The Reusability of zk-SNARKs in the Zcash Protocol

Cordian A. Daniluk
Laboratoire Jean Kuntzmann
Grenoble, France
cordian.daniluk@grenoble-inp.org

Supervised by: Aude Maignan

I understand what plagiarism entails and I declare that this report is my own, original work.
Cordian A. Daniluk, March 30th, 2022:

Abstract

A method to speed up repeated creation of a zk-SNARK in JoinSplit descriptions of Zcash transactions given another similar, previously created zk-SNARK is presented. The cryptocurrency Zcash and zk-SNARKs are introduced, explaining why optimizing the repeated creation of zk-SNARKs is useful in the context of Zcash. A mechanism using zk-SNARKs to send money in Zcash called JoinSplit description is explained, with some focus on the Groth16 proving system. Conceivable situations are described when such optimization can be beneficial to performance. The optimization techniques themselves are detailed.

1 Introduction

Cryptocurrencies such as Bitcoin [Nakamoto, 2008] were designed with the goal to decentralize money transfer. There should not be a central trusted party such as a bank, through which passes every transaction to remove the need for trust. However, without any trusted party, transactions are not checked for validity such as the fact that a spender must really be the owner of the spent money and that a spender cannot spend money twice.

A commonly used solution by cryptocurrencies to enforce these rules and yet others is a public ledger called the blockchain [Nakamoto, 2008]. The public nature of this data structure enables any node in the cryptocurrency's network to verify rules such as ownership of spent money or double-spending money. However, without appropriate measures a public ledger can lead to compromised privacy by revealing information such as the identities of transaction participants. In the special case of Bitcoin, a spent amount of money is associated with a destination address, effectively a user's pseudonym. A user may employ an arbitrary number of pseudonyms to hide their true identity and additionally use methods similar to money laundering, but even these fail to guarantee anonymity [Reid and Harrigan, 2011]. Hence, people have tried improving anonymity in new cryptocurrencies such as Monero [van Saberhagen, 2013] or Zcash [Hopwood *et al.*, 2023]. Zcash is the one this internship is focusing on.

Zcash still uses a blockchain, but to make stronger anonymity guarantees than Bitcoin, Zcash offers partially or fully shielded transactions as well as transparent transactions inherited from Bitcoin. While the latter leak data such as the pseudonyms of senders and receivers, the former hide it by either encrypting sensitive information on the blockchain in an anonymity-preserving way or not including it at all. This lack of inclusion needs additional care, however, since classical Bitcoin nodes need some public information such as a sender's pseudonym to enforce the blockchain's validity. To this end, a cryptographic construct called *zero-knowledge succinct non-interactive argument of knowledge* (zk-SNARK) is used. A zk-SNARK has public and secret inputs. The former are publicly known, the latter are only known to the sender. The sender adds all sensitive information as the secret input and some other as the public input, creates a zk-SNARK from these, and adds it to a shielded transaction. The zk-SNARK expresses the statement that the sender knows secret inputs such as the monetary value of the inputs such that the transaction is valid: for example, it is verified that the sender rightfully claims ownership of a given amount of money. Upon inclusion in the blockchain, anyone can verify the zk-SNARK to enforce the transaction's validity. Hence, nodes do not do this on public data directly included in the blockchain as in Bitcoin, but on a zk-SNARK, which enables validation of transactions without revealing its secret inputs.

The various properties of zk-SNARKs render them useful in Zcash: they are zero-knowledge, meaning that they prove statements given secret inputs without revealing them. Their succinctness makes for proofs of small size and short verification time by nodes in the Zcash network. Since they are non-interactive, unlike proposed by, for example, the foundational paper of zero-knowledge proofs [Goldwasser *et al.*, 1985], they can be included in the blockchain without further communication between sender and other parties. Finally, they are arguments, meaning that a malicious, computationally bounded sender cannot fake proofs to mislead nodes in the network. More specifically, they are arguments of knowledge, which means that not only do secret inputs exist that satisfy the proven statement, but also that the prover knows them. For a formal definition of zk-SNARKs, see [Groth,

2016], for example.

Beyond its theoretical definition, zk-SNARKs steadily evolve to make them even more practical. In Zcash alone, three different zk-SNARKs ([Ben-Sasson *et al.*, 2014] in the Sprout version, [Groth, 2016] in the Sapling version, [Bowe, 2020] in the Orchard version) have been used throughout the protocol's history. One reason for this evolution is the effort to reduce time and space requirements of the creation of zk-SNARKs: we have measured JoinSplit descriptions, a certain method to send money in a shielded way (see Section 2) using a zk-SNARK, to take about 25 s in time and about 1 GiB in space (the computer used for these measurements is given in Section 3.3) on a current Zcash node (zcashd v5.5.1).

There might be situations where the creation of zk-SNARKs does not have to be done from scratch: the network might have rejected a transaction containing a JoinSplit description and the user wants to resend it, so information from the previous transaction could be reusable. This document will focus on two reasons for rejection: insufficient transaction fees, which leads to no miner wanting to include the transaction in a block, and chain reorganizations, where blocks become stale because another branch with higher total work performed by miners has overtaken them (more detail is given in Section 3.1). It will investigate under what circumstances and to what degree the zk-SNARKs in the resent transactions have to be entirely recalculated, modified, or can be left as is. The used zk-SNARK is Groth16 [Groth, 2016], which is used since the Sapling network upgrade. While it is used for other sending mechanisms than JoinSplit descriptions, only these shall be considered in this report. Additionally, resending will only be considered for transactions sent by a Zcash full node on that same node; there are also lightweight clients, which store less information and are therefore usable on less powerful computers, but these are not considered. With an efficient resending mechanism, we hope to improve performance of the otherwise costly zk-SNARK creation of JoinSplit descriptions in Sprout and Sapling.

The report is structured as follows: Section 2 gives some background information on JoinSplit descriptions in Zcash, including the Groth16 zk-SNARK used since Sapling. Section 3 explains the actual optimization methods. This includes a description of the handled cases of transaction rejection as well as practical considerations of these optimizations. Section 4 concludes the report, summarizing the results and evaluating their effectiveness. Section 5 provides some suggestions for future work.

2 JoinSplit Descriptions

In general, Sprout transactions consist of several fields such as version information or a signature, as well as JoinSplit descriptions (in the following also abbreviated as JoinSplit). Only the latter are relevant to this paper, so we will not detail any further the other parts and only informally explain JoinSplits. For a full listing and detailed explanation of transaction elements, see [Hopwood *et al.*, 2023].

JoinSplit descriptions are composed of the elements in Table 1, which also contains public inputs to the zk-SNARK $\pi_{\text{ZKJoinSplit}}$. The secret inputs to $\pi_{\text{ZKJoinSplit}}$ are listed in Table 2. All fields mentioned in this section are detailed in these two tables.

As the name says, a JoinSplit joins the value of two old *shielded notes* that represent some monetary value and splits that sum into two new shielded notes. A note is owned by a Zcash user, so JoinSplits can be used to send money (the owner of old and new notes differs) to one or two recipients without publicly revealing any user's identity or to arrange money in different ways in one user's wallet (the owner of old and new notes is the same). Concretely, a node that sends a JoinSplit creates new shielded notes $\boldsymbol{n}_1^{\text{new}}, \boldsymbol{n}_2^{\text{new}}$, which contain elements given in their description in Table 2, and spends two old shielded notes $\boldsymbol{n}_1^{\text{old}}, \boldsymbol{n}_2^{\text{old}}$. Instead of revealing $\boldsymbol{n}_1^{\text{new}}, \boldsymbol{n}_2^{\text{new}}$, which would reveal the future owner's public key harming anonymity, the node creates cryptographically hiding and binding commitments of $\boldsymbol{n}_1^{\text{new}}, \boldsymbol{n}_2^{\text{new}}$ denoted by $\text{cm}_1^{\text{new}}, \text{cm}_2^{\text{new}}$. These commitments are publicly stored in the leaves of a binary Merkle tree [Merkle, 1988], the *commitment tree*. This tree evolves with each JoinSplit added to the blockchain and has well-defined states after every block and every JoinSplit. There is a one-to-one association between a shielded note and its commitment.

If a user wants to spend two shielded notes, he proves ownership of these by proving knowledge of the secret spending keys $a_{\text{sk},1}^{\text{old}}, a_{\text{sk},2}^{\text{old}}$ and their associated commitments' existence by proving their membership in the commitment tree of some block or JoinSplit. Furthermore, the user reveals nullifiers $\text{nf}_1^{\text{old}}, \text{nf}_2^{\text{old}}$ that correspond one-to-one to the shielded notes $\boldsymbol{n}_1^{\text{old}}, \boldsymbol{n}_2^{\text{old}}$. Publishing the nullifiers enables other nodes to check if shielded notes with the same nullifiers have already been spent and prevent double-spending. While the public only learns $\text{cm}_1^{\text{new}}, \text{cm}_2^{\text{new}}$, the receiver needs the full shielded notes $\boldsymbol{n}_1^{\text{new}}, \boldsymbol{n}_2^{\text{new}}$, so that he can spend them later on by using them as $\boldsymbol{n}_1^{\text{old}}, \boldsymbol{n}_2^{\text{old}}$ in a JoinSplit. The spender encrypts these new shielded notes and puts them as $C_1^{\text{enc}}, C_2^{\text{enc}}$ in the JoinSplit in a manner such that the receiver can decrypt them, but no one else can. Finally, measures against malleability are taken and $\pi_{\text{ZKJoinSplit}}$ is created.

A JoinSplit always has two nullifiers and two new commitments. To use only one of either, *dummy notes* can be used.

The most prominent element of a JoinSplit is $\pi_{\text{ZKJoinSplit}}$ that proves a statement about the JoinSplit's validity such that others can verify that the JoinSplit is indeed valid. $\pi_{\text{ZKJoinSplit}}$ enforces rules such as

$$v_{\text{pub}}^{\text{old}} + v_1^{\text{old}} + v_2^{\text{old}} = v_{\text{pub}}^{\text{new}} + v_1^{\text{new}} + v_2^{\text{new}} \tag{1}$$

or the aforementioned membership of commitments in the commitment tree: for $i \in \{1, 2\}$, if $\text{enforceMerklePath}_i = 1$, then path_i is a valid Merkle path from the commitment corresponding to $\boldsymbol{n}_i^{\text{old}}$ at index pos_i to the root $\text{rt}^{\text{Sprout}}$. Other rules are enforced, such as whether $a_{\text{sk},i}^{\text{old}}$ fits $a_{\text{pk},i}$ to prove the spender's possession of the shielded note. However, detailed knowledge of the rules is not relevant for the remainder and can be acquired from [Hopwood *et al.*, 2023].

$\pi_{\text{ZKJoinSplit}}$ is created using the Groth16 [Groth, 2016] proving system, so the next section will explain informally how proof creation works. For a rigorous treatment, see the original paper.

Name	Description
$v_{\text{pub}}^{\text{old}\,*}$	A value that is drawn from a transparent input and can be used for new commitments in JoinSplits. 0 if $v_{\text{pub}}^{\text{new}} \neq 0$.
$v_{\text{pub}}^{\text{new}\,*}$	A value that is drawn from a shielded note and can be used for a transparent output. 0 if $v_{\text{pub}}^{\text{old}} \neq 0$.
$\text{rt}^{\text{Sprout}\,*}$	A root of the commitment tree at some blockheight in the past or a root produced by a previous JoinSplit in the same transaction.
$\text{nf}_1^{\text{old}}, \text{nf}_2^{\text{old}\,*}$	Nullifiers that refer to shielded notes and are made public to prevent double-spending.
$\text{cm}_1^{\text{new}}, \text{cm}_2^{\text{new}\,*}$	Commitments to new shielded notes. For $i \in \{1,2\}$, cm_i^{new} is calculated from $\boldsymbol{n}_i^{\text{new}}$ as follows: $\text{cm}_i^{\text{new}} = \text{SHA-256}(\text{0xB0} \mid a_{\text{pk},i}^{\text{new}} \mid v_i^{\text{new}} \mid \rho_i^{\text{new}} \mid \text{rcm}_i^{\text{new}})$ where $\mid$ denotes bitwise concatenation.
epk	A public key relevant to the creation of $C_1^{\text{enc}}, C_2^{\text{enc}}$.
randomSeed	A random value that is used in the creation of $C_1^{\text{enc}}, C_2^{\text{enc}}$ and h_1, h_2.
h_1, h_2^*	MAC tags that help enforcing non-malleability of the JoinSplit.
$\pi_{\text{ZKJoinSplit}}$	An encoding of the zk-SNARK.
$C_1^{\text{enc}}, C_2^{\text{enc}}$	The new shielded notes in encrypted form destined for the receiver.

Table 1: A JoinSplit description. Public inputs to $\pi_{\text{ZKJoinSplit}}$ are a subset of these fields and marked with *. The fields' names are the same ones as in the original [Hopwood *et al.*, 2023].

Name	Description
φ	A randomly sampled value used to verify uniqueness of $\rho_1^{\text{new}}, \rho_2^{\text{new}}$.
$\boldsymbol{n}_1^{\text{old}}, \boldsymbol{n}_2^{\text{old}}$	$\forall i \in \{1,2\}\colon \boldsymbol{n}_i^{\text{old}} = (a_{\text{pk},i}^{\text{old}}, v_i^{\text{old}}, \rho_i^{\text{old}}, \text{rcm}_i^{\text{old}})$ The content of the shielded notes about to be spent by revealing their associated nullifier. $a_{\text{pk},i}^{\text{old}}$ is a public key identifying the owner. v_i^{old} is the monetary value contained in the shielded note. ρ_i^{old} is a unique identifier to avoid double-spending. $\text{rcm}_i^{\text{old}}$ is a random commitment trapdoor that is part of the commitment scheme.
$a_{\text{sk},1}^{\text{old}}, a_{\text{sk},2}^{\text{old}}$	Secret keys that match $a_{\text{pk},1}^{\text{old}}, a_{\text{pk},2}^{\text{old}}$. Amongst others, used to prove ownership of $\boldsymbol{n}_1^{\text{old}}, \boldsymbol{n}_2^{\text{old}}$.
$\boldsymbol{n}_1^{\text{new}}, \boldsymbol{n}_2^{\text{new}}$	$\forall i \in \{1,2\}\colon \boldsymbol{n}_i^{\text{new}} = (a_{\text{pk},i}^{\text{new}}, v_i^{\text{new}}, \rho_i^{\text{new}}, \text{rcm}_i^{\text{new}})$ The content of the shielded notes about to be produced. $a_{\text{pk},i}^{\text{new}}$ is a public key identifying the receiver. v_i^{new} is the monetary value sent to the receiver. ρ_i^{new} is a unique identifier to avoid double-spending. $\text{rcm}_i^{\text{new}}$ is a random commitment trapdoor that is part of the commitment scheme.
$\text{enforceMerklePath}_1$, $\text{enforceMerklePath}_2$	A bit that indicates if the path in the commitment tree should be verified. Set to 0 for dummy notes.
$\text{path}_1, \text{path}_2$	The path of the respective commitment in the commitment tree.
$\text{pos}_1, \text{pos}_2$	The index of the respective commitment in the leaves of the commitment tree. The leaves are enumerated from left to right starting from 0. Empty leaves have the value 0.

Table 2: The secret inputs to $\pi_{\text{ZKJoinSplit}}$. The fields' names are the same ones as in the original [Hopwood *et al.*, 2023].

2.1 The Groth16 zk-SNARK

The Groth16 proving system provides a way to prove statements of the form $x \in L$ for some language $L \in \mathsf{NP}$. For Zcash, x would be the public input to $\pi_{\text{ZKJoinSplit}}$ marked with * in Table 1 and L is the statement "x is a valid JoinSplit according to the rules mentioned in Section 2". In this section, we will show informally how a zk-SNARK is produced from such a statement $x \in L$ by reducing L to another problem ARITH-SAT and then encoding ARITH-SAT as a so-called QAP for the creation of $\pi_{\text{ZKJoinSplit}}$.

$L \in \mathsf{NP}$, so there exists an efficient algorithm $A(x, w)$ such that

$$L = \{x \in \{0,1\}^*\colon \exists w \in \{0,1\}^*\colon A(x,w) = 0\}$$

For Zcash, w corresponds to the secret inputs to $\pi_{\text{ZKJoinSplit}}$ in Table 2. Now, let q be prime and $\mathbb{F}_q$ be a q-order finite field. For an arithmetic circuit[1] C, let $m+1$ be the number of wires of C, including inputs. Let l be the number of public inputs and w.l.o.g. assign wires 1 through l to them. For practical purposes, wire 0 is assumed to carry the value 1. Now, define ARITH-SAT to be the following language:

$$\text{ARITH-SAT} := \left\{ \begin{array}{r} \text{An arithmetic circuit } C \text{ and } x \in \mathbb{F}_q^l\colon \\ \exists w \in \mathbb{F}_q^{m-l}\colon C(x,w) = 0 \end{array} \right\}$$

$C(x, w)$ represents the output of C if wires are assigned the values in x and w. x is called the "primary input", w the "auxiliary input" or "witness". Denote the value of wire i, including both inputs and wire 0, by a_i. Then $x = (a_i)_{i=1}^l$ and $w = (a_i)_{i=l+1}^m$. ARITH-SAT is NP-complete (similarly to how 3-SAT is NP-complete), so $L \leq_p$ ARITH-SAT and $x \in L$ can be efficiently converted into a form $x_C \in$ ARITH-SAT.

For n multiplication gates in C, choose $r_1, \ldots, r_n$ pairwise distinct in $\mathbb{F}_q$.

Next, define a quadratic arithmetic program (QAP) Q as

$$Q := \left(\begin{array}{r} \{u_i \in \mathbb{F}_q[x]\colon i \in [m]\}, \{v_i \in \mathbb{F}_q[x]\colon i \in [m]\}, \\ \{w_i \in \mathbb{F}_q[x]\colon i \in [m]\}, t \in \mathbb{F}_q[x] \end{array} \right)$$

Set $t(x) := \prod_{i=1}^n (x - r_i)$. Q computes an arithmetic circuit C if $x_C = (C, x) \in$ ARITH-SAT $\iff$ there exist a_i such that

$$t(x) \mid \overbrace{\underbrace{\sum_{i=0}^{m} a_i u_i(x)}_{\text{left input}} \cdot \underbrace{\sum_{i=0}^{m} a_i v_i(x)}_{\text{right input}} - \underbrace{\sum_{i=0}^{m} a_i w_i(x)}_{\text{output}}}^{P_Q(x)}$$

The idea is that $P_Q(r_i) = 0$ corresponds to the statement "left input times right input minus output equals zero" for the i^{th} multiplication gate. Note that the inputs and output are actually not only mere wire values a_i, but linear combinations of many wire values. Practically, this means that every multiplication gate can subsume many addition and scalar multiplication gates (i.e., linear operations). The coefficients of these linear combinations are hardcoded into Q's polynomials using Lagrange interpolation or discrete Fourier transforms (see Section 3.2), for example.

Every multiplication gate is satisfied if and only if $r_1, \ldots, r_n$ are zeros of P_Q. This is given by divisibility by t. For full details on QAPs, see [Gennaro *et al.*, 2012].

[1] For an illustrative image of an arithmetic circuit, see [Gennaro *et al.*, 2012] Figure 1.

Before proving, we need to setup some shared information σ between prover and verifier called the *common reference string*. Let $(G_1, +), (G_2, +)$ be two cyclic q-order subgroups of points on certain elliptic curves with generators g_1, g_2, respectively. Zcash uses the BLS12-381[Barreto *et al.*, 2002] curve, so G_1, G_2 are chosen accordingly. For full detail, see [Hopwood *et al.*, 2023].

Sample randomly $\alpha, \beta, \gamma, \delta, \tau \in \mathbb{F}_q$. Define $\sigma := (\sigma_1, \sigma_2)$ as

$$\sigma_1 := \left\{ \begin{array}{c} \left(g_1 \frac{\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau)}{\delta}\right)_{i=l+1}^{m} =: \boldsymbol{\sigma}_{1,1}, \\ \left(g_1 \frac{\tau^i t(\tau)}{\delta}\right)_{i=0}^{n-2} =: \boldsymbol{\sigma}_{1,2}, \left(g_1 \tau^i\right)_{i=0}^{n-1} =: \boldsymbol{\sigma}_{1,3}, \\ g_1\alpha, g_1\beta, g_1\delta, \left(g_1 \frac{\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau)}{\gamma}\right)_{i=0}^{l} \end{array} \right\}$$

$$\sigma_2 := \left\{ g_2\beta, g_2\gamma, g_2\delta, (g_2\tau^i)_{i=0}^{n-1} =: \boldsymbol{\sigma}_{2,4} \right\}$$

Given Q, the prover now calculates $h \in \mathbb{F}_q[x]$ as

$$h(x) := \frac{\sum_{i=0}^{m} a_i u_i(x) \cdot \sum_{i=0}^{m} a_i v_i(x) - \sum_{i=0}^{m} a_i w_i(x)}{t(x)} \tag{2}$$

and obtains the vector of coefficients of $h(x)$ denoted by $\boldsymbol{h}$. Then, the proof π (or, more concretely, in our case $\pi_{\text{ZKJoinSplit}}$) for the statement $x \in L$ is $\pi := (g_1 A =: \pi_A, g_2 B =: \pi_B, g_1 C =: \pi_C)$, where, given randomly sampled $r, s \in \mathbb{F}_q$,

$$A := \alpha + \sum_{i=0}^{m} a_i u_i(\tau) + r\delta \tag{3}$$

$$B := \beta + \sum_{i=0}^{m} a_i v_i(\tau) + s\delta \tag{4}$$

$$C := \frac{\sum_{i=l+1}^{m} a_i(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau)) + h(\tau)t(\tau)}{\delta} + As + Br - rs\delta \tag{5}$$

Given the evaluations of $(g_1 u_i(\tau))_{i=0}^{m}$, $(g_1 v_i(\tau))_{i=0}^{m}$, and $(g_2 v_i(\tau))_{i=0}^{m}$, which can be computed as they are composed of the elements of $\boldsymbol{\sigma}_{1,3}$ and $\boldsymbol{\sigma}_{2,4}$, the prover can evaluate the elements using σ as follows:

$$g_1 A = g_1\alpha + \sum_{i=0}^{m} (g_1 u_i(\tau)) a_i + (g_1\delta) r \tag{6}$$

$$g_1 B = g_1\beta + \sum_{i=0}^{m} (g_1 v_i(\tau)) a_i + (g_1\delta) s \tag{7}$$

$$g_2 B = g_2\beta + \sum_{i=0}^{m} (g_2 v_i(\tau)) a_i + (g_2\delta) s \tag{8}$$

$$g_1 C = \boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^{m} + \boldsymbol{\sigma}_{1,2}\boldsymbol{h} + (g_1 A)s + (g_1 B)r + (g_1\delta)(-rs) \tag{9}$$

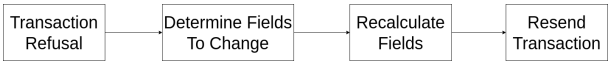

Figure 1: Steps involved in resending a transaction.

Finally, π is verified by using a pairing with G_1 and G_2 as source groups, in Zcash the optimal ate pairing[Vercauteren, 2008], specifically.

3 Resending Shielded Transactions

We assume the series of events in Figure 1: a Zcash full node creates and sends a transaction containing a JoinSplit description to its peers. At some later point, it learns that the transaction has not been included into the blockchain. The node wants to resend the transaction: it checks what fields need to change, recalculates these and finally resends the transaction. This process can be repeated until the transaction is in the blockchain.

We assume that the node is capable of sending and resending to a sufficient number of peers, such that failure of the network never prevents the transaction's inclusion. Hence, we will not deal with the first and the last step of Figure 1. A node can assume that its transaction has been rejected by waiting for the transaction's inclusion and then for its confirmation. If the node does not find its transaction in its local blockchain after some time, the transaction has been rejected. This can happen for insufficient transaction fees, for example. If it is included, but not fully confirmed by a sufficient number of blocks, it has been rejected as well. This can happen for a chain reorganization. Once it is certain that a transaction has been rejected, it must be checked what fields of the JoinSplit must change to maximize the chances of inclusion. All JoinSplit fields, which include public inputs to $\pi_{\text{ZKJoinSplit}}$, as well as all secret inputs are listed in two tables in Table 1 and 2.

Once it is clear what fields must change, the sender must recalculate them and all other related wire values in the circuit. Based on the new wire values, a new proof $\pi'_{\text{ZKJoinSplit}}$ is calculated. Let I_{changed} be the set of indices of coefficients a_i that may change. Denote the new coefficients by a'_i, where $a'_i \neq a_i \implies i \in I_{\text{changed}}$. We want to find relationships between the calculations made for $\pi_{\text{ZKJoinSplit}} := (\pi_A, \pi_B, \pi_C)$ and those made for $\pi'_{\text{ZKJoinSplit}} := (\pi'_A, \pi'_B, \pi'_C)$ such that a maximal amount from the former can be reused for the latter.

3.1 Finding I_{changed}

The set I_{changed} should be kept as small as possible; for the JoinSplit circuit, $l = 9$ and $m = 1956949$, so I_{changed} should be a small percentage of m. As we will see in Section 3.2, this allows for the biggest gains in performance since much of the previous proof can be reused. Indeed, with reasonable assumptions, the two cases mentioned in the introduction allow I_{changed} to be small.

By far the biggest number of wires, about 80% of the circuit, is involved in the verification of the Merkle paths for the input notes as briefly described in Section 2: the hash function used for the Merkle tree is the SHA-256 compression function, which needs many wires. Should any input change that

this computation depends on such as path_1 or path_2, then all involved wires need to be recalculated and I_{changed} becomes huge. Otherwise, it will stay small.

First, we consider the case of changing transaction fees sent to a transparent address. Anyone sending a transaction pays a small fee to miners in order to incentivize the transaction's inclusion in a block. Such fees are paid implicitly by keeping the transaction outputs too low: if the sum of input values to a transaction exceeds the sum of output values, then the miner can take ownership of the difference. However, if the transaction fee is set too low, the miner might decide to give preference to other transactions with higher fees. Hence, the goal is to choose the minimal fee such that the transaction is included, but not too much money is spent on that either. If the spender finds the transaction to be rejected, it must be resent with a higher transaction fee.

Paying transaction fees from a JoinSplit description happens by setting $v_{\text{pub}}^{\text{new}}$ to some non-zero value. This amount is then available to the transparent outputs of the containing transaction. They can spend some of it and the remainder can later be claimed by the miner. This last step is out of the control of the JoinSplit description, however, so the fundamental mechanism to set and change a transaction fee paid from a JoinSplit transaction is the adjustment of $v_{\text{pub}}^{\text{new}}$.

Changing $v_{\text{pub}}^{\text{new}}$ affects the balance equation (1) in Section 2, verified by $\pi_{\text{ZKJoinSplit}}$. Increasing $v_{\text{pub}}^{\text{new}}$ necessitates a decrease in either the left-hand side or $v_1^{\text{new}} + v_2^{\text{new}}$. We have $v_{\text{pub}}^{\text{old}} = 0$ (see Table 1) and changing $v_1^{\text{old}} + v_2^{\text{old}}$ would imply a lot more changes because of the tight link to $\boldsymbol{n}_1^{\text{old}}, \boldsymbol{n}_2^{\text{old}}$, so we confine ourselves to changing $v_1^{\text{new}} + v_2^{\text{new}}$. Assuming in addition that one of v_1^{new} and v_2^{new} is sufficient, w.l.o.g. we choose v_1^{new}. In addition, v_1^{new} is included in $\boldsymbol{n}_1^{\text{new}}$, so cm_1^{new} must be recalculated.

In total, this makes for a set I_{changed} of size

$$|I_{\text{changed}}| = \underbrace{1}_{v_{\text{pub}}^{\text{new}} \text{ in } \mathbb{F}_q} + \underbrace{64}_{v_{\text{pub}}^{\text{new}} \text{ in binary}} + \underbrace{64}_{v_1^{\text{new}} \text{ in binary}} + \underbrace{51510}_{\text{SHA-256 calculation of } \text{cm}_1} = 51639$$

which is only $\approx 2.6\%$ of all wires.

The second scenario we would like to consider is a blockchain reorganization, frequently called a "reorg". Every node stores in reality not a chain but a tree, since for each height in the blockchain, mined blocks may be proposed by multiple nodes. The active blockchain is a path from the tree's root to one of its leaves and is the path with the largest accumulated difficulty: each block has an associated difficulty value indicating the effort a miner has to put into mining the block. It happens that a node receives a mined block that is at a lower height than the tip of its active blockchain. This new block now is a branch off the active blockchain. If this branch becomes active after some time because yet other blocks are added and it ends up having higher accumulated difficulty, a blockchain reorganization happens: all blocks that are in the old active chain but not in the new one become stale and its JoinSplits are invalidated. Because of that invalidation, a user might want to resend a JoinSplit.

Whether $\pi_{\text{ZKJoinSplit}}$ needs to be recalculated or not depends on how $\text{rt}^{\text{Sprout}}$ was chosen. If $\text{rt}^{\text{Sprout}}$ refers to a commitment tree at some previous blockheight and it is before the branch, no change is needed since $\text{rt}^{\text{Sprout}}$ refers to a tree far enough in the past. If it is contained in the branch, $\text{rt}^{\text{Sprout}}$ changes and so do $\text{path}_1, \text{path}_2$, which requires recalculation of the path verification, dramatically increasing $|I_{\text{changed}}|$. Therefore, $\text{rt}^{\text{Sprout}}$ should always be chosen far enough in the past. If $\text{rt}^{\text{Sprout}}$ refers to a commitment tree at some other JoinSplit in the same transaction, $\text{rt}^{\text{Sprout}}$ of the first JoinSplit which actually refers to a past block must be checked as described above. In any case, either $\text{rt}^{\text{Sprout}}$ changes and the optimizations provide practically no advantage, or $\pi_{\text{ZKJoinSplit}}$ can be fully reused.

3.2 Optimizing The zk-SNARK's Recalculation

The steps involved in the calculation of any proof as described in Section 2.1 are

1. Circuit synthesis; given the public and secret inputs of a JoinSplit, all wire values a_i must be found,
2. calculation of h in (2), its coefficients $\boldsymbol{h}$ in particular, and
3. the calculation of the proof elements π_A, π_B, π_C.

Circuit synthesis aims to calculate all wire values a_i that are neither public nor secret inputs such as the intermediate results of the repeated application of SHA-256 to verify that $\text{path}_1, \text{path}_2$ are correct.

Calculating h involves operations on polynomials. To avoid a time complexity of $\mathcal{O}(d^2)$ for d-degree polynomials in coefficient representation for multiplication and division, the polynomials are converted to point representation and operations are performed on the points in linear time. That conversion is done quickly using fast Fourier transforms (FFT), which computes discrete Fourier transforms in sub-quadratic time. For a more detailed description of this method, see [Cormen *et al.*, 2009].

The proof elements are calculated as described in (6) through (9). The main operation here is *multiexponentiation*: given an integer k and base values $b_1, \ldots, b_k$ and exponents $e_1, \ldots, e_k$, compute $b_1^{e_1} b_2^{e_2} \cdots b_k^{e_k}$. The sum $\sum_{i=0}^{m} (g_1 u_i(\tau)) a_i$ in (6) is one example of this operation (we chose additive notation for G_1, G_2, so here, multiexponentiation actually is a dot product between a vector of points on an elliptic curve and a vector of scalars).

Synthesis can be optimized by only recalculating a_i' for $i \in I_{\text{changed}}$ and copying all other a_i.

A similar technique of substitution can be applied to π_A and π_B as well as $g_1 B$ and $\boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^{m}$ in π_C. Given π_A, π_B, $g_1 B$, and $\boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^{m}$ in (9), we want to remove all a_i with $i \in I_{\text{changed}}$ (or $i \in I_{\text{changed}} \cap \{l+1, \ldots, m\}$ for $\boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^{m}$) and replace them with a_i'. Define $\pi_A^- := g_1 \sum_{i \in I_{\text{changed}}} a_i u_i(\tau)$. Then, for π_A', this replacement is fulfilled by the calculation

$$\pi_A - \pi_A^- + g_1 \sum_{i \in I_{\text{changed}}} a_i' u_i(\tau) =: \pi_A'$$

as can be seen from the definition of A in (3). We have created π'_A by doing a multiexponentiation using the indices in $i \in I_{\text{changed}}$ instead of the full range of indices. If I_{changed} is sufficiently small, this may lead to a performance boost. By defining "subtraction terms" like π_A^- for π_B, g_1B, and $\boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^m$, very similar calculations can be performed for these other values as well as can be seen from (4) and (5).

However, by changing $\pi_A = g_1A$ and g_1B, we must recalculate $(g_1A)s + (g_1B)r$ in π_C. This requires knowledge of r and s. These would need to be stored for each sent transaction for potential resending until its full confirmation, at which moment r and s can be dropped.

For the multiexponentiation $\boldsymbol{\sigma}_{1,2}\boldsymbol{h}$, such a simple optimization cannot be performed. If there exists any i such that $a_i \neq a'_i$, the coefficients of the product polynomial $\sum_{i=0}^m a_i u_i(x) \sum_{i=0}^m a_i v_i(x)$ in (2) may all change, which means that all coefficients in $\boldsymbol{h}$ are also affected. Therefore, there is no clear, exploitable relationship between the wire values a_i and $\boldsymbol{h}$, unlike the structure of π_A, for example, where a changed a_i affects π_A in a very predictable way.

For similar reasons, we did not optimize the computation of h either. We did not find a relationship between a change in a_i and the results of the FFTs used.

3.3 Practicality Of The Optimizations

In order to understand the relative usefulness of speeding up each computation, proof creation of $\pi_{\text{ZKJoinSplit}}$ without our optimizations was benchmarked. The used system was a 11th Gen Intel(R) Core(TM) i7-1165G7 CPU at 2.80 GHz with 16 GiB of RAM. Most calculations are parallelized, and if they are, it was made sure that calculations do not overlap as they do, however, in the original implementation. The obtained measurements were the following:

Calculated Data	Time (in ms)
Synthesis (finding all a_i)	754
$\boldsymbol{h}$ from h in (2)	3564
$\boldsymbol{\sigma}_{1,2}\boldsymbol{h}$ in (9)	13795
$\boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^m$ in (9)	1398
$\sum_{i=0}^m (g_1 u_i(\tau)) a_i$ in (6)	1396
$\sum_{i=0}^m (g_1 v_i(\tau)) a_i$ in (7)	576
$\sum_{i=0}^m (g_2 v_i(\tau)) a_i$ in (8)	1675

The multiexponentiation $\boldsymbol{\sigma}_{1,2}\boldsymbol{h}$ and the calculation of h take by far the longest time. Unfortunately, as explained in Section 3.2, we have not found a viable optimization for these operations. This severely limits the usefulness of our optimizations in the real world.

The values needed to store after sending a transaction and before notification of full confirmation are r, s and π_A^-, as well as the similar values for π_B, g_1B, and $\boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^m$. Choosing the encoding of elements in G_1 and G_2 used for Zcash [Hopwood *et al.*, 2023] any element in G_1 takes 48 bytes and any element in G_2 takes 96 bytes. $q \approx 2^{255}$, so 32 bytes suffice to represent any value in $\mathbb{F}_q$.

This makes for a total additional storage requirement in bytes per transaction of

$$\underbrace{2 \cdot 32}_{r,\, s \,\in\, \mathbb{F}_q} + \underbrace{3 \cdot 48}_{\pi_A^- \,\in\, G_1 \text{ and similar values for } g_1B \text{ and } \boldsymbol{\sigma}_{1,1}(a_i)_{i=l+1}^m} + \underbrace{96}_{\text{The equivalent of } \pi_A^- \text{ for } \pi_B \text{ in } G_2} = 304$$

4 Conclusion

We have provided some ways to optimize the recalculation of a zk-SNARK in previously sent JoinSplit descriptions. They incur additional temporary storage requirements per transaction that should not be dramatic for systems capable of running full Zcash nodes. Furthermore, the optimizations are fully backwards-compatible such that any user may decide to apply them or not, leaving all other users unaffected.

Even though no measurements of the optimizations have been made, we believe that the improvements in time are minor, given that the most expensive computation has not been optimized. Additionally, creating new JoinSplit transactions is not supported anymore by recent Zcash nodes (zcashd v5.5.1).

5 Future Work

Most importantly, the theoretical optimizations have to be implemented. Then, a performance comparison between resending a transaction from scratch and using the optimizations is required, preferably for different real-world scenarios.

More generally, there are potentially other optimizations that can be made when recalculating the proof. While this report only dealt with fully backward-compatible changes, perhaps there are optimizations that require changes of the CRS or the content of JoinSplit descriptions, which would break backward compatibility. Furthermore, more recent versions of Zcash support other transaction types such as Spend and Output since Sapling, which are still proved using Groth16 but come with their own circuits. I_{changed} would need to be adjusted to these more recent circuits. Other scenarios than the two treated in this report may be examined as well, which would require further adjustment of I_{changed}.

References

[Barreto *et al.*, 2002] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. Cryptology ePrint Archive, Paper 2002/088, 2002. https://eprint.iacr.org/2002/088.

[Ben-Sasson *et al.*, 2014] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 781–796, USA, 2014. USENIX Association.

[Bowe, 2020] Sean Bowe. The halo2 Book. https://zcash.github.io/halo2/, 2020. Accessed: 2023-04-07.

[Cormen *et al.*, 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Gennaro *et al.*, 2012] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. Cryptology ePrint Archive, Paper 2012/215, 2012. https://eprint.iacr.org/2012/215.

[Goldwasser *et al.*, 1985] 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, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[Groth, 2016] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[Hopwood *et al.*, 2023] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. https://zips.z.cash/protocol/protocol.pdf, 2023. Accessed: 2023-04-07.

[Merkle, 1988] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[Nakamoto, 2008] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008. Accessed: 2023-04-07.

[Reid and Harrigan, 2011] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 1318–1326, 2011.

[van Saberhagen, 2013] Nicolas van Saberhagen. Cryptonote v 2.0. https://www.getmonero.org/resources/research-lab/pubs/whitepaper_annotated.pdf, 2013. Accessed: 2023-04-07.

[Vercauteren, 2008] F. Vercauteren. Optimal pairings. Cryptology ePrint Archive, Paper 2008/096, 2008. https://eprint.iacr.org/2008/096.