Fine-Tuning Decentralized Anonymous Payment Systems based on Arguments for Arithmetic Circuit Satisfiability

Kamil Kluczniak and Man Ho Au

Department of Computing, The Hong Kong Polytechnic University
kamil.kluczniak@polyu.edu.hk, man-ho-allen.au@polyu.edu.hk

Abstract

Digital currencies like Bitcoin and other blockchain based systems provide means to record monetary transfers between accounts. In Bitcoin like systems transactions are published on a decentralized ledger and reveal the sender, receiver and amount of a transfer, hence such systems give only moderate anonymity guarantees.

Payment systems like ZCash attempt to offer much stronger anonymity by hiding the origin, destination and value of a payment. The ZCash system is able to offer strong anonymity, mainly due to use of Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (ZK-SNARK) of arithmetic circuit satisfiability. One drawback of ZCash is that the arithmetic circuit is rather large, thus requires a large common reference string and complex prover for the ZK-SNARK. In fact, the memory and prover complexity is dominated by the ZK-SNARK in use and is mainly determined by the complexity of the circuit.

In this paper we design a Decentralized Anonymous Payment system (DAP), functionally similar to ZCash, however with significantly smaller arithmetic circuits, thus greatly reducing the memory and prover complexity of the system. Our construction is based on algebraic primitives, from the realm of elliptic curve and lattice based cryptography, which satisfiability might be efficiently verified by an arithmetic circuit.

1 Introduction

Bitcoin and other so called digital currencies, at time of writing this article, have got much attention by media, industry and academia. At the core of these digital currencies are distributed ledgers which realize a tamper resistant append only list structure. Additionally, parties sustaining the distributed ledger may perform some additional verification of the data appended on the list. We may view such protocol simply as a distributed state machine where every party involved may update the state according to globally accepted rules, and keep a replica of the current state and the history of previous states and inputs to the machine. The fact, that all nodes in the network keep the history of states and inputs gives the ability to verify the consistency of the current state to parties which dynamically join and leave the protocol. In a digital currency users are given the possibility to create accounts on a distributed ledger and register monetary transactions. In Bitcoin for instance, an account is identified by a public key of a signature scheme and is associated with some value. Users might exchange value simply by signing the amount they wish to send and the recipient address. When nodes of the network obtain a transaction as above, they verify whether the signature is correct and whether the sender has enough value on his account. Then the system changes the state by subtracting the value from the senders' account and advancing the receivers account. Obviously, this change in state is then replicated among all nodes in the system. For clarity, we omit here animosities related to realizing the distributed ledger, consensus or the network layer. What is worth noting however, is that every participant, not only needs to posses a replica of the current state, but also the history of all states in order to verify the consistency of the transaction record.

Since the public keys on a distributed ledger, such as Bitcoin, have no tight link with their legal identity and can be created "on-the-fly", it has become a popular claim that Bitcoin and other digital currencies, offer strong privacy. However, it is easy to see that such claims are actually not strongly grounded in reality. In Bitcoin type currencies a transaction does not hide the values of the transactions nor the public keys of the sender or recipient. In fact methods for tracing Bitcoin transactions are becoming more advanced and effective [47, 3, 48, 42]. Therefore, most digital currencies offer only moderate privacy guaranties, sometimes called pseudonymity.

There have been proposals of digital currencies which offer much stronger privacy guarantees. One line of solutions is to use mixing services, which in essence operate very similarly to mix nets, or onion routing. Such approaches have been applied in protocols [41, 51, 50] and incorporated in some digital currencies like for instance Dash. However, the very idea requires to involve third parties or additional distributed infrastructure which results in less scalable systems and higher operational costs.

Another approach is to design a payment system which by design does not reveal anything except the correctness of the transaction. We will call such systems Decentralized Anonymous Payment systems (DAP)[1]. In principle, DAPs are supposed to offer the same functionality as Bitcoin type payment systems, however without revealing any additional information about a transaction despite that a transaction happened and is correctly formed. In particular, the goal is to hide the sender and recipient addresses, and the amount of the transfer. On the other hand, we want to have a way to verify the consistency of such transfers and ensure a correct update of the current state of the ledger. In this approach a transaction does not reveal the sender or recipient public keys in

[1]The terminology in literature is inconsistent. For instance in [44] such systems called Ring Confidential transactions. In this article we adopt the terminology from [53], since its the closest to what this article is about.

contrast to Bitcoin, but instead exploits the properties of zero-knowledge proofs to provide consistency of the transaction while simultaneously hiding the sender and recipient addresses, and values transferred between them. One notable protocol called RingCT [44] and based on CryptoNote [52], has been implemented by the Monero digital currency. The protocol makes use of some clever tricks with Schnorr type Σ-protocols [25] for proving statements about linear relations of discrete logarithms. In short, the RingCT protocol uses so called one-time ring signatures, i.e. the signer produces a signature of knowledge that he has a private key related to one of the public keys in a given set. The signature is one-time in the sense that attempts to use the same secret signing key will lead to the same serial number what can be detected. Furthermore, the coin public keys for the ring signature are created in a Diffie-Hellman manner, where, in contrast to the Diffie-Hellman protocol, only the recipient can compute the secret key. Since, the protocol applies Schnorr type Σ-protocols for discrete logarithm statements and proofs of set membership, the size of the signature depends linearly on the size of the ring, i.e. the anonymity set. This is a serious obstacle since the protocol does not scale well and in practice the anonymity set includes 2 to 4 coins. Several proposals have been made to improve RingCT in order to make the protocol scale better. For instance [32] proposed a set membership proof of logarithmic size. In [54] the authors design a constant size proof, but use pairing based accumulators to accumulate integers. The asymptotic proof size from [54] is very promising, however system parameters grow linearly in the maximal size of the anonymity set. Moreover, in order to instantiate the system in practice the parameters for the system need to be relatively large, since the accumulator needs to accumulate group elements (integers) in which the discrete logarithm is hard. Another obstacle is hiding the mounts of a transaction, which is realised by commitment schemes and expensive, in terms of size, range proofs.

To address the scalability problem, Ben-Sasson et al. [53] introduce Zerocoin, where the public parameters depend only logarithmically on the anonymity set and the size of a transaction is constant, i.e. depends only on the security parameter. The Zerocoin protocol relies on Zero-Knowledge Succinct Non-interactive ARgument of Knowledge (ZK-SNARK) for general statements instead of Schnorr type Σ-protocols. Briefly, Non-Interactive Zero-Knowledge proofs (NIZK), produce a proof of the truth of a statement without revealing any other information, in particular not revealing any information about the witness. When a NIZK is a proof of knowledge, we additionally assume there is a (hypothetical) extractor which can extract a witness satisfying the statement. Since, there exists such an extractor we belief that the prover must have known the witness. We call a system an argument system instead of poof system, if the soundness or the existence of the extractor is conditioned on some computational assumptions. Finally, a ZK-SNARK is a non-interactive zero-knowledge argument of knowledge which is additionally succinct, what means that the size of the argument is polylogarithmic in the size of the statement [10, 30, 11, 13]. Practical ZK-SNARKs [29, 12, 26, 33, 31] and implementations of those systems [45, 6, 7, 5] allow to proof statements about satisfiability of arithmetic circuits and the size of the argument is a few group elements. For instance the argument size of the system from Parno et al. [45] is only 8 group elements. Later improvements resulted only 3 groups elements [31, 33]. Succinctness of the arguments make ZK-SNARKs especially attractive for applications like distributed ledgers. The major drawbacks of practical ZK-SNARKs are related to their reliance on a common reference string (CRS). First of all, the CRS has to be generated by a trusted party. In practice, this may be accomplished by a distributed setup process [21, 22]. Second, the size of the common reference string and the prover complexity depend on the complexity of the circuit. Moreover, the security assumptions are not falsifiable and the strength of the assumptions also depends on the circuit complexity. In other words, the more complex the circuit, the bigger the system parameters, the more complex the prover and the weaker the soundness guarantees of the argument system. Therefore it is crucial to have a relatively small circuit.

In Zerocoin [53] the authors exploit the succinctness and the generality of the argument system, and give a generic construction of a Decentralized Payment System (DAP). The generic construction is build from collision-resistant hash functions, commitment schemes and pseudorandom functions. Later these building blocks may be instantiated by any primitive satisfying the requirements. In Zerocoin and later ZCash [36], these building blocks are instantiated by standard symmetric primitives like the SHA256 hash function. The ZK-SNARK then proofs that the transaction is consistent and that the user knows all the secret keys, without revealing any additional information. The ZK-SNARK in use in ZCash is based on [7] and shows satisfiability of arithmetic circuits. Most of the statements circuitry in the ZCash protocol checks satisfiability of boolean operations, hence the arithmetic circuit for the ZK-SNARK needs to realize boolean operations what is quite expensive in terms of the circuit size. Therefore, in order to reduce the circuit size, it seems to be reasonable to redesign the protocol such that its build from primitives which can directly be implemented by the arithmetic circuit of the ZK-SNARK.

Our Contribution. We propose a semi-generic construction of a Decentralized Anonymous Payments (DAP) scheme. Our construction relies on similar design principles as ZCash [53, 36], however we do not follow the ZCash generic construction faithfully. The main difference lies in the way we construct addresses, coins and their serial numbers. Below we give an (over) simplified description of our system. In our case an address is a single group element $X = g^x \in \mathbb{G}$, where $x \in \mathbb{Z}_q$ is the address secret key. Then a coin is build also from a coin public key $pk = g^x g_1^a g_2^b$, where a will be used to compute the serial number and b is an additional blinding, and a commitment of the coins value. The coin public key is created by a payer, who chooses the secret masks $a, b \in \mathbb{Z}_p$. In order to "bind" and compress the commitment and the coin public key, a collision-resistant hash of these values is published on the ledger. Double spending is prevented by computing a serial number of the coin public key. In our case the serial numbers are computed using the Dodis-Yampolskiy PRF

[27], i.e $sn = g^{1/x+a}$. Similarly, as in ZCash, in our scheme a spender creates a non-interactive zero-knowledge argument of knowledge that, his coin is well formed, he knows all secrets of the coin, the serial number if well formed and the coin is in a hash tree. Moreover, the spender also has to create a new coin with a public key as described above and a commitment of its value. We describe our system for two input coins and two output coins, thus the non-interactive argument also needs to show that the values of the coins are consistent.

Our scheme is semi-generic. In particular, we may instantiate commitment schemes, hash functions and hash trees with different building blocks. A significant part of our contribution includes two instantiations from discrete logarithm based primitives and lattice based primitives.

To instantiate discrete logarithm based primitives we follow the suggestions given by Kosba et al. in [38], and instantiate the underlying group on twisted Edwards curves. Moreover, we apply a few optimisations which result in smaller arithmetic circuits like scalar multiplication using the windowed method when the based point is known. Using the windowed method was suggested at a blog post [58] with a window of size $w = 4$. We show that actually the optimal window size is $w = 3$. This is because the cost of multiplexing precomputed elements grows much faster. Our construction also requires to perform general scalar multiplications on unknown points. In this case we cannot precompute a table for the windowed method, therefore we provide a few optimisations for general scalar multiplication by exploiting doubling formulas with slightly smaller arithmetic circuits. As mentioned earlier, the membership proofs are usually done by showing that a coin is in a hash tree. Previous work [5, 53], focused mostly on binary Merkle trees. In our work we investigate whether 3-ary or 4-ary hash trees may give better performance, and we found that 3-ary trees result in smaller arithmetic circuits.

Additionally, we provide an instantiation on lattice based primitives. In particular, we instantiate the hash functions on the Ajtai hash [1]. The use of Ajtai hash was first suggested by Ben-Sasson et al. in [5], where the authors build Merkle trees. However, the parameter choices for collision-resistance made in [5] were further questioned by Kosba et al. in [38], who provide additional analysis for the Ajtai hash. In our case we complement the analysis given in [38] with additional analysis using combinatorial methods suggested for lattice based hash functions in [43]. Beside collision-resistant hash functions, we additionally investigate the use of Ajtai hash for building statistically-hiding commitment schemes. In [37] Kawachi et al. give a commitment scheme which is essentially the Ajtai hash where half of the input bits are the randomness bits and the other half is are the bits of the message. The authors show that if the hash input is big enough relative to the output, then the distribution of the output bits are statistically close to uniform. In our setting the message space of the commitment scheme is small. Therefore, using directly the constraints on the parameters from [37] would result in a bigger arithmetic circuit than necessary. For our construction we set the number of bits in the message, and we extend the randomness space of the randomness to "blur" the committed message.

Finally, we evaluate our instantiation in terms of size of the arithmetic circuit for the argument system. In order for our results to be accountable, we show the design of the arithmetic circuits and calculate their cost.

Concurrent work. We are aware that some parallel work on a similar system is in progress by the ZCash team. However, little is known by far about the details. At a blog post [57], the ZCash team announced a major update codenamed "Sampling", which aims to improve performance and usability of the payment system. In a series of blog posts [17, 20, 19] Sean Bowe described the BLS12-381 curve which will be used to instantiate the ZK-SNARK [31] for the "Sampling" update. In a later blog post [18] by Sean Bowe, we learn that the new system for the arithmetic circuit will not be instantiated by symmetric primitives, but with elliptic curve based primitives. Additionally, there is an explainer page given [58] with the description of a twisted Edwards curve called Jubjub. The explainer page announces plans of using windowed exponentiation at window size $w = 4$ and hash functions based on the Pedersen commitment. At this point however, we don't know whether the designed system will follow the generic construction from [53], and, if so, what other building blocks (especially which PRF's) will be used.

In order to provide an easier comparison in the future, we evaluate the size of our circuits for the bit size of the field and group order over the Jubjub curve.

2 Preliminaries

In this section we give our denotations and recall the fundamental definitions we use throughout the article. Some definitions, however, are given in other sections where there will be needed. We denote as $x = [x_i \in \mathcal{S}]_{i=0}^{m-1}$ a string of elements x_i from the set $\mathcal{S}$. In particular we will denote as $x = [x_i \in \{0,1\}]_{i=0}^{m-1}$ a bit string of length m s.t. $x = \sum_{i=0}^{m-1} 2^i x_i$. When such a string in input to an algorithm, then we implicitly input its string representations, e.g. bit representation and not the corresponding integer, unless its clearly marked. Later $\mathbb{F}$ denotes a finite field and $\mathbb{F}_q$ denotes a finite field with q elements. We denote as $\mathbb{G}$ a cyclic group usually of order p. The operator $a||b$ denotes concatenation of two bit strings.

Definition 1 (Commitment Scheme)**.** *A commitment scheme consists of probabilistic polynomial-time algorithms $C = (\mathsf{CommSetup}, \mathsf{Comm})$ defined as follows:*

$\mathsf{CommSetup}(1^\lambda)$*: The key generation algorithm takes as input a security parameter 1^λ and outputs the commitment public key pk. The commitment public key determines the message space $\mathcal{M}$ and randomness space $\mathcal{R}$.*

$\mathsf{Comm}(pk, m, r)$*: The commitment algorithm takes as input the commitment public key pk, a message $m \in \mathcal{M}$ and randomness $r \in \mathcal{R}$, and outputs the commitment cm.*

Definition 2 (Hiding)**.** *A commitment scheme is hiding if it is infeasible to distinguish which message is in the commitment. More formally, for a commitment scheme $C = (\mathsf{CommSetup}, \mathsf{Comm})$ and an adversary $\mathcal{A}$, we definite the* *advantage $Adv^{Hide}_{C,\mathcal{A}}$ as follows.*

$$\left| \Pr\left[\begin{array}{l} pk \leftarrow \mathsf{CommSetup}(1^\lambda); (m_1, m_2) \leftarrow \mathcal{A}(pk); r \xleftarrow{R} \mathcal{R} : \\ \qquad \mathcal{A}(\mathsf{Comm}(pk, m_0, r)) \end{array}\right] - \Pr\left[\begin{array}{l} pk \leftarrow \mathsf{CommSetup}(1^\lambda); (m_1, m_2) \leftarrow \mathcal{A}(pk); r \xleftarrow{R} \mathcal{R} : \\ \qquad \mathcal{A}(\mathsf{Comm}(pk, m_1, r)) \end{array}\right] \right| \leq Adv^{Hide}_{C,\mathcal{A}}(1^\lambda)$$

We say that C is perfect hiding if for all $\mathcal{A}$ and λ, $Adv^{Hide}_{C,\mathcal{A}}(1^\lambda) = 0$. We say that C is computationally hiding if for all PPT adversaries $\mathcal{A}$, $Adv^{Hide}_{C,\mathcal{A}}(1^\lambda)$ is negligible in λ.

Definition 3 (Binding)**.** *A commitment scheme is binding if it is hard to find a collision, i.e. two message/randomness pairs which result in the same commitment. More formally, for a commitment scheme $C = (\mathsf{CommSetup}, \mathsf{Comm})$ and an adversary $\mathcal{A}$, we definite the advantage $Adv^{Bind}_{C,\mathcal{A}}$ as follows.*

$$\Pr\left[\begin{array}{l} pk \leftarrow \mathsf{CommSetup}(1^\lambda); (m_0, r_0, m_1, r_1) \leftarrow \mathcal{A}(pk) : \\ (m_0, r_0) \neq (m_1, r_1) \wedge \mathsf{Comm}(pk, m_0, r_0) = \mathsf{Comm}(pk, m_1, r_1) \end{array}\right] \leq Adv^{Bind}_{C,\mathcal{A}}(1^\lambda)$$

We say that C is computationally binding if for all PPT adversaries $\mathcal{A}$, $Adv^{Bind}_{C,\mathcal{A}}(1^\lambda)$ is negligible in λ.

Definition 4 ((Keyed) Hash Function)**.** *A (Keyed) Hash Function consists of algorithms $H = (\mathsf{Setup}_H, \mathsf{H}_{pk})$ defined as follows:*

$\mathsf{Setup}_H(1^\lambda)$*: The key generation algorithm takes as input a security parameter 1^λ and outputs the hash public key pk. The public key determines the message space $\mathcal{M}$.*

$\mathsf{H}_{pk}(m)$*: The commitment algorithm takes as input the public key pk, a message $m \in \mathcal{M}$ and outputs the hash h.*

Definition 5 (Collision-Resistance)**.** *Similarly as for binding commitment schemes, a hash function if collision-resistant if it is hard to find collisions. For a (keyed) hash function $H = (\mathsf{Setup}_H, \mathsf{H}_{pk})$ and an adversary $\mathcal{A}$, we definite the the advantage $Adv^{CR}_{H,\mathcal{A}}$ as follows.*

$$\Pr\left[\begin{array}{l} pk \leftarrow \mathsf{Setup}_H(1^\lambda); (m_0, m_1) \leftarrow \mathcal{A}(pk) : \\ \qquad (m_0) \neq (m_1) \wedge \mathsf{H}_{pk}(m_0) = \mathsf{H}_{pk}(m_1) \end{array}\right] \leq Adv^{CR}_{H,\mathcal{A}}(1^\lambda)$$

We say that K is a collision-resistant (keyed) hash function (CRHF) if for all PPT adversaries $\mathcal{A}$, $Adv^{CR}_{H,\mathcal{A}}(1^\lambda)$ is negligible in λ.

Definition 6 (Pseudorandom Function)**.** *Let $PRF : K \times M \to R$ be a efficiently computable function, where K is the key space, M is the domain space and R is the range. For an adversary $\mathcal{A}$ define the advantage of braking the PRF as follows.*

$$|\Pr[k \xleftarrow{R} K; \mathcal{A}^{PRF(k,.)} = 1] - \Pr[\mathcal{A}^{\mathcal{O}(.)} = 1]| \leq Adv^{PRF}_{\mathcal{A}}(\lambda),$$

were the oracle $\mathcal{O}$ on input a massage from M chooses a uniformly random output and saves the output for future queries on the same message.

We say that the the function PRF is a pseudorandom function if $Adv_{\mathcal{A}}^{PRF}(\lambda)$ is negligible in the security parameter.

Non-interactive zero-knowledge arguments of knowledge. We recall the definition of non-interactive zero-knowledge arguments of knowledge from [31]. A NP-relation $R(x, \omega)$ is a binary relation which can be tested in polynomial time, i.e. having x and ω we can verify whether $R(x, \omega) = 1$ in polynomial time. Let $L_R = \{x : \exists \omega \ R(x, \omega) = 1\}$ denote the language defined by R. We define a relation generator $\mathcal{R}_\lambda$ which on input 1^λ outputs a relation R from the set of possible relations.

Definition 7. *An non-interactive proof system $\Pi_R = (\mathsf{SetupCRS}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim})$ for an NP-relation $R(x, \omega)$ is defined by the following PPT algorithms.*

$\mathsf{SetupCRS}(R, 1^\lambda)$: *This procedure given a relation R and security parameter λ, outputs a common reference string crs and a simulation trapdoor τ.*

$\mathsf{Prove}(R, \mathsf{crs}, x, \omega)$: *The prover takes as input the relation R, a common reference string crs and $(x, \omega) \in R$, and returns an argument π.*

$\mathsf{Verify}(R, \mathsf{crs}, x, \pi)$: *The verification algorithm takes as input the relation R, the common reference string crs, the statement x and argument π, and outputs 0 (reject) or 1 (accept).*

$\mathsf{Sim}(R, \tau, x)$: *The simulator takes as input the relation R, the trapdoor τ and statement x, and outputs an argument x.*

Definition 8. *Perfect Completeness Completeness says that, given any true statement, an honest prover should be able to convince an honest verifier. For all $\lambda \in \mathbb{N}$, $R \in L_R$, $(x, \omega) \in R$*

$$\Pr[(\mathsf{crs}, \tau) \leftarrow \mathsf{SetupCRS}(R, 1^\lambda); \pi \leftarrow \mathsf{Prove}(R, \mathsf{crs}, x, \omega) : \mathsf{Verify}(R, \mathsf{crs}, x, \pi)] = 1.$$

Definition 9. *Perfect zero-knowledge. An argument is zero-knowledge if it does not leak any information besides the truth of the statement. We say Π_R = $(\mathsf{SetupCRS}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim})$ is perfect zero-knowledge if for all $\lambda \in \mathbb{N}$, $(R, z) \leftarrow R(1^\lambda)$, $(x, \omega) \in R$ and all adversaries $\mathcal{A}$.*

$$\begin{aligned} &\Pr\left[\begin{array}{c} (\mathsf{crs}, \tau) \leftarrow \mathsf{SetupCRS}(R, 1^\lambda); \pi \leftarrow \mathsf{Prove}(R, \mathsf{crs}, x, \omega) : \\ \mathcal{A}(R, z, \mathsf{crs}, \tau, \pi) = 1 \end{array}\right] \\ = &\Pr\left[\begin{array}{c} (\mathsf{crs}, \tau) \leftarrow \mathsf{SetupCRS}(R, 1^\lambda); \pi \leftarrow \mathsf{Sim}(R, \tau, x) : \\ \mathcal{A}(R, z, \mathsf{crs}, \tau, \pi) = 1 \end{array}\right] \end{aligned}$$

Definition 10. *Computational Knowledge Soundness We say that an argument system has computation knowledge soundness if there exists an extractor which can extract a valid witness from the proof. More formally, let $\Pi_R = (\mathsf{SetupCRS},$*

Prove, Verify, Sim) *be an argument system and let* $\mathcal{X}_{\mathcal{A}}$ *be a non-uniform polynomial time extractor. We define the advantage for any non-uniform polynomial time adversaries* $\mathcal{A}$ *as*

$$\Pr\left[\begin{array}{c}(R,z)\leftarrow R(1^{\lambda});(\mathsf{crs},\tau)\leftarrow\mathsf{SetupCRS}(R);\\((x,\pi);\omega)\leftarrow(\mathcal{A}||\mathcal{X}_{\mathcal{A}})(R,z,\mathsf{crs}):\\(x,\omega)\notin R\wedge\mathsf{Verify}(R,\mathsf{crs},x,\pi)\end{array}\right]\leq Adv^{Ext}_{\mathcal{A},\Pi}(1^{\lambda}]).$$

We say the proof system Π_R *is computationally knowledge sound if* $Adv^{Ext}_{\mathcal{A},\Pi}(1^{\lambda}])$ *is negligible in the security parameter* λ*.*

Definition 11. *We call a non-interactive zero-knowledge argument of knowledge a ZK-SNARK if for all* $\lambda\in\mathbb{N}$, $R\in L_R$, $(x,\omega)\in R$, $(crs,\tau)\leftarrow\mathsf{SetupCRS}(R,1^{\lambda})$ *and honest executions* $\pi\leftarrow\mathsf{Prove}(R,\mathsf{crs},x,\omega)$, *we have that* $|\pi|=polylog(|x|)$, *i.e. the size of the proofs is bounded poly-logarithmically in the size of the statement.*

Quadratic Constraints. In this article we consider arithmetic circuits consisting of addition and multiplication gates over a finite field $\mathbb{F}_q$. We may view some of the publicly known wires of such arithmetic circuit as specifying a statement and all other wires as specifying the witness. The arithmetic circuit specifies a binary relation where a statement consisting of public wires and a witness consisting of all other wires belong to the relation only if these variables corresponding to the wires satisfy the arithmetic circuit. Throughout the paper we will revere to wires corresponding to the witness as variables, whereas to wires corresponding to the statement as constants.

We may represent the satisfiability problem of an arithmetic circuit as a system of equations. The equations will takes variables $x_0,\ldots,x_{n-1}\in\mathbb{F}$ and be of the form

$$(\sum_{i=0}^{n}x_ia_i)\cdot(\sum_{i=0}^{n}x_ib_i)=(\sum_{i=0}^{n}x_ic_i),$$

where $a_i,b_i,c_i\in\mathbb{F}$ for all $0\geq i\geq n$ and $x_n=1$.

Analogically, Ben-Sasson et al. [6], characterizes such systems using vector operation and calls such systems Rank-1 Constraint Systems (R1CS).

The arithmetic circuits we build in this paper are rather big and complicated. Thus we will build them from smaller procedures and develop convenient notation. We will usually denote variables which are on input to a procedure in bold e.g. $\mathbf{a}$. Additional variables which appear within a procedure are denoted with fraktur font e.g. $\mathfrak{a}$. Constants are denoted in plain. A procedure will have the following form.

Procedure-Name(Set of input variables: Short description of the relation):

1. Quadratic constrains in form of

$$(\sum_{i=0}^{n} x_i a_i) \cdot (\sum_{i=0}^{n} x_i b_i) = (\sum_{i=0}^{n} x_i c_i),$$

2. and subprocedures.

Cost: (Constraint number) constraints and (Variable number) additional variables.

We present our constraint system in Appendix A.

3 Decentralized Anonymous Payments

Definition 12. *A Decentralized Anonymous Payment system consists of algorithms* (Setup_{DAP}, $\mathsf{CreateAddress}_{DAP}$, $\mathsf{CreateCoin}_{DAP}$, Mint_{DAP}, Pour_{DAP}, Verify_{DAP}, $\mathsf{Receive}_{DAP}$) *defined as follows.*

$\mathsf{Setup}_{DAP}(1^\lambda)$*: On input a security parameter λ outputs the public parameters* pp_{DAP}.

$\mathsf{CreateAddress}_{DAP}(\mathsf{pp}_{DAP})$*: On input the public parameters* pp_{DAP}*, outputs address* addr *and address secret key* addrSK.

$\mathsf{CreateCoin}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{addr}, \mathsf{addrSK}, v)$*: On input the public parameters* pp_{DAP}*, an address* addr *and value v, this algorithm outputs a coin* coin*, a proof π and coin spending key* coinSK.

$\mathsf{Mint}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{coin}, v, \pi, L)$*: On input the public parameters* pp_{DAP}*, a coin* coin*, value v and a proof π, this algorithm appends* coin *to the ledger L, or outputs $\perp$.*

$\mathsf{Pour}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{coin}_1, \mathsf{coin}_2, \mathsf{coinSK}_1, \mathsf{coinSK}_2, \mathsf{addr}_1, \mathsf{addr}_2, v_1, v_2, v^{pub}, L)$*: This algorithm takes as input the public parameters* pp_{DAP}*, two input coins* coin_1*,* coin_2 *and the corresponding spending keys* coinSK_1*,* coinSK_2*, two output addresses* addr_1*,* addr_2*, output values v_1, v_2, the public value v^{pub} and the ledger L. The algorithm outputs two output coins* coin_1^{out}*,* coin_2^{out} *and their decodings* $\mathsf{coinDec}_1^{out}$*,* $\mathsf{coinDec}_2^{out}$*, and a transaction* tr_{POUR}.

$\mathsf{Verify}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{tr}_{POUR}, \mathsf{coin}_1, \mathsf{coin}_2, L)$*: On input the public parameters* pp_{DAP}*, a pour transaction* tr_{POUR}*, two coins* $\mathsf{coin}_1, \mathsf{coin}_2$ *and the ledger L, this algorithm appends* $\mathsf{tr}_{POUR}, \mathsf{coin}_1, \mathsf{coin}_2$ *to the ledger, or outputs $\perp$.*

$\mathsf{Receive}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{coin}, v, \mathsf{coinDec}, \mathsf{addr}, \mathsf{addrSK})$*: On input the public parameters* pp_{DAP}*, a coin* coin*, a value v, a coin decoding* $\mathsf{coinDec}$ *and an address* addr *and its secret key* addrSK*, this algorithm returns the spending key* coinSK*, or $\perp$.*

The Zerocash paper [53] defines security of a DAP system by three properties called ledger indistinguishability, transaction non-malleability and balance. Informally, ledger indistinguishability captures the fundamental privacy requirement, that is, that the DAP scheme reveals no information beyond what is publicly-revealed. In particular, a DAP scheme hides everything except the values of minted coins, public values and the total number of transactions. Note, that the public values are necessary in practice to pay transaction fees. In this case the adversary may sent coins to an address, and receive payments from an address. Transaction non-malleability requires that transaction data returned by a pour transaction cannot be altered by a third party. Finally, balance requires that the amount of value on the ledger is determined only by the mint transaction. In practice, that means that the pour transaction cannot create new value or delete existing value. Moreover, this property also captures the infeasibility of double spending (a double spend would create new value).

Unfortunately, there have been issues with the definitions given in [53], some of them pointed out by Kosba et al. [39]. In particular, the ledger indistinguishability game can be won by any adversary since the adversarial model leaks unintended information. Garman et al. [28] introduce a simulation based definition and outline a ideal functionality. Then, they present an incomplete sketch of a proof for the Zerocoin [53]. This leaves us without a reliable security definition and security analysis for any DAP scheme. Unfortunately, because of time constraints we also have to leave the definition and security analysis for a further iteration of the article.

4 Our DAP Protocol

In this section we describe our basic scheme.

$\mathsf{Setup}_{DAP}(1^\lambda)$: The procedure takes as input the security parameter λ based on which it generates the following:

1. A group $\mathbb{G}$ of prime order p.
2. Random generators $g, \bar{g}_1, \bar{g}_2 \in \mathbb{G}$.
3. A bit commitment function $\mathsf{Comm} : \{0,1\}^{\ell_R + \ell_{max}} \to \mathsf{D}_{\mathsf{Comm}}$.
4. Collision-resistant functions: $\mathsf{H} : \{0,1\}^{\ell_I} \to \mathcal{D}_{\mathsf{H}}$, $\mathsf{H}_S\{0,1\}^* \to \mathbb{Z}_p$, $\mathsf{H}_P : \{0,1\}^{\log(p) \times \ell_{PRF}} \to \mathbb{Z}_p$.
5. Set ℓ_{max} to be the maximal value of a coin.
6. Set the height of the Merkle tree as N. If necessary define an additional hash function.
7. A non-interactive proof system Π_{crs} with common reference string crs.
8. A one-time signature scheme $S = (\mathsf{Setup}_{Sig}, \mathsf{Sign}_{Sig}, \mathsf{Verify}_{Sig})$.

The procedure outputs the public parameters $\mathsf{pp}_{DAP} = (\mathbb{G}, p, g, \bar{g}_1, \bar{g}_2, \mathsf{Comm}, \mathsf{H}, \mathsf{H}_S, \ell_{max}, N, crs, S)$.

$\mathsf{CreateAddress}_{DAP}(\mathsf{pp}_{DAP})$: A user chooses uniformly random $x \in \mathbb{Z}_p$, computes $X = g^x$ sets the address $\mathsf{addr} = (X)$ and address secret key $\mathsf{addrSK} = (x)$.

$\mathsf{CreateCoin}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{addr}, \mathsf{addrSK}, v)$: The procedure takes as input the public parameters pp_{DAP}, an address $\mathsf{addr} = (X)$ and its secret key $\mathsf{addrSK} = (x)$, and value $v = [v_i \in \{0,1\}]_{i=0}^{\ell_{max}-1}$. A user chooses $y \in \{0,1\}^{\ell_{PRF}}$, $b \overset{R}{\leftarrow} \mathbb{Z}_p$, computes $a \leftarrow \mathsf{H}_{PRF}(y)$, sets $sk \leftarrow (x, y, b)$ and $pk \leftarrow g^x \bar{g}_1^a, \bar{g}_2^b$. Then the user chooses a bit string $r = [r_i \in \{0,1\}]_{i=0}^{\ell_R-1}$ uniformly at random, computes $c \leftarrow \mathsf{Comm}(r, v)$ and $cm = \mathsf{H}(pk||c)$. Finally, the user sets $\mathsf{coin} = (cm)$, a proof $\pi = (pk, c, r, v)$ and coin spending key $\mathsf{coinSK} = (sk, pk, c, r, v)$.

$\mathsf{Mint}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{coin}, v, \pi, L)$: The verifier takes as input the public parameters pp_{DAP}, a coin $\mathsf{coin} = (cm)$, value $v = [v_i \in \{0,1\}]_{i=0}^{\ell_{max}-1}$ and a proof $\pi = (pk, c, r, v)$. If $cm \neq \mathsf{H}(pk||c)$ or $c \neq \mathsf{Comm}(r, v)$ or the user didn't pay v, the procedure outputs $\perp$ and aborts. Otherwise the verifier appends (coin, π) to the ledger L.

$\mathsf{Pour}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{coin}_1, \mathsf{coin}_2, \mathsf{coinSK}_1, \mathsf{coinSK}_2, \mathsf{addr}_1, \mathsf{addr}_2, v^{out,1}, v^{out,2}, v^{pub}, L)$: The prover takes the public parameters pp_{DAP} the ledger L and for $j, k \in \{1, 2\}$ the following:

- The input coins $\mathsf{coin}_j = (cm^{in,j} = \mathsf{H}(pk^{in,j}||c^{in,j}))$.
- The input coin spending key $\mathsf{coinSK}_j = (sk^{in,j}, pk^{in,j}, c^{in,j}, r^{in,j}, v^{in,j})$ where $sk^{in,j} = (x^{in,j}, y^{in,j}, b^{in,j}) \in \mathbb{Z}_p$, $a^{in,j} = \mathsf{H}_{PRF}(y^{in,j}||g^{x^{in,j}})$, $pk^{in,j} = g^{x^{in,j}} \bar{g}_1^{a^{in,j}} \bar{g}_2^{b^{in,j}}$, $c^{in,j} = \mathsf{Comm}(r^{in,j}, v^{in,j})$, $r^{in,j} = [r_i^{in,j} \in \{0,1\}]_{i=0}^{\ell_R-1}$ and $v^{in,j} = [v_i^{in,j} \in \{0,1\}]_{i=0}^{\ell_{max}-1}$. We denote $X^{in,j} = g^{x^{in,j}}$.
- The public values $v^{pub} = [v_i^{pub} \in \{0,1\}]_{i=0}^{\ell_{max}-1}$
- Output values $v^{out,k} = [v_i^{out,k} \in \{0,1\}]_{i=0}^{\ell_{max}-1}$, s.t. $v^{out,1} + v^{out,2} = v^{pub} + v^{in,1} + v^{in,2}$.
- Output addresses $\mathsf{addr}_k = (X^{out,k} \in \mathbb{G})$.

To spend the input coins and create new output coins with values $v^{out,1}$ and $v^{out,2}$, for $j, k \in \{1, 2\}$ the user does the following:

1. Compute hash a Merkle tree root M_{root} s.t. $cm^{in,j}$ are leaves in the tree and M_{path}^j are the corresponding paths to M_{root}. The computation is done using the ledger L.
2. Compute $\hat{sk}_j = 1/(x^{in,j} + a^{in,j}) \mod p$ and the serial numbers $sn_j \leftarrow g^{\hat{sk}_j}$. Additionally, compute $\hat{pk}^j \leftarrow g^{x^{in,j} + a^{in,j}}$.
3. Compute the output coins:
   - Choose $y^{out,k} \in \{0,1\}^{\ell_{PRF}}$, $b^{out,k} \in \mathbb{Z}_p$ and $r^{out,k} = [r_i^{out,k} \in \{0,1\}]_{i=0}^{\ell_R-1}$ uniformly at random.

- Compute the coin public keys, by computing $a^{out,k} \leftarrow \mathsf{H}_{PRF}(y^{out,k}||X_k)$ and $pk^{out,k} = \bar{g}_1^{a^{out,k}} \bar{g}_2^{b^{out,k}} \cdot X_k$.
- Compute the commitments $c^{out,k} = \mathsf{Comm}(r^{out,k}, v^{out,k})$.
- Compute $cm^{out,k} = \mathsf{H}(pk^{out,k}||c^{out,k})$.
- Set $\mathsf{coin}_k^{out} = (cm^{out,k})$ and $\mathsf{coinDec}_k^{out} = (y^{out,k}, b^{out,k}, pk^{out,k}, c^{out,k}, r^{out,k}, v^{out,k}))$.

4. Run $(PK_{Sig}, SK_{Sig}) \leftarrow \mathsf{Setup}_{Sig}(1^\lambda)$ and compute the following:
   - $h_S \leftarrow \mathsf{H}_S(PK_{Sig})$,
   - $sk_j^S = 1/(x_{in,j} + a_{in,j} + h_S) \mod p$ and
   - $sn'_j \leftarrow g^{sk_j^S}$.
5. Set
   - the statement $stmt = (\mathsf{pp}_{DAP}, M_{root}, sn_1, sn_2, h_S, sn'_1, sn'_2, cm^{out,1}, cm^{out,2}, v^{pub})$, and
   - the secret input $\omega = (M_{path}^1, M_{path}^2\ pk^{in,1}, pk^{in,2}, c^{in,1}, c^{in,2}, a^{in,j}, b^{in,j}, x^{in,j}, X^{in,j}, \hat{pk}^j, \hat{sk}_1, \hat{sk}_2, sk_1^S, sk_2^S, r^{in,1}, r^{in,2}, v^{in,1}, v^{in,2}, pk^{out,1}, pk^{out,2}, c^{out,1}, c^{out,2}, r^{out,1}, r^{out,2}, v^{out,1}, v^{out,2})$.
6. Compute the proof $\pi \leftarrow \Pi_{crs}.\mathsf{Prove}(stmt, \omega)$.
7. Compute the signature $\sigma \leftarrow \mathsf{Sign}_{Sig}(SK_{Sig}; stmt, \pi)$.
8. Output the transaction $\mathsf{tr}_{POUR} = (M_{root}, sn_1, sn_2, sn'_1, sn'_2, \pi, PK_{Sig}, \sigma, v^{pub})$, the output coins coin_1, coin_2 and send $\mathsf{coinDec}_1$ and $\mathsf{coinDec}_2$ to the new coin owners.

$\mathsf{Verify}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{tr}_{POUR}, \mathsf{coin}_1, \mathsf{coin}_2, L)$: On input the public parameters pp_{DAP}, a transaction $\mathsf{tr}_{POUR} = (M_{root}, sn_1, sn_2, sn'_1, sn'_2, \pi, PK_{Sig}, \sigma, v^{pub})$ and two coins for the transaction $\mathsf{coin}_1 = (cm^{out,1})$, $\mathsf{coin}_2 = (cm^{out,2})$, the verifier

1. restores the Merkle tree with root M_{root},
2. checks whether $sn_1 \neq sn_2$ or sn_1 or sn_2 appears on L,
3. computes $h_S \leftarrow \mathsf{H}_S(PK_{Sig})$,
4. sets the statement as $stmt = (\mathsf{pp}_{DAP}, M_{root}, sn_1, sn_2, h_S, sn'_1, sn'_2, cm^{out,1}, cm^{out,2})$,
5. verifies the proof $\Pi_{crs}.\mathsf{Verify}(stmt, \pi) = 1$,
6. verifies the signature $\mathsf{Verify}_{Sig}(PK_{Sig}; stmt, \pi; \sigma) = 1$,
7. updates the Merkle tree to contain $cm^{out,1}$ and $cm^{out,2}$, and
8. appends $\mathsf{coin}_1, \mathsf{coin}_2$ and tr_{POUR} to the ledger.

$\mathsf{Receive}_{DAP}(\mathsf{pp}_{DAP}, \mathsf{coin}, v, \mathsf{coinDec}, \mathsf{addr}, \mathsf{addrSK})$: a user with $\mathsf{addr} = (X)$ and address secret key $\mathsf{addrSK} = (x)$, receives a coin $\mathsf{coin} = (cm)$ and the coin decoding $\mathsf{coinDec} = (y, b, pk, c, r, v)$. In particular, the user obtains a share of the coin secret key $y \in \{0,1\}^{\ell_{PRF}}$, the blinding $b \in \mathbb{Z}_p$ the commitment random bits $r = [r_i \in \{0,1\}]_{i=0}^{\ell_R - 1}$ and the bits $v = [v_i \in \{0,1\}]_{i=0}^{\ell_{max}-1}$ of the coin value. If any of the given values is outside in the required domain, the user rejects. If $a = \mathsf{H}_P(y||X)$ is on the users internal list, i.e. he got the same randomness earlier or $a = 0$, then return $\perp$. The user checks $pk = g^x \bar{g}_1^a \bar{g}_2^b$ and $c = \mathsf{Comm}(r, v)$ and whether $cm = \mathsf{H}(pk||c)$ is on the ledger. The user sets key $sk = (x, y, b)$ and the coin secret key as $\mathsf{coinSK} = (sk,\ pk,\ c,\ r,\ v)$.

- Public input: $stmt = (\mathsf{pp}_{DAP},\ M_{root},\ sn_1,\ sn_2,\ h_S,\ sn'_1,\ sn'_2,\ cm^{out,1},\ cm^{out,2},\ v^{pub})$,
- Secret input: $\omega = (M^1_{path},\ M^2_{path}\ pk^{in,1},\ pk^{in,2},\ c^{in,1},\ c^{in,2},\ a^{in,j},\ b^{in,j},\ x^{in,j},\ X^{in,j},\ \hat{pk}^j,\ \hat{sk}_1,\ \hat{sk}_2,\ sk^S_1,\ sk^S_2,\ r^{in,1},\ r^{in,2},\ v^{in,1},\ v^{in,2},\ pk^{out,1},\ pk^{out,2},\ c^{out,1},\ c^{out,2},\ r^{out,1},\ r^{out,2},\ v^{out,1},\ v^{out,2})$.
- Prove that the inputs which are supposed to be bits are bits.
- For $j \in \{1,2\}$ and $k \in \{1,2\}$ proof that:

  1. $pk^{in,j}$ are well formed and the prover knows the secret keys.
     - $g^{x^{in,j}} = X^{in,j}$,
     - $\bar{g}_1^{a^{in,j}} \bar{g}_2^{b^{in,j}} \cdot X^{in,j} = pk^{in,j}$ and
     - $X^{in,j} \cdot g^{a^{in,j}} = \hat{pk}^j$.
  2. sn_j are well formed and the prover knows the secret key.
     - $(\hat{pk}^j)^{\hat{sk}_j} = g$ and
     - $sn_j = g^{\hat{sk}_j}$.
  3. Prove that sn'_j is well formed and the prover knows the secret key.
     - $(\hat{pk}^j \cdot g^{h_S})^{sk^S_j} = g$ and
     - $sn'_j = g^{sk^S_j}$.
  4. The commitments and values are correct
     - $c^{in,j} = \mathsf{Comm}(r^{in,j}, v^{in,j})$,
     - $c^{out,k} = \mathsf{Comm}(r^{out,k}, v^{out,k})$,
     - $cm^{out,k} = \mathsf{H}(pk^{out,k}||c^{out,k})$,
     - $v^{pub} + v^{in,1} + v^{in,2} = v^{out,1} + v^{out,2}$.
  5. $cm^{in,j} = \mathsf{H}(pk^{in,j}||c^{in,j})$ are in the Merkle tree rooted at M_{root}.

Scheme 1: The Prove statement Π_{crs}

Remark 1. *The function H needs to be collision resistant, but we don't require it to be hiding. In particular, we could publish pk^{out} and c^{out} alongside $cm = \mathsf{H}(pk||c)$. The function of the cm is to "bind" both pk and c together. Hence we may consider a slight optimisation and resign of showing*

$cm^{out,k} = \mathsf{H}(pk^{in,k}||c^{in,k})$ for both output coins. However then we need to set the output coins to $\mathsf{coin}^{out,k} = (pk^{in,k}, c^{in,k})$ what may cost more space on the ledger. Moreover, we would also have to make sure $\mathsf{coin}^{out,k}$ is signed, to avoid substitution attacks.

Remark 2. *In our description we didn't specify how the coin decodings $\mathsf{coinDec}$ are transported to the new owner. It can obviously be realized similarly as in the ZCash protocol [53], by using a key-private encryption scheme [4] or it may be just sent out-of-band. In case of using a key-private encryption scheme, we need to alter the user address, and add an encryption public key. In order to avoid malleability attacks, we have to make sure that the ciphertexts are signed in the pour procedure. Additionally, the $\mathsf{Receive}_{DAP}$ procedure, would have to be altered to reject payments which incorrectly encrypt the secrets.*

Furthermore, using public key encryption allows us a user to delegate the decryption key to a third party, who then might monitor incoming transactions to this address.

Remark 3. *As noticed above using public key encryption to transport $\mathsf{coinDec}$ through a public ledger may allow chosen third parties holding the decryption key, to monitor incoming transactions.*

With some modifications we can also allow to monitor output transactions. First we may reveal that the coin has been spent, by publishing an encryption of the coin public key. In order to enable to monitor to who the coin has been sent, we may also publish the encryption of the share $a \in \mathbb{Z}_p$, s.t. the output coin public key $pk^{out} = g^a \cdot X^{out}$ for the output address X^{out}. In the above method one must take care to sign the ciphertexts to avoid malleability attacks.

Note however, that the above method relies on "good will" of the coin owner. In order, to provide some sort of accountability and force the coin owner to provide correct encryptions we would have to make some more serious modifications. First of all the proof from the poor transaction would have to include a membership proof of the outgoing address. Without such membership proof, the payment sender can always send the coin to an address which is not on the ledger. Second, the pour proof should also include a proof that the values in the ciphertexts are consistent with the values used to spend the coin and create the output coins.

Remark 4. *We can drastically simplify the protocol by making an address $X = g_1^x g_2^r$ and the public key coin $pk = g_0^b g_1^x g_2^r$. Then, we have to proof knowledge of b, x, r, and we can use a generic generic PRF to compute the serial numbers, i.e. $sn = PRF_x(b)$ and $sn' = PRF_x(b + h_S \mod p)$. The PRF then may be instantiated with SHA256 as in [36] or AES with 256 bit key and 256 bit block (the key and message are $\log(p)$ bits long). Such construction does not have the drawback of the Dodis-Yampolskiy [27] PRF, that it requires the message space to be relatively small. However, the cost of proving a symmetric primitive is much higher.*

5 Instantiation

In this section we show two instantiations of the building blocks. In particular, we discuss the instantiation of the elliptic curve E over $\mathbb{F}_q$, the CRHF H, the commitment scheme Comm and the Merkle tree.

Both instantiations share a common ground, which includes operations in group $\mathbb{G}$, the signature scheme S and H_S and H_{PRF}. What differs, is the way Comm, H and the Merkle tree are instantiated. Thus we will first present the common operations and then, in Section 5.1 we show an instantiation solely based on discrete logarithms commitments and in Section 5.3 we show an instantiation exploiting lattice based primitives. The design of most constraint systems for the ZK-SNARK are given in Appendix A.

The group $\mathbb{G}$. As noticed in the construction of our DAP scheme, $\mathbb{G}$ is a group of points on an elliptic curve E over $\mathbb{F}_q$, where q is prime. Moreover, we require the order p of $\mathbb{G}$ to be and odd prime.

We will mainly focus on Edwards and twisted Edwards curves [9, 34] since, as noticed in [38], the arithmetic on these curves is very efficient when implemented on arithmetic circuits over $\mathbb{F}_q$.

Definition 13. *Let $\mathbb{F}$ be a finite field of characteristic char$(\mathbb{F}) \neq 2$. Let $a, d \in \mathbb{F}$. The Edwards curve with coefficient d is defined as follows.*

$$E_d(\mathbb{F}) = x^2 + y^2 = 1 + dx^2y^2.$$

The group is instantiated over the Twisted Edwards curves.

Definition 14. *Let $\mathbb{F}$ be a finite field of characteristic char$(\mathbb{F}) \neq 2$. Let $a, d \in \mathbb{F}$ and $a \neq d$. The Twisted Edwards curve with coefficients a, d is defined as follows.*

$$E_{a,d}(\mathbb{F}) = ax^2 + y^2 = 1 + dx^2y^2.$$

We say that an Twisted Edwards curve is a Edwards curve if $a = 1$. The neutral element is $\mathcal{O} = (0, 1)$ and negative of (x_1, y_1) is $(-x_1, y_1)$.

Let us now focus on the general case of Twisted Edwards curves, and recall the affine addition formulas.

Definition 15. *Affine Addition Formula Let $(x_1, y_1), (x_2, y_2)$ be points on $E_{a,d}(\mathbb{F})$. Then*

$$(x_1, y_1) + (x_2, y_2) = ((\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}), (\frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2})) = (x_3, y_3). \quad (1)$$

Definition 16. *Affine Doubling Formula Let (x_1, y_1) be a point on $E_{a,d}(\mathbb{F})$. Then*

$$2(x_1, y_1) = ((\frac{2x_1y_1}{1 + dx_1^2y_1^2}), (\frac{y_1^2 - ax_1^2}{1 - dx_1^2y_1^2})) = (x_3, y_3). \quad (2)$$

As shown in [8], the formulas 15 and 16 are complete if d is not a square in $\mathbb{F}$.

Additionally, we will consider the addition formulas from given in [34].

Definition 17. *Affine Addition Formula independent of d Let $(x_1, y_1), (x_2, y_2)$ be points on $E_{a,d}(\mathbb{F})$. Then*

$$(x_1, y_1) + (x_2, y_2) = ((\frac{x_1y_1 + x_2y_2}{y_1y_2 + ax_1x_2}), (\frac{x_1y_1 - x_2y_2}{x_1y_2 - y_1x_2})) = (x_3, y_3). \tag{3}$$

Definition 18. *Affine Doubling Formula independent of d Let (x_1, y_1) be a point on $E_{a,d}(\mathbb{F})$. Then*

$$2(x_1, y_1) = ((\frac{2x_1y_1}{ax_1^2 + y_1^2}), (\frac{y_1^2 - ax_1^2}{2 - ax_1^2 - y_1^2})) = (x_3, y_3). \tag{4}$$

Note, that the addition formula given by Definition 17 doesn't work for point doubling. Moreover, as shown in [34] there are exceptional cases even when d is not a square in $\mathbb{F}$. On the other hand, if both points are not equal and are of odd order, then the the formula is correct as shown in [34]. As for doubling formula given by Definition 18 the same exceptional cases apply as for the formula given by Definition 16. Note also that the $E_{a,d}(\mathbb{F}_q)$ has another special point $(0, -1)$ of order 2.

In our instantiation we will only consider the field $\mathbb{F}_q$ where q is an odd prime. Additionally, we will exploit point compression on twisted Edwards curves by reducing the x coordinate to its lest significant bit. In other words, given a point $(x, y) \in E_{a,d}(\mathbb{F}_q)$, the compressed point is $(x \mod 2, y)$. We can rewrite the curve equation for $E_{a,d}$ as

$$\frac{1 - y^2}{(a - dy^2)} = x^2.$$

First, note that if $a = dy^2$, then the curve equation would become $ax^2 + y^2 = 1 + ax^2$, what is only satisfied by $y = 1$ and $y = -1$, for any x. However, $y^2 = 1$ would imply that $ax^2 + 1 = 1 + dx^2$, what holds for $x = 0$ (the neutral element) and the point of order 2. But for $x \neq 0$ we have that $a = d$, what contradicts the definition of $E_{a,d}$. Lets denote $\hat{x} = \sqrt{\frac{1-y^2}{(a-dy^2)}}$, then $q - \hat{x} \mod q$ is another solution. Since q is a odd prime, we have that $x \mod 2$ uniquely encodes the x coordinate. In this paper we will only be interested in the above fact about point compression. In particular, we will not explicitly use any point decompression algorithms.

Proving consistency of the coin public key. Bellow, we show the constraint system for showing knowledge of $\mathbf{a} = [\mathbf{a}_i \in \{0,1\}]_{i=0}^{\log(p)-1}$, $\mathbf{b} = [\mathbf{b}_i \in \{0,1\}]_{i=0}^{\log(p)-1}$, $\mathbf{x} = [\mathbf{x}_i \in \{0,1\}]_{i=0}^{\log(p)-1}$, and $\mathbf{X}$, $\mathbf{pk}$, $\hat{\mathbf{pk}} \in \mathbb{G}$, satisfying $g^{\mathbf{x}} = \mathbf{X}$, $\bar{g}_1^{\mathbf{a}}\bar{g}_2^{\mathbf{b}} \cdot \mathbf{X} = \hat{\mathbf{pk}}$ and $\mathbf{X} \cdot g^{\mathbf{a}} = \hat{\mathbf{pk}}$. For brevity, we will assume the procedure takes only the bits of the scalars as input.

PK-Consistency($\mathbf{a}, \mathbf{b}, \mathbf{x}, \mathbf{X}, \mathbf{pk}, \hat{\mathbf{pk}}$: $g^{\mathbf{x}} = \mathbf{X} \wedge \bar{g}_1^{\mathbf{a}}\bar{g}_2^{\mathbf{b}} \cdot \mathbf{X} = \mathbf{pk} \wedge \mathbf{X} \cdot g^{\mathbf{a}} = \hat{\mathbf{pk}}$):

1. WindowMul($\mathbf{x}, \mathbf{X}$: $g^{\mathbf{x}} = \mathbf{X}$)
2. WindowMul($\mathbf{a}, \mathfrak{g}_1$: $\bar{g}_1^{\mathbf{a}} = \mathfrak{g}_1$)
3. WindowMul($\mathbf{b}, \mathfrak{g}_2$: $\bar{g}_2^{\mathbf{b}} = \mathfrak{g}_2$)
4. WindowMul($\mathbf{a}, \mathfrak{g}_3$: $g^{\mathbf{a}} = \mathfrak{g}_3$)
5. EdAdd($\mathbf{X}, \mathfrak{g}_3$: $\hat{\mathbf{pk}} = \mathbf{X} \cdot \mathfrak{g}_3$)
6. EdAdd($\mathbf{X}, \mathfrak{g}_1$: $\mathfrak{g}_4 = \mathbf{X} \cdot \mathfrak{g}_1$)
7. EdAdd($\mathfrak{g}_4, \mathfrak{g}_2$: $\mathbf{pk} = \mathfrak{g}_4 \cdot \mathfrak{g}_2$)

Cost: $\frac{4\log(p)}{w}(f_{pm}+7) - 7$ constraints and $\frac{4\log(p)}{w}(g_{pm}+9) - 19$ additional variables.

Proving the consistency of the serial numbers. Note, that for the construction presented in Section 4, we compute the serial numbers as $sn \leftarrow g^{1/(a+x)}$, where g^x is the address public key and $a = \mathsf{H}_{PRF}(y||g^x)$. Moreover, the coin secret key sk, is constructed as $sk = x + a$, where $X = g^x$ is a users address and a is chosen by the sender of the coin, thus potentially by the adversary. This construction resembles the Dodis-Yampolskiy [27] verifiable pseudorandom function. In our case however, we do not need pairings to verify the correctness of the PRF output and thus we may reduce the pseudorandomness to the weaker q-DDHI assumption.

Definition 19. *(q-DDHI problem [15, 27]) Let $(g, g^x, \ldots, g^{x^q}, g^{1/x}) \in \mathbb{G}$. We define the advantage of an adversary $\mathcal{A}$ of breaking the q-DDHI problem as*

$$|\Pr[\mathcal{A}(g, g^x, \ldots, g^{x^q}, g^{1/x})] - \Pr[\Gamma \xleftarrow{R} \mathbb{G}; \mathcal{A}(g, g^x, \ldots, g^{x^q}, \Gamma)]| \leq Adv_{q\text{-}DDHI}(\lambda)$$

We say the q-DDHI problem is hard in $\mathbb{G}$ if $Adv_{q\text{-}DDHI}(\lambda)$ is negligible in the security parameter.

One important thing to note, is that this PRF is only able to process a relatively small message space. This is because, in the reduction of pseudorandomness to the q-DDHI problem, the solver needs to guess the challenged message. In our case, the messages for the PRF are $\log(p)$ bits long, however there are only ℓ_{PRF} such messages since we require that a message $a = \mathsf{H}_{PRF}(y||X)$, for $y \in \{0,1\}^{\ell_{PRF}}$. Hence in the reduction we may choose the these messages beforehand and the program the ROM to output the messages we need. Additionally, we have serial numbers obtained from the messages $a + \mathsf{H}(PK_{Sig})$, were PK_{Sig} is a public key of the signature scheme. In this case, we may also program the ROM, s.t. the sum will fall into the desired value from the q-DDHI reduction. We recall the reduction in Appendix B. In [16] Boneh et al. generalize the Dodis-Yampolskiy PRF and apply it into a cascade construction, which may process a messages from a larger space, but divided into chunks of size ℓ_{max}. In our construction we use only one chunk.

Below we show the construction of the constraint system for checking $sn_j = g^{1/\hat{sk}_j}$ and $sn'_j = g^{1/(\hat{sk}_j+h_S)}$. Both systems use procedures defined in Appendix A. Unfortunately, in both procedures will not be able to exploit the efficiency of the window scalar multiplication. This is because, we need to operate on the basepoint $\hat{pk}^j$ which is secret and therefore we cannot precompute a lookup table.

$\mathsf{InvPRF}(\mathbf{sk} = [\mathbf{sk}_i \in \{0,1\}]_{i=0}^{\log(p)-1}, \mathbf{pk} \in \mathbb{G}\text{: } \mathbf{pk}^{\mathbf{sk}} = g \wedge sn = g^{\mathbf{sk}})$**:**

1. $\mathsf{Double\text{-}Add}(\mathbf{sk}, \mathbf{pk}\text{: } \mathbf{pk}^{\mathbf{sk}} = g)$
2. $\mathsf{WindowMul}(\mathbf{sk}\text{: } g^{\mathbf{sk}} = sn)$

Cost: $14\log(p) + \frac{\log(p)}{w}(f_{pm}+7) - 14$ constraints and $14\log(p) + \frac{\log(p)}{w}(g_{pm}+9) - 18$ additional variables.

$\mathsf{InvPRF+}(\mathbf{sk} = [\mathbf{sk}_i \in \{0,1\}]_{i=0}^{\log(p)-1}, \mathbf{pk} \in \mathbb{G}\text{: } (\mathbf{pk} \cdot c)^{\mathbf{sk}} = g \wedge sn = g^{\mathbf{sk}})$**:**

1. $\mathsf{EdAddConst}(\mathfrak{g}, \mathbf{pk}\text{: } \mathfrak{g} = \mathbf{pk} \cdot c)$
2. $\mathsf{Double\text{-}Add}(\mathbf{sk}, \mathfrak{g}\text{: } \mathfrak{g}^{\mathbf{sk}} = g)$
3. $\mathsf{WindowMul}(\mathbf{sk}\text{: } g^{\mathbf{sk}} = sn)$

Cost: $14\log(p) + \frac{\log(p)}{w}(f_{pm}+7) - 11$ constraints and $14\log(p) + \frac{\log(p)}{w}(g_{pm}+9) - 15$ additional variables.

The Circuit. Now we will sum up the sub-procedures of the constraint system we have to proof.

- For $pk^{in,1}, pk^{in,2}$ we have in total 2 times $\mathsf{PK-Consistency}$, on input bit strings $x^{in,1}$, $x^{in,2}$, $a^{in,1}$, $a^{in,2}$, $b^{in,1}$,$b^{in,2}$ and points $X^{in,1}$, $X^{in,2}$, $\hat{pk}^1$, $\hat{pk}^2$.
- For sn_1, sn_2, we have in total 2 times InvPRF, on input the bit strings of $\hat{sk}_1$ and $\hat{sk}_2$ and the points $pk^{in,1}$ and $pk^{in,2}$.
- For sn'_1, sn'_2, we have in total 2 times $\mathsf{InvPRF+}$, on input the bit strings of sk^S_1 and sk^S_2 and the points $pk^{in,1}$ and $pk^{in,2}$.
- For the bit strings $v^{in,1}$, $v^{in,2}$, $v^{out,1}$ and $v^{out,2}$, we run $\mathsf{CheckValues}$.

At this point we know we have $4 \cdot (\log(p) + \ell_{max}) + 4$ variables and we to verify that $4 \cdot (\log(p) + \ell_{max})$ variables are bits.

Furthermore, the parts which are instantiation specific are as follows.

- For $c^{in,1}, c^{in,2}, c^{out,1}, c^{out,2}$, we have in total 4 times Comm, on input the bit strings $r^{in,1}$, $r^{in,2}$, $r^{out,1}$, $r^{out,2}$, $v^{in,1}$, $v^{in,2}$, $v^{out,1}$ and $v^{out,2}$.
- For $cm^{in,1}$, $cm^{in,2}$, $cm^{out,1}$ and $cm^{out,2}$, we have to check in total 4 time H on input two points. Note that we will need to add additional variables only for $cm^{in,1}$ and $cm^{in,2}$.
- Verifying 2 k-ary trees of height N.

Later we have to remember that we have additionally $4 \cdot \ell_R$ variables which are bits. Furthermore, the paths for the k-ary tree of height N require in total $2N(k-1)$ bits. The commitments, hash outputs and the hash elements on the paths depend on the instantiation.

5.1 Instantiating from DLP based Primitives

In this section we briefly describe how to instantiate Comm and H using Discrete Logarithm based primitives. As mentioned earlier we give the design for the constraint systems in Appendix A.

The commitment scheme Comm. The Pedersen commitment scheme $\mathsf{Comm} : \{0,1\}^{\ell_R+\ell_{max}} \to \mathsf{D}_{\mathsf{Comm}}$ may be defined as follows.

$\mathsf{SetupP}(\mathbb{G})$: Choose g, $h \overset{R}{\leftarrow} \mathbb{G}$ and output $pp_{Com} = (\mathbb{G}, g, h)$.

$\mathsf{CommP}(pp_{Com}, m, r)$: On input a message $m \in \mathbb{Z}_p$ and randomness $r \in \mathbb{Z}_p$, return $cm = g^m \cdot h^r$.

The domain of the Pedersen commitment CommP is $\mathsf{D}_{\mathsf{Comm}} = \mathbb{G}$. Note, that we omitted to recall the opening algorithm. We omit this intentionally to reduce the number of procedures, however knowledge the opening is actually used implicitly in the ZK-SNARK verification, where we proof the knowledge of (m, r), s.t. $cm = g^m \cdot h^r$. According to our notation from Section 4, the bit length of the randomness ℓ_R is $\log(p)$, where the max bit length of the values ℓ_{max} may be arbitrary.

The Collision Resistant Function. As our hash function H we use the function introduced by Chaum et al. [23] and further analysed in [56, 35].

$\mathsf{SetupH}_P(\mathbb{G}, 1^\ell)$: Choose $h_i \overset{R}{\leftarrow} \mathbb{G}$ for $i \in \{0, \dots, \ell\}$ and output $pp_{\mathsf{H}} = (\mathbb{G}, [h_i]_{i=0}^{\ell})$.

$\mathsf{H}_P(pp_{\mathsf{H}}, m)$: The hash takes as input a bit string $[m_i]_{i=0}^{\ell-1}$ and returns $h = h_\ell \prod_{i=0}^{\ell-1} h_i^{m_i}$.

Putting the DLP Based instantiation together. Although it may be easy to see how the building block fit to each other "outside" the ZK-SNARK, here we will discuss how to fit them together for statement of the ZK-SNARK.

How to fit the commitment is straightforward. In order to verify the commitment of will simply use the

$$\mathsf{WindowCommP}([\mathbf{x}_i \in \{0,1\}]_{i=0}^{\ell_{max}-1}, [\mathbf{r}_i \in \{0,1\}]_{i=0}^{\log(p)-1}, \mathbf{c} : g^{\mathbf{x}} \cdot h^{\mathbf{r}} = \mathbf{c})$$

constraint system, where the number of message bits is ℓ_{max} and the number of randomness bits is $\log(p)$. Then we alter the plain DLP hash function. Since we input 2 points, we would need to hash $4\log(q)$ bits. We may optimise that by hashing the compressed points, which would result in hashing only $2(\log(q)+1)$ bits. We do this simply putting only the least significant bit of the x coordinate into the DLP hash. Note, that if the secret key bit size is not divisible by the window size, we need to complement the secret key with zeros.

For the hash function for the Merkle tree, we may use the same hash as above, but as for now we will assume the the hash is more general and takes k points, thus $k \cdot 2\log(q)$ bits. As before, we will hash only the compressed points, thus the DLP hash needs to consume $k(\log(q)+1)$ bits. This setting will later be necessary for analysing the applicability of k-ary trees for $k > 2$. Note, that for $k = 2$, we can simply reuse the precomputations from the hash H.

To sum up, we input 4 commitments in total, thus we add 8 variables in $\mathbb{F}_q$ and $4 \cdot \log(p)$ variables which are bits (for the commitment randomness). Furthermore, we have additionally 4 variables in $\mathbb{F}_q$ for $cm^{in,1}$ and $cm^{in,2}$. Note that $cm^{out,1}$ and $cm^{out,2}$ are public. For the hash functions we assume the inputs are already verified bits. Therefore, we additionally have to count the cost of splitting 16 variables into $\log(q)$ bit strings. The input to the Merkle tree does not assume the variables are already split into bit strings.

5.2 Performance Estimates for DLP Instantiation

In this section we discuss the performance of our instantiations. We will first set the bit size of the field underlying the arithmetic circuit as $\log(q) = 255$ and the order of $\mathbb{G}$ as $\log(p) = 251$. This corresponds to the Jubjub curve given in [58]. At Table 1 we show the costs of scalar multiplication, and the InvPRF and $\mathsf{InvPRF+}$ procedures for different windows values. Table 2 depicts calculations for the $\mathsf{PK-Consistency}$ procedure. At Table 3 we present the calculations for the DLP based commitment scheme and in Table 4, Table 5 and at Table 6 we show the costs for the DLP based hash with different input size. For window size $w = 1$ we count the $\mathsf{ScalarMul}$ constraint system instead of the $\mathsf{WindowMul}$ constraint system. As we may notice from our calculations the optimal window size is $w = 3$ instead of $w = 4$ as suggested in [58].

Later in Table 7 we give our calculations for k-ary hash trees offering different anonymity sets. The hash function used in these calculations was the DLP hash with window size $w = 3$.

In Table 8 we summarize the costs for our system, but without counting the costs of the hash trees. Finally, in Table 9 we give the total costs for 2 and 3-ary hash trees and sizes of anonymity sets $\approx 2^{29}$ and 2^{64}.

To give a comparison with existing schemes we recall the cost estimates for ZCash [36] and the original Zerocoin paper [53] in Table 10. However, these costs include only the number of constraints (multiplication gates) and do not provide the number of variables. On the other hand, in practice the number of variables should approximately be the same as the number of multiplication gates.

w	Scalar	Lookup table	Multiplication		InvPRF		InvPRF	
			Const	Vars	Const	Vars	Const	Vars
1	251	251	1252	1250	-	-	-	-
2	252	504	1127	1125	4648	4644	4651	4647
3	252	672	1085	1083	4606	4602	4609	4605
4	252	1008	1316	1314	4837	4833	4840	4836
5	255	1632	1880	1878	5443	5439	5446	5442
6	252	2688	2891	2889	6412	6408	6415	6411

Table 1: Cost estimates for scalar multiplication, InvPRF and $\mathsf{InvPRF}+$. For scalar multiplication at window size $w = 1$ we summarize the cost for $\mathsf{ScalarMul}$. For all other window sizes we count the windowed versions of the operations. The scalar is of size $\log(p) = 251$, however in the scalar column we rounded up the scalar to be divisible by the window size w.

w	Scalar	Lookup table size	Const	Vars
2	252	1512	4529	4517
3	252	2016	4361	4349
4	252	3024	5285	5273
5	255	4896	7541	7529

Table 2: Cost of $\mathsf{PK} - \mathsf{Consistency}$ with $\log(p) = 251$ bit randomness rounded up to be divisible by w. Lookup table size includes tables for g, $\bar{g}_1$, $\bar{g}_2$, thus we need 3 tables.

w	ℓ_R	Lookup table size	Const	Vars
2	252	504	1415	1413
3	251	669	1358	1356
4	252	1008	1652	1650
5	251	1606	2324	2322
6	254	2709	3650	3648

Table 3: Cost for the commitment CommP with $\ell_{max} = 64$ and $\log(p) = \ell_R = 251$ bit randomness rounded up to be divisible by w.

w	Bit input	Lookup table size	Const	Vars
1	512	512	2560	2558
2	512	1024	2297	2295
3	513	1368	2216	2214
4	512	2048	2681	2679
5	515	3296	3804	3802
6	516	5504	5927	5925

Table 4: Cost for the hash $\mathsf{WindowCRHFDLP}$ with $2(\log(q)+1) = 512$ bit input rounded up to be divisible by w.

w	Bit input	Lookup table size	Const	Vars
1	768	768	3840	3838
2	768	1536	3449	3447
3	768	2048	3321	3319
4	768	3072	4025	4023
5	770	4928	5691	5689
6	768	8192	8825	8823

Table 5: Cost for the hash $\mathsf{WindowCRHFDLP}$ with $3(\log(q)+1) = 768$ bit input rounded up to be divisible by w.

w	Bit input	Lookup table size	Const	Vars
1	1024	1024	5120	5118
2	1024	2048	4601	4599
3	1026	2736	4439	4437
4	1024	4096	5369	5367
5	1025	6560	7578	7576
6	1026	10944	11792	11790

Table 6: Cost for the hash WindowCRHFDLP with $4(\log(q)+1) = 1024$ bit input rounded up to be divisible by w.

k	N	Anonymity Set	Path		Tree		Total
			Const	Vars	Const	Vars	Const/Vars
2	29	2^{29}	29	145	94076	93960	94105
3	19	$2^{31} \geq 3^{19} \geq 2^{29}$	38	114	92435	92359	92473
4	15	$2^{30} = 4^{15}$	45	105	97485	97425	97530
2	64	2^{64}	64	320	207616	207360	207680
3	41	$2^{65} \geq 3^{41} \geq 2^{64}$	82	246	199465	199301	199547
4	33	$2^{66} = 4^{33}$	99	231	214467	214335	214566

Table 7: Cost of a k-ary Merkle tree instantiated with WindowCRHFDLP at window size $w = 3$. The path includes the nodes (hash outputs) and the control bits. The constraints on the path include verifying that the control bits are bits. The path verification has not been included in the total constraint number for the tree.

Operations	Const	Vars
$2 \cdot \mathsf{PK-Consistency}$	8722	8698
$2 \cdot \mathsf{InvPRF}$	9212	9204
$2 \cdot \mathsf{InvPRF+}$	9218	9210
$4 \cdot \mathsf{Comm}$	5432	5424
$4 \cdot \mathsf{H}$	8864	8856
$16 \cdot \mathsf{SplitIntegerToBit}$ for splitting $pk^{in,j}$, $pk^{out,k}$, $c^{in,j}$, $c^{out,k}$ for input to H	4096	4080
$\mathsf{CheckBit}$ for $\hat{sk}_j$, sk^S_j, $r^{in,j}$, $r^{out,k}$, $x^{in,j}$, $a^{in,j}$, $b^{in,j}$ and the coin values (rounded to fit the window $w = 3$)	3784	3784
$\mathsf{CheckValues}$	67	65
Other variables: $pk^{in,j}$, $pk^{out,k}$, $c^{in,j}$, $c^{out,k}$, sn_j, sn'_j, $cm^{in,j}$, $X^{in,j}$, $\hat{pk}^j$	0	18
Total:	49395	49339

Table 8: Size of the arithmetic circuit for our pour transaction without counting the hash trees.

k	N	Total lookup table			**Total Const**	**Total Vars**
		Elements	Plain points	Compressed		
2	29	4053	≈ 0.26MB	≈ 0.13MB	237605	237549
3	19	5501	≈ 0.35MB	≈ 0.17MB	234341	234285
2	64	4053	≈ 0.26MB	≈ 0.13MB	464755	464699
3	41	5501	≈ 0.35MB	≈ 0.17MB	448489	448433

Table 9: Cost of the system with two k-ary Merkle trees and window size $w = 3$. For $k = 3$ we assume the CRHF for the hash tree has a separate lookup table.

Operations	Const
Check cm_1^{old}, cm_2^{old} are in the Merkle tree for 29 levels	$2 \cdot 816669$
Check cm_1^{old}, cm_2^{old} are in the Merkle tree for 64 levels	$2 \cdot 1802304$
(Cost of a single level)	(28161)
Check computation of sn_1^{old}, sn_2^{old}	$2 \cdot 27904$
Check computation of $a_{pk,1}^{old}, a_{pk,2}^{old}$	$2 \cdot 27904$
Check computation of $cm_1^{old}, cm_2^{old}, cm_1^{new}, cm_2^{new}$	$4 \cdot 83712$
Ensure that $v_1^{new} + v_2^{new} + v_{pub} = v_1^{old} + v_2^{old}$	1
Ensure that $v_1^{old} + v_2^{old} < 2^{64}$	65
Verifying the computation of h_1 and h_2	$2 \cdot 27904$
Misc	2384
Total for $k = 29$ according to [36]:	2138060
Total for $k = 64$ according to [53]:	4109330

Table 10: The estimated number of constraints according to the ZCash specification [36] (smaller anonymity set) and the Zerocash paper [53].

5.3 Instantiation from SIS

In this section we describe an instantiation of the commitment scheme and CRHF using the Ajtai function [1]. First we recall some fundamental definitions.

Definition 20. *A full-dimensional lattice in $\mathcal{R}^m$ is a discrete subgroup $\mathcal{L} = \{\mathbf{B}x : x \in \mathbb{Z}^m\}$, where $\mathbf{B} = [\vec{b}_1, \ldots, \vec{b}_m] \in \mathbb{Z}^{m \times m}$ is a matrix of linearly independent vectors. The matrix $\mathbf{B}$ is called the basis of $\mathcal{L}$. The rank of $\mathcal{L}$ is the rank of $\mathbf{B}$ and if the rank equal m, the lattice is called full rank.*

Definition 21. *A lattice $\mathcal{L}$ is called a q-ary if $q\mathbb{Z} \subseteq \mathcal{L}$. For $q \in \mathbb{N}$ and $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, we define the following q-ary's.*

$$\Lambda_q^{\perp}(\mathbf{A}) = \{\vec{v} \in \mathbb{Z}^m : \mathbf{A}\vec{v} = \vec{0} \mod q\}.$$

Definition 22. *Short Integer Solution (SIS) Given $n, m, q \in \mathbb{N}$, a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ chosen uniformly at random and a norm bound $1 \leq \beta < q$, the $\boldsymbol{SIS}_{n,q,\beta,m}$ problem is to find $\vec{v} \in \Lambda_q^{\perp}(\mathbf{A})$ with $0 < ||\vec{v}|| \leq \beta$.*

Definition 23. *Ajtai Hash Given $n, m, q \in \mathbb{N}$, a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ chosen uniformly at random, the Ajtai hash function $\mathsf{H}_A : \{0,1\}^m \to \mathbb{Z}_q^n$ is defined as follows.*

$$\mathsf{H}_{\mathbf{A}}(\vec{z}) \to \mathbf{A}\vec{z} \in \mathbb{Z}_q^n.$$

It is easy to see that given a collision in the Ajtai hash, i.e. vectors $\vec{v}_1, \vec{v}_2 \in \{0,1\}^m$ s.t. $\mathbf{A}\vec{v}_1 - \mathbf{A}\vec{v}_2 = \vec{0}$, we may easily compute $\vec{v}_1 - \vec{v}_2 = \vec{v} \in \{-1, 0, 1\}^m$, s.t. $\mathbf{A}\vec{v} = \vec{0}$. The vector $\vec{v}$ is then the solution to the $\mathbf{SIS}_{n,q,\beta,m}$, where $0 < \beta \leq \sqrt{m}$. Therefore, it is immediate that we may use the Ajtai hash as a CRHF.

In order to evaluate the security level of lattice-based hash functions Micciancio and Regev in [43] suggest to use a variant of the generalized birthday attack [14, 55], which has been used to choose parameters for the SWIFFT hash function [40]. The method solves the SIS problem for a q-ary $\Lambda_q^{\perp}(\mathbf{A})$. We recall this method at Algorithm 1.

Algorithm 1 Combinatorial algorithm for SIS from [43, 40].

```
1:  procedure SIS-Solver($\mathbf{A} \in \mathbb{Z}_q^{n,m}, b \in \mathbb{N}, k \in \mathbb{N}$)
2:      Divide columns of $\mathbf{A}$ in $2^k$ groups $[T_i \in \mathbb{Z}_q^{m/2^k}]_{i=0}^{2^k-1}$.
3:      for $i \in \{0, \dots, 2^k - 1\}$ do
4:          Create a list $L_i^0$ from linear combinations of columns in $T_i$ with
            coefficients in $\{-b, \dots, b\}$.
5:      end for
6:      for $j \in \{0, \dots, k-1\}$ do
7:          for $i \in \{0, \dots, 2^{k-j-1} - 1\}$ do
8:              For all $\vec{x} \in L_{2\cdot i}^j$ and $\vec{y} \in L_{2\cdot i+1}^j$, let $\vec{z} = \vec{x} + \vec{y} = [\vec{0}\ \vec{z'}]^t$. If
                $dim(\vec{z'}) = log_q(2b+1)^{m/2^k}$, then add $\vec{z}$ to $L_i^{j+1}$.
9:          end for
10:     end for
11:     Return the linear combination corresponding to the $\vec{0}$ vector in
        $L_0^k$ if there is such.
12: end procedure
```

As shown in [43] the parameter k in Algorithm 1, is chosen such that $n \approx (k+1)\log_q(2b+1)^{m/2^k}$. Then, we can expect the list L_0^k to contain a all zero vector which is given by a combination of the columns of $\mathbf{A}$ bounded by b, hence with coefficients in $\{-1, 0, 1\}$. This combination is the desired short lattice vector. In our case we will only be interested in the case $b = 1$.

Let us now, assess the complexity of Algorithm 1. After Step 2, we obtain 2^k groups $T_i \in \mathbb{Z}_q^{m/2^k}$. This step is for free. Then we create 2^k lists L_i^0, each containing $(2b+1)^{m/2^k}$ linear combinations of the columns from T_i. Thus we have to create $2^k \cdot (2b+1)^{m/2^k}$ vectors in $\mathbb{Z}_q^n$. Then we create a binary tree from the leaves L_i^0. To create a parent node we sum two vectors from both its children nodes and add the summed vector if it starts with $j\log_q(2b+1)^{m/2^k}$ zeros, where j is the level of the tree. Note that the parent also contains around $(2b+1)^{m/2^k}$ vectors. Since, we build a full-binary tree we have need to create $2^{k-1} - 1$ nodes (not counting the leaves), thus going up the tree we need to create $(2^{k-1} - 1)(2b+1)^{m/2^k}$ vectors giving us in total $(2^k - 1)(2b+1)^{m/2^k}$ vectors to create.

At Table 11 we show the cost in vector operations for different configurations of the Ajtai hash.

m	n	k	$(k+1)\log_q(2b+1)^{m/2^k}$	vectors
512	1	4	0.9944	$\approx 2^{2^{54}}$
512	2	3	1.5911	$\approx 2^{104}$
512	3	2	2.3867	$\approx 2^{204}$
768	1	5	0.8950	$\approx 2^{42}$
768	2	4	1.4917	$\approx 2^{79}$
768	3	3	2.3867	$\approx 2^{154}$
1024	1	5	1.1933	$\approx 2^{55}$
1024	2	4	1.9889	$\approx 2^{105}$
1024	3	3	3.1823	$\approx 2^{205}$

Table 11: Cost of executing algorithm 1 for modulus $q \approx 2^{255}$. In our case we are only interested in $b = 1$.

Kosba et al. in [38] gave an analysis for the SIS problem based on lattice reduction attacks. In more detail, they repeat the experiments from [49] on BKZ with modulus $q = 2^{254}$ and provide an estimate security level based on [2] and [24]. We recall the method proposed in [38] for calculating the bit security in Algorithm 2.

Algorithm 2 Calculating SIS bit security [38].

```
1: procedure SIS-Bit($n, m, q, \beta$)
2:     Compute $\lceil m^* \leftarrow \frac{2n\log(q)}{\log(\beta)} \rceil$.
3:     Compute $\delta_0 \leftarrow (\frac{\beta}{q^{n/m^*}})^{1/m^*}$.
4:     Compute BKZ2.0 estimation for $T_{ACF15}(\delta_0)$.
5: end procedure
```

In Algorithm 2, the function $T_{ACF15}(\delta_0)$ introduced by Albreht et al. in [2] is defined as $T_{ACF15}(\delta_0) = 2^{0.009/(\log^2(\delta_0)+4.1}$.

m	n	m^*	δ_0	T_{ACF15}
512	1	114	1.0138	26.93
512	2	227	1.0069	95.44
512	3	340	1.0045	209.61
512	4	454	1.0034	369.45
765	1	107	1.0157	21.89
765	2	213	1.0078	75.27
765	3	320	1.0052	164.23
765	4	426	1.0039	288.77
1024	1	102	1.0171	19.08
1024	2	204	1.0085	64.02
1024	3	306	1.0056	138.93
1024	4	408	1.0042	243.81

Table 12: Estimates for the security level according to Algorithm 2 for modulus $q \approx 2^{255}$ and $\beta = \sqrt{m}$.

The commitment scheme. Kawachi et al. [37] study the usability of the Ajtai hash as a statistically hiding commitment scheme. Let us consider the following construction.

$\mathsf{Setup}_A(1^{\ell_m}, 1^{\ell_R}, n_C, q)$: The algorithm takes as input the message bit length ℓ_m and the modulus and the output length n_C. The algorithm chooses a uniformly random matrix $\mathbf{A} \overset{R}{\leftarrow} \mathbb{Z}_q^{n_C \times \ell_m + \ell_R}$ and outputs the parameters $pp = (\mathbf{A})$.

$\mathsf{CommA}(pp, m)$: Given the parameters pp and the message $\vec{m} \in \{0,1\}^{\ell_m}$, this algorithm chooses $\vec{r} \overset{R}{\leftarrow} \{0,1\}^{\ell_R}$ uniformly at random, computes $\vec{c} \leftarrow \mathsf{H}_{\mathbf{A}}(\vec{r}||\vec{m})$ and outputs the commitment $\vec{c}$.

The computationally binding property immediately follows from the collision-resistance of $\mathsf{H}_{\mathbf{A}}$. Kawachi et al. in [37] showed that the above construction is also statistically-hiding for $\ell_m = \ell_R$ and $2\ell_m \geq 10 n_C \log(q)$. Let us recall the definition of statistical distance.

Definition 24. *Let ϕ_1 and ϕ_2 be probability density functions on a finite set S. We define the satatistical distance between ϕ_1 and ϕ_2 as*

$$\Delta(\phi_1, \phi_2) = \frac{\sum_{x \in S} |\phi_1(x) - \phi_2(x)|}{2}.$$

Their the analysis of statistical hiding from [37], the authors rely on a claim from [46], which we recall below.

Claim 1. *Let $\mathbb{G}$ be some finite Abelian group and let ℓ be some integer. For any ℓ elements $g_1, \ldots, g_\ell \in \mathbb{G}$ consider the statistical distance between the uniform distribution on $\mathbb{G}$ and the distribution given by the sum of a random subset* *of $g_1, \ldots, g_\ell$. Then the expectation of this statistical distance over a uniform choice of $g_1, \ldots, g_\ell \in \mathbb{G}$ is at most $\sqrt{|\mathbb{G}|/2^\ell}$. In particular, the probability that this statistical distance is more than $\sqrt[4]{|\mathbb{G}|/2^\ell}$ is at most $\sqrt[4]{|\mathbb{G}|/2^\ell}$.*

In our case the modulus q will be rather big and the requirement that $2\ell_m \geq 10n_C \log(q)$ makes the parameter choices rather impractical. For example, even for $n_C = 1$, we would have that $\ell_m = 1275$, which is a huge waste. In our setting we will set $\ell_m = 64$. Then, we allow ℓ_R to be a multiple of ℓ_m. In particular we denote $\ell_R = c \cdot \ell_m$ for some $c \in \mathbb{N}$. Moreover, we will assume the $\ell_m = \log(q)/4$. Taking the above into account we can write $\ell_R = c \cdot \log(q)/4$. By Claim 1 we have the following.

$$\Big(\frac{qn_C}{2^{\ell_R}}\Big)^{1/4} = \Big(\frac{qn_C}{2^{c\log(q)/4}}\Big)^{1/4}$$

Let $c = 4(4c' + 1)$, for some $c' \in \mathbb{N}$. Then

$$\Big(\frac{qn_C}{2^{c\log(q)/4}}\Big)^{1/4} = \Big(\frac{qn_C}{2^{(4c'+1)\log(q)}}\Big)^{1/4} = \Big(\frac{n_C}{2^{(4c')\log(q)}}\Big)^{1/4} = \Big(\frac{\sqrt[4]{n_C}}{q^{c'}}\Big)$$

Now we can estimate $\ell_R = (4c'+1)log(q)$. So the statistical distance between $\mathsf{CommA}(pp, \vec{0})$ and the uniform distribution is $\leq \frac{\sqrt[4]{n_C}}{q^{c'}}$. Equivalently we may write that for $\ell_R = c \cdot \log(q)/4$ bits of randomness, the statistical distance is $\frac{\sqrt[4]{n_C}}{q^{(c-4)/16}}$. Note, that in case the bits of the message would be distributed uniformly, then we have $\ell_R + \ell_{max} = c\log(q)/2$, and the the distance is $\leq \frac{\sqrt[4]{n_C}}{q^{(c-2)/8}}$. Hence, we may write that for any messages $\vec{m} \in \{0,1\}^{\ell_{max}}$, the distance of $\mathsf{CommA}(pp, \vec{m})$ and the uniform distribution with high probability is at most $\frac{\sqrt[4]{n_C}}{q^{c'}}$, thus the statistical distance between commitments of two messages is as most $\frac{2\sqrt[4]{n_C}}{q^{c'}}$. At Table 13 we present the results of estimating the number of random bits ℓ_R. In our calculation we take only small n_C into account. In particular $n_C \leq 16$, so we round up $\sqrt[4]{n_C} \leq 2$. We denote the probability $P_{stat}(q, c') = \frac{2}{q^{c'}}$.

ℓ_{max}	c'	$P_{stat}(q,c')$	ℓ_R	m^*	δ_0	T_{ACF15}
64	1/3	$\approx 2^{-84}$	594	327	1.0050	179.62
64	2/5	$\approx 2^{-101}$	663	322	1.0051	169.24
64	1/2	$\approx 2^{-126}$	765	316	1.0053	156.71

Table 13: Estimates for the number of randomness bits for modulus $q \approx 2^{255}$. The security parameter for binding is calculated for $m = \ell_{max} + \ell_R$ and $n_C = 3$.

The Collision Resistant Hash. The hash function will simply be instantiated with $\mathsf{H}_{\mathbf{A}}$. Here however the input to the hash is one point and the commitment.

We will use the same optimisation as in the DLP case and input only the compressed point to $\mathsf{H}_{\mathbf{A}}$. In the DLP commitment version we have $2(\log(q)+1)$ bit input. For the SIS based commitment the input to this hash function will be $\log(q)+1+n_C\log(q)$ bits.

Later we will need to define a hash function for the k-ary Merkle tree. In this case we will need to set the output of the hash to fit the requirements of a k-ary hash tree. In particular, the dimensions for $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$, will be set to $n = km\log(q)$.

Putting the SIS instantiation together. According to our security estimates a reasonable way to chose the parameter n_C and n for the commitment scheme and hash is $n_c = n = 3$. The, we need to remember to count 3 split gates for each commitment output. Then the commitment scheme will take as input in total 727 bits ($\ell_{max} = 64$ and $\ell_R = 663$). The hash H will take as input $(n_C+1)\cdot\log(q)+1 = 1021$ bits ($3n_C$ elements in $\mathbb{Z}_q$ and one compressed point). For a 2 and 3-ary binary tree be will instantiation H with the same hash function as for the binary tree with a random public padding.

For the hash tree may consider to following parametrization for the Ajtai hash. For a binary tree $n = 3$ and $m = 1530$. For a 3-ary tree $n = 3$ and $m = 2295$. Finally, a 4-ary tree $n = 4$ and $m = 4080$.

At Table 14, we show the security estimate for the above parameter choices.

m	n	m^*	δ_0	T_{ACF15}	k	$(k+1)\log_q(2b+1)^{m/2^k}$	vectors
727	3	322	1.0051	169.24	3	2.2593	$\approx 2^{145}$
1021	3	307	1.0057	139.17	3	3.1730	$\approx 2^{204}$
1530	3	290	1.0064	111.74	4	2.9718	$\approx 2^{154}$
2295	3	275	1.0071	90.89	5	2.6746	$\approx 2^{117}$
4080	4	341	1.0061	119.92	6	2.7736	$\approx 2^{105}$

Table 14: Estimates for the security level according to the combinatorial method at Algorithm 1 and Algorithm 2 for modulus $q \approx 2^{255}$ and $\beta = \sqrt{m}$.

5.4 Performance Estimates for SIS Instantiation

In this section we show the performance estimates for the parameter choices given in Section 5.3. First at Table 15 we present the cost of the hash trees. Again we may see, that 3-ary trees perform slightly better, however in this case we need to remember that the security of the underlying hash is much weaker. At Table 16 we summarize the cost of all checks except for the hash trees. Finally, at Table 17 we summarize the cost of the system.

k	N	Anonymity Set	Path		Tree		Total	
			Const	Vars	Const	Vars	Const	Vars
2	29	2^{29}	29	116	44805	44631	44834	44747
3	19	$2^{31} \geq 3^{19} \geq 2^{29}$	38	95	44061	43947	44099	44042
4	15	$2^{30} = 4^{15}$	45	105	61860	61740	61905	61845
2	64	2^{64}	64	256	98880	98496	98944	98752
3	41	$2^{65} \geq 3^{41} \geq 2^{64}$	82	205	95079	94833	95161	95038
4	33	$2^{66} = 4^{33}$	99	231	136092	135828	136191	136059

Table 15: Cost of a k-ary Merkle tree instantiated with Ajtai hash with $n = 3$ for 2 and 3-ary trees and $n = 4$ for 4-ary trees. The path includes the nodes (hash outputs) and the control bits. The constraints on the path include verifying that the control bits are bits. The path verification has not been included in the total constraint number for the tree.

Operations	Const	Vars
$2 \cdot \mathsf{PK-Consistency}$	8722	8698
$2 \cdot \mathsf{InvPRF}$	9212	9204
$2 \cdot \mathsf{InvPRF+}$	9218	9210
$4 \cdot \mathsf{Comm}$	12	0
$4 \cdot \mathsf{H}$	12	0
$20 \cdot \mathsf{SplitIntegerToBit}$ for splitting $pk^{in,j}$, $pk^{out,k}$, $c^{in,j}$, $c^{out,k}$ for input to H	5120	5100
$\mathsf{CheckBit}$ for $\hat{sk}_j$, sk_j^S, $r^{in,j}$, $r^{out,k}$, $x^{in,j}$, $a^{in,j}$, $b^{in,j}$ and the coin values (rounded to fit the window $w = 3$)	5172	5172
$\mathsf{CheckValues}$	67	65
Other variables: $pk^{in,j}$, $pk^{out,k}$, $c^{in,j}$, $c^{out,k}$, sn_j, sn'_j, $cm^{in,j}$, $X^{in,j}$, $\hat{pk}^j$	0	38
Total:	37535	37487

Table 16: Size of the arithmetic circuit based on Ajtai hash and SIS commitment scheme for our pour transaction without counting the hash trees. Scalar multiplication for InvPRF and $\mathsf{InvPRF+}$ is counted for window $w = 3$.

k	N	Table size			**Total Const**	**Total Vars**
		Elements	Plain points	Compressed		
2	29	$2016 \cdot \mathbb{G} + 6771 \cdot \mathbb{Z}_q$	≈ 0.34MB	≈ 0.28MB	127203	126981
3	19	$2016 \cdot \mathbb{G} + 9066 \cdot \mathbb{Z}_q$	≈ 0.42MB	≈ 0.35MB	125733	125571
2	64	$2016 \cdot \mathbb{G} + 6771 \cdot \mathbb{Z}_q$	≈ 0.34MB	≈ 0.28MB	235423	234991
3	41	$2016 \cdot \mathbb{G} + 9066 \cdot \mathbb{Z}_q$	≈ 0.42MB	≈ 0.35MB	227857	227563

Table 17: Cost of the system with two k-ary Merkle trees and window size $w = 3$. The table size column includes lookup table for scalar multiplication and matrices for the Ajtai hashes.

6 Conclusions

To sum up our performance evaluation. At an anonymity level 2^{29} our DLP based instantiation costs less than 2^{18} constraints and SIS based instantiation less than 2^{17} constraints. When instantiated with 3-ary trees instead of binary trees, we can save 3264 constraints in case of DLP instantiation and 1470 in case of SIS based instantiation. In comparison, Zerocash [53], needs about 2^{29} constraints. At anonymity level 2^{64} our DLP based instantiation costs roughly twice as much constraints as at anonymity level 2^{19}, but still less than 2^{19} constraints and SIS based instantiation less than 2^{18} constraints. What is interesting, the SIS based system performs slightly better at this anonymity set, than the DLP based at anonymity set 2^{29}. When instantiated with 3-ary trees instead of binary trees, we can reduce the size of the arithmetic constraint by 16266 constraints in case of DLP instantiation and 7566 in case of SIS based instantiation. In comparison, Zerocash [53], needs about 2^{22} constraints. What is worth noting, is that for the 3-ary tree case, the anonymity set is actually bigger than for binary trees.

In the current version of this article we provide mainly the complexity analysis of the arithmetic circuits. In near future we plan to analyse the system when instantiated by concrete argument systems. Some argument systems, may however require to slightly change the arithmetic circuit. Any necessary redesign will mostly be dictated by the security proofs and properties of the argument system under consideration. Additionally, we are missing a reductionist security analysis and a formal model, as it turns out the original definition of Zerocash [53] has some issues pointed out in [39]. In Section 3 we only give an informal description of the algorithms and security requirements.

References

[1] M. Ajtai. "Generating Hard Instances of Lattice Problems (Extended Abstract)". In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing.* STOC '96. Philadelphia, Pennsylvania, USA: ACM,

1996, pp. 99–108. ISBN: 0-89791-785-5. DOI: 10.1145/237814.237838. URL: http://doi.acm.org/10.1145/237814.237838.

[2] Martin R. Albrecht et al. “On the Complexity of the BKW Algorithm on LWE”. In: *Des. Codes Cryptography* 74.2 (Feb. 2015), pp. 325–354. ISSN: 0925-1022. DOI: 10.1007/s10623-013-9864-x. URL: http://dx.doi.org/10.1007/s10623-013-9864-x.

[3] Simon Barber et al. “Bitter to Better — How to Make Bitcoin a Better Currency”. In: *Financial Cryptography and Data Security*. Ed. by Angelos D. Keromytis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 399–414. ISBN: 978-3-642-32946-3.

[4] Mihir Bellare et al. “Key-Privacy in Public-Key Encryption”. In: *Advances in Cryptology — ASIACRYPT 2001*. Ed. by Colin Boyd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 566–582. ISBN: 978-3-540-45682-7.

[5] Eli Ben-Sasson et al. *Scalable Zero Knowledge via Cycles of Elliptic Curves*. Cryptology ePrint Archive, Report 2014/595. http://eprint.iacr.org/2014/595. 2014.

[6] Eli Ben-Sasson et al. “SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge”. In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 90–108. ISBN: 978-3-642-40084-1.

[7] Eli Ben-Sasson et al. “Succinct Non-interactive Zero Knowledge for a Von Neumann Architecture”. In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC’14. San Diego, CA: USENIX Association, 2014, pp. 781–796. ISBN: 978-1-931971-15-7. URL: http://dl.acm.org/citation.cfm?id=2671225.2671275.

[8] Daniel J. Bernstein and Tanja Lange. “Faster Addition and Doubling on Elliptic Curves”. In: *Advances in Cryptology – ASIACRYPT 2007*. Ed. by Kaoru Kurosawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 29–50. ISBN: 978-3-540-76900-2.

[9] Daniel J. Bernstein et al. “Twisted Edwards Curves”. In: *Progress in Cryptology – AFRICACRYPT 2008*. Ed. by Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 389–405. ISBN: 978-3-540-68164-9.

[10] Nir Bitansky and Alessandro Chiesa. “Succinct Arguments from Multi-prover Interactive Proofs and Their Efficiency Benefits”. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 255–272. ISBN: 978-3-642-32009-5.

[11] 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.

[12] Nir Bitansky et al. "Succinct Non-interactive Arguments via Linear Interactive Proofs". In: *Theory of Cryptography*. Ed. by Amit Sahai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 315–333. ISBN: 978-3-642-36594-2.

[13] Nir Bitansky et al. "The Hunting of the SNARK". In: *Journal of Cryptology* 30.4 (Oct. 2017), pp. 989–1066. ISSN: 1432-1378. DOI: 10.1007/s00145-016-9241-9. URL: https://doi.org/10.1007/s00145-016-9241-9.

[14] Avrim Blum, Adam Kalai, and Hal Wasserman. "Noise-tolerant Learning, the Parity Problem, and the Statistical Query Model". In: *J. ACM* 50.4 (July 2003), pp. 506–519. ISSN: 0004-5411. DOI: 10.1145/792538.792543. URL: http://doi.acm.org/10.1145/792538.792543.

[15] Dan Boneh and Xavier Boyen. "Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles". In: *Advances in Cryptology - EUROCRYPT 2004*. Ed. by Christian Cachin and Jan L. Camenisch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 223–238. ISBN: 978-3-540-24676-3.

[16] Dan Boneh, Hart William Montgomery, and Ananth Raghunathan. "Algebraic Pseudorandom Functions with Improved Efficiency from the Augmented Cascade". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 131–140. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866323. URL: http://doi.acm.org/10.1145/1866307.1866323.

[17] Sean Bowe. *ZCash blog. BLS12-381: New zk-SNARK Elliptic Curve Construction?* Mar. 11, 2017. URL: https://z.cash/blog/new-snark-curve.html.

[18] Sean Bowe. *ZCash blog. Cultivating Sapling: Faster zk-SNARKs*. Sept. 13, 2017. URL: https://blog.z.cash/cultivating-sapling-faster-zksnarks/.

[19] Sean Bowe. *ZCash blog. Cultivating Sapling: New Crypto Foundations*. July 26, 2017. URL: https://blog.z.cash/cultivating-sapling-new-crypto-foundations/.

[20] Sean Bowe. *ZCash blog. The Near Future of Zcash*. Apr. 4, 2017. URL: https://blog.z.cash/the-near-future-of-zcash/.

[21] Sean Bowe, Ariel Gabizon, and Matthew D. Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. Cryptology ePrint Archive, Report 2017/602. `https://eprint.iacr.org/2017/602`. 2017.

[22] Sean Bowe, Ariel Gabizon, and Ian Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Report 2017/1050. `https://eprint.iacr.org/2017/1050`. 2017.

[23] David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graaf. "An Improved Protocol for Demonstrating Possession of Discrete Logarithms and Some Generalizations". In: *Advances in Cryptology — EUROCRYPT' 87*. Ed. by David Chaum and Wyn L. Price. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 127–141. ISBN: 978-3-540-39118-0.

[24] Yuanmi Chen and Phong Q. Nguyen. "BKZ 2.0: Better Lattice Security Estimates". In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–20. ISBN: 978-3-642-25385-0.

[25] Ivan Damgård. *On Σ-protocols*. 2002.

[26] George Danezis et al. "Square Span Programs with Applications to Succinct NIZK Arguments". In: *Advances in Cryptology – ASIACRYPT 2014*. Ed. by Palash Sarkar and Tetsu Iwata. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 532–550. ISBN: 978-3-662-45611-8.

[27] Yevgeniy Dodis and Aleksandr Yampolskiy. "A Verifiable Random Function with Short Proofs and Keys". In: *Public Key Cryptography - PKC 2005: 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005. Proceedings*. Ed. by Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 416–431. ISBN: 978-3-540-30580-4. DOI: `10.1007/978-3-540-30580-4_28`. URL: `https://doi.org/10.1007/978-3-540-30580-4_28`.

[28] Christina Garman, Matthew Green, and Ian Miers. *Accountable Privacy for Decentralized Anonymous Payments*. Cryptology ePrint Archive, Report 2016/061. `https://eprint.iacr.org/2016/061`. 2016.

[29] Rosario Gennaro et al. "Quadratic Span Programs and Succinct NIZKs without PCPs". In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 626–645. ISBN: 978-3-642-38348-9.

[30] Craig Gentry and Daniel Wichs. "Separating Succinct Non-interactive Arguments from All Falsifiable Assumptions". In: *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*. STOC '11. San Jose, California, USA: ACM, 2011, pp. 99–108. ISBN: 978-1-4503-0691-1. DOI: `10.1145/1993636.1993651`. URL: `http://doi.acm.org/10.1145/1993636.1993651`.

[31] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology – EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326. ISBN: 978-3-662-49896-5. DOI: `10.1007/978-3-662-49896-5_11`. URL: `https://doi.org/10.1007/978-3-662-49896-5_11`.

[32] Jens Groth and Markulf Kohlweiss. "One-Out-of-Many Proofs: Or How to Leak a Secret and Spend a Coin". In: *Advances in Cryptology - EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 253–280. ISBN: 978-3-662-46803-6.

[33] Jens Groth and Mary Maller. "Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs". In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 581–612. ISBN: 978-3-319-63715-0.

[34] Huseyin Hisil et al. "Twisted Edwards Curves Revisited". In: *Advances in Cryptology - ASIACRYPT 2008*. Ed. by Josef Pieprzyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 326–343. ISBN: 978-3-540-89255-7.

[35] Dennis Hofheinz and Eike Kiltz. "Programmable Hash Functions and Their Applications". In: vol. 25. 3. July 2012, pp. 484–527. DOI: `10.1007/s00145-011-9102-5`. URL: `https://doi.org/10.1007/s00145-011-9102-5`.

[36] Daria Hopwood et al. *ZCash Protocol Specification Verion 2017-0-beta-27*. 2017. URL: `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`.

[37] Akinori Kawachi, Keisuke Tanaka, and Keita Xagawa. "Concurrently Secure Identification Schemes Based on the Worst-Case Hardness of Lattice Problems". In: *Advances in Cryptology - ASIACRYPT 2008: 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*. Ed. by Josef Pieprzyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 372–389. ISBN: 978-3-540-89255-7. DOI: `10.1007/978-3-540-89255-7_23`. URL: `https://doi.org/10.1007/978-3-540-89255-7_23`.

[38] Ahmed Kosba et al. *C∅C∅: A Framework for Building Composable Zero-Knowledge Proofs*. Cryptology ePrint Archive, Report 2015/1093. `http://eprint.iacr.org/2015/1093`. 2015.

[39] A. Kosba et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 839–858. DOI: `10.1109/SP.2016.55`.

[40] Vadim Lyubashevsky et al. "SWIFFT: A Modest Proposal for FFT Hashing". In: *Fast Software Encryption*. Ed. by Kaisa Nyberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 54–72. ISBN: 978-3-540-71039-4.

[41] Gregory Maxwell. *CoinJoin: Bitcoin privacy for the real world*. Aug. 22, 2013. URL: `https://bitcointalk.org/index.php?topic=279249`.

[42] Sarah Meiklejohn et al. "A Fistful of Bitcoins: Characterizing Payments Among Men with No Names". In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. Barcelona, Spain: ACM, 2013, pp. 127–140. ISBN: 978-1-4503-1953-9. DOI: `10.1145/2504730.2504747`. URL: `http://doi.acm.org/10.1145/2504730.2504747`.

[43] Daniele Micciancio and Oded Regev. "Lattice-based Cryptography". In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 147–191. ISBN: 978-3-540-88702-7. DOI: `10.1007/978-3-540-88702-7_5`. URL: `https://doi.org/10.1007/978-3-540-88702-7_5`.

[44] Shen Noether. *Ring Signature Confidential Transactions for Monero*. Cryptology ePrint Archive, Report 2015/1098. `https://eprint.iacr.org/2015/1098`. 2015.

[45] B. Parno et al. "Pinocchio: Nearly Practical Verifiable Computation". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 238–252. DOI: `10.1109/SP.2013.47`.

[46] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*. STOC '05. Baltimore, MD, USA: ACM, 2005, pp. 84–93. ISBN: 1-58113-960-8. DOI: `10.1145/1060590.1060603`. URL: `http://doi.acm.org/10.1145/1060590.1060603`.

[47] Fergal Reid and Martin Harrigan. "An Analysis of Anonymity in the Bitcoin System". In: *Security and Privacy in Social Networks*. Ed. by Yaniv Altshuler et al. New York, NY: Springer New York, 2013, pp. 197–223. ISBN: 978-1-4614-4139-7. DOI: `10.1007/978-1-4614-4139-7_10`. URL: `https://doi.org/10.1007/978-1-4614-4139-7_10`.

[48] Dorit Ron and Adi Shamir. "Quantitative Analysis of the Full Bitcoin Transaction Graph". In: *Financial Cryptography and Data Security*. Ed. by Ahmad-Reza Sadeghi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 6–24. ISBN: 978-3-642-39884-1.

[49] Markus Rückert and Michael Schneider. *Estimating the Security of Lattice-based Cryptosystems*. Cryptology ePrint Archive, Report 2010/137. `https://eprint.iacr.org/2010/137`. 2010.

[50] Tim Ruffing and Pedro Moreno-Sanchez. *Mixing Confidential Transactions: Comprehensive Transaction Privacy for Bitcoin*. Cryptology ePrint Archive, Report 2017/238. `https://eprint.iacr.org/2017/238`. 2017.

[51] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. "CoinShuffle: Practical Decentralized Coin Mixing for Bitcoin". In: *Computer Security - ESORICS 2014*. Ed. by Mirosław Kutyłowski and Jaideep Vaidya. Cham: Springer International Publishing, 2014, pp. 345–364. ISBN: 978-3-319-11212-1.

[52] Nicolas van Saberhagen. *CryptoNote v 2.0*. 2013. URL: `https://cryptonote.org/whitepaper.pdf`.

[53] E. B. Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. May 2014, pp. 459–474. DOI: `10.1109/SP.2014.36`.

[54] Shi-Feng Sun et al. "RingCT 2.0: A Compact Accumulator-Based (Linkable Ring Signature) Protocol for Blockchain Cryptocurrency Monero". In: *Computer Security – ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Cham: Springer International Publishing, 2017, pp. 456–474. ISBN: 978-3-319-66399-9. DOI: `10.1007/978-3-319-66399-9_25`. URL: `https://doi.org/10.1007/978-3-319-66399-9_25`.

[55] David Wagner. "A Generalized Birthday Problem". In: *Advances in Cryptology — CRYPTO 2002*. Ed. by Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 288–304. ISBN: 978-3-540-45708-4.

[56] Brent Waters. "Efficient Identity-Based Encryption Without Random Oracles". In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 114–127. ISBN: 978-3-540-32055-5.

[57] Zooko Wilcox, Nathan Wilcox, and Jack Gavigan. *ZCash blog. The Near Future of Zcash*. Feb. 21, 2017. URL: `https://blog.z.cash/the-near-future-of-zcash/`.

[58] ZCash. *ZCash blog. What is Jubjub?* URL: `https://z.cash/technology/jubjub.html`.

A Quadratic Constraints

In this section we describe the construction of the quadratic constraints which need to be implemented in the arithmetic circuit of the ZK-SNARK.
Let us remind, that the following constraints are over $\mathbb{F}_q$. In case, the input is not clearly marked do be constrained to other values (e.g. bits), we assume that the inputs are in $\mathbb{F}_q$.

We start by designing some basic constraint systems for bit operations.

```
CheckBit(b: b ∈ {0,1}):
```

1. $(\mathbf{b}) \cdot (\mathbf{b} - 1) = (0)$

Cost: 1 constraints and 0 additional variables.

$\mathsf{SplitIntegerToBit}(\mathsf{a}, [\mathsf{b}_i]_{i=0}^{n-1}\colon\ \mathsf{a} = \sum_{i=0}^{n-1} 2^i \mathsf{b}_i \wedge [\mathsf{b}_i \in \{0,1\}]_{i=0}^{n-1})$**:**

1. $\forall_{i \in \{0,\ldots,n-1\}}$
   (a) $\mathsf{CheckBit}(\mathbf{b} : \mathbf{b} \in \{0,1\})$
2. $(\sum_{i=0}^{n-1} \mathbf{b}_i 2^i) \cdot (1) = (\mathbf{a})$

Cost: $n+1$ constraints and 0 additional variables.

$\mathsf{CheckIntegerToBits}(\mathsf{a}, [\mathsf{b}_i \in \{0,1\}]_{i=0}^{n-1}\colon\ \mathsf{a} = \sum_{i=0}^{n-1} 2^i \mathsf{b}_i)$**:**

1. $(\sum_{i=0}^{n-1} \mathbf{b}_i 2^i) \cdot (1) = (\mathbf{a})$

Cost: 1 constraints and 0 additional variables.

The constraint below checks whether $\mathbf{v}^{in,1} + \mathbf{v}^{in,2} + v^{pub} = \mathbf{v}^{out,1} + \mathbf{v}^{out,2}$.

$\mathsf{CheckValues}([\mathsf{v}^{in,j} = [\mathsf{v}_i^{in,j} \in \{0,1\}]_{i=0}^{n-1}]_{j=1}^{2}, [\mathsf{v}^{out,j} = [\mathsf{v}_i^{out,j} \in \{0,1\}]_{i=0}^{n-1}]_{j=1}^{2}\colon\ \mathsf{v}^{in,1} + \mathsf{v}^{in,2} + v^{pub} = \mathsf{v}^{out,1} + \mathsf{v}^{out,2})$**:**

1. $(\sum_{i=0}^{n-1} 2^i \mathbf{v}_i^{out,1} + \sum_{i=0}^{n-1} 2^i \mathbf{v}_i^{out,2}) \cdot (1) = (\mathfrak{c})$
2. $\mathsf{SplitIntegerToBit}(\mathfrak{c}, [\mathfrak{c}_i]_{i=0}^{n-1}\colon\ \mathfrak{c} = \sum_{i=0}^{n-1} 2^i \mathfrak{c}_i \wedge [\mathfrak{c}_i \in \{0,1\}]_{i=0}^{n-1})$
3. $(\sum_{i=0}^{n-1} 2^i \mathbf{v}_i^{in,1} + \sum_{i=0}^{n-1} 2^i \mathbf{v}_i^{in,2} + v^{pub}) \cdot (1) = (\mathfrak{c})$

Cost: $n+3$ constraints and $n+1$ additional variables.

Multiplexers. In this paragraph we will show constructions for multiplexing integers and points. In order to conveniently denote a choice of an element a form a set S where the choice is encoded by a a set of bits $[b]$, we will write $a \in S_{[b]}$. In some cases however, it will be more convenient to denote a choice by an algebraic operation involving the control bits of the multiplexer. We will denote a "IF" statement as $(\texttt{Statement1})_{\mathbf{b}=0}, (\texttt{Statement2})_{\mathbf{b}=0}$.

We start by showing a 2-to-1 multiplexer and a 4-to-1 multiplexer.

2-1-Multiplex($\mathbf{b} \in \{0,1\}, \mathbf{x}, \mathbf{y}, \mathbf{z}$: $(\mathbf{z} \in \{\mathbf{x}, \mathbf{y}\}_{\mathbf{b}})$:

1. $(\mathbf{b}) \cdot (\mathbf{y} - \mathbf{x}) = (\mathbf{z} - \mathbf{x})$

Cost: 1 constraints and 0 additional variables.

4-1-Multiplex($(\mathbf{b}_0, \mathbf{b}_1 \in \{0,1\}), \mathbf{y}$: $\mathbf{y} \in \{x_0, x_1, x_2, x_3\}_{[\mathbf{b}_i]_{i=0}^{1}}$):

1. $(\mathbf{b}_0) \cdot (-x_0 + \mathbf{b}_1 x_0 + x_1 - \mathbf{b}_1 x_1 - \mathbf{b}_1 x_2 + \mathbf{b}_1 x_3) = (\mathbf{y} - x_0 + \mathbf{b}_1 x_0 - \mathbf{b}_1 x_2)$

Cost: 1 constraints and 0 additional variables.

Note that, in the 4-to-1 multiplexer we can switch only between known values. This is because for a control bit number ≥ 1, we get squared bits in the equation $\mathbf{z} = (1-\mathbf{b}_0)(1-\mathbf{b}_1)x_0 + (\mathbf{b}_0)(1-\mathbf{b}_1)x_1 + (1-\mathbf{b}_0)(\mathbf{b}_1)x_2 + (\mathbf{b}_0)(\mathbf{b}_1)x_3$. Hence, if the x values would be variables, we would end with a qubic equation.

The relation 2-1-Multiplex($\mathbf{b} \in \{0,1\}, \mathbf{x}, \mathbf{y}$: $z \in \{\mathbf{x}, \mathbf{y}\}_{[\mathbf{b}]}$), i.e. where z is constant, is realized by the same constrained system as in the basic 2-to-1 multiplexer. The same applies for larger multiplexers.

In order to construct larger multiplexers we can simply chain smaller multiplexers together. For, example in order to build a 4-to-1 multiplexer for variables instead of constants, we can simply take two 2-to-1 multiplexers for the initial inputs which feed the output multiplexer. Analogously, we can construct a 8-to-1 multiplexer, but taking two 4-to-1 multiplexers and chaining their output to a 2-to-1 multiplexer.

Additionally, we will need a constraint system for switching points on elliptic curves, i.e. take 2 input points and 1 output point. This is realized as follows.

2-1-PSwitch($\mathbf{b} \in \{0,1\}, \mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3 \in \mathbb{G}$: $\mathbf{g}_3 \in \{\mathbf{g}_1, \mathbf{g}_2\}_{[\mathbf{b}]}$):

1. 2-1-Multiplex($\mathbf{b}, \mathbf{x}_1, \mathbf{x}_2$: $(\mathbf{x}_3 \in \{\mathbf{x}_1, \mathbf{x}_2\}_{[\mathbf{b}]})$
2. 2-1-Multiplex($\mathbf{b}, \mathbf{y}_1, \mathbf{y}_2$: $(\mathbf{y}_3 \in \{\mathbf{y}_1, \mathbf{y}_2\}_{[\mathbf{b}]})$

Cost: 2 constraints and 0 additional variables.

In general for multiplexing n vectors to 1, where each vector has m coordinates, we need m, n-to-1 multiplexers. Note also that a 4-to-1 point multiplexer may be constructed analogously, and since the 4-to-1 integer multiplexer costs just 1 constraint, we have that the 4-to-1 point multiplexer cost is 2 constraints. We depict a point multiplexer for 2^n public points. As we see, the construction is recursive and calls itself for a decremented n. When we get a call for $n = 2$ we use the 4-to-1 point multiplexer. At Table A we depict some initial values of the f_{pm} and g_{pm} functions.

2^n-1-PSwitch($\mathbf{b} = [\mathbf{b}_i \in \{0,1\}]_{i=0}^{n-1}, \mathbf{g} \in \mathbb{G}$: $\mathbf{g} = \{[g_i]_{i=0}^{2^n-1}\}_{[\mathbf{b}_i]_{i=0}^{n-1}}$):

1. 2^{n-1}-1-PSwitch($[\mathbf{b}_i]_{i=0}^{n-2}, [\mathbf{g}_i]_{i=0}^{2^n/2-1}, \mathfrak{g}_0$: $\mathfrak{g}_0 \in \{[\mathbf{g}_i]_{i=0}^{2^n/2-1}\})_{[\mathbf{b}_i]_{i=0}^{n-2}}$)
2. 2^{n-1}-1-PSwitch($[\mathbf{b}_i]_{i=0}^{n-2}, [\mathbf{g}_i]_{i=2^n/2}^{2^n-1}, \mathfrak{g}_1$: $\mathfrak{g}_1 \in \{[\mathbf{g}_i]_{i=2^n/2}^{2^n-1}\}_{[\mathbf{b}_i]_{i=0}^{n-2}}$)
3. 2-1-PSwitch($\mathbf{b}_{n-1}, \mathfrak{g}_0, \mathfrak{g}_1, \mathbf{g}$: $\mathbf{g} \in \{\mathfrak{g}_0, \mathfrak{g}_1\}_{[\mathbf{b}_{n-1}]}$)

Cost: $f_{pm}(n)$ constraints and $g_{pm}(n)$ additional variables.

The costs functions $f_{pm}(n)$ and $g_{pm}(n)$ of the 2^n-to-1 point multiplexer are as follows.

$$f_{pm}(n) = \begin{cases} n & \text{if } n \leq 2 \\ 2f_{pm}(n-1) + 2 & \text{if } n \geq 3 \end{cases}$$

$$g_{pm}(n) = \begin{cases} 0 & \text{if } n \leq 2 \\ 4 & \text{if } n = 3 \\ 2g_{pm}(n-1) + 4 & \text{if } n \geq 4 \end{cases}$$

n	2	3	4	5	6
$f_{pm}(n)$	2	6	14	30	62
$g_{pm}(n)$	0	4	12	28	60

Table 18: Cost of point multiplexers

Elliptic Curve Arithmetic. Below we show constraints for operations on the group $\mathbb{G}$ as specified in Section 5, i.e. operation on points on Twisted Edwards curves. In particular we cover point addition and scalar multiplication. For simplicity of the notation, we will denote $g_i = (x_i, y_i) \in \mathbb{G}$, i.e. a point g_i indexed with i, has coordinates (x_i, y_i).

Let us first focus on the formulas which involve the coefficient d, i.e. formulas given by Definition 15 and Definition 16. Below we have the constraints for adding a secret point to a known point.

EdAddConst($\mathbf{g}_1, \mathbf{g}_3 \in \mathbb{G}$: $\mathbf{g}_3 = \mathbf{g}_1 \cdot g_2$):

1. $(\mathbf{x}_1) \cdot (\mathbf{y}_1) = (\mathfrak{a})$
2. $(1 + dx_2y_2\mathfrak{a}) \cdot (\mathbf{x}_3) = (\mathbf{x}_1y_2 + \mathbf{y}_1x_2)$
3. $(1 - dx_2y_2\mathfrak{a}) \cdot (\mathbf{y}_3) = (\mathbf{y}_1y_2 - \mathbf{x}_1ax_2)$

Cost: 3 constraints and 1 additional variables.

Note that the relation EdAddConst($\mathbf{g}_1 \in \mathbb{G}$: $g_3 = \mathbf{g}_1 \cdot g_2$) might be done exactly as EdAddConst($\mathbf{g}_1, \mathbf{g}_3 \in \mathbb{G}$: $\mathbf{g}_3 = \mathbf{g}_1 \cdot g_2$), with that difference that g_3 is a known constant. Addition of two secret points is slightly more complex.

EdAdd($\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3 \in \mathbb{G}$: $\mathbf{g}_3 = \mathbf{g}_1 \cdot \mathbf{g}_2$):

1. $(\mathbf{x}_1) \cdot (\mathbf{y}_2) = (\mathfrak{a})$
2. $(\mathbf{y}_1) \cdot (\mathbf{x}_2) = (\mathfrak{b})$
3. $(\mathbf{y}_1) \cdot (\mathbf{y}_2) = (\mathfrak{c})$
4. $(\mathbf{x}_1) \cdot (\mathbf{x}_2) = (\mathfrak{d})$
5. $(\mathfrak{c}) \cdot (\mathfrak{d}) = (\mathfrak{e})$
6. $(1 + d\mathfrak{e}) \cdot (\mathbf{x}_3) = (\mathfrak{a} + \mathfrak{b})$
7. $(1 - d\mathfrak{e}) \cdot (\mathbf{y}_3) = (\mathfrak{c} - a\mathfrak{d})$

Cost: 7 constraints and 5 additional variables.

Fast point doubling may be done by using the formula without the d coefficient as given by Definition 18.

EdDouble($\mathbf{g}_1, \mathbf{g}_3 \in \mathbb{G}$: $\mathbf{g}_3 = 2\mathbf{g}_1$):

1. $(\mathbf{x}_1) \cdot (\mathbf{y}_1) = (\mathfrak{a})$
2. $(\mathbf{x}_1) \cdot (\mathbf{x}_1) = (\mathfrak{b})$
3. $(\mathbf{y}_1) \cdot (\mathbf{y}_1) = (\mathfrak{c})$
4. $(a\mathfrak{b} + \mathfrak{c}) \cdot (\mathbf{x}_3) = (2\mathfrak{a})$
5. $(2 - a\mathfrak{b} - \mathfrak{c}) \cdot (\mathbf{y}_3) = (\mathfrak{c} - a\mathfrak{b})$

Cost: 5 constraints and 3 additional variables.

Unfortunately, it turns out that using the formula from Definition 17 results in a more complex constraint system, than addition using the formula from Definition 15.

Additionally, we will need point addition which is conditioned on a bit. This is done by simply adding two points, and then chaining the output with a 2-1 point multiplexer. Below we show conditional point addition for two cases. The first case is when the added point is a known constant, whereas the second case is when both input points are variables. The design concept for both cases is exactly the same, the difference is the cost of both circuits.

$\mathsf{EdAddConstIF}(\ \mathbf{g}_1, \mathbf{g}_3 \in \mathbb{G}, \mathbf{b} \in \{0,1\}\text{: } (\mathbf{g}_3 = \mathbf{g}_1)_{\mathbf{b}=0}, (\mathbf{g}_3 = \mathbf{g}_1 \cdot g_2)_{\mathbf{b}=1})\textbf{:}$

1. $\mathsf{EdAddConst}(\mathbf{g}_1, \mathfrak{g} \in \mathbb{G} : \mathfrak{g} = \mathbf{g}_1 \cdot g_2)$
2. $\mathsf{2\text{-}1\text{-}PSwitch}(\mathbf{b}, \mathbf{g}_1, \mathfrak{g}, \mathbf{g}_3 : (\mathbf{g}_3 = \mathbf{g}_1)_{\mathbf{b}=0}, (\mathbf{g}_3 = \mathfrak{g})_{\mathbf{b}=1})$

Cost: 5 constraints and 3 additional variables.

$\mathsf{EdAddIF}(\ \mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3 \in \mathbb{G}, \mathbf{b} \in \{0,1\}\text{: } (\mathbf{g}_3 = \mathbf{g}_1)_{\mathbf{b}=0}, (\mathbf{g}_3 = \mathbf{g}_1 \cdot \mathbf{g}_2)_{\mathbf{b}=1})\textbf{:}$

1. $\mathsf{EdAdd}(\mathbf{g}_1, \mathfrak{g} \in \mathbb{G} : \mathfrak{g} = \mathbf{g}_1 \cdot \mathbf{g}_2)$
2. $\mathsf{2\text{-}1\text{-}PSwitch}(\mathbf{b}, \mathbf{g}_1, \mathfrak{g}, \mathbf{g}_3 : (\mathbf{g}_3 = \mathbf{g}_1)_{\mathbf{b}=0}, (\mathbf{g}_3 = \mathfrak{g})_{\mathbf{b}=1})$

Cost: 9 constraints and 7 additional variables.

Similarly as in case of the multiplexers, we may implement the relation $\mathsf{EdAddConstIF}(\ \mathbf{g}_1 \in \mathbb{G}, \mathbf{b} \in \{0,1\}\text{: } g_3 = \mathbf{g}_1 \cdot g_2 \text{ if } \mathbf{b} = 0, g_3 = \mathbf{g}_1 \text{ if } \mathbf{b} = 1)$ the same way as the conditioned addition above, simply by using a different kind of multiplexer. The cost remains the same.

Now we will construct a constraint system for a hash function based on DLP. The constraint system uses a precomputed table $\{h_0, h_1, h_2, \ldots, h_n\}$. Note that we always add h_n and the bit size of the input to the CRHF is n.

$\mathsf{CRHFDLP}([\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, \mathbf{h}\text{: } h_n \prod_{i=0}^{n-1} h_i^{\mathbf{x}_i} = \mathbf{h})\textbf{:}$

1. $\mathsf{2\text{-}1\text{-}PSwitch}(\mathbf{x}_0, \mathfrak{g}_0 : (\mathfrak{g}_0 = (0,1))_{\mathbf{x}_0=0}, (\mathfrak{g}_0 = h_0)_{\mathbf{x}_0=1})$
2. $\forall_{i \in \{1,\ldots,n-1\}}$
    (a) $\mathsf{EdAddConstIF}(\mathfrak{g}_{i-1}, \mathfrak{g}_i, \mathbf{x}_i : (\mathfrak{g}_i = \mathfrak{g}_{i-1})_{\mathbf{x}_i=0}, (\mathfrak{g}_i = \mathfrak{g}_{i-1} \cdot h_i)_{\mathbf{x}_i=1})$
3. $\mathsf{EdAddConst}(\mathfrak{g}_{n-1}, \mathbf{h} : \mathbf{h} = \mathfrak{g}_{n-1} \cdot h_n)$

Cost: $5n$ constraints and $5n - 2$ additional variables.

An analogical constraint system, for which the target point h is known proceeds as above, except the last step is done by $\mathsf{EdAddConstIF}(\mathfrak{g}_{n-2}, \mathbf{x}_{n-1} \in \{0,1\} : h = \mathfrak{g}_{n-2} \text{ if } \mathbf{x}_{n-1} = 0, h = \mathfrak{g}_{n-2} \cdot h_{n-1} \text{ if } \mathbf{x}_{n-1} = 1)$. It is easy to see that the cost is exactly the same.

Now, we will construct a constraint system for scalar multiplication where the base point is unknown. Basically, the constrained system represents the Double-and-Add algorithm.

```
Double-Add($[\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, \mathbf{g}, \mathbf{h}$: $\mathbf{g}^x = \mathbf{h}$):
  1. EdDouble($\mathbf{g}, \mathfrak{h}_0 : \mathfrak{h}_0 = \mathbf{g} \cdot \mathbf{g}$)
  2. $\forall_{i \in \{1,\dots,n-1\}}$
      (a) EdDouble($\mathfrak{h}_{i-1}, \mathfrak{h}_i : \mathfrak{h}_i = \mathfrak{h}_{i-1} \cdot \mathfrak{h}_{i-1}$)
  3. 2-1-PSwitch($\mathbf{x}_0, \mathbf{g} : (\mathfrak{h}_0 = (0,1))_{\mathbf{x}_0=0}, (\mathfrak{h}_0 = \mathbf{g})_{\mathbf{x}_0=1}$)
  4. $\forall_{i \in \{1,\dots,n-2\}}$
      (a) EdAddIF($\mathfrak{g}_{i-1}, \mathfrak{g}_i, \mathbf{x}_i : (\mathfrak{g}_i = \mathfrak{g}_{i-1})_{\mathbf{x}_i=0}, (\mathfrak{g}_i = \mathfrak{g}_{i-1} \cdot \mathfrak{h}_i)_{\mathbf{x}_i=1}$)
  5. EdAddIF($\mathfrak{g}_{n-2}, \mathbf{h}, \mathbf{x}_{n-1} : (\mathbf{h} = \mathfrak{g}_{n-2})_{\mathbf{x}_{n-1}=0}, (\mathbf{h} = \mathfrak{g}_{n-2} \cdot \mathfrak{h}_{n-1})_{\mathbf{x}_{n-1}=1}$)
Cost: $14n - 7$ constraints and $14n - 9$ additional variables.
```

We will also need a system for scalar multiplication where the base point is constant. We can use the fact the base point is constant and perform some precomputation which might significantly improve the efficiency of the circuit. The first method proceeds similarly to CRHFDPL. We will have a precomputed table $[g^{2^i}]_{i=0}^n$, and given the bits of a scalar $[\mathbf{x}_i \in \{0,1\}]_{i=0}^n$ we check that $h = \prod_{i=0}^n g^{2^i \mathbf{x}_i}$. As may be expected, the cost is the same as for CRHFDPL. Let us denote $g^{2^i} = h_i$ for $i \in \{0,\dots,n-1\}$.

```
ScalarMul($[\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, \mathbf{h}$: $\prod_{i=0}^{n-1} g^{2^i \mathbf{x}_i} = \mathbf{h}$):
  1. 2-1-PSwitch($\mathbf{x}_0, \mathfrak{g}_0 : (\mathfrak{g}_0 = (0,1))_{\mathbf{x}_0=0}, (\mathfrak{g}_0 = h_0)_{\mathbf{x}_0=1}$)
  2. $\forall_{i \in \{1,\dots,n-2\}}$
      (a) EdAddConstIF($\mathfrak{g}_{i-1}, \mathfrak{g}_i, \mathbf{x}_i : (\mathfrak{g}_i = \mathfrak{g}_{i-1})_{\mathbf{x}_i=0}, (\mathfrak{g}_i = \mathfrak{g}_{i-1} \cdot h_i)_{\mathbf{x}_i=1}$)
  3. EdAddConstIF($\mathfrak{g}_{n-2}, \mathbf{h}, \mathbf{x}_{n-1} : (\mathbf{h} = \mathfrak{g}_{n-2})_{\mathbf{x}_i=0}, (\mathbf{h} = \mathfrak{g}_{n-2} \cdot h_{n-1})_{\mathbf{x}_i=1}$)
Cost: $5n - 3$ constraints and $5(n - 1)$ additional variables.
```

In particular, we will use the sliding window method. The idea is that we divide the secret bit string into smaller bit strings, precompute all possible output points for these smaller bit strings. Then in the circuit, instead of checking multiplications within the small bit strings, we sample out the precomputed point using a multiplexer. In more detail, to check $\mathbf{h} = g^{\sum_{i=0}^n 2^i \mathbf{x}_i}$, we will set a window size w and divide the input bits $\{\mathbf{x}_0, \dots, \mathbf{x}_n\}$ into groups of w bits. Finally, the relation which needs to be checked may be written as $\mathbf{h} = \prod_{j=0}^{n/w-1} g^{\sum_{k=0}^w 2^{jw+k} \mathbf{x}_{jw+k}}$. Note that if we precompute $g^{\sum_{k=0}^w 2^{jw+k} \mathbf{x}_{jw+k}}$ for all possible bits in $[\mathbf{x}_{jw+k}]_{k=0}^w$, then we do not need to perform the check in the circuit, but we will use a 2^w-to-1 point multiplexer to sample out the correct value. However, due to the pre-computation step, we will need to publish a lookup table of size $\frac{n2^w}{w}$ group elements. The general procedure is depicted below.

```
WindowMul($[\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, \mathbf{h}$: $g^x = \mathbf{h}$):
  1. $2^w$-PSwitch($[\mathbf{x}_k]_{k=0}^{w}, \mathfrak{h}_0 : \mathfrak{h}_0 = g^{\sum_{k=0}^{w} 2^k \mathbf{x}_k}$)
  2. $\forall_{j \in \{1,\ldots,n/w-2\}}$
     (a) $2^w$-PSwitch($[\mathbf{x}_{jw+k}]_{k=0}^{w}, \mathfrak{g}_j : \mathfrak{g}_j = g^{\sum_{k=0}^{w} 2^{jw+k} \mathbf{x}_{jw+k}}$)
     (b) EdAdd($\mathfrak{h}_{j-1}, \mathfrak{g}_j, \mathfrak{h}_j : \mathfrak{h}_j = \mathfrak{h}_{j-1} \cdot \mathfrak{g}_j$)
  3. $2^w$-PSwitch($[\mathbf{x}_{n/w-1+k}]_{k=0}^{w}, \mathfrak{g}_{n/w-1} : \mathfrak{g}_{n/w-1} = g^{\sum_{k=0}^{w} 2^{n-w+k} \mathbf{x}_{n-w+k}}$)
  4. EdAdd($\mathfrak{h}_{n/w-2}, \mathfrak{g}_{n/w-1}, \mathbf{h} : \mathbf{h} = \mathfrak{h}_{n/w-2} \cdot \mathfrak{g}_{n/w-1}$)
Cost: $\frac{n}{w}(f_{pm}(w) + 7) - 7$ constraints and $\frac{n}{w}(g_{pm}(w) + 9) - 9$ additional variables.
```

The constraint complexity and size complexity of the above heavily depends on the window size w. The window determines what kind of multiplexer we need to use and how many additions we will need to perform.

For sake of completeness, below we show an analogous constraint system for the DPL CRHF. In this case the relation is rewritten as $\mathbf{h} = h_n \prod_{j=0}^{n/w-1} \prod_{k=0}^{w} h_{jw+k}^{\mathbf{x}_{jw+k}}$. Thus, for every $j \in \{0,\ldots,n/w-1\}$ we will need to pre-compute $\prod_{k=0}^{w} h_{jw+k}^{\mathbf{x}_{jw+k}}$ for every combination of the bits. In order to save one multiplication by the constant point h_n, we simply add this point in one of the windows.

```
WindowCRHFDLP($[\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, \mathbf{h}$: $\prod_{i=0}^{n-1} h_i^{\mathbf{x}_i} = \mathbf{h}$):
  1. $2^w$-PSwitch($[\mathbf{x}_k]_{k=0}^{w}, \mathfrak{h}_0 : \mathfrak{h}_0 = \prod_{k=0}^{w} h_k^{\mathbf{x}_k}$)
  2. $\forall_{j \in \{1,\ldots,n/w-2\}}$
     (a) $2^w$-PSwitch($[\mathbf{x}_{jw+k}]_{k=0}^{w}, \mathfrak{g}_j : \mathfrak{g}_j = \prod_{k=0}^{w} h_{jw+k}^{\mathbf{x}_{jw+k}}$)
     (b) EdAdd($\mathfrak{h}_{j-1}, \mathfrak{g}_j, \mathfrak{h}_j : \mathfrak{h}_j = \mathfrak{h}_{j-1} \cdot \mathfrak{g}_j$)
  3. $2^w$-PSwitch($[\mathbf{x}_{n/w-1+k}]_{k=0}^{w}, \mathfrak{g}_{n/w-1} : \mathfrak{g}_{n/w-1} = \prod_{k=0}^{w} h_{n-w+k}^{\mathbf{x}_{n-w+k}}$)
  4. EdAdd($\mathfrak{h}_{n/w-2}, \mathfrak{g}_{n/w-1}, \mathbf{h} : \mathbf{h} = \mathfrak{h}_{n/w-2} \cdot \mathfrak{g}_{n/w-1}$)
Cost: $\frac{n}{w}(f_{pm}(w) + 7) - 7$ constraints and $\frac{n}{w}(g_{pm}(w) + 9) - 9$ additional variables.
```

Finally, using the previous constraint systems, we are now able to design a system for the Pedersen commitment scheme. In our case, however, we make the description more general, i.e. we show a system for an arbitrary number of bits of the message to be committed.

```
$\mathsf{WindowCommP}([\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, [\mathbf{r}_i \in \{0,1\}]_{i=0}^{\log(p)-1}, \mathbf{c}\colon g^{\mathbf{x}} \cdot h^{\mathbf{r}} = \mathbf{c})$:
    1. $\mathsf{WindowMul}([\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, \mathfrak{g}\colon g^{\mathbf{x}} = \mathfrak{g})$
    2. $\mathsf{WindowMul}([\mathbf{x}_i \in \{0,1\}]_{i=0}^{n-1}, \mathfrak{h}\colon h^{\mathbf{r}} = \mathfrak{h})$
    3. $\mathsf{EdAdd}(\mathfrak{g}, \mathfrak{h}, \mathbf{c}\colon \mathbf{c} = \mathfrak{g} \cdot \mathfrak{h})$
Cost: $\frac{n+\log(p)}{w}(f_{pm}(w) + 7) - 7$ constraints and $\frac{n+\log(p)}{w}(g_{pm}(w) + 9) - 9$ additional variables.
```

Ajtai Hash Function. In this paragraph we show a simple constraint system for the Ajtai hash function.

```
$\mathsf{Ajtai}([\mathbf{b}_i \in \{0,1\}]_{i=0}^{m-1}, [\mathbf{z}_j]_{j=0}^{n-1}\colon \forall_{j=0}^{n-1} z_j = \sum_{i=0}^{m-1} \mathbf{b}_i a_{i,j})$:
    1. $\forall_{j=0}^{n-1}$
        (a) $(\sum_{i=0}^{n-1} \mathbf{b}_i a_j) \cdot (1) = (\mathbf{z}_j)$
Cost: $n$ constraints and 0 additional variables.
```

Merkle Trees. In this paragraph we will design constraint systems to show that an element is in a Merkle tree. In order to keep our design as general as possible, we will assume that we have a black box CRHF H. Later H can be instantiated with any hash function which matches our specification. In particular, we will use the DLP and Ajtai hash functions. We specify the general hash function as $\mathsf{H} : \{0,1\}^n \to \mathbb{F}_q^m$, thus it maps a bit vector of length n into a vector of integers of length m. Since, we will use the hash function in k-ary hash trees we require that $n = km \cdot \log(q)$. The hash tree is binary, for the special case of $K = k$. We denote the constraint system for checking the input/output consistency of H as $\mathsf{Check\text{-}H}(x \in \{0,1\}^{km\log(q)}, h \in \mathbb{F}_q^m\colon \mathsf{H}(x) = h)$. Moreover, we denote as $f_{\mathsf{H}}(k,m)$ and $g_{\mathsf{H}}(k,m)$, the number of constraints and additional variables of this verification procedure respectively.

In our case, the constraint system will take as secret input a leaf and its path to the root. The goal is to show that the leaf and its path belongs to the tree, but we cannot reveal which leaf and path it is. Hence the constraint system cannot, by itself, reveal whether a node is the right or left input to its parent. Therefore, we will need to show that a particular shift of the inputs to H is correct.

We start with a constraint system which checks whether two output integers are a correct shift of the two input integers.

2-Shift($\mathbf{b} \in \{0,1\}, \mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}'$: $((\mathbf{x}', \mathbf{y}') = (\mathbf{x}, \mathbf{y}))_{b=0}, ((\mathbf{x}', \mathbf{y}') = (\mathbf{y}, \mathbf{x}))_{b=1}$):

1. $(\mathbf{b})(\mathbf{y} - \mathbf{x}) = (\mathbf{x}' + \mathbf{x})$
2. $(\mathbf{b})(\mathbf{x} - \mathbf{y}) = (\mathbf{y}' + \mathbf{y})$

Cost: 2 constraints and 0 additional variables.

Now we can generalize the above shift method, to the case of k elements.

k-Shift($[\mathbf{b}_i \in \{0,1\}]_{i=0}^{k-2}, [\mathbf{x}_i]_{i=0}^{k-1}, [\mathbf{x}'_i]_{i=0}^{k-1}$: …):

1. 2-Shift($\mathbf{b}_0, \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}'_0, \mathfrak{y}_1$: $((\mathbf{x}'_0, \mathfrak{y}_1) = (\mathbf{x}_0, \mathbf{x}_1))_{\mathbf{b}_0=0}, ((\mathbf{x}'_0, \mathfrak{y}_1) = (\mathbf{x}_1, \mathbf{x}_0))_{\mathbf{b}_0=1}$)
2. $\forall_{i \in \{1,\ldots,k-3\}}$
   (a) 2-Shift($\mathbf{b}_i, \mathfrak{y}_i, \mathbf{x}_{i+1}, \mathbf{x}'_i, \mathfrak{y}_{i+1}$: $((\mathbf{x}'_i, \mathfrak{y}_{i+1}) = (\mathbf{x}_{i+1}, \mathfrak{y}_i))_{\mathbf{b}_i=0}, ((\mathbf{x}'_i, \mathfrak{y}_{i+1}) = (\mathfrak{y}_i, \mathbf{x}_{i+1}))_{\mathbf{b}_i=1}$)
3. 2-Shift($\mathbf{b}_{k-2}, \mathfrak{y}_{k-2}, \mathbf{x}_{k-1}, \mathbf{x}'_{k-2}, \mathbf{x}'_{k-1}$: $((\mathbf{x}'_{k-2}, \mathbf{x}'_{k-1}) = (\mathbf{x}_{k-1}, \mathfrak{y}_{k-2}))_{\mathbf{b}_{k-2}=0}, ((\mathbf{x}'_{k-2}, \mathbf{x}_{k-1}) = (\mathfrak{y}_{k-2}, \mathbf{x}_{k-1}))_{\mathbf{b}_{k-2}=1}$)

Cost: $2(k-1)$ constraints and $k-2$ additional variables.

Below, we show a strightforward application of the k-Shift constraint system to vectors of m elements. The idea is simply to make a k-Shift for every coordinate, all with the same shift specification (i.e. the same control bits). We denote as $\mathbf{x}_{i,j}$ the j-th coordinate in the i-th vector $\mathbf{x}_i$.

(m,k)-Shift($\vec{\mathbf{b}} = [\mathbf{b}_i \in \{0,1\}]_{i=0}^{k-2}, [\vec{\mathbf{x}}_i, \vec{\mathbf{x}}'_i \in \mathbb{F}_q^m]_{i=0}^{k}$: …):

1. $\forall_{j=0}^{m-1}$
   (a) k-Shift($\vec{\mathbf{b}}, [\mathbf{x}_{i,j}]_{i=0}^{k-1}, [\mathbf{x}'_{i,j}]_{i=0}^{k-1}$: …)

Cost: $2m(k-1)$ constraints and $m(k-2)$ additional variables.

An example use case is shifting points. For points we will have an $m = 2$ dimensional vector, thus we shift all coordinates by the same position. It is easy to see that the cost of shifting two points, is 4 constraints and 0 additional variables, given that the control bits are already checked.

$\mathsf{SplitVector}([\mathbf{a}]_{i=0}^{m-1}, [\mathbf{b}_{i,j}]_{i=0,j=0}^{\ell-1,m-1}\colon\ \forall_{i\in\{0,\ldots,m-1\}}\mathbf{a}_i = \sum_{j=0}^{\ell-1} 2^j\mathbf{b}_{i,j} \wedge [\mathbf{b}_{i,j} \in \{0,1\}]_{j=0}^{\ell-1})$**:**

1. $\forall_{i\in\{0,\ldots,m\}}$
    (a) $\mathsf{SplitIntegerToBit}(\mathbf{a}_i, [\mathbf{b}_{i,j}]_{j=0}^{\ell-1}\colon\ \mathbf{a}_i = \sum_{j=0}^{\ell-1} 2^j\mathbf{b}_{i,j} \wedge [\mathbf{b}_{i,j} \in \{0,1\}]_{j=0}^{\ell-1})$

Cost: $m(\ell+1)$ constraints and 0 additional variables.

Finally, we show the constraint system for verifying membership of a k-ary hash tree of height N. The notational convention is as follows. $\mathbf{x}_0 \in \mathbb{F}_q^m$ is the element which we want to proof membership of. The idea is to go through each level of the tree, and shift the input element to its right position, hiding the subtree in which the input element is. As each level, the input element is treated as the first element for the shift. Then, we split all the integers into bits and pass this bits to the hash function, obtaining an input element for the next level.

$(N,k,m,\mathsf{H})\text{-}\mathsf{Tree}([\mathbf{b}_j \in \{0,1\}^{k-1}]_{j=0}^{N-1}, [\mathbf{x}_j \in \mathbb{F}_q^m]_{j=0}^{N-1}, M_{root} \in \mathbb{F}_q^m\colon\ [\mathbf{x}_j]_{j=0}^{N-1}$ is a path to M_{root})**:**

1. $\forall_{j=0}^{N-2}$
    (a) $(k,m)\text{-}\mathsf{Shift}(\mathbf{b}_j, \mathbf{x}_j, [\mathfrak{y}_{j,i}]_{i=1}^{k-1}, \mathfrak{s}_j = [\mathfrak{s}_{j,i}]_{i=0}^{k-1}\colon \ldots)$
    (b) $\mathsf{SplitVector}(\mathfrak{s}_j, \mathfrak{b}_j\colon\ \mathfrak{b}_j = \mathfrak{s}_j \wedge \mathfrak{b}_j \in \{0,1\}^{km\log(q)})$
    (c) $\mathsf{Check\text{-}H}(\mathfrak{b}_j, \mathbf{x}_{j+1}\colon\ \mathsf{H}(\mathfrak{b}_j) = \mathbf{x}_{j+1})$
2. $(k,m)\text{-}\mathsf{Shift}(\mathbf{b}_{N-1}, \mathbf{x}_{N-1}, [\mathfrak{y}_{N-1,i}]_{i=1}^{k-1}, \mathfrak{s}_j = [\mathfrak{s}_{N-1,i}]_{i=0}^{k-1}\colon \ldots)$
3. $\mathsf{SplitVector}(\mathfrak{s}_{N-1}, \mathfrak{b}_{N-1}\colon\ \mathfrak{b}_{N-1} = \mathfrak{s}_{N-1} \wedge \mathfrak{b}_{N-1} \in \{0,1\}^{km\log(q)})$
4. $\mathsf{Check\text{-}H}(\mathfrak{b}_{N-1}, M_{root}\colon\ \mathsf{H}(\mathfrak{b}_{N-1}) = M_{root})$

Cost: $N(mk(\log(q)+3) + f_{\mathsf{H}}(k,m) - 2m)$ constraints and $N(mk(\log(q)+3) + g_{\mathsf{H}}(k,m) - 3m)$ additional variables.

Counting the cost of checking the Merkle tree is a bit more complicated. Let us first observe that we are repeating tree operations N times, i.e. one on each level of the tree. The shift operation costs $2m(k-1)$ constraints and $m(k-2)$ additional variables, however we introduce additionally $m(2k-1)$ variables. Now all the mk variables are treated as a single vector (concatenation is for free), and splitting it into bits costs $mk(\log(q)+1)$ constraints and we introduce $mk\log(q)$ new variables. Finally, checking the hash cost $f_{\mathsf{H}}(k,m)$ constraints and $g_{\mathsf{H}}(k,m)$ new variables.

For the path we need $N(k-1)$ control bits and $N \cdot m$ elements (N hash values).

B Dodis-Yampolskiy PRF

In this section we recall the security reduction for the Dodis-Yampolskiy Verifiable PRF. However, we slightly alter the reduction. In the original work [27] the the reduction is made to the q-DBDHI assumption. However, as we don't need pairings in the group on which the PRF is instantiated we reduce the pseudorandomness to the weaker q-DHI problem. In [27] the reduction also assumed that the input space is small, in order to guess the point from the challenge. In our case the input space will be the output of a programmable random oracle and the guess will be made for one of the oracle queries. beside, this changes the reduction proceeds identically.

Proof. Input to the reduction: The solver $\mathcal{S}$ is given a tuple $(g, g^{\alpha}, \ldots, g^{\alpha^q}) \in \mathbb{G}$ and an element $\Gamma \in \mathbb{G}$. The goal is to distinguish whether $\Gamma = g^{1/\alpha}$ or random.

key generation: We guess that $\mathcal{A}$ will choose to distinguish $x_0 \in \mathbb{Z}_p$, i.e. one of the outputs of the q_{H} queries to the hash oracle. Let $\beta = \alpha - x_0$. We can compute $(g^{\beta}, \ldots, g^{\beta^q})$ form $(g^{\alpha}, \ldots, g^{\alpha^q})$. Let $f(z)$ be define as follows.

$$f(z) = \prod_{w_i, x_i \neq x_0} (z + w_i) = \sum_{j=0}^{q-1} c_j z^j,$$

for some $c_0, \ldots, c_{q-1}$ and whether w_i are the outputs of the hash oracle. Then, we compute $h = g^{f(\beta)} = \prod_{j=0}^{q-1} (g^{\beta^j}) c^j$ and $h^{\beta} = \prod_{j=1}^{q} (g^{\beta^j}) c^{j-1}$. We set the public key as h^{β} and the basepoint as h.

responding to oracle queries: For the i-th query, if $x_i = x_0$ we fail. Otherwise, we compute

$$g^{f_i(\beta)} = \prod_{j=0}^{q-2} g^{d_j \beta^j} = h^{1/(x_i + \beta)}$$

where $f_i(z)$ is defined as

$$f_i(z) = f(z)/(z + x_i) = \sum_{j=0}^{q-2} d_j z^j,$$

for some coefficients $d_0, \ldots, d_{q-2}$.

Challenge: For the challenge query if $x^* \neq x_0$, then we fail. Otherwise, we have that x_0 is not a root of f, thus

$$f(z)/(z + x_0) = \sum_{j=0}^{q-2} \gamma_j z^j + \gamma_{-1}/(z + x_0)$$

where $\gamma_{-1} \neq 0$. Then we output

$$\Gamma^* = \Gamma^{\gamma_{-1}} \cdot \prod_{j=0}^{q-2} g^{\gamma_j \beta^j}.$$

Note that if $\Gamma = g^{1/\alpha}$, then $\Gamma^* = g^{f(\beta)/(\beta - x_0)}$. Otherwise, Γ^* is uniformly distributed.

□