tnnl: A Privacy Layer for Standards-Compliant Fungible Tokens

Alex Geiger
alex@cryptar.io

Julian Geiger
julian@cryptar.io

Cryptographic Applications Research, LLC

October 28, 2018
v0.9.3

Abstract

The current **Ethereum** token-oriented ecosystem is lacking in privacy. With the 'Byzantium' hard-fork to the **Ethereum** network, it became possible to implement a confidential transaction system using zk-SNARKs; however, with a preexisting token infrastructure, it is a non-trivial amount of work for current tokens that are in want of privacy features to incorporate the technology enabled by the latest hard-fork. We propose a system that provides *all* preexisting ERC20-compliant, ERC223-compliant, ERC677-compliant, and ERC777-compliant tokens a means of operating in a confidential manner.

1 Introduction

tnnl is a privacy layer for the **Ethereum** [1, 2, 3] network that is derived from **Zcash** [4]. The structure of this document is intended to provide some foundational information (in the form of informal definitions, a brief background overview, a high-level description of zk-SNARKs, and a notional summary of **Zcash**) before laying out the primary challenges of transplanting the **Zcash** protocol to the **Ethereum** network and finally, a protocol specification for **tnnl**. Those already with an understanding of **Zcash** and **Ethereum** may skip to § 8.

1.1 Definitions

We first provide a few relevant but informal definitions. More formal definitions, while more informative, are too technical for a high-level protocol overview. Basic familiarity with blockchains, particularly **Bitcoin** [5] and **Ethereum**, is assumed.

Zero-Knowledge Proof [6]

Information provided by one party, the *prover*, that convinces another party, the *verifier*, that the prover is in possession of some secret; the information provided does not impart knowledge of the secret to the verifier. A malicious prover additionally is unable to convince the verifier with a false proof.

zk-SNARK [7, 8, 9, 10]
(Zero-Knowledge 'Succinct Non-interactive ARgument of Knowledge.') A class of zero-knowledge proving protocols suitable for blockchains, where an arbitrary program goes through several transformations. The result of these transformations is a circuit, proving key, and verification key; the proving key and verification key can be used respectively with the circuit to generate and verify concise zero-knowledge proofs. zk-SNARKs are given a more in-depth, albeit non-specific and only mildly-technical, summary in § 2.

Cryptographically-Secure Hash Function [11]
A *hash function* is a one-way function that operates on an arbitrary-length binary string ('message') that produces a fixed-length output ('message digest'). Two properties of a *cryptographically-secure* hash function are: (1) *first preimage resistance*, where the message digest cannot be used to determine the inputted message; and (2) *second preimage resistance*, where no input can produce the same message digest as another input.

Merkle Tree [12]
A form of binary trie where nodes are assigned values of the message digests of their leaves or child-nodes. All nodes and leaves are hashed in this manner until a single root hash is produced. When viewed as a hash function itself, a Merkle tree of this description is **not** cryptographically-secure, as, by definition, it does not meet the second preimage resistance criterion.

Fixed-Depth Merkle Tree [13]
A subset of Merkle tree where all appendations to the tree are at a predetermined depth—that is to say, all leaf-nodes are "at the same level." Only a finite amount of entries can be made to an instance of a fixed-depth Merkle tree. When viewed as a hash function, a fixed-depth Merkle tree *is* cryptographically-secure provided that the hash function used is also cryptographically secure.

1.2 Notation

Let E denote an elliptic curve and $\mathbb{F}_n$ denote a finite field of size n. Let $\vec{v}$ denote a vector and v_1 denote the first entry of $\vec{v}$. Let $\circ$ denote concatenation.

1.3 ZK Digital Currency: From Theory to Reality

While the concept of using zero-knowledge ('zk') proofs in cash-analogue cryptosystems predates even the first successful cryptocurrency, **Bitcoin**, by at least twelve years [14], a practical implementation of such a system was not possible until relatively recently. After a series of many theoretical proposals of systems using various iterations of zero-knowledge proofs had been proposed [15, 16, 17], the flagship practical implementation utilizing zero-knowledge proofs as a means of performing confidential transactions was realized and deployed as **Zcash**[1] on October 28, 2016.

Zcash is, in essence, a fork of **Bitcoin** that has privacy-preserving capabilities. The initial version of **Zcash**, referable as **"Sprout" Zcash** (in order to differentiate from the **"Sapling" Zcash** protocol upgrades), used the BCTV14 proving system [9], a concrete implementation of a zk-SNARK proving protocol.

The **Ethereum** network, having launched on July 30, 2015, is a public-ledger that — as it was developed prior to **Sprout Zcash** — has formed a prevailing 'standard token' lacking the level of privacy enabled via zk-SNARKs. The result is *more than* $13,000,000,000 USD worth of tokens (at the time of publication) where all the token holders are determinable to powerful adversaries.

After **Sprout Zcash** demonstrated that its form of zero-knowledge proofs are a viable means of enabling confidential transactions, **Ethereum** developers worked to add the capability to perform the same class of zero-knowledge proofs to its already-existing network. In the latest hard-fork to the **Ethereum** network known as 'Byzantium,' the capabilities to perform three elliptic curve operations were added in the form of 'precompiled contracts.' These three operations are elliptic curve addition, scalar multiplication, and pairing checks. Together these three operations enable on-chain verifications of the same types of zero-knowledge proofs used by **Sprout Zcash**[2].

2 High-Level Overview of zk-SNARKs

While zk-SNARKs are a very dense topic with a lot of ongoing, technical research, a conceptual understanding of the properties of zk-SNARK proving systems is crucial for the comparison of **tnnl** to **Zcash** in following sections. In this document, we only provide an 'as-surface-as-possible' comprehension of zk-SNARKs, largely because there are a variety of zk-SNARK proving systems, many of which differ in their constructions of their secure parameters.

[1]**Zcash** is going through a major protocol change, so it is important to differentiate between the two versions. In this subsection (§ 1.3), **"Zcash"** refers to the project as a whole; in the remaining sections, **"Zcash"** refers to the initial version of the **Zcash** protocol.

[2]The elliptic curve operations provided in the 'Byzantium' fork do not enable a reasonably secure version of the **Sapling Zcash** protocol to be implemented (which is why **tnnl** is derived from the earlier **Sprout Zcash** protocol), as the **Ethereum** operations currently provided are implemented only for a particular elliptic curve. The curve used in **Sapling Zcash** has what is called an 'embedded curve' (which is an elliptic curve implemented over the order of the actual curve acting as a finite field) of a high prime order, meaning that more sophisticated mechanisms can be used within zero-knowledge proofs. The **Sprout Zcash** curve that the 'Byzantium' fork provided does not have an embedded curve of prime order, meaning that the embedded curve is only as secure as its order's biggest prime factor. The **tnnl** protocol does not rely on an embedded curve.

2.1 zk-SNARK Programs

zk-SNARK proving systems are designed in such a manner that one can start with an arbitrary, high-level computer program that is intended to fulfill a confidential role within the broader context of an encompassing system. Concisely stated, the intended function of a zk-SNARK scheme is to produce a verifiable proof that the program in question was executed properly without necessarily knowing the inputs, outputs, or intermediary variables used in the construction of the proof.

2.1.1 Key components

Most zk-SNARK proving systems provide two *keys.* One is used by the prover to generate a proof and is referred to as the *proving key.* The other key is used by the verifier to verify a proof and is called the *verifying* or *verification key.* In contrast to the public-private key-pairs in asymmetric cryptosystems, both the key components are public. The relation of the keys to each other is to remain unknown to all.

2.1.2 Private/public input differentiation

When some inputs to a zk-SNARK program are intended to be made public, they are referred to as *primary inputs.* The other arguments that are intended to remain private are referred to as *auxiliary inputs.*

2.1.3 Code flattening and constraints

Variables are not 'reused' in a zk-SNARK program. Instead of reassignment of a variable, a new intermediate variable is used in a process called *flattening.* Flattening occurs because, when designing a zk-SNARK program, one must define *constraints* describing the relations between intermediary variables that must be met in order for the execution of the program to be considered valid. The assignments to all the inputs and intermediary variables from an execution of a program are together known as the *witness.*

2.1.4 Common domain-specific library

Much of the leading zk-SNARK research has produced `libsnark`, an open-source C++ library with implementations of many proving systems. Most zk-SNARK-related projects either use `libsnark` directly or consume other libraries that utilize `libsnark`.

2.2 Algorithms

zk-SNARK schemes typically define three algorithms, all of which depend on the program in the form of a circuit (which is elaborated on in § 2.3). The three core algorithms are:

1. `KeyGen`: Where two sets of parameters are generated that are utilized in the remaining two algorithms. The first set of parameters is the *proving key* and is denoted as pk. The second set of parameters the *verification key* or the *verifying key* and is denoted as vk. This key generation process is usually a one-time process, since the generated parameters are reusable. (This algorithm is often referred to as the "set-up phase.")

2. `Compute`: Where the proving key pk is used to generate a *proof* (which is denoted as π in zero-knowledge literature).

3. `Verify`: Where the proof π is checked using the verifying key vk.

2.3 Transformations

To achieve zero-knowledge, the high-level program goes through a series of complex transformations.

- First, the high-level program is transformed into an arithmetic circuit C. While there are tools that allow high-level programs to be compiled into an arithmetic circuit, it is currently the case that "hand-written" arithmetic circuits are usually more optimized to the point where manual transformations of a program (via encapsulation) is standard practice. Constraints for the inevitably-resulting witness are defined alongside the logic that populates the witness.

- The circuit and constraints are then transformed into a set of three matrices corresponding to the variables and constraints in the original program. This form is referred to as a *rank-1 constraint system (R1CS)*.

- The R1CS is then transformed into a *quadratic arithmetic program (QAP)*, denoted as σ, using Lagrange Interpolation on each constraint. σ is a tuple that comprises of three sets of polynomials ($\vec{A}$, $\vec{B}$, and $\vec{C}$).

  This transformation of the R1CS to σ is a bi-directional transformation, and evaluating the polynomials at a point x produces the x^{th} constraint.

- The QAP σ, along with the circuit C, can at this point be used to complete a zk-SNARK protocol specification. zk-SNARK protocols typically leverage a bilinear map[3] $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are cyclic order-r subgroups of $E(\mathbb{F}_p)$ and $E'(\mathbb{F}_{p^{k/d}})$ respectively. Let p denote a large prime number, k denote an embedding degree, and d denote the degree of the twist making E' the d^{th} twist of E [18]. Let g_1 and g_2 denote the generators for $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively; all other elements (with the exception of key components) of $\mathbb{G}_1$ and $\mathbb{G}_2$ are denoted as calligraphic letters such as $\mathcal{P}$ and $\mathcal{Q}$. $\mathbb{G}_T$ is the subgroup of the r^{th} roots of unity in $\mathbb{F}_{p^k}$. The bilinearity of e is expressed by:

[3]The security models of zk-SNARK proving systems typically assume symmetric bilinear maps, but in practice more-efficient asymmetric bilinear maps are used.

$$e(u\mathcal{P}, v\mathcal{Q}) = e(\mathcal{P}, \mathcal{Q})^{uv}$$

Depending on the exact specification of a proving system, pk comprises of points in $\mathbb{G}_1$ and $\mathbb{G}_2$ that are the results of evaluating and transforming $\vec{A}$, $\vec{B}$, and $\vec{C}$ at and by uniformly random secret points $\vec{s}$ before performing a group multiplicative operation with either g_1 or g_2.

vk loosely comprises of points that are computed by taking the product of the secret points $\vec{s}$ and the generator points of isomorphic counterparts to the points computed in pk. The secret points $\vec{s}$ are destroyed, and the only known way that a prover can generate a proof π that satisfies the checks of the `Verify` algorithm is by using a valid witness $\vec{w}$ with the proving key pk.

Example 1. *As part of its `KeyGen` algorithm, one proving system evaluates each polynomial A_i in $\vec{A}$ at a randomly sampled point τ. Another point α is also randomly sampled.*

The following proving key components are generated:

$$\mathsf{pk}_{A_i} := A_i(\tau)g_1 \qquad \mathsf{pk}'_{A_i} := A_i(\tau)g_1\alpha$$

The verification key has a corresponding component:

$$\mathsf{vk}_A := \alpha g_2$$

The prover, using a witness $\vec{w}$ and following the `Compute` algorithm, generates the following proof-components:

$$\pi_A := \langle \vec{w}, \mathsf{pk}_A \rangle \qquad \pi'_A := \langle \vec{w}, \mathsf{pk}'_A \rangle$$

The `Verify` algorithm for this proving system prescribes that we check (amongst other things):

$$e(\pi_A, \mathsf{vk}_A) = e(\pi'_A, g_2)$$

This check can be expanded for clarity:

$$\begin{aligned}
e(\pi_A, \mathsf{vk}_A) &= e(\pi'_A, g_2) \\
e(\langle \vec{w}, \mathsf{pk}_A \rangle, \mathsf{vk}_A) &= e(\langle \vec{w}, \mathsf{pk}'_A \rangle, g_2) \\
e(\langle \vec{w}, A(\tau)g_1 \rangle, \alpha g_2) &= e(\langle \vec{w}, A(\tau)g_1\alpha \rangle, g_2) \\
e(\langle \vec{w}, A(\tau)g_1 \rangle, \alpha g_2) &= e(\langle \vec{w}, A(\tau)g_1 \rangle \alpha, g_2) && \textit{(Factor out constant } \alpha \textit{ in the RHS linear combination)} \\
e(\mathcal{U}, \alpha g_2) &= e(\mathcal{U}\alpha, g_2) && \textit{(Substitute } \mathcal{U} \textit{ for } \langle \vec{w}, A(\tau)g_1 \rangle\textit{)} \\
e(\mathcal{U}, g_2)^\alpha &= e(\mathcal{U}, g_2)^\alpha && \square
\end{aligned}$$

Under the assumption that τ and α were destroyed, the only way (probabilistically) that a prover could provide a valid knowledge-commitment pairing for the $\vec{A}$ query *was if the prover used a witness $\vec{w}$ and computed the linear combinations using the proving key pk.*[4]

3 Ideally Private Tokens

We define an ideally private token as a transactable digital asset where transactions bear the following three properties:

- **Sender Privacy**: Where the initiator of a transaction is not immediately identifiable.
- **Payment Confidentiality**: Where the amount transacted is unknown to non-participants of the transaction.
- **Receiver Privacy**: Where the intended recipient of a transaction is not immediately identifiable.

4 Zcash Summary

In this section, we give an overview of the **Zcash** protocol. Since **Zcash**'s primary goal is to achieve privacy, we first emphasize on how **Zcash**'s *shielded* protocol meets the criteria of ideal privacy before getting into some of the necessary steps for transitioning from a transparent protocol such as **Bitcoin** into the *shielded* protocol that enables ideally private transactions. We do not provide an overview of the other aspects of the **Zcash** protocol (mining, block subsidy, scripting, etc.).

Private transactions in **Zcash** are *note-oriented*, where the transactions create and/or combine *notes*. A note is created (or *minted*) when an amount of currency is taken "out of circulation" from the transparent system to produce a unique commitment of the information required to "spend" (transfer ownership of the currency amount or reintroduce into the transparent system) the associated value. More specifically, the commitment *is* the SHA-256 digest of this "information required to spend." The information required to spend comprises of the amount, the owner's paying key (similar to a *public key* in asymmetric cryptography), and some additional information that will be noted once other aspects of the **Zcash** protocol are described.

[4]This proving system requires other checks to ensure that the witness $\vec{w}$ is the result of a *valid* execution of the program. In actuality, the proving system (BCTV14 [9]) adds another parameter, ρ_A to both of the explicitly mentioned proving key components. Similar components and verification steps are used for the $\vec{B}$ and $\vec{C}$ queries. Another set of key components and verification checks (which use ρ_A, ρ_B, and ρ_C from the homomorphic trapdoor commitments) ensure that the same coefficients were used for the $\vec{A}$, $\vec{B}$, and $\vec{C}$ queries. Finally, another set of key components and checks are used to ensure that the $\vec{A}$, $\vec{B}$, and $\vec{C}$ queries use a witness that is the result of a valid execution of the program.

4.1 Sender Privacy

Zcash achieves its sender privacy via an incremental, fixed-depth Merkle tree inside the zk-SNARK's circuit, which fulfills the role of a *cryptographic accumulator*. All notes, upon validation of the transaction that results in their creation, are appended to the *note commitment tree* at the next available index.

A prover wishing to generate a transaction that spends a note first finds the commitment to that note in the tree, and then determines what is called the *Merkle path* — a list of all the digests that the commitment must be hashed with to produce the root hash of the tree — and provides this private information along with the root hash of the tree as a primary input.

The program uses the note commitment and Merkle path inputs to compute a root hash and provides constraints that the computed root hash must be equal to the publicly-inputted root hash. As long as a cryptographically-secure hash function is used for the Merkle tree, this approach to zero-knowledge set membership is cryptographically-sound since there is no other way to produce the root hash. Given the *second preimage resistance* assumption, only either the true hashes of all the nodes and leaves (or a Merkle branch from the tree) can produce the root hash.

4.2 Payment Confidentiality

Purely private **Zcash** transactions take in *two* notes as inputs and produce two notes as outputs. Values of two input notes can be combined to produce a new note equal to the sum of the inputs and a dummy note or they can be combined to produce two notes of different denominations also equal to the sum of the inputs. Input and output value equality is sometimes referred to as "conservation-of-money." With the conservation-of-money constraints, the actual amounts of notes never need to be made public, meaning that transactions can occur where only the parties involved know the amounts transacted.

4.3 Receiver Privacy

The *recipient* of a transaction of an asset is the party to which *ownership* of the asset is transferred to. In this case, ownership is more specifically the authority to spend a note.

The authority of the party initiating the transaction to spend the notes is determined by whether or not they know the *spending key* corresponding to the *paying key* of the note. The spending key and paying key for each note are private inputs to the program. The privately-inputted spending key is used to derive a computed paying key and constraints are in place to ensure that the computed paying key matches the privately-inputted spending key. Ownership of the output notes is simply determined by the paying keys, which are private inputs, used for the commitment of the new notes.

In addition to the paying key, the initiator of the transaction must privately provide all of the remaining information used to generate the commitment for the note. The information required to spend each new note is encrypted using the *transmission key* of the recipient. A party determines if they are a recipient of a transaction by listening for all transactions and attempting to decrypt the encrypted data with their *viewing key*; if the decryption is successful, they are the recipient.

4.4 Additional Protocol Concerns

Zcash also solves the traditional problem that all digital currencies must solve: *double-spend protection.*

Zcash's transparent protocol is essentially the **Bitcoin** protocol, so double-spend protection of transparent transactions is the same as in **Bitcoin** (where the first transaction is the accepted one), but the privacy-centric protocol provides double-spend protection in a different manner.

Notes, when spent, must publicly reveal a *nullifier.* This nullifier is deterministically computed from the blinded information required to spend the note and is unique. In the zk-SNARK program, there are constraints that ensure that the nullifier is correctly computed, and the **Zcash** protocol also stipulates that a nullifier may only be used once.

4.5 Transitioning From Transparent Protocol

Transitioning from a transparent, publicly-verifiable digital payment protocol, similar to **Bitcoin**, to the shielded protocol described above requires the introduction of two more types of transactions. At this point, we are familiar with two types of transactions, but there are a total of four, as described by the **Zcash** community:

1. **Public**: Where both parties and the amounts are known. The transparent protocol of **Bitcoin** falls into this classification. Tokens implementing the ERC20 interface also fall into this classification.

2. **Shielding**: Where the amount transacted is easy to determine, but the receiving address stores the balance essentially as a blinded hash, where any subsequent transaction amounts will not be exactly knowable (though, they cannot exceed the initial amount if no other spends occur involving the address in question).

3. **Deshielding**: The inverse of shielding; where a blinded amount is unblinded and transacted to a transparent address.

4. **Private**: Where the transaction occurs between two private addresses, and the amount transacted is not known. The shielded protocol described in the previous sections falls into this classification.

The shielding transaction type is necessary for "setting the stage." A visible amount is removed from the transparent system and two notes are created containing the removed amount. This transaction itself does not have the properties of ideal privacy since the transaction was initiated in a transparent system, but because the receiving system's protocol allows for sender privacy, subsequent transactions spending the notes will be ideally private.

The deshielding transaction type is necessary for the re-introduction of the shielded amounts back into the transparent system. These types of transactions still have sender privacy, but there is a loss of payment confidentiality and receiver privacy.

In **Zcash**, all transactions involving shielded values — whether shielding, private, or deshielding — use the same circuit. The shielding transaction type is enabled via a v_{pub}^{old} parameter, and the deshielding transaction type is enabled via a v_{pub}^{new} parameter.

5 Challenges

A direct 'port' of **Zcash**'s circuit is unsuitable for a token-shielding system. There are primarily two compatibility challenges. While a random **Zcash** transaction was successfully verified on an **Ethereum** testnet, (1) there is a severe privacy concern with the fundamental nature of the **Ethereum Virtual Machine (EVM)** when the technology is sought to be used by real users on the **Ethereum** network. There is also a concern (2) that many of the other operations that would need to be performed by an **Ethereum** port of **Zcash** upon a successful verification of a proof would bear too high of a gas cost.

5.1 The "Gas Leak" Challenge

Because **Ethereum** network transactions require that the "broadcaster" of the transaction pay for the execution of the invoked code, and that the network currency, Ether, used to pay for the transaction lacks privacy features, any privacy-focused networks, systems, and tokens implemented on top of the **Ethereum** network will be lacking in sender privacy.

5.1.1 "Gas leak" solution

The solution to the issue of gas payments leaking information about a transaction requires a distinction between the **Ethereum** network transaction, which can be viewed as an interaction on a relatively low-level protocol, and a transaction on the privacy layer network. The privacy layer network is implemented on top of the **Ethereum** network.

To get around the issue of "sender pays the gas," the privacy layer network's protocol provides a mechanism for some other party to broadcast the transaction to the **Ethereum** network so that "someone else to pay the gas." This means that the broadcaster of a transaction can be distinct from the initiator of a transaction.

To simultaneously solve a similar networking-level issue, we note that most leading **Ethereum** clients also include Whisper, a mesh-net intended for "dapp"-level messaging. This is prime for an onion-routing implementation. A broadcast-path is to be pseudo-randomly chosen, and the final node of the broadcast-path broadcasts the transaction to the **Ethereum** network without any knowledge of the party that originally initiated the transaction. The last node is able to insert an **Ethereum** address and claim the reward. The details of this protocol are specified in § 6.

The privacy network also ensures that there are always peers available to broadcast transactions, by both requiring a market-determined amount of privacy layer network tokens to be "spent" to initiate the transaction, and providing a deterministic amount as a reward to the broadcaster.

It should also be noted that the privacy-layer network token must bear the same level of privacy as the non-network token seeking privacy. This is so that the initiator of the transaction can still spend with sender privacy.

5.2 Challenge of Incompatible Optimizations

The second issue is the result of "optimizations" within the gadgets in the circuit used in constructing a zero-knowledge proof in the **Zcash** protocol. The **Zcash** protocol uses the SHA-256 *compression* algorithm, which is not the same as the full secure hashing algorithm as specified in NIST 180.4. The difference between full SHA-256 and the SHA-256 compression algorithm is that the compression algorithm omits the padding step. These optimizations were made because all uses of the compression algorithm in **Zcash** operate on blocks that are already appropriate lengths (modulo 512 bits in length).

To understand the difficulty here requires knowledge of how many cryptographic primitives are accessed in **Ethereum**.

5.2.1 Precompiled contracts

The EVM operates on bytecode that corresponds to opcodes of a virtual machine, where each opcode has a cost proportional to the number of computational steps that the operation requires.

Some routines that are common to the blockchain paradigm (such as hash functions) are implemented in a different form. These are operations that are implemented in the **Ethereum** client itself — which is "closer to the metal" and more efficient to execute. A smart contract address is associated with an operation. These types of special operations, which are implemented in the client and specified with the use of an agreed-upon contract address, are called *precompiled contracts*.

The SHA-256 algorithm is implemented in this manner, as it would be somewhat cost prohibitive to implement in bytecode within a broader context of larger system that also performs other operations. The incompatability between this optimized approach to **Ethereum** and the optimization in the **Zcash** circuit is that **only** the *full* SHA-256 algorithm is implemented, meaning various aspects of the **Zcash** protocol cannot be transplanted to the **Ethereum** network without modifications.

5.2.2 Incompatible optimizations solution

The current solution to this problem, short of submitting an **Ethereum Improvement Proposal (EIP)** for a SHA-256 Compression precompiled contract[5], using the current technology at our disposal is to get rid of the optimization in **Zcash**'s Merkle tree and use the *full* SHA-256 algorithm instead.

This unfortunately comes at a significant computational cost, and in order to be computationally equivalent to the **Zcash** circuit, the depth of the Merkle tree used must be halved (since full SHA-256 would result in twice the amount of constraints due to the "unnecessary" padding imposed in Merkle–Damgård constructions).

5.3 Impact of Both "Solutions" to Aforementioned Challenges

The dramatic impact of both solutions — recall that the first solution was the introduction of a network token that is simultaneously transacted and the second was the substitution of full SHA-256 for SHA-256 compression in the Merkle tree — is that, in order to be computationally equivalent to **Zcash**, the depth of the Merkle tree must be reduced by 75%. This is because the two aforementioned solutions *both* double the amount of constraints.

The **Zcash** Merkle tree uses a depth of 29, meaning it can contain 2^{28} notes, or $268,435,456$ notes. The depth of a Merkle tree for a rough computational equivalence to the **Zcash** Merkle tree, using the same proving system and given the aforementioned solutions, would be approximately 7, or 64 notes.

The depth of the Merkle tree can be increased and bear the same computational equivalence (roughly) if we substitute a more efficient proving system. Amongst the various protocol changes for **Sapling Zcash**, one change that is compatible with the **Ethereum** precompiles is the switch away from the BCTV14 proving system [9] to the Groth16 proving system [10]. Groth16 generates proofs in about 65% of the time of BCTV14, meaning that a depth of 11 can be used with the Groth16 proving system and be roughly computationally equivalent[6] to a depth of 7, meaning that a Groth16 Merkle tree can contain 1024 notes.

One consequence of these changes is the loss of *absolute* sender privacy. Instead of a spent note being any note, a spent note is now identifiable to a pool of 1024 other notes. This is still superior to the level of privacy enabled by the **CryptoNote** protocol (in terms of the set of the potential initiators of a transaction), used by **Monero** and other Ring-Signature based privacy coins.

With the switch to Groth16, the overall depth of the Merkle tree is larger, but it is still too shallow to be usable by a large network, as it would be exhausted fairly quickly. The naïve solution to this problem is to allow for multiple Merkle trees to be used, where notes are appended to a primary tree and once the primary

---

[5]The use of precompiled contracts as a means of extending the capabilities of the EVM is an unsustainable approach and only highlights the flawed design of the EVM.

[6]We say "roughly" as in actuality it is ideal for the degree of the QAP (number of constraints in the circuit) to be as close as possible to a power of two (but not exceeding). Depending on the optimizations that can be made to the new constraints in the circuit, the final depth may be greater than 11.

tree is exhausted, all notes are appended to a *new* primary tree, and so on. The issue with this solution is that it allows for an adversary to infringe upon a user's sender privacy by filling a primary tree with notes that they control in order to reduce the set of notes that a user could be spending from within a tree. This attack, with the shallow depth of the primary tree and an assumed cost of \$3 per transaction, is very cheap at \$768 USD. To resolve this issue, instead of ***a single*** primary tree for notes to be appended to, ***multiple*** "primary" trees are available to be appended to. In order to ensure that only one Merkle root needs to be recomputed per transaction, all output notes from a transaction are appended to the same tree (the number of outputs must be a power of two that is root (square, cubic, etc.) of a tree). The manner of determining the tree a transaction appends to is done by taking the hash of the 251st-last block number modulo the (prime) number of concurrent trees.

6 Whisper Onion Router

A user wishing to send a confidential payload selects three nodes on a particular Whisper topic, and, using the public keys of the selected nodes, encrypts the confidential transaction three times over using `ECDH(...)` and `AEAD_CHACHA20_POLY130(...)` (these algorithms are defined in § 8).

After discovering a set of peers able to broadcast a transaction to the Ethereum network, the selection of the nodes from the discovered set is done in a non-interactive manner. Specifically, the path taken and which nodes are selected is determined by:

1. Running the encoded transaction through a cryptographically secure hash function, specifically `Keccak256(...)` (also defined in § 8), to compute pseudo-entropy source ξ.

2. The number of discovered peers ν_{R_1} is used to find the $\nu_{R_1}^{th}$ prime number ω_{R_1}. Let N denote the collection of all discovered peers.

3. The index i_{R_1} of the first relay-node R_1 is determined by computing:
$$i_{R_1} \equiv \xi \bmod \nu_{R_1}$$

4. R_1 is then removed from N, and ν_{R_1} is decremented to produce ν_{R_2}. ω_{R_2} is calculated using ν_{R_2}.

5. The index i_{R_2} of the second relay-node R_2 is determined by computing:
$$i_{R_2} \equiv \xi \bmod \nu_{R_2}$$

6. R_2 is then removed from N, and ν_{R_2} is decremented to produce ν_B. ω_B is calculated using the new ν_B.

7. The index i_B of the broadcasting node B is determined by computing:
$$i_B \equiv \xi \bmod \nu_B$$

7 Network Incentive

As opposed to the staggered reward having of **Bitcoin** (which is intended to occur every four years), distribution of the **tnnl** network token occurs via a reversed and elongated discrete logistic "curve."

Let I_r be the reward iterator that is incremented with each recorded transaction. Let S_T be the number of steps per reward tier and let C_T be the current reward and C_{T_i} be the current reward tier's index. Let $\mathscr{L}_m$ be a logistic map where $\mathscr{L}_m(x_{n+1}) := rx_n(1 - \frac{x_n}{M})$. Typically, $x_n \in \mathbb{R}$ and $r \in \mathbb{R}$ and $M \in \mathbb{R}$, but because the EVM lacks support for floating point numbers, it must be the case that $x_n \in \mathbb{Z}$ and $r \in \mathbb{Z}$ and $M \in \mathbb{Z}$ (a *discrete* discrete logistic map is required).

Assuming that the chosen values for r and M produce a curve resembling the standard logistic equation and reach stability at a point x_s, values for $\mathscr{L}_m$ are computed up to x_s and comprise $\mathbb{T}$ (the set of reward tiers).

As each transaction is recorded, I_r is incremented (each increment is considered a step). The current reward tier index C_{T_i} is determined using I_r and S_T. (In practice $C_{T_i} := \sharp\mathbb{T} - (\left\lfloor \frac{I_r}{S_T} \right\rfloor + 1)$.) Let the next reward tier N_T be the reward tier immediately preceding C_T. (That is to say, $N_{T_i} := \sharp\mathbb{T} - (\left\lfloor \frac{I_r}{S_T} \right\rfloor + 2)$.)

The *maximum* value rewarded r_{max} to the broadcaster of the transaction is calculated as the mapping of $I_r \mod S_T$ to $[C_T, N_T]$.

7.1 Precompile Gas Cost Reduction Foresight

To address the possibility of the gas costs of the precompiled contracts used within **tnnl** being reduced (as **Ethereum** clients arrive upon more optimized implementations) and impacting the smoothness of the **tnnl** network token distribution, the gas g used in the transaction is an argument that determines the actual value rewarded r_{act} to the broadcaster of the transaction.

With each transaction, the average gas cost g_{avg} (a moving average) is recomputed, and r_{act} is the mapping of g from $[1, g_{avg}]$ to $[0, r_{max}]$.[7]

8 tnnl Protocol Specification

NOTE: *Being derived from **Zcash**, much of the protocol specification that follows is derived from the **Zcash** protocol specification [4].*

As a privacy *layer* for **Ethereum**, **tnnl** relies on an **Ethereum** client and communicates with the client via **Ethereum**'s standard JSON-RPC API. Part of the **tnnl** privacy layer is implemented via a smart contract that exists on the **Ethereum** network. Because this contract maintains the state of the Merkle trees and the Merkle paths used in crafting shielded transactions are determined via JSON-RPC, it is

[7]This specification of the calculation of r_{act} is subject to change with experimentation of gas price tracking (as a measurement of transaction priority).

recommended that those seeking strong privacy do not rely on light clients that query other parties or public services such as Infura.

8.1 Primitive Types

All type-identifiers are denoted in sans serif lettering, such as Uint64 and Uint256. Let a type-specifier Type with trailing square-brackets, such as Type[] denote a vector of Type. If no numeric literal is within the square brackets, the vector is of dynamic length; else if a numeric literal is provided (such as Type[7]), then the number literal specifies the length of the Type vector.

Primitive types used in this specification are:

Bit
: A binary digit, where a literal value is expressed with 0 or 1.

Uint256
: An unsigned integer 256-bits in length.

Uint252
: An unsigned integer 252-bits in length.

Uint208
: An unsigned integer 208-bits in length.

Uint160
: An unsigned integer 160-bits in length.

Uint96
: An unsigned integer 96-bits in length.

Uint64
: An unsigned integer 64-bits in length.

String
: An arbitrary-length array of bytes represented as US-ASCII characters. Literal values are represented in double quotes such as "`abc`". A String is implicitly convertible to a BString.

BString
: An arbitrary-length binary string.

BStringMod512
: An binary string where the length must be modulo 512.

Secp256k1Element
: A tuple (x : Uint256, y : Uint256) denoting a point on the $secp256k1$[19] curve.

8.2 Requisite Algorithms

In this protocol specification, algorithms are denoted in monospaced lettering and a list of arguments, such as `Sha256Compression`($message$).

`Sha256`($message$)

Denotes the full SHA-256 algorithm as specified in [20].

Inputs:

- $message$: BString

Outputs:

- $messageDigest$: Uint256

`Sha256Compression`($message$)

Denotes the SHA-256 *compression* algorithm (essentially `Sha256`, but without the preprocessing-padding step). The assumption is made that the bit-length of $message$ is a multiple of 512.

Inputs:

- $message$: BStringMod512

Outputs:

- $messageDigest$: Uint256

`PRF`(a, b, c, d, x, y)

Pseudo-random function that uses its arguments in a concatenated form $\kappa := a \circ b \circ c \circ d \circ x \circ y$ as an input to `Sha256Compression`(κ).

Inputs:

- a : Bit
- b : Bit
- c : Bit
- d : Bit
- x : Uint252
- y : Uint256

Outputs:

- $pseudoRandomNumber$: Uint256

`Keccak256`($message$)

Denotes the `Keccak-256` algorithm as specified in [21].

Inputs:

- $message$: BString

Outputs:

- $messageDigest$: Uint256

8.3 Key Components and Addresses

8.3.1 Low-level key components

Algorithms mentioned in this subsection are defined in § 8.3.2. A **tnnl** account consists of four low-level key components:

spending key (a_{sk} : Uint252)
The value of a_{sk} is uniformly random and intended to remain private at all times.

paying key (a_{pk} : Uint256)
Derived from a_{sk} using `DeriveAccountPublicKey`(a_{sk}).

receiving key (sk_{enc} : Uint256)
Derived from a_{sk} using `DeriveSealPrivateKey`(a_{sk}). The value of sk_{enc} is to remain private at all times.

transmission key (pk_{enc} : Secp256k1Element)
Derived from sk_{enc} using `DeriveSealPublicKey`(sk_{enc}).

8.3.2 Key component derivation algorithms

Low-level key components are derived with the following algorithms:

`DeriveAccountPublicKey`(a_{sk})
Computes an account public key a_{pk} from a_{sk}, where:

$$a_{pk} := \texttt{PRF}(1, 1, 0, 0, a_{sk}, 0)$$

Inputs:

- a_{sk} : Uint252

Outputs:

- a_{pk} : Uint256

`DeriveSealPrivateKey`(a_{sk})
Internally computes an intermediate, unformatted private key μ, where:

$$\mu := \texttt{PRF}(1, 1, 0, 0, a_{sk}, 2^{248})$$

After μ computed, μ is then padded to the requisite length and returned as sk_{enc}.
Inputs:

- a_{sk} : Uint252

Outputs:

- sk_{enc} : Uint256

`DeriveSealPublicKey`(sk_{enc})
Performs scalar multiplication of generator point $g_{secp256k1}$ with sk_{enc}.
Inputs:

- sk_{enc} : Uint256

OUTPUTS:

- pk_{enc} : Secp256k1Element

8.3.3 User-level key components

It is not intended for **tnnl** users to deal with all of the aforementioned low-level key components directly. Instead, users are only required to understand their client maintains knowledge of the account's a_{sk}, and the user only interacts with the following composite key-components:

shielded payment address

Denoted as $addr_{pk}$, the *shielded payment address* is a tuple (a_{pk}, pk_{enc}). A user wishing to receive a payment presents $addr_{pk}$. This address is reusable; although, as noted in the **Zcash** protocol specification, colluding payers are able to determine if they have sent to the same address, so users may want to create separate addresses for each invoice.

incoming viewing key

Denoted as ivk, the *incoming viewing key* is a tuple (a_{pk}, sk_{enc}). A user wishing to determine if they are a recipient of a particular note uses this key to attempt to decrypt the ciphertext which stores the necessary information to produce the commitment of the note.

8.4 Notes

A *note* is a tuple $(a_{pk}, c_{addr}, v, \rho, r)$, where:

- a_{pk} : Uint256 is the (low-level) owner's shielded paying key.
- c_{addr} : Uint160 is the contract address of the token that is shielded. This field does not exist in **Zcash**.
- v : Uint64 is the value of the shielded token, in its smallest denomination.
- ρ : Uint256 is used as an input to derive the *nullifier* of the note. The derivation of ρ is stated in § 8.6.3.
- r : Uint208 is a random blinding factor (also known as a commitment trapdoor). The length of r differs from **Zcash** to a slight detriment (in **Zcash** it is 256) in order to accommodate the new c_{addr} field without adding another block to the note commitment algorithm.

Let Note be the type-identifier for an instance of a *note* tuple.

8.4.1 Note commitments

Notes are appended to a Merkle tree in the form of a *commitment* ς. The following algorithm is used to compute the commitment of a Note.

`CommitToNote`(n)

Generates a commitment ς for a Note n. n is unpacked to produce $(a_{pk}$,

$c_{addr}, v, \rho, r)$. The result of $\texttt{Sha256}(1 \circ 0 \circ 1 \circ 1 \circ 0 \circ 0 \circ 0 \circ 0 \circ a_{pk} \circ c_{addr} \circ v \circ \rho \circ r)$ is assigned to ς.

INPUTS:

- n : Note

OUTPUTS:

- ς : Uint256

8.4.2 Note nullifiers

To ensure that notes are only spent once, every Note n has a corresponding $nullifier$ θ that must be publicly revealed when n is spent. The revelation of a $nullifier$ does not indicate which Note is being spent, since the nullifier depends on a_{sk}, which is never public. Additionally, because $nullifiers$ rely on a_{sk}, they can only be generated by the owner of n. Nullifiers are computed using the following algorithm:

$\texttt{ComputeNullifier}(n, a_{sk})$

Computes a nullifier θ for a Note n. n is unpacked to retrieve ρ, which is used alongside a_{sk} for this computation.

$$\theta := \texttt{PRF}(1, 1, 1, 0, a_{sk}, \rho)$$

INPUTS:

- n : Note
- a_{sk} : Uint252

OUTPUTS:

- θ : Uint256

8.4.3 Note encryption and decryption

The sender of a transaction generates an *ephemeral keypair*, which is a tuple (eph_{sk}, eph_{pk}), where:

- eph_{sk} : Uint256 is the *ephemeral private key*, which uniformly random and is intended to stay secret.
- eph_{pk} : Secp256k1Element is the *ephemeral public key*, which is included as part of the transaction along with the Note's transmitted ciphertext. eph_{pk} is derived from eph_{sk} using $\texttt{DeriveSealPublicKey}(eph_{sk})$.

The following algorithm denotes the $Elliptic\text{-}curve\ Diffie\text{–}Hellman$ key-agreement algorithm, which is used by both parties to arrive upon a shared secret $\mathcal{S}$:

$\texttt{ECDH}(a, \mathcal{B})$

Group multiplication of public $\mathcal{B} \in \mathbb{G}_{secp256k1}$ by a secret scalar a.

INPUTS:

- a : Uint256

- $\mathcal{B}$: Secp256k1Element

Outputs:

- $\mathcal{S}$: Secp256k1Element

The sender of a note uses their $ephemeral\ private\ key\ eph_{sk}$ as the first argument and the recipient's $transmission\ key\ pk_{enc}$ as the second argument. The recipient uses their sk_{enc} as the first argument and the sender's $ephemeral\ public\ key\ eph_{pk}$ as the second argument. Assuming that the recipient *is* the intended recipient, then the following equality will hold and both parties have arrived at the same $shared\ secret\ \mathcal{S}$:

$$\texttt{ECDH}(eph_{sk}, pk_{enc}) = \texttt{ECDH}(sk_{enc}, eph_{pk})$$

The shared secret $\mathcal{S}$ is used as an input to the following $key\ derivation\ function$:

$\texttt{KDF}(i, h_{sig}, \mathcal{S}, eph_{pk}, pk_{enc})$

Derives a key, denoted as δ, used for the symmetric encryption and decryption of a Note corresponding to index i. (Observe the seven zero-bits preceding i which, when concatenated with i, occupy a byte.)

$\delta := \texttt{Keccak256}(\text{``}\texttt{tnnlKDF}\text{''} \circ 0 \circ 0 \circ 0 \circ 0 \circ 0 \circ 0 \circ 0 \circ i \circ h_{sig} \circ \mathcal{S} \circ eph_{pk} \circ pk_{enc})$

Inputs:

- i : Bit
- h_{sig} : Uint256
- $\mathcal{S}$: Secp256k1Element
- eph_{pk} : Secp256k1Element
- pk_{enc} : Secp256k1Element

Outputs:

- δ : Uint256

Once δ is derived, it is used as the key with the `AEAD_CHACHA20_POLY130` symmetric encryption scheme as per RFC-7539[22]. The 96-bit nonce is "all-zero" (this acceptable because the $ephemeral\ key$ is never reused) and the "associated data" is unassigned. The following algorithms denote symmetrical encryption and decryption of a Note n:

$\texttt{Seal}(n, \delta, memo)$

Inputs:

- n : Note
- δ : Uint256
- $memo$: String

Outputs:

- $ciphertext$: BString

$\texttt{Open}(ciphertext, \delta)$

Inputs:

- $ciphertext$: BString
- δ : Uint256

Outputs:

- n : Note
- $memo$: String

8.5 Merkle Paths

Let MerklePath be the type-identifier for a $Merkle\ path$, a tuple $(\vec{\eta}, \lambda)$ where:

- $\vec{\eta}$: MerkleNode[11]
- λ : MerkleLeaf

and MerkleNode is a tuple where:

- $value$: Uint256 is the intermediary digest value
- $index$: Bit is the value indicating which ***side*** the intermediary digest value corresponds to.

When a Note n's commitment ς is appended to the primary Merkle tree (which exists on the **Ethereum** mainnet in the form of a smart contract), the $tree\ index$ t_i and $leaf\ index$ l_i of the commitment is logged as an **Ethereum** event. When the owner of n wishes to spend n, the logged information is used to determine a valid MerklePath in the following algorithm:

`GetMerklePath`(t_i, l_i)

Determines a MerklePath ψ and root hash rt for the commitment ς at the specified indices.

Inputs:

- t_i : Uint256
- l_i : Uint64

Outputs:

- ψ : MerklePath
- rt : Uint256

8.6 JoinSplit Transactions

A $JoinSplit$ transaction is a shielded transaction spending (by revealing their nullifiers) two input Notes and creating two output Notes.

8.6.1 JoinSplit statements

Because a $JoinSplit$ transaction is shielded via a zk-SNARK, it is attested to with a proof π. In this context, a proof can be referred to as a statement. Let a $JoinSplit$ $statement$'s type-identifier be JoinSplit$_\pi$, which denotes a specific proof π with

- rt : Uint256[2]
- $\vec{\theta}^{old}$: Uint256[2]

- $\vec{\varsigma}$: Uint256[2]
- v_{pub}^{old} : Uint64
- v_{pub}^{new} : Uint64
- c_{addr} : Uint160
- h_{sig} : Uint256
- $\vec{h}$: Uint256[2]

as the *primary input* and

- $\vec{\psi}$: MerklePath[2]
- $\vec{n}^{old}$: Note[2]
- $\vec{a}_{sk}$: Uint252[2]
- $\vec{n}^{new}$: Note[2]
- φ : Uint252

as the *auxiliary input* that satisfies the following constraints:

- **Merkle path validity** — (which is enforced if $v_{pub}^{old} \neq 0$) where each $n_i \in \vec{n}^{old}$, when hashed with the values of ψ_i, MUST produce rt_i.
- **Contract address equivalence** — (which is enforced if $v_{pub}^{old} \neq 0 \vee v_{pub}^{new} \neq 0$) where c_{addr} MUST match each $n_i^{old}.c_{addr} \in \vec{n}^{old}$ and $n_i^{new}.c_{addr} \in \vec{n}^{new}$.
- **Balance** — $v_{pub}^{old} + \sum_{i=0}^{1} n_i^{old}.v = v_{pub}^{new} + \sum_{i=0}^{1} n_i^{new}.v \in \{0..2^{64} - 1\}$.
- **Nullifier integrity** — for each $i \in \{0, 1\}$: $\theta_i^{old} = \texttt{ComputeNullifier}(n_i, a_{ski})$.
- **Spend authority** — for each $i \in \{0, 1\}$: $n_i^{old}.a_{pk} = \texttt{DeriveAccountPublicKey}(a_{ski})$.
- **Non-malleability** — for each $i \in \{0, 1\}$: $h_i = \texttt{ComputeAuthCode}(i, a_{ski}, h_{sig})$ (where `ComputeAuthCode(...)` is as described in § 8.6.2.1).
- **Uniqueness of $\vec{\rho}^{new}$** — for each $i \in \{0, 1\}$: $n_i.\rho^{new} = \texttt{DeriveRho}(i, \varphi, h_{sig})$ (where `DeriveRho(...)` is as described in § 8.6.3).
- **Note commitment integrity** — for each $i \in \{0, 1\}$: $\varsigma = \texttt{CommitToNote}(n_i)$.

8.6.2 Derivation and role of h_{sig}

There are couple of measures in place to ensure that a transaction cannot be tampered with. The signature scheme described in § 8.7.2 is one measure to prevent tampering with a JoinSplit$_\pi$ $\jmath$. Another measure is the manner in which $\jmath$'s h_{sig} is computed and how h_{sig} is used to generate authentication codes $\vec{h}$ for $\vec{n}^{old}$.

h_{sig} is computed with the following algorithm:

$\texttt{ComputeHSig}(randomSeed, \vec{\theta}^{old}, \gamma_{pk})$

Computes a unique identifier h_{sig} for a shielded transaction, where:

$$h_{sig} := \texttt{Keccak256}(\text{“}\texttt{tnnlComputehSig}\text{”} \circ randomSeed \circ \vec{\theta}^{old} \circ \gamma_{pk})$$

b Inputs:

- $randomSeed$: Uint256
- $\vec{\theta}^{old}$: Uint256[]
- γ_{pk} : Uint160

Outputs:

- h_{sig} : Uint256

8.6.2.1 Non-malleability authentication codes

For each n_i in $\vec{n}^{old}$, a non-malleability authentication code h (that ensures the owner of n_i has authorized the use of their private key a_{sk} in the transaction) is generated using the following algorithm:

`ComputeAuthCode`(i, a_{sk}, h_{sig})

Computes an authentication code h for the private key a_{sk} corresponding to the Note n at zeroed-index i, where:

$$h := \texttt{PRF}(0, i, 0, 0, a_{sk}, h_{sig})$$

INPUTS:

- i : Bit
- a_{sk} : Uint252
- h_{sig} : Uint256

OUTPUTS:

- h : Uint256

8.6.3 Derivation of ρ

Since authentic notes (as opposed to dummy notes) are created as the result of a transaction where there exists an h_{sig}, their commitments incorporate this h_{sig} as an input into the following algorithm that derives the ρ (that is used to compute a nullifier at a later point in time):

`DeriveRho`(i, φ, h_{sig})

Derives a unique ρ for the commitment ς of a Note n^{new} at index i, using φ as a random private seed, where:

$$\rho := \texttt{PRF}(0, i, 0, 0, \varphi, h_{sig})$$

INPUTS:

- i : Bit
- φ : Uint252
- h_{sig} : Uint256

OUTPUTS:

- ρ : Uint256

8.7 tnnlJoinSplit Descriptions

A tnnlJoinSplit$_\pi$ transaction contains *two* JoinSplit$_\pi$s, where:

- *network* denotes the JoinSplit$_\pi$ statement for the network token paid to the broadcaster of the transaction
- *shielded* denotes the arbitrary token being privately transacted in an unknown manner

For the purpose of efficiency, the primary inputs for both $\mathsf{JoinSplit}_\pi$ statements are concatenated and computed and verified as a single proof π.

8.7.1 tnnlJoinSplit serialization

The following algorithms are used for the serialization of a $\mathsf{JoinSplit}_\pi$ and $\mathsf{tnnlJoinSplit}_\pi$ $\jmath$ respectively.

`SerializeJoinSplit`($\jmath_{JS}$)

$\jmath_{JS}$ is unpacked to produce $(rt, \vec{\theta}^{old}, \vec{\varsigma}, h_{sig}, \vec{h})$.

$$packed := rt_0 \circ rt_1 \circ h_{sig} \circ \theta_0^{old} \circ h_0 \circ \theta_1^{old} \circ h_1 \circ \varsigma_0 \circ \varsigma_1$$

Inputs:

- $\jmath_{JS}$: $\mathsf{JoinSplit}_\pi$

Outputs:

- $packed$: $\mathsf{BString}$

`SerializeTnnlJoinSplit`($\jmath_{tJS}$)

`SerializeJoinSplit`(is used with each property of $\jmath_{tJS}$. In addition, $\jmath_{tJS}.\mathit{network}$ is unpacked to produce $(\mathit{network}_{v^{old}_{pub}}, \mathit{network}_{v^{new}_{pub}}, \mathit{network}_{c_{addr}})$ and, similarly, $\jmath_{tJS}.\mathit{shielded}$ is unpacked to produce $(\mathit{shielded}_{v^{old}_{pub}}, \mathit{shielded}_{v^{new}_{pub}}, \mathit{shielded}_{c_{addr}})$.

$$packed_{\mathit{network}} := \mathtt{SerializeJoinSplit}(\jmath_{tJS}.\mathit{network})$$
$$packed_{\mathit{shielded}} := \mathtt{SerializeJoinSplit}(\jmath_{tJS}.\mathit{shielded})$$
$$packed_{\mathit{all}} := packed_{\mathit{network}} \circ packed_{\mathit{shielded}} \circ \mathit{network}_{v^{old}_{pub}} \circ \mathit{network}_{v^{new}_{pub}} \circ \mathit{shielded}_{v^{old}_{pub}} \circ \mathit{shielded}_{v^{net}_{pub}} \circ \mathit{network}_{c_{addr}} \circ \mathit{shielded}_{c_{addr}}$$

Inputs:

- $\jmath_{tJS}$: $\mathsf{tnnlJoinSplit}_\pi$

Outputs:

- $packed_{\mathit{all}}$: $\mathsf{BString}$

8.7.2 JoinSplit signatures

To ensure that no party can tamper with a $\mathsf{tnnlJoinSplit}_\pi$ transaction $\jmath$, $\jmath$ is cryptographically signed in a signature scheme referred to as JoinSplitSig.

Because these $\mathsf{tnnlJoinSplit}_\pi$ transactions are verified on the **Ethereum** network, there is an associated gas cost with the signature verification. In order to be as efficient as possible, the `ECRECOVER` precompiled contract (specified in [3]) is used, as it bears the lowest possible gas cost for a signature scheme.

A JoinSplitKeyPair is a tuple $(\gamma_{sk}, \gamma_{pk})$, where:

- γ_{sk} : Uint256 is the signer's private key
- γ_{pk} : Uint160 is the signer's public key, which takes the form of an **Ethereum** address, although it is not used as such.

In practice, JoinSplitSig is really an implementation of the standard **Ethereum** digital signature algorithm (EIP 191 formatted) (where the result of `SerializeTnnlJoinSplit`$(\jmath)$ is passed as the "message" and a random "ephemeral" JoinSplitKeyPair γ is generated and γ's γ_{sk} component is used as the private key).

9 Acknowledgments

Special thanks go to the inventors of **Zerocash** — Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza — and the designers of the **Zcash** protocol, which are the **Zerocash** inventors and also Daira Hopwood, Sean bBowe, Jack Grigg, Simon Liu, Taylor Hornby, Nathan Wilcox, Zooko Wilcox, Jay Graber, Ariel Gabizon, and George Tankersley. The designers of the **tnnl** protocol are Alex Geiger and Julian Geiger.

Many of the ideas utilized by the aforementioned privacy-focused protocols were first applied to digital currencies by Tomas Sander and Amnon Ta–Shma. To a certain extent, **Zerocash**, **Zcash**, and **tnnl** are refinements of their "Auditable, Anonymous Electronic Cash" proposal in [23].

We would also like to thank the designers and developers of both **Bitcoin** and **Ethereum**.

References

[1] Vitalik Buterin. *Ethereum: The Ultimate Smart Contract and Decentralized Application Platform.* 2013. URL: `http://web.archive.org/web/20131228111141/http://vbuterin.com/ethereum.html`.

[2] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform.* 2018. URL: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[3] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger (Byzantium Version).* 2018. URL: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[4] Daira Hopwood et al. *Zcash Protocol Specification (2018.0-beta-21).* 2018. URL: `https://github.com/zcash/zips/releases/tag/2018.0-beta-21`.

[5] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* 2008. URL: `http://bitcoin.org/bitcoin.pdf`.

[6] S Goldwasser, S Micali, and C Rackoff. "The Knowledge Complexity of Interactive Proof-systems". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing.* STOC '85. Providence, Rhode Island, USA: ACM, 1985, pp. 291–304. ISBN: 0-89791-151-2. DOI: `10.1145/22145.22178`. URL: `http://doi.acm.org/10.1145/22145.22178`.

[7] Nir Bitansky et al. “From Extractable Collision Resistance to Succinct Non-interactive Arguments of Knowledge, and Back Again”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference.* ITCS ’12. Cambridge, Massachusetts: ACM, 2012, pp. 326–349. ISBN: 978-1-4503-1115-1. DOI: 10.1145/2090236.2090263. URL: http://doi.acm.org/10.1145/2090236.2090263.

[8] Bryan Parno et al. *Pinocchio: Nearly Practical Verifiable Computation.* Cryptology ePrint Archive, Report 2013/279. 2013. URL: https://eprint.iacr.org/2013/279.

[9] Eli Ben-Sasson et al. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture.* Cryptology ePrint Archive, Report 2013/879. 2013. URL: https://eprint.iacr.org/2013/879.

[10] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments.* Cryptology ePrint Archive, Report 2016/260. 2016. URL: https://eprint.iacr.org/2016/260.

[11] Ralph C. Merkle. *Secrecy, Authentication and Public Key Systems.* 1979. URL: http://www.merkle.com/papers/Thesis1979.pdf.

[12] Ralph C. Merkle. “(US4309569) Method of providing digital signatures”. Pat. 1982. URL: https://patentscope.wipo.int/search/en/detail.jsf?docId=US37291157&redirectedID=true.

[13] Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. “Proving Properties of Incremental Merkle Trees”. In: *Proceedings of the 20th International Conference on Automated Deduction.* CADE’ 20. Tallinn, Estonia: Springer-Verlag, 2005, pp. 424–440. ISBN: 3-540-28005-7, 978-3-540-28005-7. DOI: 10.1007/11532231_31. URL: http://dx.doi.org/10.1007/11532231_31.

[14] Laurie Law, Susan Sabett, and Jerry Solinas. *How to Make a Mint: The Cryptography of Anonymous Electronic Cash.* Tech. rep. National Security Agency Office of Information Security Research and Technology, June 1996. URL: http://groups.csail.mit.edu/mac/classes/6.805/articles/money/nsamint/nsamint.htm.

[15] Ian Miers et al. “Zerocoin: Anonymous Distributed E-Cash from Bitcoin”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* The Johns Hopkins University Department of Computer Science, Baltimore, USA. Oakland, 2013. URL: http://spar.isi.jhu.edu/~mgreen/ZerocoinOakland.pdf.

[16] George Danezis et al. “Pinocchio Coin: Building Zerocoin from a Succinct Pairing-based Proof System”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* Microsoft Research. Oakland, 2013. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.673.8208&rep=rep1&type=pdf.

[17] Eli Ben-Sasson et al. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *Proceedings of the IEEE Symposium on Security & Privacy.* 459-474. Oakland, 2014. URL: http://zerocash-project.org/media/pdf/zerocash-oakland2014.pdf.

[18] Michael Scott. *A note on twists for pairing friendly curves.* URL: http://indigo.ie/~mscott/twists.pdf.

[19] Certicom Research. *SEC 2: Recommended Elliptic Curve Domain Parameters.* 2010. URL: http://www.secg.org/sec2-v2.pdf.

[20] *FIPS PUB 180-4, Secure Hash Standard (SHS).* U.S.Department of Commerce/National Institute of Standards and Technology. 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[21] Guido Bertoni et al. *The Keccak Reference.* 2011. URL: https://keccak.team/files/Keccak-reference-3.0.pdf.

[22] Yoav Nir and Adam Langley. *Request for Comments 7539: ChaCha20 and Poly1305 for IETF Protocols.* 2015. URL: https://tools.ietf.org/html/rfc7539 (visited on 07/18/2018).

[23] Tomas Sander and Amnon Ta-Shma. "Auditable, Anonymous Electronic Cash". In: *Advances in Cryptology - CRYPTO '99. Proceedings of the 19th Annual International Cryptology Conference (Santa Barbara, California, USA, August 15–19, 1999).* Ed. by Michael Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 555–572. ISBN: 978-3-540-66347-8. DOI: 10.1007/3-540-48405-1_35. URL: https://link.springer.com/content/pdf/10.1007/3-540-48405-1_35.pdf (visited on 06/05/2018).