destination
payment_number
value

The zero knowledge proof enforces the following constraints:

- For the transaction as a whole:

all public input values used to create the proof are the same as the public input values used to verify the proof
sum(input billet values) = sum(output billet values) + witness_donation

- For each input billet:

value <= input_value_maximum
serial_number = zkhash(commitment, spend_secret)
where:
commitment = zkhash(destination, payment_number , value, commitment_iv)
and destination = zkhash(enforce_spendspec_hash, enforce_spendspec_hash * spendspec_hashed, zkhash(spend_secret))
if the commitment is not a public input, then the commitment and its Merkle path must be included in the hidden inputs and must hash to the Merkle root

- For each output billet:

value >= output_value_minimum
value <= output_value_maximum
address = zkhash(destination, payment_number)
value_enc = value ^ zkhash64(destination, payment_number)
commitment = zkhash(destination, payment_number, value, commitment_iv)
where commitment_iv = lower_128_bits(merkle_root)

28. The Payor constructs a transaction in which the following information is publicly published:

The blockchain level of the Merkle root
The zero knowledge proof key id
The zero knowledge proof
witness_donation

- For each input billet:

serial_number
if the billet's Merkle path was not input into the zero knowledge proof:
the commitment

- For each output billet:

address
value_enc
the commitment

29. The Payor possibly adds a “proof-of-work” to the transaction, which consists of:

a 64-bit timestamp
eight 40-bit nonces

The eight nonces are used to compute eight hashes, all of which must be less than the proof-of-work threshold. Computing eight hashes reduces the variance in the computation time by 65%. The hashes are computed by:

hash[i] = siphash24_mod(key, txhash)
where the 128-bit key = (((i << 40) | nonce) << 64) | timestamp
txhash = first 128 bits of blake2b(transaction payload)
and siphash24_mod is the standard siphash24 modified to not append a size byte to the end of the hash input

30. The Payor either submits the transaction to the network, or sends it to the Payee for submission to the network.

31. If the Payor used a random payment_number, then the Payor must send this payment_number to the Payee. If the Payor used a constant or sequential payment_number as specified by the Payee in the payspec, then sending it to the Payee is optional. The payment advice may be sent using a wallet-to-wallet protocol that has not yet been determined but will likely involve a Tor hidden service provided by the Payee’s wallet and encoding the service’s onion address in the payspec provided to the Payor. Along with the payment_number, the Payor may optionally send other key transaction information such as the amount, the commitment_iv and the commitment.

32. If the Payor did not add a proof-of-work to the transaction, the Payee or the receiving node (if it is willing) adds a proof-of-work.

33. Every node in the network (relay and witness) checks the following:

The proof-of-work satisfies the minimum requirement.
The blockchain level of the Merkle root is a valid value from the last 48 hours.
The zero knowledge proof key id specified in the transaction is valid and has sufficient capacity for the number of inputs and outputs in the transaction.
The zero knowledge proof verifies.
All input commitments published in the transaction (if any) are valid outputs of prior transactions.
No serial number has been used as a prior input of this transaction or an earlier transaction in the same block or any predecessor block.

34. If the transaction fails any of these tests, it is discarded. If it passes, a witness strips out the proof-of-work and adds the transaction to a block.

35. If the Payor wishes to check for the transaction to clear, it checks any one of the serial numbers published in the transaction to see if it has been added to the permanent list of spent serial numbers, which occurs when the transaction appears in an indelible block in the blockchain.

36. If the Payee wishes to check if a payment has been made and it specified a sequence_type of "0" or "1" in the payspec, it may compute the assumed payment_address and watch for transactions to the resulting address. When a new payment is detected, it may retrieve the value_enc and then compute value using the formula

value = value_enc ^ zkhash64(destination, payment_number)

Privacy note: Anyone who knows the destination and payment_number can similarly monitor the blockchain for a transaction and decode the value. For that reason, the Payee should always specify a sequence_type of "9" when sending a payspec to more than one unrelated person, in which case the Payor should always use a random payment_number.

Usage note: Wallets should consider generating billet spend_secrets from a master_secret, using an equation such as

spend_secret[i] = hash256(master_secret || i)

For best security, the master_secret should be randomly generated or generated from a user passphrase using a strong key derivation function such as PBKDF2 with a cryptographically random 256 bit salt that is stored in a secure location. By generating billet secrets in the predictable manner, the wallet can use the master_secret and the information in the blockchain to recover all billets that use zero or sequential values for the payment_number. For payments that use a random payment_number, the wallet should consider immediately rolling the billet over into a billet with a sequential payment_number so it also will be recoverable.

Design Rationale and Security Analysis

Because the commitment is published in the blockchain and included in the Merkle tree, no one can spend a non-existent output or change the commitment, the output value or the spend_secret without finding a hash collision. Because the spend_secret is included in the zero knowledge proof, no one can spend the output unless they can either find a hash collision, reverse the hash function to find the spend_secret, or find a "collision" in the 288 byte zero knowledge proof. Because the serial_number is published in the transaction and checked for a prior spend, no one can spend an output twice without finding a hash collision. In short, since it is very unlikely if not impossible to find a collision or reverse the hash function, the zero knowledge proof ensures the integrity of the system while keeping the transaction information private.

Along with verifying the zero knowledge proof, every node on the network also verifies that the serial_number published in the transaction has not been already been spent in an earlier transaction in the blockchain. This prevents a payment from being spent twice. The serial number cannot however be publicly connected to the commitment published

when the payment is made because the two are not published together in the same transaction (as long as the commitment's Merkle path is provided as a hidden input), and the only information that ties them together, the value and spend_secret, are kept private by the zero knowledge proof and never published.

It would have been possible to not publish address or value_enc in the blockchain, and instead simply require the Payor to notify the Payee of the commitments for all payments. It was desired however that the wallet should be able to detect and gather sufficient information to spend incoming payments only by monitoring the blockchain, and for that reason, an address known to the Payee and the output value (required to spend the payment) in encrypted form were added to the blockchain. This formulation also allows wallets to recover their contents from a single master_secret if all of the addresses are generated from a master_secret in a predicable way. This allows the blockchain to serve as a continuous backup of every wallet to help prevent the loss of billets.

It would have been possible to set simply address = zkhash(spend_secret). However, this formulation would not have provided privacy for public addresses (for example, addresses posted on public internet sites) that were intended for purposes such as receiving donations. In that case, any person would be able to monitor the blockchain and detect payments to the public address, compromising privacy. For this reason, the Payor is asked to add a large random nonce to the address computation for payments to public destinations.

In the public payment scenario above, when the Payor selects a random payment_number, it is required that the Payor notify the Payee of at least the payment_number so that the Payee will know that a payment has been made and will have sufficient information to spend it. It would have been possible instead for the Payee to provide a persistent ECDH (elliptic curve Diffie-Hellman) key in the payspec, for the Payor to add a session ECDH key to the transaction, and for the transaction

address and value to be encrypted using the shared ECDH secret. However, this formulation would have required the Payee to monitor and attempt to decrypt every transaction in the blockchain to see which payments were sent to it, and was therefore deemed impractical for most Payees. If the Payor is willing to make a payment, it seems reasonable that the Payor would also be willing to notify the Payee of the payment, and for that reason, this latter formulation was adopted.

In certain cases when making a payment to a private destination, the Payor may be required to make more than one payment. This might arise for example when the amount to be paid is too large for a single billet, for subscriptions, for installment sales, for deposit plus final payment sale (for example, half up front, half on delivery), to correct payment errors, etc. For these situations, the payment_number formulation above was adopted to provide separate addresses for each payment in order to prevent the payments from being linked. Since the Payor generates sequential payment_numbers, the Payee is able to predict the payment_numbers and compute the addressees so it can monitor the blockchain for incoming payments, or if necessary, recover the payments using its master_secret.

As previously discussed, anyone who knows the destination and payment_number is able to find payments for that destination in the blockchain and determine their values, compromising privacy. For this reason, the Payor should either limit distribution of a payspec (which contains the destination), or generate the paypec with sequence_type = "9" to request the Payor use a random payment_number.

One potential attack is to attempt to create two billets with different values but the same commitment, then enter the lower value billet into the blockchain and spend the higher value billet. This would require finding a hash collision, i.e., two sets of inputs that hash to the same commitment. While that itself is very unlikely, the commitment_iv, which comes from the value of a recent Merkle root, was added to the commitment's hash inputs to limit an attacker's ability to exploit a collision.

The "zkhash" Algorithm

The zkhash is computed as follows:

1. First, all of the inputs are concatenated into one long string of n bits, which will be called b1, …, bn

2. From the inputs bits, two pseudo-independent knapsack values are computed:

```
kx = b1*x1 + b2*x2 + … + bn*xn
ky = b1*y1 + b2*y2 + … + bn*yn
```

where each of the coefficients xi and yi are 254-bit values in the prime field P, and the sum is computed modulo the prime P.

3. A Diophantine polynomial of degree 256 is computed in the prime field from the pseudo-independent knapsack values as follows:

```
s = kx + ky
for (i = 1; i <= 8; ++i)
{
        kx = kx*kx + kx + 1
        ky = ky*ky - ky + 1
}
s = s + kx + ky
```

The multiplications and additions above are performed modulo the prime P.

4. The sum s above is broken into the number of bits desired for the final output, i.e., if h bits are desired in the final output, s is broken down into the bits s1, …, sh with the remainder (the higher order bits) discarded. Generally, all of the bits of the prime field are desired in the final output, but in certain situations (such as the computation of value_enc), only a smaller number of bits such as 64 are needed.

5. A third knapsack value is computed from the bits of s, as follows:

```
kz = s1*z1 + s2*z2 + … + sh*zh
```

6. The zkhash output is the value of the final knapsnack, kz. In situations where fewer bits than the full field are desired, that output is broken down into bits again and the remainder discarded.

7. CredaCash has precomputed 2304 coefficients for use in the knapsacks: 1024 values for kx, 1024 values for ky, and 256 values for kz. The coefficients are computed from

ci = sha256("basis i.j")

where j is the smallest integer that gives a ci less than the prime P.

8. Each of the zkhash values computed in the zero knowledge proof uses a different offset into the kx and ky coefficient tables. This is equivalent to using a different initialization vector or a different nonce input in a standard hash, and ensures the hash outputs are independent.

Rationale and Security Analysis

The zkhash uses only multiplications and additions performed in the prime field P so it can be efficiently verified using a zero knowledge proof.

The first two knapsacks are each used as a compression/expansion function and a pre-pseudo-randomizer. It compresses or expands the input bits to 254 bits, and spreads the input entropy uniformly through those 254 bits. According to [BCTV14b] and the references cited therein, a single knapsnack (which recent versions of the paper call a "subset sum" function) in a prime field is cryptographically secure. However, it has a relatively simple structure. If an attacker is attempting to manipulate one 254 bit input to generate a collision, there are only 254 input vectors in the sum, and adding or removing a vector requires only a single addition or subtraction. The accumulator will only alias or wrap around the modulus a maximum of 254 times, so it might be possible to extend the accumulator by 8 bits, do a high-speed parallel comparison against all aliases, and then combine these into some sort of differential or distance based-analysis.

Due to the simplicity, it would seem imprudent to rely only on a single knapsack. It does however do a very good job of spreading the input entropy out across the accumulator, which greatly enhances the cascade effect of any subsequent stages. For that reason, it makes a very good pre-pseudo-randomizer.

The second stage is a Diophantine polynomial of degree 256 with two semi-independent inputs. Diophantine equations have been extensively studied for many years, and arbitrary high-degree bivariate Diophantine equations are considered to be unsolvable. A Diophantine in the prime field P should be even more difficult to solve. Each time the 254 bit input is squared, it potentially aliases around the prime 2^254 times. This makes enumerating the aliases at least as difficult as enumerating the output values. The coefficients and degree of the Diophantine were derived from simulations using smaller 8 to 16 bit inputs in a scaled prime field (P ~= 0.756 * 2^nbits), and those parameters were found by themselves, without any input conditioning, to produce a pseudo-random output close in quality to a good pseudo-random function.

The last knapsack ensures the output appears completely random, and makes it that much more difficult to recover the inputs from the output. It may not be necessary but the cost is manageable and it seems prudent to include.

The Commitment Merkle Tree

Nodes in the network are required to maintain a Merkle tree of all commitments which is used to prove a commitment exists without revealing the specific commitment. In the CredaCash block assembly process, once a block is followed by a sufficient number of additional blocks, it becomes “indelible” and will never be replaced by another block. When a block becomes indelible, the commitments in the block are then placed into the Merkle tree. That choice was made so the nodes do not have to maintain an “undo” data structure to correct the Merkle tree when an alternate block is chosen for the chain.

The Merkle tree is binary with a height of 48 and can hold up to 2^48 commitments. That is sufficient capacity to hold 890,000 commitments generated continuously every second for 10 years. To keep the size of the Merkle tree and the spent serial number lists from growing without bound, commitments will expire in 5 to 10 years (the exact time to be determined) and removed from the Merkle tree. Before they expire, they should be rolled over into new commitments so they can still be spent. An additional transaction type is contemplated to spend expired commitments, but it would come with a considerably higher processing cost.

Commitments are entered into the Merkle tree from left to right in the same order they appear in the blockchain. Commitments are assigned consecutive commitment numbers, with the first commitment placed in the tree assigned commitment_number = 0. At the tree leaves, the commitments are first hashed with the commitment number:

leaf_hash = zkhash(commitment, commitment_number)

This makes it difficult for an attacker to predict the leaf hash input, which makes potential “chosen hash” attacks more difficult.

In the Merkle tree, a slightly modified version of zkhash is used as a compression function. It takes two 254 bit inputs and computes a single 254 bit output. When computing the inner hashes, instead of forming one long bit string from the two inputs, the bits from each input are instead multiplied by the same coefficients and then added together, i.e., if the two inputs are a and b, the initial knapsacks are computed by:

kx' = kx(a) + ky(b) = (a1*x1 + … + an*xn) + (b1*x1 + … + bn*xn)
ky' = ky(a) + ky(b) = (a1*y1 + … + an*yn) + (b1*y1 + … + bn*yn)

and then the Diophantine is computed using kx' and ky'. This makes the resulting hash independent of the order of the inputs a and b which simplifies the constraint system needed in the zero knowledge proof to verify a Merkle path.

The tree structure is pruned, meaning no hash value is computed for positions in the tree for which no commitments exist in both the left and right input paths. When a hash is computed that has commitments in the path of only one of its two inputs, a null_input is used on the side with no commitments. The null_input is changed each time commitments are added to the tree by setting it equal to the lower 256 bits of the blake2b hash of the block that contains the commitments being added, modulo the prime P. This adds an element of randomization to the Merkle root and makes attacks that might seek to create a collision in the Merkle root more difficult.

The Zero Knowledge Proof

The zero knowledge proof verifies the prover is able to satisfy a set of arithmetic constraints. The operations that can be used in the constraint system include addition, subtraction, multiplication, division; bitwise and, or, exclusive-or, one's complement; comparisons for equality, less than, greater than, etc. The prover must provide a set of hidden values that, when combined with the public values provided by the verifier, satisfy the constraints determined in the construction of the proof system. In general, this is most useful when the prover is required to provide the inputs to one or more publicly known hash outputs, since this proves knowledge of values (the hash inputs) that cannot be determined from the publicly known values (the hash outputs). The zero knowledge proof verifies the prover provided these hash inputs without revealing the inputs used. The references listed below provide further details.

CredaCash uses a zero knowledge proof construction similar to [BCTV14a] and used in their [libsnark] reference library, with the correction suggested by [P15]. It is similar to the construction chosen for [zerocash]. CredaCash uses modified versions of the [snarklib] and [snarkfront] libraries which were developed directly from [libsnark]. The elliptic curve system is the Barreto-Naehrig pairing that is claimed to provide 128 bits of security. The modifications implemented in CredaCash perform arithmetic operations directly in 254-bit prime field for efficiency, similar to [BCTV14b]. Many

of the constraints are coded directly for efficiency, rather than using an algebraic generator.

The proving and verification keys were generated from a cryptographically-secure random number generator and then all initial and intermediary values were immediately discarded for security. CredaCash provides proving and verification keys that handle up to 16 outputs, 16 inputs with Merkle paths, and 2 inputs with the commitment publicly published. The time to generate a proof on a single core of an Intel Core i5-520M @ 2.4 GHz (a circa 2010 midrange laptop CPU) is 0.20 seconds per output, plus 3.8 seconds per input with a Merkle path, and 0.24 seconds per input with commitment published. The memory requirements are roughly 85 KB per Merkle path input plus 50 KB of overhead. The zero knowledge proofs themselves consist of 288 bytes per transaction, and the verification time is 0.035 seconds per transaction on the same CPU, independent of the number of inputs and outputs.

References

[GGPR13] Quadratic span programs and succinct NIZKs without PCPs, Rosario Gennaro, Craig Gentry, Bryan Parno, Mariana Raykova, http://eprint.iacr.org/2012/215

[BCIOP12] Succinct Non-Interactive Arguments via Linear Interactive Proofs, Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, Omer Paneth, http://eprint.iacr.org/2012/718

[PGHR13] Pinocchio: Nearly Practical Verifiable Computation, Bryan Parno, Craig Gentry, Jon Howell, Mariana Raykova, http://eprint.iacr.org/2013/279

[BCTV14a] Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture, Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, Madars Virza, http://eprint.iacr.org/2013/879

[BCTV14b] Scalable Zero Knowledge via Cycles of Elliptic Curves, Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, Madars Virza, https://eprint.iacr.org/2014/595

[P15] A Note on the Unsoundness of vnTinyRAM's SNARK, Bryan Parno, https://eprint.iacr.org/2015/437

[zerocash] Zerocash: Decentralized Anonymous Payments from Bitcoin, Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, Madars Virza, http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf

[libsnark] https://github.com/scipr-lab/libsnark

[snarklib] https://github.com/jancarlsson/snarklib

[snarkfront] https://github.com/jancarlsson/snarkfront